

AI Handbook v.1-2  
prelim. ed. produced  
at CST.



Handbook of Artificial Intelligence

Edited by Avron Barr and Edward A. Feigenbaum  
Handbook of Artificial Intelligence

— To Ed —

Avron Barr  
Jan Smith

Rune Gordon  
Patricia  
Bill Conroy  
Jan Atkins  
Steve Feppel  
S. J. Westfold

Helen Tognetti  
Mel Abt  
Jan Davidson  
Larry Long  
Bob Eschlyer  
Sara Buchanan  
Doug Lenat  
Terry

August, 1979



# Handbook of Artificial Intelligence

Edited by Avron Barr and Edward A. Feigenbaum

PRELIMINARY EDITION

Computer Science Department

Stanford University



**August 1979  
Stanford, California**

**This research was supported by both the Defense Advanced Research Projects Agency (ARPA Order No. 3423, Contract No. MDA 903-77-C-0322) and the National Institutes of Health (Contract No. NIH RR-00785-06). The views and conclusions of this document should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Institutes of Health, or the United States Government.**

**Copyright Notice: The material herein is copyright protected. Permission to quote or reproduce in any form must be obtained from the Editors. Such permission is hereby granted to agencies of the United States Government.**



# Contents

*The material in this preliminary edition will appear in the first two volumes of the Handbook, to be published in Winter 1979-80.*

Foreword	9
List of Contributors	11
<b>Search</b>	
A. Overview	15
B. Problem representation	23
1. State-space representation	23
2. Problem-reduction representation	26
3. Game trees	32
C. Search Methods	34
1. Blind state-space search	34
2. Blind AND/OR graph search	41
3. Heuristic state-space search	45
a. Basic concepts in heuristic search	45
b. A*: optimal search for an optimal solution	49
c. Relaxing the optimality requirement	51
d. Bidirectional search	55
4. Heuristic search of an AND/OR graph	57
5. Game tree search	65
a. Minimax	65
b. Alpha-beta	68
c. Heuristics in game tree search	72
D. Example search programs	83
1. Logic Theorist	83
2. General Problem Solver	86
3. Gelernter's geometry theorem-proving machine	91
4. Symbolic Integration Programs	94
5. STRIPS	98
6. ABSTRIPS	104
<b>Representation of Knowledge</b>	
A. Issues and problems in representation theory	121
B. Survey of representation techniques	129
C. Representation schemes	137
1. Logic	137
2. Semantic nets	151
3. Production systems	160
4. Procedural representations	173
5. Semantic primitives	183
6. Direct (Analogical) representations	191
7. Higher-level knowledge structures	199



## Natural Language Understanding

A. Overview	223
B. Mechanical translation	228
C. Grammars	233
1. Formal grammars	233
2. Transformational grammar	238
3. Systemic grammar	242
4. Case Grammars	244
D. Parsing techniques	247
1. Overview of parsing techniques	247
2. Augmented transition nets	252
3. GSP	256
E. Text generation	261
F. Natural language processing systems	267
1. Early NL systems	267
2. Wilks's machine translation work	273
3. LUNAR	276
4. SHRDLU	279
5. MARGIE	283
6. SAM and PAM	287
7. LIFER	296

## Speech Understanding Systems

A. Overview	319
1. HEARSAY I	326
B. Some early ARPA speech systems	329
1. DRAGON	329
2. SPEECHLIS	329
C. Recent Speech Systems	335
1. HARPY	335
2. HEARSAY II	338
3. HWIM	342
4. SRI-SDC System	345

## AI Programming Languages

A. Historical Overview of AI Languages	355
B. AI Language Features	370
1. Overview of Language Features	370
2. Data Structures	376
3. Control Structures	388
4. Pattern Matching	399
5. Environment	407
C. Example AI Programming Languages	416
1. LISP	416
2. PLANNER and CONNIVER	433
3. QLISP	435
4. SAIL	436
5. POP-2	437

### Applications-oriented AI Research -- Part 1

A. Overview of Applications-oriented AI Research	443
B. TEIRESIAS - Issues in Expert Systems Design	449
C. Mathematics	462
1. MACSYMA	462
2. AM	468
D. Miscellaneous Applications Research	482
1. SRI Computer-based consultant	482
2. PROSPECTOR	487
3. RITA	495
4. AI Applications in Information Retrieval	500

### Applications-oriented AI Research -- Part 2: Chemistry

A. Overview of Applications in Chemistry	521
B. Applications in Chemical Analysis	524
C. The DENDRAL Programs	527
1. DENDRAL	527
2. CONGEN and its extensions	531
3. Meta-DENDRAL	536
D. CRYVALIS	545
E. Applications in Organic Synthesis	555

### Applications-oriented AI Research -- Part 3: Medicine

A. Overview of Medical Applications Research	575
B. MYCIN	581
C. CASNET	589
D. INTERNIST	593
E. Present Illness Program (PIP)	598
F. Digitalis Advisor	602
G. IRIS	605

### Applications-oriented AI Research -- Part 4: Education

A. Historical Overview	617
B. Components of Intelligent CAI Systems	620
C. ICAI Systems	626
1. SCHOLAR	626
2. WHY	631
3. SOPHIE	636
4. WEST	642
5. WUMPUS	649
6. BUGGY	655
7. EXCHECK	659

### Automatic Programming

A. Automatic Programming Overview	675
B. Techniques for Program Specification	684
C. Approaches to AP	689
D. AP Systems	699
1. PSI	699
2. SAFE	708
3. Programmer's Apprentice	714
4. PECOS	721
5. DAEDALUS	726
6. PROTOSYSTEM-1	734
7. NLPQ	739
8. LIBRA - Program Optimization	744



*The following sections are not in the preliminary edition.  
They will appear in the third volume of the Handbook.*

**Theorem Proving**

**Vision**

**Robotics**

**Learning and Inductive Inference**

**Information Processing Psychology**

**Planning and Related Problem-solving Techniques**

## Foreword

Those of us involved in the creation of the Handbook of Artificial Intelligence, both writers and editors, have attempted to make the concepts, methods, tools, and main results of artificial intelligence research accessible to a broad scientific and engineering audience. Currently, AI work is familiar mainly to its practicing specialists and other interested computer scientists. Yet the field is of growing interdisciplinary interest and practical importance. With this book we are trying to build bridges that are easily crossed by engineers, scientists in other fields, and our own computer science colleagues.

In the Handbook we intend to cover the breadth and depth of AI, presenting general overviews of the scientific issues, as well as detailed discussions of particular techniques and important AI systems. Throughout we have tried to keep in mind the reader who is not a specialist in AI.

As the cost of computation continues to fall, new areas of computer applications become potentially viable. For many of these areas, there do not exist mathematical "cores" to structure calculational use of the computer. Such areas will inevitably be served by symbolic models and symbolic inference techniques. Yet those who understand symbolic computation have been speaking largely to themselves for twenty years. We feel that it is urgent for AI to "go public" in the manner intended by the Handbook.

Several other writers have recognized a need for more widespread knowledge of AI and have attempted to help fill the vacuum. Lay reviews, in particular Margaret Boden's *Artificial Intelligence and Natural Man*, have tried to explain what is important and interesting about AI, and how research in AI progresses through our programs. In addition, there are a few textbooks that attempt to present a more detailed view of selected areas of AI, for the serious student of computer science. But no textbook can hope to describe all of the sub-areas, to present brief explanations of the important ideas and techniques, and to review the forty or fifty most important AI systems.

The Handbook contains several different types of articles. Key AI ideas and techniques are described in core articles (e.g., basic concepts in heuristic search, semantic nets). Important individual AI programs (e.g., SHRDLU) are described in separate articles that indicate, among other things, the designer's goal, the techniques employed, and the reasons why the program is important. Overview articles discuss the problems and approaches in each major area. The overview articles should be particularly useful to those who seek a summary of the underlying issues that motivate AI research.

Eventually the Handbook will contain approximately two hundred articles. We hope that the appearance of this material will stimulate interaction and cooperation with other AI research sites. We look forward to being advised of errors of omission and commission. For a field as fast moving as AI, it is important that its practitioners alert us to important developments, so that future editions will reflect this new material. We intend that the Handbook of Artificial Intelligence be a living and changing reference work.

The articles in this edition of the Handbook were written primarily by graduate students in AI at Stanford University, with assistance from graduate students and AI professionals at other institutions. We wish particularly to acknowledge the help from those at Rutgers University, SRI International, Xerox Palo Alto Research Center, MIT, and the RAND Corporation.

Avron Barr  
Edward Feigenbaum

Stanford University  
August, 1979



# Contributors

The following people made the Handbook a reality. Together, over the last four years, they have combed the entire literature of AI and attempted to make a coherent presentation of this very diverse field.

## Section Editors

James Bennett  
Victor Ciesielski, Rutgers  
William Clancey  
Paul Cohen  
James Davidson  
Robert Elschlager

Lawrence Fagan  
Anne Gardner  
Jorge Phillips  
Steve Tappel  
Stephen Westfold

and Helen Tognetti, Publications Editor

## Contributors

Jan Aikins  
Robert Anderson, RAND  
Douglas Appelt  
David Arnold  
Donald Biesel, Rutgers  
Lee Blaine, IMSSS  
Rodney Brooks  
Richard Chestek  
Randall Davis, MIT  
Gerard Dechen  
Richard Duda, SRI  
Ramez El-Masri  
Robert Filman  
Fritz Fisher  
Christian Freksa, UC Berkeley  
Peter Friedland  
Richard Gabriel  
Michael Genesereth, MIT  
Neil Goldman, ISI  
Ira Goldstein, Xerox PARC

George Heidorn, IBM  
Douglas Hofstadter  
Elaine Kant  
William Laaser  
Douglas Lenat  
Pamela McCorduck  
Richard Pattis  
Gregory Ruth, MIT  
Daniel Sagalowicz, SRI  
Behrokh Samadi  
William Scherlis  
Andrew Silverman  
Donald Smith, Rutgers  
Phillip Smith, Waterloo  
Reid Smith  
William Swartout, MIT  
William Van Melle  
Richard Waldinger, SRI  
Richard Waters, MIT  
David Wilkins

## Reviewers

Saul Amarel, Rutgers  
Robert Balzer, ISI  
David Barstow, Yale  
Thomas Binford  
Daniel Bobrow, Xerox PARC  
John Seely Brown, Xerox PARC  
Bruce Buchanan  
Richard Burton, Xerox PARC  
Lewis Creary  
Daniel Dolata, UC Santa Cruz  
Lee Erman, ISI  
Cordell Green, SCI  
Casimir Kulikowsky, Rutgers

Brian McCune  
Donald Michle, Edinburgh  
Nils Nilsson, SRI  
Glen Ouchi, UC Santa Cruz  
Ira Pohl, UC Santa Cruz  
Herbert Simon, CMU  
Mark Stefik  
Albert Stevens, BBN  
Perry Thorndyke, RAND  
Don Walker, SRI  
Keith Wescourt, ONR  
Terry Winograd



**Search**



# Search

## Table of Contents

A. Overview . . . . .	15
B. Problem Representation . . . . .	23
1. State-space Representation . . . . .	23
2. Problem-reduction Representation . . . . .	26
3. Game Trees . . . . .	32
C. Search Methods . . . . .	34
1. Blind State-space Search . . . . .	34
2. Blind AND/OR Graph Search . . . . .	41
3. Heuristic State-space Search . . . . .	45
a. Basic Concepts in Heuristic Search . . . . .	45
b. A*-Optimal Search for an Optimal Solution . . . . .	49
c. Relaxing the Optimality Requirement . . . . .	51
d. Bidirectional Search . . . . .	55
4. Heuristic Search of an AND/OR Graph . . . . .	57
5. Game Tree Search . . . . .	65
a. Minimax Procedure . . . . .	65
b. Alpha-beta Pruning . . . . .	68
c. Heuristics in Game Tree Search . . . . .	72
D. Example Search Programs . . . . .	83
1. Logic Theorist . . . . .	83
2. General Problem Solver . . . . .	86
3. Gelernter's Geometry Theorem-proving Machine . . . . .	91
4. Symbolic Integration Programs . . . . .	94
5. STRIPS . . . . .	98
6. ABSTRIPS . . . . .	104
References . . . . .	108
Index . . . . .	116

## A. Overview

In Artificial Intelligence the terms *problem solving* and *search* refer to a large body of core ideas that deal with deduction, inference, planning, commonsense reasoning, theorem proving, and related processes. Applications of these general ideas are found in programs for natural language understanding, information retrieval, automatic programming, robotics, scene analysis, game playing, expert systems, and mathematical theorem proving. In this chapter we examine search as a tool for problem solving in a more limited area. Most of the examples to be considered in detail are problems that are relatively easy to formalize. Some typical problems are

- finding the solution to a puzzle;
- finding a proof for a theorem in logic or mathematics;
- finding the shortest path connecting a set of nonequidistant points (the traveling-salesman problem);
- finding a sequence of moves that will win a game, or the best move to make at a given point in a game; and
- finding a sequence of transformations that will solve a symbolic integration problem.

## Organization of the Chapter

This overview takes a general look at search in problem solving, indicating some connections with topics considered in other chapters. The articles in the next section, Section B, describe the problem representations that form the basis of search techniques. The detailed examples there of state-space and problem-reduction representations will clarify what is meant by words like "search" and "problem solving" in AI. Readers to whom the subject of search is new are encouraged to turn to those articles for more concrete presentations of the fundamental ideas. Section B also discusses game trees, which are a historically and conceptually important class of representations.

Section C, Search Methods, deals with the algorithms that use these various problem representations. *Blind* search algorithms, which treat the search space syntactically, are contrasted with *heuristic* methods, which use information about the nature and structure of the problem domain to limit the search. Various search algorithms are presented in full.

Finally, Section D reviews some well-known early programs based on search. It also describes two programs, STRIPS and ABSTRIPS, that introduce the closely related topic of *planning* in problem solving. This general topic, however, is treated more fully under Planning.

## Components of Search Systems

Problem-solving systems can usually be described in terms of three main components. The first of these is a *database*, which describes both the current task-domain situation and

the goal. The database can consist of a variety of different kinds of data structures including arrays, lists, sets of predicate calculus expressions, property list structures, and semantic networks. In theorem proving, for example, the current task-domain situation consists of assertions representing axioms, lemmas, and theorems already proved; the goal is an assertion representing the theorem to be proved. In information retrieval applications, the current situation consists of a set of facts, and the goal is the query to be answered. In robot problem solving, a current situation is a *world model* consisting of statements describing the physical surroundings of the robot, and the goal is a description that is to be made true by a sequence of robot actions.

The second component of problem-solving systems is a set of *operators* that are used to manipulate the database. Some examples of operators include:

- in theorem proving, rules of inference such as modus ponens and resolution;
- In chess, rules for moving chessmen;
- in symbolic integration, rules for simplifying the forms to be integrated, such as integration by parts or trigonometric substitution.

Sometimes the set of operators consists of only a few general rules of inference that generate new assertions from existing ones. Usually it is more efficient to use a large number of very specialized operators that generate new assertions only from very specific existing ones.

The third component of a problem-solving system is a *control strategy* for deciding what to do next--in particular, what operator to apply and where to apply it. Sometimes control is highly centralized, in a separate control executive that decides how problem-solving resources should be expended. Sometimes control is diffusely spread among the operators themselves.

Another aspect of control strategy is its effect on the contents and organization of the database. In general, the object is to achieve the goal by applying an appropriate sequence of operators to an initial task-domain situation. Each application of an operator modifies the situation in some way. If several different operator sequences are worth considering, the representation often maintains data structures showing the effects on the task situation of each alternative sequence. Such a representation permits a control strategy that investigates various operator sequences in parallel or that alternates attention among a number of sequences that look relatively promising. This is the character of most of the algorithms considered in this chapter; they assume a database containing descriptions of multiple task-domain situations or *states* (see, e.g., C1, Blind State-space Search). It may be, however, that the description of a task-domain situation is too large for multiple versions to be stored explicitly; in this case, a *backtracking* control strategy may be used (see AI Programming Languages). A third possibility, available in some types of problems such as theorem proving, exists where the application of operators can add new assertions to the description of the task-domain situation but never can require the deletion of existing assertions. In this case, the database can describe a single, incrementally changing task-domain situation; multiple or alternative descriptions are unnecessary. (See Theorem Proving.)

### Reasoning Forward and Reasoning Backward

The application of operators to those structures in the database that describe the task-domain situation--to produce a modified situation--is often called *reasoning forward*. The object is to bring the situation, or problem state, forward from its initial configuration to one satisfying a goal condition. For example, an initial situation might be the placement of chessmen on the board at the beginning of the game; the desired goal, any board configuration that is a checkmate; and the operators, rules for the legal moves in chess.

An alternative strategy, reasoning backward, involves using another type of operator, which is applied not to a current task-domain situation but to the goal. The goal statement, or problem statement, is converted to one or more subgoals that are (one hopes) easier to solve and whose solutions are sufficient to solve the original problem. These subgoals may in turn be reduced to sub-subgoals, and so on, until each of them is either accepted to be a trivial problem or its solution is accomplished by the solution of its subproblems. For example, given an initial goal of integrating  $1/(\cos x)^2 dx$ , and an operator permitting  $1/(\cos x)$  to be rewritten as  $(\sec x)$ , one can work backward toward a restatement of the goal in a form whose solution is immediate: The integral of  $(\sec x)^2$  is  $\tan x$ .

The former approach is said to use *forward reasoning* and to be *data-driven* or *bottom-up*. The latter uses *backward reasoning* and is *goal-directed* or *top-down*. The distinction between forward and backward reasoning assumes that the current task-domain situation or state is distinct from the goal. If one chooses to say that a current state is the state of having a particular goal, the distinction naturally vanishes.

Much human problem-solving behavior is observed to involve reasoning backward, and many artificial intelligence programs are based on this general strategy. In addition, combinations of forward and backward reasoning are possible. One important AI technique involving forward and backward reasoning is called *means-ends analysis*; it involves comparing the current goal with a current task-domain situation to extract a *difference* between them. This difference is then used to index that (forward) operator most relevant to reducing the difference. If this especially relevant operator cannot be immediately applied to the present problem state, subgoals are set up to change the problem state so that the relevant operator can be applied. After these subgoals are solved, the relevant operator is applied and the resulting, modified situation becomes a new starting point from which to solve for the original goal. (See D2, GPS; and D5, STRIPS.)

### State Spaces and Problem Reduction

A problem-solving system that uses forward reasoning and whose operators each work by producing a single new object--a new state--in the database is said to represent problems in a *state-space representation* (see B1).

A distinction may be drawn between two cases of backward reasoning. In one, each application of an operator to a problem yields exactly one new problem, whose size or difficulty is typically slightly less than that of the previous problem. Systems of this kind will also be referred to, in this chapter, as employing state-space representations. Two instances of such representations are presented later in the chapter. One example is the Logic Theorist program (D1); the other is the backward-reasoning part of Pohl's *bidirectional search* (C1 and C3d).

A more complex kind of backward reasoning occurs if applying an operator may divide the problem into a set of subproblems, perhaps each significantly smaller than the original. An example would be an operator changing the problem of integrating  $2/(x^2-1) dx$  into the three subproblems of integrating  $1/(x-1) dx$ , integrating  $-1/(x+1) dx$ , and adding the results. A system using this kind of backward reasoning, distinguished by the fact that its operators can change a single object into a conjunction of objects, will be said to employ a *problem-reduction representation*. The relationship between problem-reduction and state-space representations is examined further at the end of Article B2.

There may or may not be constraints on the order in which the subproblems generated by a problem-reduction system can be solved. Suppose, for example, that the original problem is to integrate  $(f(x) + g(x)) dx$ . Applying the obvious operator changes it to the new problem consisting of two integrations,  $f(x) dx$  and  $g(x) dx$ . Depending on the representation, the new problem can be viewed as made up of either (a) two integration subproblems that can be solved in any order, or (b) two integration subproblems plus the third subproblem of summing the integrals. In the latter case, the third task cannot be done until the first two have been completed.

Besides the state-space and problem-reduction representation approaches, other variations on problem representation are possible. One occurs in connection with game-playing problems, which differ from most other problems by virtue of the existence of adversary moves. A game-playing problem must be represented so as to take into account the opponent's possible moves as well as the player's own. The usual representation is a *game tree* (see B3), which shares many features of a problem-reduction representation. Another variation is relevant to theorem-proving systems, many of which use forward reasoning and operators (rules of inference) that act on conjunctions of objects in the database. Although the representations discussed here assume that each operator takes only a single object as input, it is possible to define a *theorem-proving representation* that provides for multiple-input, single-output operators (Kowalski, 1972; see Theorem Proving).

### Graph Representation

In either a state-space or a problem-reduction representation, achieving the desired goal can be equated with finding an appropriate finite sequence of applications of available operators. While what one is primarily interested in--the goal situation or the sequence that leads to it--may depend on the problem, the term *search* can always be understood, without misleading consequences, as referring to the search for an appropriate operator sequence.

Tree structures are commonly used in implementing control strategies for the search. In a state-space representation, a tree may be used to represent the set of problem states produced by operator applications. In such a representation, the root node of the tree represents the initial problem situation or state. Each of the new states that can be produced from this initial state by the application of just one operator is represented by a *successor node* of the root node. Subsequent operator applications produce successors of these nodes, etc. Each operator application is represented by a directed *arc* of the tree. In general, the states are represented by a *graph* rather than by a tree since there may exist different paths from the root to any given node. Trees are an important special case, however, and it is usually easier to explain their use than that of graphs. (See B1, State-space Representation.)



In addition to these ordinary trees and graphs used for state-space representations, specialized ones called *AND/OR graphs* are used for problem-reduction problem-solving methods. For problems in which the goal can be reduced to sets of subgoals, AND/OR graphs provide a means for keeping track of which subgoals have been attempted and of which combinations of subgoals are sufficient to achieve the original goal (see Article B2).

### The Search Space

The problem of producing a state that satisfies a goal condition can now be formulated as the problem of searching a graph to find a node whose associated state description satisfies the goal. Similarly, search based on a problem-reduction representation can be formulated as the search of an AND/OR graph.

It should be noted that there is a distinction between the graph to be searched and the tree or graph that is constructed as the search proceeds. In the latter, nodes and arcs can be represented by explicit data structures; the only nodes included are those for which paths from the initial state have actually been discovered. This explicit graph, which grows as the search proceeds, will be referred to as a *search graph* or *search tree*.

In contrast, the graph to be searched is ordinarily not explicit. It may be thought of as having one node for every state to which there exists a path from the root. It may even be thought of, less commonly, as having one node for every state that can be described, whether or not a path to it exists. The implicit graph will be called the *state space* or, if generalized to cover non-state-space representations such as AND/OR graphs or game trees, the *search space*. Clearly, many problem domains (such as theorem proving) have an infinite search space, and the search space in others, though finite, is unimaginably large. Estimates of search space size may be based on the total number of nodes (however defined) or on other measures. In chess, for example, the number of different complete plays of the average-length game has been estimated at  $10^{120}$  (Shannon, 1950, 1956), although the number of "good" games is much smaller (see Good, 1968). Even for checkers the size of the search space has been estimated at  $10^{40}$  (Samuel, 1963).

Searching now becomes a problem of making just enough of the search space explicit in a search graph to contain a solution of the original goal. If the search space is a general graph, the search graph may be either a subgraph, or a subgraph that is also a tree, or a tree obtained by representing distinct paths to one search space node with duplicate search graph nodes.

### Limiting Search

The critical problem of search is the amount of time and space necessary to find a solution. As the chess and checkers estimates suggest, exhaustive search is rarely feasible for nontrivial problems. Examining all sequences of  $n$  moves, for example, would require operating in a search space in which the number of nodes grows exponentially with  $n$ . Such a phenomenon is called a *combinatorial explosion*.

There are several complementary approaches to reducing the number of nodes that a search must examine. One important way is to recast the problem so as to reduce the size of the search space. A dramatic, if well-known, example is the mutilated chessboard problem:

Suppose two diagonally opposite corner squares are removed from a standard 8 by 8 square chessboard. Can 31 rectangular dominoes, each the size of exactly two squares, be so placed as to cover precisely the remaining board? (Raphael, 1976, p. 31)

If states are defined to be configurations of dominoes on the mutilated board, and an operator has the effect of placing a domino, the search space for this problem is very large. If, however, one observes that every domino placed must cover both a red square and a black one and that the squares removed are both of one color, the answer is immediate. Unfortunately, little theory exists about how to find good problem representations. Some of the sorts of things such a theory would need to take into account are explored by Amarel (1968), who gives a sequence of six representations for a single problem, each reducing the search space size by redefining the states and operators.

A second aspect concerns search efficiency within a given search space. Several graph- and tree-searching methods have been developed, and these play an important role in the control of problem-solving processes. Of special interest are those graph-searching methods that use *heuristic knowledge* from the problem domain to help focus the search. In some types of problems, these *heuristic search* techniques can prevent a combinatorial explosion of possible solutions. Heuristic search is one of the key contributions of AI to efficient problem solving. Various theorems have been proved about the properties of search techniques, both those that do and those that do not use heuristic information. Briefly, it has been shown that certain types of search methods are guaranteed to find optimal solutions (when such exist). Some of these methods, under certain comparisons, have also been shown to find solutions with a minimal amount of search effort. Graph- and tree-searching algorithms, with and without the use of heuristic information, are discussed at length in Section C.

A third approach addresses the question: Given one representation of a search problem, can a problem-solving system be programmed to find a better representation automatically? The question differs from that of the first approach to limiting search in that here it is the program, not the program designer, that is asked to find the improved representation. One start on answering the question was made by the STRIPS program (D5). STRIPS augments its initial set of operators by discovering, generalizing, and remembering *macro-operators*, composed of sequences of primitive operators, as it gains problem-solving experience. Another idea was used in the ABSTRIPS program (D6), which implements the idea of *planning*, in the sense of defining and solving problems in a search space from which unimportant details have been omitted. The details of the solution are filled in (by smaller searches within the more detailed space) only after a satisfactory outline of a solution, or *plan*, has been found. Planning is a major topic itself; for further discussion, see Planning.

### The Meaning of "Heuristic" and "Heuristic Search"

Although the term "heuristic" has long been a key word in AI, its meaning has varied both among authors and over time. In general, its usage is illustrated by example better than by definition, and several of the prime examples are included in the programs of Section D. However, a brief review of the ways "heuristic" and "heuristic search" have been used may provide a useful warning against taking any single definition too seriously.

As an adjective, the most frequently quoted dictionary definition for "heuristic" is "serving to discover." As a noun, referring to an obscure branch of philosophy, the word meant the study of the methods and rules of discovery and invention (see Polya, 1957, p. 112).

When the term came into use to describe AI techniques, some writers made a distinction between methods for discovering solutions and algorithms for producing them. Thus Newell, Shaw, and Simon stated in 1957: "A process that *may* solve a given problem, but offers no guarantees of doing so, is called a *heuristic* for that problem" (Newell, Shaw, & Simon, 1963b, p. 114). But this meaning was not universally accepted. Minsky, for example, said in a 1961 paper:

The adjective "heuristic," as used here and widely in the literature, means *related to improving problem-solving performance*; as a noun it is also used in regard to any method or trick used to improve the efficiency of a problem-solving program. . . . But imperfect methods are not necessarily heuristic, nor vice versa. Hence "heuristic" should not be regarded as opposite to "foolproof"; this has caused some confusion in the literature. (Minsky, 1963, p. 407n.)

These two definitions refer, though vaguely, to two different sets: devices that improve efficiency and devices that are not guaranteed. Feigenbaum and Feldman (1963, p. 6) apparently limit "heuristic" to devices with both properties:

*A heuristic (heuristic rule, heuristic method) is a rule of thumb, strategy, trick, simplification, or any other kind of device which drastically limits search for solutions in large problem spaces. Heuristics do not guarantee optimal solutions; in fact, they do not guarantee any solution at all; all that can be said for a useful heuristic is that it offers solutions which are good enough most of the time.*

Even this definition, however, does not always agree with common usage, because it lacks a historical dimension. A device originally introduced as a heuristic in Feigenbaum and Feldman's sense may later be shown to guarantee an optimal solution after all. When this happens, the label "heuristic" may or may not be dropped. It has not been dropped, for example, with respect to the A\* algorithm (C3b). Alpha-beta pruning (C5b), on the other hand, is no longer called a heuristic.

It should be noted that the definitions quoted above, ranging in time from 1957 to 1963, refer to heuristic rules, methods, and programs, but they do not use the term "heuristic search." This composite term appears to have been first introduced in 1965 in a paper by Newell and Ernst, "The Search for Generality" (see Newell & Simon, 1972, p. 888). The paper presented a framework for comparing the methods used in problem-solving programs up to that time. The basic framework, there called heuristic search, was the one called *state-space search* in the present chapter. Blind search methods were included within the heuristic search paradigm.

A similar meaning for heuristic search appears in Newell and Simon, 1972 (pp. 91-105). Again no contrast is drawn between heuristic search and blind search; rather, heuristic search is distinguished from a problem-solving method called *generate-and-test*. The difference between the two is that the latter simply generates elements of the search space (i.e., states) and tests each in turn until it finds one satisfying the goal condition;

whereas in heuristic search the order of generation can depend both on information gained in previous tests and on the characteristics of the goal. But the Newell and Simon distinction is not a hard and fast one. By their 1976 Turing Lecture, they seem to have collapsed the two methods into one:

Heuristic Search. A second law of qualitative structure for AI is that symbol systems solve problems by generating potential solutions and testing them, that is, by searching. (Newell & Simon, 1976, p. 126)

In the present chapter, the meaning attached to "heuristic search" stems not from Newell and Simon but from Nilsson, whose 1971 book provides the most detailed and influential treatment of the subject that has yet appeared. For Nilsson, the distinction between heuristic search and blind search is the important one. Blind search corresponds approximately to the systematic generation and testing of search space elements, but it operates within a formalism that leaves room for additional information about the specific problem domain to be introduced, rather than excluding it by definition. If such information, going beyond that needed merely to formulate a class of problems as search problems, is in fact introduced, it may be possible to restrict search drastically. Whether or not the restriction is foolproof, the search is then called heuristic rather than blind.

#### References

See Amarel (1968), Feigenbaum & Feldman (1963), Good (1968), Jackson (1974), Kowalski (1972), Minsky (1963), Newell & Ernst (1965), Newell, Shaw, & Simon (1963b), Newell & Simon (1972), Newell & Simon (1976), Nilsson (1971), Polya (1957), Raphael (1976), Samuel (1963), Shannon (1950), Shannon (1956), and Vanderbrug & Minker (1975).

## B. Problem Representation

### B1. State-space Representation

A state-space representation of a problem uses two kinds of entities: *states*, which are data structures giving "snapshots" of the condition of the problem at each stage of its solution, and *operators*, which are means for transforming the problem from one state to another.

A straightforward example of state-space representation is the simple, well-known puzzle called the 8-puzzle. An 8-puzzle is a square tray containing 8 square tiles of equal size, numbered 1 to 8. The space for the 9th tile is vacant (see Figure 1).

2	1	6
4		8
7	5	3

Figure 1. An 8-puzzle.

A tile may be moved by sliding it vertically or horizontally into the empty square. The problem is to transform one tile configuration, say that of Figure 1, into another given tile configuration, say that of Figure 2.

1	2	3
8		4
7	6	5

Figure 2. A solution configuration of the 8-puzzle.

A state is a particular configuration of tiles; each state might be represented by a 3 x 3 matrix, similar to Figures 1 and 2. The operators, corresponding to possible moves, might be defined with separate operators for each of tiles 1 through 8. However, a more concise definition is made possible by viewing the empty square as the object to be moved and stating the operators in terms of the movements of this square. In this formulation, only four operators are used:

"UP"        (move the blank up one square),  
 "DOWN"     (move the blank down one square),  
 "LEFT"     (move the blank left one square),  
 "RIGHT"    (move the blank right one square).

An operator may be inapplicable in certain states, as when it would move the blank outside the tray of tiles.

The set of all attainable states of a problem is often called its *state space*. The 8-

puzzle, for example, has a state space of size  $9!/2$ --since there are  $9!$  configurations of the tiles but only half this number can be reached from any given starting configuration. This comes to only 181,440 possible states. For comparison, see the discussion of chess and checkers in the Overview article.

The four operators defined for the 8-puzzle form a set of *partial functions* on the state space: Each operator, if it applies to a given state at all, returns exactly one new state as its result. In more complex problems, however, the operators often contain variables. If, for a particular state and operator, the variables can be instantiated in more than one way, then each instantiation yields one new state, and the operators of the problem, if they are to be considered as defining functions, are more accurately termed *operator schemata*.

The complete specification of a state-space problem has three components. One is a set  $O$  of operators or operator schemata. In addition, one must define a set  $S$  of one or more *initial states* and find a predicate defining a set  $G$  of *goal states*. A state-space problem is then the triple  $(S, O, G)$ . A *solution* to the problem is a finite sequence of applications of operators that changes an initial state into a goal state.

A state space can be treated as a directed graph whose nodes are states and whose arcs are operators transforming one state to another. For example, if state 1 is a state to which any of three operators can be applied, transforming it to state 2, 3, or 4, then the corresponding graph would be as in Figure 3. Nodes 2, 3, and 4 are called the *successors* of node 1.

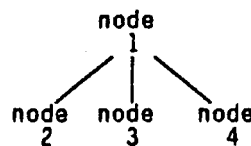


Figure 3. Directed arcs.

In graph notation, a solution to a state-space problem is a path from an initial node to a goal node. In Figure 4, one solution would be an application of operator B twice, followed by operator D, to reach the indicated goal node or final state. There may be other final states and multiple ways to reach a particular final state.

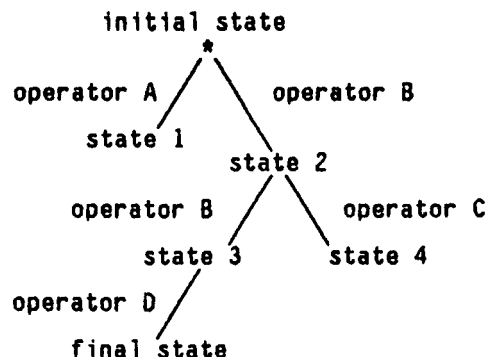


Figure 4. A state-space graph.

A common variation on state-space problems requires finding not just any path but one of minimum cost between an initial node and a goal node. In this case, each arc of the graph is labeled with its cost. An example is the traveling-salesman problem: Given a number of cities to be visited and the mileage between each pair of cities, find a minimum-mileage trip beginning and ending at city A that visits each of the other cities exactly once. An example mileage chart and the corresponding state-space graph are shown in Figure 5. Because different paths to the same city represent distinct partial solutions, each state is identified not just as a city name but as a list of the cities visited so far.

	A	B	C	D
A	-	4	6	10
B		-	7	10
C			-	5
D				-

Mileage chart

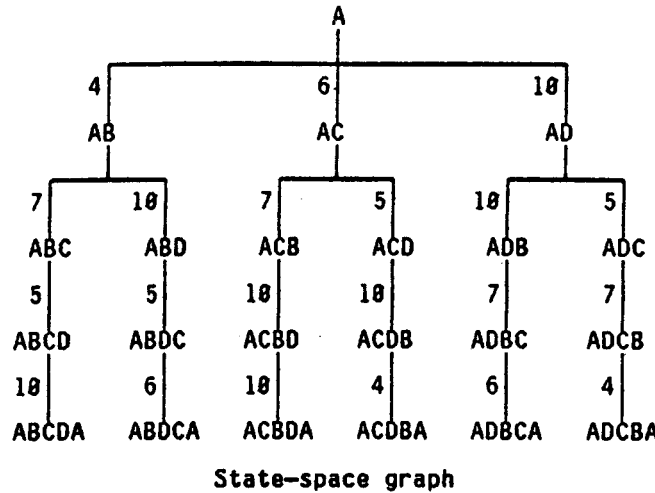


Figure 5. A traveling-salesman problem.

The desired solution is A-B-D-C-A, or its reversal, with a total mileage of 25. (The two bottom levels of the graph could be omitted, since the mileage of each tour of  $n$  cities is determined by the first  $n-1$  cities chosen to be visited.)

Because the state-space graph is usually too large to represent explicitly, the problem of searching for a solution becomes one of generating just enough of the graph to contain the desired solution path. Search methods are discussed in Article C1, Blind State-space Search, and Section C3, Heuristic State-space Search.

References

See Nilsson (1971).

## B2. Problem-reduction Representation

Often distinguished from the state-space representation of problems is a technique called *problem-reduction representation*. In the problem-reduction approach, the principal data structures are problem descriptions or *goals*. An initial problem description is given; it is solved by a sequence of transformations that ultimately change it into a set of subproblems whose solutions are immediate. The transformations permitted are defined as *operators*. An operator may change a single problem into several subproblems; to solve the former, all the subproblems must be solved. In addition, several different operators may be applicable to a single problem, or the same operator may be applicable in several different ways. In this case it suffices to solve the subproblems produced by any one of the operator applications. A problem whose solution is immediate is called a *primitive problem*. Thus, a problem representation using problem reduction is defined by a triple consisting of

- (a) an initial problem description,
- (b) a set of operators for transforming problems to subproblems, and
- (c) a set of primitive problem descriptions.

Reasoning proceeds backward from the initial goal.

### An Example

An example that lends itself nicely to problem-reduction representation is the famous Tower of Hanoi puzzle. In one common version there are three disks, A, B, and C, of graduated sizes. There are also three pegs, 1, 2, and 3. Initially the disks are stacked on peg 1, with A, the smallest, on top and C, the largest, at the bottom. The problem is to transfer the stack to peg 3, as in Figure 1, given that (a) only one disk can be moved at a time, and (b) no disk may be placed on top of a smaller disk.

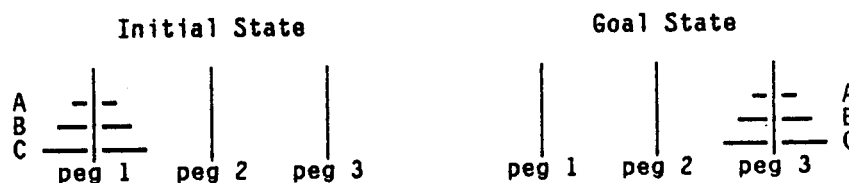


Figure 1. The Tower of Hanoi puzzle.

Only one operator need be used in the solution: Given distinct pegs  $i$ ,  $j$ , and  $k$ , the problem of moving a stack of size  $n > 1$  from peg  $i$  to peg  $k$  can be replaced by the three problems:

- (a) moving a stack of size  $n-1$  from  $i$  to  $j$ ,
- (b) moving a stack of size 1 from  $i$  to  $k$ , and
- (c) moving a stack of size  $n-1$  from  $j$  to  $k$ .

The only primitive problem is that of moving a single disk from one peg to another, provided no smaller disk is on the receiving peg. If a smaller disk were present, this problem would be unsolvable (in view of the definition of the only available operator).



Each problem description can now be given by specifying the size  $n$  of the stack to be moved, the number of the sending peg, and the number of the receiving peg. The original problem, moving a stack of three disks from peg 1 to peg 3, would then be represented as  $(n = 3, 1 \text{ to } 3)$ , and the transformation of the original problem to primitive problems can be represented by a tree:

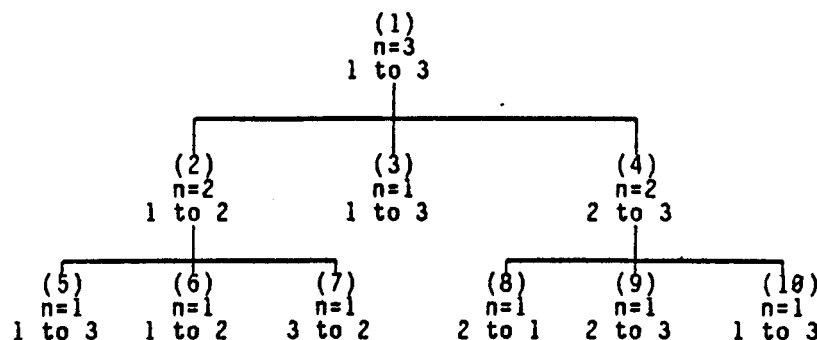


Figure 2. Solution of the Tower of Hanoi puzzle.

There happen to be two possible operator sequences that transform the original problem to primitive problems: Apply the operator to node 1, then node 2, and then node 4; or apply the operator to node 1, then node 4, and then node 2. Since node 3 is a primitive problem, it needs no further attention. Node 2 represents the subproblem of moving the top two disks on peg 1 to peg 2. This subproblem is solved by expanding it to the primitive problems at nodes (5), (6), and (7)--which are solved by moving the smallest disk to peg 3, moving the middle disk to peg 2, and finally putting the small disk back on top of the middle one.

The sequence of operators to be applied should be distinguished from the sequence of actions to be taken to achieve the goal. In the Tower of Hanoi example, the actions are the actual movements of the disks. This sequence is given by the terminal nodes of the tree, read left to right. Whether or not it is considered important to assemble such a sequence of actions depends on the particular problem domain.

### AND/OR Graphs

In the example above, a tree was used to display a problem-reduction solution to the Tower of Hanoi puzzle. The tree notation must be generalized if it is to represent the full variety of situations that may occur in problem reduction. This generalized notation for problem reduction is called an *AND/OR graph*.

According to one common formulation (Nilsson, 1971), an AND/OR graph is constructed according to the following rules:

1. Each node represents either a single problem or a set of problems to be solved. The graph contains a start node corresponding to the original problem.

2. A node representing a primitive problem, called a *terminal node*, has no descendants.
3. For each possible application of an operator to problem P, transforming it to a set of subproblems, there is a directed arc from P to a node representing the resulting subproblem set. For example, Figure 3 illustrates the reduction of P to three different subproblem sets: A, B, and C. Since P can be solved if any one of sets A, B, or C can be solved, A, B, and C are called *OR nodes*.

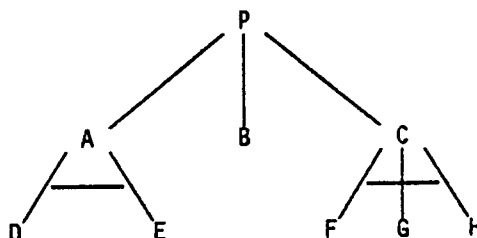


Figure 3. An AND/OR tree.

4. Figure 3 illustrates further the composition of sets A, B, and C:  $A = \{D, E\}$ , B consists of a single (unnamed) problem, and  $C = \{F, G, H\}$ . In general, for each node representing a set of two or more subproblems, there are directed arcs from the node for the set to individual nodes for each subproblem. Since a set of subproblems can be solved only if its members can *all* be solved, the subproblem nodes are called *AND nodes*. To distinguish them from OR nodes, the arcs leading to AND node successors of a common parent are joined by a horizontal line.
5. A simplification of the graph produced by rules 3 and 4 may be made in the special case where only one application of an operator is possible for problem P and where this operator produces a set of more than one subproblem. As Figure 4 illustrates, the *intermediate OR node* representing the subproblem set may then be omitted:

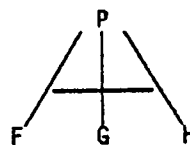


Figure 4. An AND/OR tree with one operator at problem P.

Another example of this construction was given in Figure 2.

In the figures above, every node represents a distinct problem or set of problems. Since each node except the start node has just one parent, the graphs are in fact *AND/OR trees*. As a variation on Figure 3, assume that problem A is reducible to D and E; and problem C, to E, G, and H. Then E may be represented either by two distinct nodes, or by a single

node as shown in Figure 5. The choice makes a difference in the search algorithms which are discussed later in the chapter. For example, if node E is in turn reducible to C, the general graph representation simply adds another directed arc to Figure 5, but the corresponding tree becomes infinite.

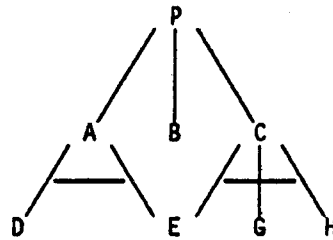


Figure 5. An AND/OR graph.

The constructions discussed so far concern graphs depicting the entire problem search space. To find a solution to the initial problem, one need only build enough of the graph to demonstrate that the start node can be solved. Such a subgraph is called a *solution graph* or, in the more restricted case of an AND/OR tree, a *solution tree*. The following rules apply:

A node is solvable if:

- (a) it is a terminal node (a primitive problem);
- (b) it is a nonterminal node whose successors are AND nodes that are all solvable; or
- (c) it is a nonterminal node whose successors are OR nodes and at least one of them is solvable.

Similarly, a node is unsolvable if:

- (a) it is a nonterminal node that has no successors (a nonprimitive problem to which no operator applies);
- (b) it is a nonterminal node whose successors are AND nodes and at least one of them is unsolvable; or
- (c) it is a nonterminal node whose successors are OR nodes and all of them are unsolvable.

Methods of searching an AND/OR graph for such a solution are discussed in Articles C2 and C4.

### Relation between Problem-reduction and State-space Representations

Some interesting general relationships can be found between problem-reduction and state-space representations. In the first place, although one representation often seems the more natural for a given problem, it is often possible to recast the problem definition so that it uses the other form. For example, the Tower of Hanoi puzzle can also be solved by a state-space search using operators that move a single disk and that represent all the legal

moves in a given configuration. In comparison to the problem-reduction representation, which in fact gives an algorithm for solving the puzzle, the state-space representation would be a poor one since it leaves room for searching down unnecessarily long paths.

Second, it is possible to translate mechanically between state-space representations and problem-reduction representations without any fundamental shift in the way a problem is viewed. The ways of making such translations can provide helpful insight into many search programs in which the concepts of state-space and problem-reduction representation appear to be intermixed. Several translation schemes are described below. (Some readers may wish to skip the following material at first reading.)

**State space to problem reduction.** Two approaches suggest themselves for translating state-space representations to problem-reduction representations. In one, the state-space graph is understood as an AND/OR graph containing only OR nodes. Each state of the state-space version corresponds to the problem of getting from that state to a goal state; and a goal state of the state space becomes the primitive problem of getting from that goal state to itself. In other words, data structures representing states are simply reinterpreted as representing problem descriptions, where a problem consists of state information together with an implicit goal.

Alternately, there is a slight variation of the first approach that requires redefining the operators of the state-space representation. Each such operator, taking state  $i$  to state  $j$ , becomes an operator applicable to the problem of getting from state  $i$  to a goal state. Its effect is to reduce the problem to a pair of subproblems: (a) go from state  $i$  to state  $j$  (a primitive problem), and (b) go from state  $j$  to a goal state. Figure 6 illustrates this correspondence.

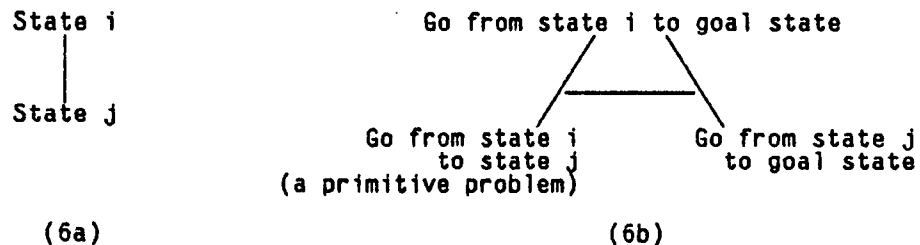


Figure 6. (a) Part of a state-space tree; (b) the corresponding part of an AND/OR (problem-reduction) tree.

**Problem reduction to state space.** The translation from a problem-reduction representation to a state-space representation is a little more complex, assuming that the problem-reduction operators in fact produce AND nodes. The initial problem of the problem-reduction representation can be understood as having two components: (a) the description of the goal to be achieved, as discussed at the beginning of this article, and (b) the description of an initial state of the world. These components will be denoted  $g_0$  and  $s_0$ , respectively. Some examples are

--  $g_0$  = a theorem to be proved, and  $s_0$  = the axioms from which to prove it;

--  $g_0$  = a configuration of objects to be achieved, and  $s_0$  = their existing configuration.

Each state  $\underline{S}$  of the corresponding state-space representation is a pair consisting of a stack of goals  $(g_1, \dots, g_0)$  to be achieved and a current state  $\underline{s}$  of the world. Thus, the initial state  $\underline{S_0}$  of the state-space representation is  $\underline{S_0} = ((g_0), s_0)$ . A final state is one in which the stack of goals to be achieved has been emptied.

For each problem-reduction operator, mapping a problem or goal  $g$  to a set of subgoals  $\{g_m, \dots, g_n\}$ , the state-space representation has a corresponding operator mapping state  $S_1$ , where  $S_1 = ((g_1, \dots, g_0), s)$ , to a state  $S_2$  in which  $\{g_m, \dots, g_n\}$  have been added to the top of the goal-stack (in the order in which they should be carried out, if relevant), and the state of the world  $\underline{s}$  is unchanged; that is,  $S_2 = ((g_m, \dots, g_n, g_1, \dots, g_0), s)$ .

The state-space representation also needs a second type of operator, which becomes applicable whenever the goal on top of the stack represents a primitive problem. Its function is to remove that primitive problem from the stack and, at the same time, to change the state  $\underline{s}$  to reflect its solution. In the Tower of Hanoi puzzle, for example, the new state would reflect the changed position of a single disk. In a theorem-proving problem, the new state would differ from the old one by the addition of one formula to those that had been given as axioms or established from having solved previous subproblems. A representation of this type is used explicitly in Fikes and Nilsson's STRIPS program, described in Article D5.

#### References

See Jackson (1974), and Nilsson (1971).

### B3. Game Trees

Most games played by computer programs, including checkers, chess, *go*, and tic-tac-toe, have several basic features in common. There are two players who alternate in making moves. At each turn, the rules define both what moves are legal and the effect that each possible move will have; there is no element of chance. In contrast to card games in which the players' hands are hidden, each player has complete information about his opponent's position, including the choices open to him and the moves he has made. The game begins from a specified state, often a configuration of men on a board. It ends in a win for one player and a loss for the other, or possibly in a draw.

A complete *game tree* is a representation of all possible plays of such a game. The root node is the initial state, in which it is the first player's turn to move. Its successors are the states he can reach in one move; their successors are the states resulting from the other player's possible replies; and so on. Terminal states are those representing a win, loss, or draw. Each path from the root node to a terminal node gives a different complete play of the game.

An important difference between a game tree and a state-space tree (Article B1) is that the game tree represents moves of two opposing players, say A and B. An AND/OR tree (Article B2), however, is sufficient to reflect this opposition. The game tree is ordinarily drawn to represent only one player's point of view. In a game tree drawn from A's standpoint, A's possible moves from a given position are represented by OR nodes since they are alternatives under his own control. The moves that B might make in return are AND nodes, since they represent sets of moves to which A must be able to respond. Because the players take turns, OR nodes and AND nodes appear at alternate levels of the tree. In the language of AND/OR graphs, the tree displays the search space for the problem of showing that A can win. A node representing a win for A corresponds to a primitive problem; a node representing a win for B or a draw, to an unsolvable problem. Unlike the usual AND/OR graph terminology, both of these kinds of nodes will be called *terminal nodes*.

As an example, Figure 1 shows a portion of the game tree for tic-tac-toe. The players are X and O, X has the first move, and the tree is drawn from X's standpoint. Positions are considered identical if one can be obtained from the other by rotation or reflection of the grid. The tree could also be drawn from O's standpoint, even though X has the first move. In this case, the AND nodes would become OR nodes, and vice versa, and the labels "win" and "lose" would be reversed. An alternate formulation of game trees, not explicitly distinguishing between AND and OR nodes, is given in Article C5a, Minimax.

Methods of searching a game tree for a winning strategy are discussed in Section C5. As with search in other domains, the source of difficulty in challenging games is the unimaginably large search space. A complete game tree for checkers, for instance, which is harder than tic-tac-toe but far simpler than chess or *go*, has been estimated as having about  $10^{40}$  nonterminal nodes (Samuel, 1963). If one assumed that these nodes could be generated at the rate of 3 billion per second, generation of the whole tree would still require around  $10^{21}$  centuries!

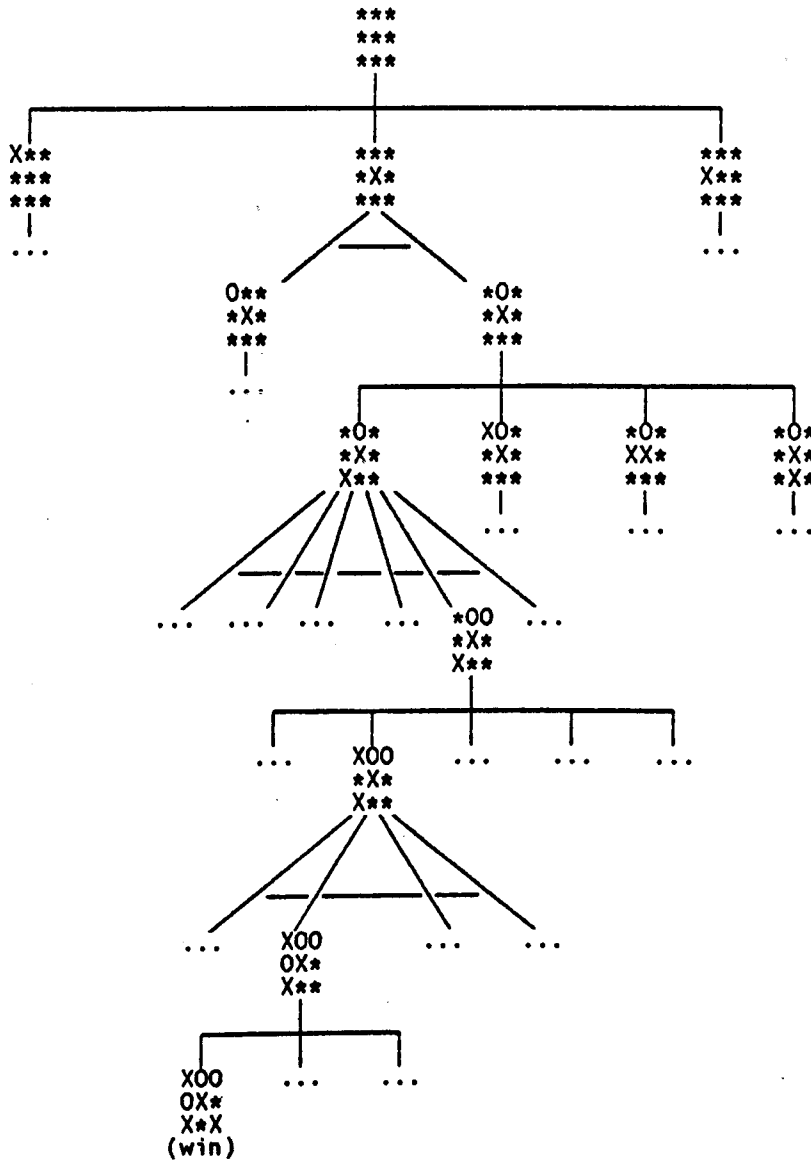


Figure 1. A game tree for Tic-tac-toe.

References

See Nilsson (1971), and Samuel (1963).

### C. Search Methods

#### C1. Blind State-space Search

As discussed in Article B1, a problem in the state-space search paradigm is defined by a triple (S, O, G), where

S is a set of one or more initial states,  
O is a set of operators on states, and  
G is a set of goal states.

The state space is commonly identified with a directed graph in which each node is a state and each arc represents the application of an operator transforming a state to a successor state. A solution is a path from a start state to a goal state. Goal states may be defined either explicitly or as the set of states satisfying a given predicate.

The search for a solution is conducted by making just enough of the state-space graph explicit to contain a solution path. If the order in which potential solution paths are considered is arbitrary, using no domain-specific information to judge where the solution is likely to lie, the search is called *blind search*. Although blind search is impracticable for nontrivial problems, because of the large proportion of the state space it may explore, it provides a useful foundation for the understanding of *heuristic search* techniques, discussed in Section C3.

Several blind-search methods are described below; they differ from one another mainly in the order in which nodes are examined. In each case, it is assumed that a procedure exists for finding all the *successors* of a given node--that is, all the states that can be reached from the current state by a single operator application. Such a procedure is said to *expand* the given node.

The first three algorithms also make two other assumptions:

- (a) The state-space graph is a tree. The implication is that there is only one start state (the root) and that the path from the start node to any other node is unique. Modifications to the search methods needed for a general directed graph are noted in Nilsson (1971) and in Article C3a, Basic Concepts in Heuristic Search.
- (b) Whenever a node is expanded, creating a node for each of its successors, the successor nodes contain pointers back to the parent node. When a goal node is finally generated, this feature makes it possible to trace the solution path.

#### Breadth-first Search

The breadth-first method expands nodes in order of their proximity to the start node, measured by the number of arcs between them. In other words, it considers every possible operator sequence of length  $n$  before any sequence of length  $n+1$ . Thus, although the search may be an extremely long one, it is guaranteed eventually to find the shortest possible solution sequence if any solution exists.

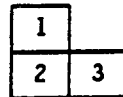


Breadth-first search is described by the following algorithm:

- (1) Put the start node on a list, called OPEN, of unexpanded nodes. If the start node is a goal node, the solution has been found.
- (2) If OPEN is empty, no solution exists.
- (3) Remove the first node,  $n$ , from OPEN and place it in a list, called CLOSED, of expanded nodes.
- (4) Expand node  $n$ . If it has no successors, go to (2).
- (5) Place all successors of node  $n$  at the end of the OPEN list.
- (6) If any of the successors of node  $n$  is a goal node, a solution has been found. Otherwise, go to (2).

As an example of breadth-first search, consider a world consisting of a table and three toy blocks. The initial state of the world is that blocks 2 and 3 are on the table, and block 1 is on top of block 2 (see Figure 1). We wish to reach a goal state in which the three blocks are stacked with block 1 on top, block 2 in the middle, and block 3 on the bottom.

Initial state



Goal state

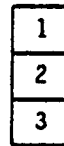


Figure 1. An example problem for breadth-first search.

The only operator is MOVE  $X$  to  $Y$ , which moves object  $X$  onto another object,  $Y$ . As preconditions to applying the operator, it is required (a) that  $X$ , the object to be moved, be a block with nothing on top of it, and (b) that if  $Y$  is a block, there must be nothing on  $Y$ . Finally, the operator is not to be used to generate the same state more than once. (This last condition can be checked from the lists of expanded and unexpanded nodes.)

Figure 2 shows the search tree generated by the breadth-first algorithm. The nodes are states  $S_0$  through  $S_{10}$ ; node  $S_1$ , for example, corresponds to the successor state of  $S_0$  reached by "MOVE block 1 to the table." The nodes are generated and expanded in the order given by their state numbers, i. e.,  $S_0, S_1, S_2, \dots, S_{10}$ . When the algorithm terminates, finding  $S_{10}$  to be the goal state, the list of expanded nodes contains  $S_0$  through  $S_5$ , and the OPEN list still contains  $S_6$  through  $S_{10}$ .

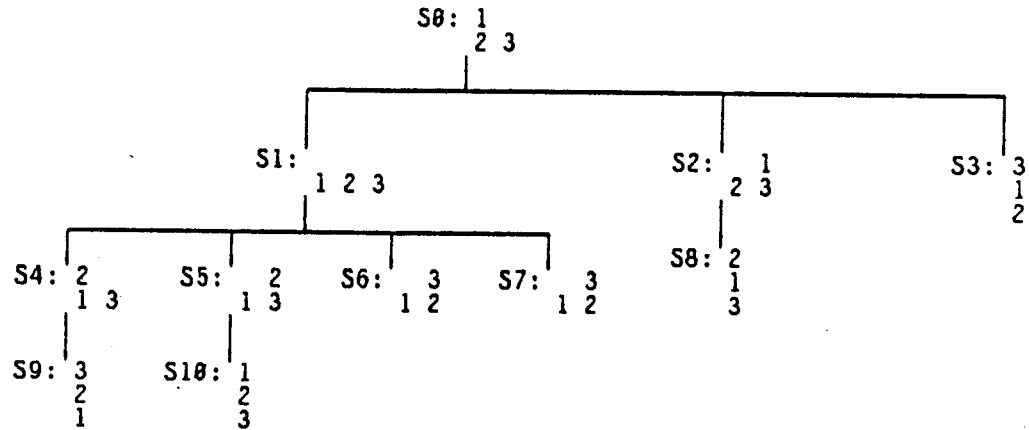


Figure 2. The search tree for Figure 1.

### Uniform-cost Search

The breadth-first algorithm can be generalized slightly to solve the problem of finding the cheapest path from the start state to a goal state. A nonnegative cost is associated with every arc joining two nodes; the cost of a solution path is then the sum of the arc costs along the path. The generalized algorithm is called a *uniform-cost search*. If all arcs have equal cost, the algorithm reduces to breadth-first search. The need for assigning costs to the arcs is illustrated by the traveling-salesman problem, described in Article B1, where the different distances between cities correspond to the arc costs and the problem is to minimize the total distance traveled.

In the uniform-cost algorithm given below, the cost of the arc from node  $i$  to node  $j$  is denoted by  $c(i,j)$ . The cost of a path from the start node to any node  $i$  is denoted  $g(i)$ .

- (1) Put the start node,  $s$ , on a list called OPEN of unexpanded nodes. If the start node is a goal node, a solution has been found. Otherwise, set  $g(s) = 0$ .
- (2) If OPEN is empty, no solution exists.
- (3) Select from OPEN a node  $i$  such that  $g(i)$  is minimum. If several nodes qualify, choose node  $i$  to be a goal node if there is one; otherwise, choose among them arbitrarily. Move node  $i$  from OPEN to a list, CLOSED, of expanded nodes.
- (4) If node  $i$  is a goal node, the solution has been found.
- (5) Expand node  $i$ . If it has no successors, go to (2).
- (6) For each successor node  $j$  of node  $i$ , compute  $g(j) = g(i) + c(i,j)$  and place all the successor nodes  $j$  in OPEN.
- (7) Go to (2).

### Depth-first Search

Depth-first search is characterized by the expansion of the most recently generated, or deepest, node first. Formally, the *depth* of a node in a tree is defined as follows:

The depth of the start node is 0.

The depth of any other node is one more than the depth of its predecessor.

As a consequence of expanding the deepest node first, the search follows a single path through the state space downward from the start node; only if it reaches a state that has no successors does it consider an alternate path. Alternate paths systematically vary those previously tried, changing only the last  $n$  steps while keeping  $n$  as small as possible.

In many problems, of course, the state-space tree may be of infinite depth, or at least may be deeper than some known upper bound on the length of an acceptable solution sequence. To prevent consideration of paths that are too long, a maximum is often placed on the depth of nodes to be expanded, and any node at that depth is treated as if it had no successors. It should be noted that, even if such a *depth bound* is used, the solution path found is not necessarily the shortest one.

The following algorithm describes depth-first search with a depth bound:

- (1) Put the start node on a list, OPEN, of unexpanded nodes. If it is a goal node, a solution has been found.
- (2) If OPEN is empty, no solution exists.
- (3) Move the first node,  $n$ , on OPEN to a list CLOSED of expanded nodes.
- (4) If the depth of node  $n$  is equal to the maximum depth, go to (2).
- (5) Expand node  $n$ . If it has no successors, go to (2).
- (6) Place all successors of node  $n$  at the beginning of OPEN.
- (7) If any of the successors of node  $n$  is a goal node, a solution has been found. Otherwise go to (2).

As an example, consider the following simple problem: A pawn is required to move through the matrix in Figure 3 from top to bottom. The pawn may enter the matrix anywhere in the top row. From a square containing 0, the pawn must move downward if the square below contains 0; otherwise, it must move horizontally. From a square containing 1, no further moves are possible. The goal is to reach a square containing zero in the bottom row. A depth bound of 5 is assumed.

	1	2	3	4
1	1	0	0	0
2	0	0	1	0
3	0	1	0	0
4	1	0	0	0

Figure 3. An example problem for depth-first search.

The search tree generated by the depth-first algorithm is shown in Figure 4. At node  $S_0$ , the pawn has not yet entered the grid. At the other nodes, its position is given as a (row number, column number) pair. The numbering of nodes gives the order in which they are moved out of the OPEN list of unexpanded nodes. When the algorithm terminates, the OPEN list contains  $S_{17}$  (a goal node) and  $S_{18}$ ; all other nodes are on the expanded list. The solution found, which is one move longer than the minimum, calls for the pawn to enter at  $(1,3)$ , move one square right, and then go straight down to  $(4,4)$ . Had no depth bound been used, the tree would have been one level deeper since node  $S_{12}$  has a successor,  $(4,1)$ . Since the algorithm treats the state space as a tree, not a general graph, it does not discover that the distinct nodes  $S_2$  and  $S_9$  in fact represent the same state. Consequently, the search downward from  $S_9$  duplicates the work already done from  $S_2$ .

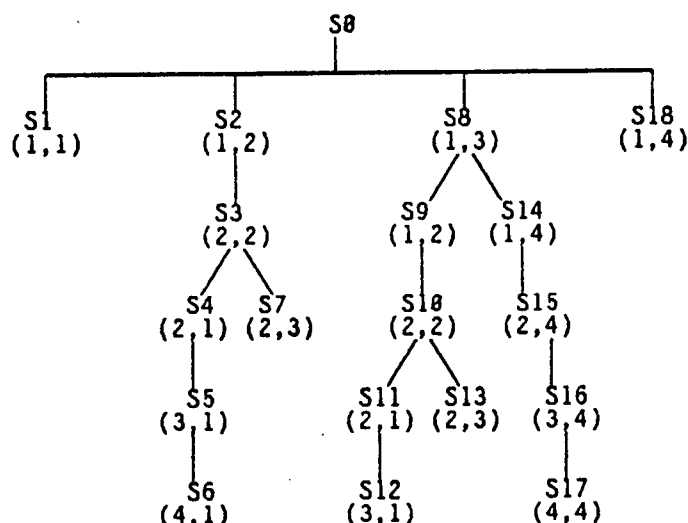


Figure 4. The search tree for Figure 3.

### Bidirectional Search

Each of the algorithms given above uses *forward reasoning*, working from the start node of a state-space tree towards a goal node and using operators that each map a node  $i$  to a successor node  $j$ . In some cases, the search could equally well use *backward reasoning*, moving from the goal state to the start state. An example of this is the 8-puzzle, in which

- (a) the goal state can be fully described in advance, and
- (b) it is easy to define inverse operators--each applicable operator mapping node  $j$  to a predecessor node  $i$ .

Since backward search through a tree is trivial, it is assumed that node  $j$  can have more than one predecessor--that is, several inverse operators may apply at node  $j$ . For example, in the pawn maze problem, Figure 4, position  $(1,2)$  [at nodes  $S_2$  and  $S_9$ ] would have both nodes  $S_0$  and  $S_8$  as predecessors.

Forward and backward reasoning can be combined into a technique called *bidirectional*

*search*. The idea is to replace a single search graph, which is likely to grow exponentially, by two smaller graphs: one starting from the initial state and one starting from the goal. The search terminates (roughly) when the two graphs intersect.

A bidirectional version of the uniform-cost algorithm, guaranteed to find the shortest solution path through a general state-space graph, is due to Pohl (1969, 1971). Empirical data for randomly generated graphs showed that Pohl's algorithm expanded only about one-fourth as many nodes as unidirectional search.

An algorithm for blind bidirectional search is given in detail below. A related algorithm for *heuristic* bidirectional search is discussed in Article C3d.

The following notation is used in the algorithm:

The start node is  $s$ ; the goal or terminal node,  $t$ .

S-OPEN and S-CLOSED are lists of unexpanded and expanded nodes, respectively, generated from the start node.

T-OPEN and T-CLOSED are lists of unexpanded and expanded nodes, respectively, generated from the terminal node.

The cost associated with the arc from node  $n$  to node  $x$  is denoted  $c(n,x)$ .

For a node  $x$  generated from the start node,  $gs(x)$  measures the shortest path found so far from  $s$  to  $x$ .

For a node  $x$  generated from the terminal node,  $gt(x)$  measures the shortest path found so far from  $x$  to  $t$ .

The algorithm is as follows:

- (1) Put  $s$  in S-CLOSED, with  $gs(s) = 0$ . Expand node  $s$ , creating a node for each of its successors. For each successor node  $x$ , place  $x$  on S-OPEN, attach a pointer back to  $s$ , and set  $gs(x) = c(s,x)$ . Correspondingly, put  $t$  in T-CLOSED, with  $gt(t) = 0$ . Expand node  $t$ , creating a node for each of its predecessors. For each predecessor node  $x$ , place  $x$  on T-OPEN, attach a pointer forward to  $t$ , and set  $gt(x) = c(x,t)$ .
- (2) Decide whether to go forward or backward. If forward, go to (3); if backward, to (4). (One way to implement this step is to alternate between forward and backward moves. Another way, which Pohl found to give better performance, is to move backward if T-OPEN contains fewer nodes than S-OPEN; otherwise, forward. It is assumed that a solution path does exist, so the chosen list will be nonempty.)
- (3) Select from S-OPEN a node  $n$  at which  $gs(n)$  is minimum. Move  $n$  to S-CLOSED. If  $n$  is also in T-CLOSED, go to (5). Otherwise, for each successor  $x$  of  $n$ :
  - (a) If  $x$  is on neither S-OPEN nor S-CLOSED, then add it to S-OPEN. Attach a pointer back to  $n$  and the path cost  $gs(x) = gs(n) + c(n,x)$ .

- (b) If  $x$  was already on S-OPEN, a shorter path to  $x$  may have just been found. Compare the previous path cost,  $gs(x)$ , with the new cost  $gs(n) + c(n,x)$ . If the latter is smaller, set  $gs(x)$  to the new path cost and point  $x$  back to  $n$  instead of its predecessor on the longer path.
- (c) If  $x$  was already on S-CLOSED, do nothing; although a new path to  $x$  has been found, its cost must be at least as great as the cost of the path already known. (For further consideration of this point, see Article C3b.)

Return to (2).

- (4) Select from T-OPEN a node  $n$  at which  $gt(n)$  is minimum. Move  $n$  to T-CLOSED. If  $n$  is also in S-CLOSED, go to (5). Otherwise, for each predecessor  $x$  of  $n$ :
  - (a) If  $x$  is on neither T-OPEN nor T-CLOSED, then add it to T-OPEN. Attach a pointer forward to  $n$  and the path cost  $gt(x) = gt(n) + c(x,n)$ .
  - (b) If  $x$  was already on T-OPEN and a shorter path from  $x$  to  $t$  has just been found, reduce the stored value of  $gt(x)$ , and point  $x$  forward to  $n$  (instead of to its successor on the longer path).
  - (c) If  $x$  was already on T-CLOSED, do nothing.
 Return to (2).

- (5) Consider the set of nodes that are in both S-CLOSED and either T-CLOSED or T-OPEN. Select from this set a node  $n$  for which  $gs(n) + gt(n)$  is minimum; and exit with the solution path obtained by tracing the path from  $n$  back to  $s$  and forward to  $t$ .

#### References

See Nilsson (1971), Pohl (1969), and Pohl (1971).

## C2. Blind AND/OR Graph Search

A problem to be solved using AND/OR-graph search can be defined by specifying a start node (representing an initial goal or problem description), a set of terminal nodes (descriptions of primitive problems), and a set of operators for reducing goals to subgoals. The rules for constructing an AND/OR graph, together with the use of such graphs for problem-reduction representation, were discussed in Article B2. To recapitulate briefly, each possible application of an operator at a node  $n$  (see Figure 1) is represented by a directed arc from node  $n$  to a successor node; these successor nodes are called *OR nodes*, since only one of the operator applications will ever be needed to solve the problem that node  $n$  represents. Each OR node successor of node  $n$  represents a set of subproblems. If the set of subproblems represented by an OR node  $m$  has more than one element, then there are directed arcs from  $m$  to nodes representing the individual elements of the set. These successors are called *AND nodes*, because all of the elements of the set must be solved in order to solve the subproblem set represented by node  $m$ . To distinguish AND nodes visually from OR nodes, the arcs in the graph from  $m$  to its AND successors are joined by a horizontal line.

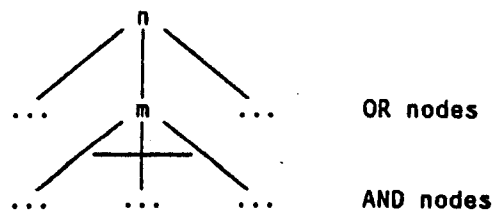


Figure 1. AND/OR graph notation.

Formally, a node or problem is said to be *solved* if one of the following conditions holds:

1. The node is in the set of terminal nodes (primitive problems). (In this case, the node has no successors.)
2. The node has AND nodes as successors and all these successors are solved.
3. The node has OR nodes as successors and any one of these successors is solved.

A solution to the original problem is given by a subgraph of the AND/OR graph sufficient to show that the start node is solved. In Figure 2, for example, assuming that nodes 5, 6, 8, 9, 10, and 11 are all terminal, there are three possible solution subgraphs: {1, 2, 4, 8, 9}, {1, 3, 5, 6, 7, 10}, and {1, 3, 5, 6, 7, 11}.

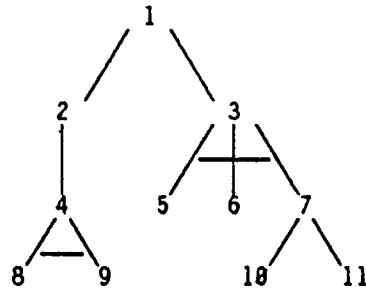


Figure 2. An AND/OR graph.

A node is said to be *unsolvable* if one of the following conditions is true:

1. The node has no successors and is not in the set of terminal nodes. That is, it is a nonprimitive problem to which no operator can be applied.
2. The node has AND nodes as successors and one or more of these successors is unsolvable.
3. The node has OR nodes as successors and all of these successors are unsolvable.

Again in Figure 2, node 1 would be unsolvable if all nodes in any of the following sets were unsolvable: {8, 5}, {8, 6}, {8, 10, 11}, {9, 5}, {9, 6}, {9, 10, 11}.

Two algorithms for the blind search of an AND/OR tree (breadth-first and depth-first) are given at the end of this article. They have several features in common with blind state-space search algorithms (Article C1): The operation of *expanding* a node is again present, and again the algorithms differ mainly in the order in which nodes are considered for expansion. It should be noted that the expansion of a node may differ slightly from the case of state-space search. In Figure 2, for example, two operators apply at node 1: One reduces it to a single equivalent problem (node 2) and the other to a set (node 3) of three subproblems (nodes 5, 6, and 7). In this case, nodes 2, 3, 5, 6, and 7 would all be generated in expanding node 1, and each new node would be given a pointer to its immediate predecessor, but only nodes 2, 5, 6, and 7 would be placed on the list of unexpanded nodes.

In contrast to the state-space search algorithms, most of which use forward reasoning, the search algorithms below reason backward from the initial goal. The algorithms described here make two important simplifying assumptions: (a) The search space is an AND/OR tree and not a general graph, and (b) when a problem is transformed to a set of subproblems, the subproblems may be solved in any order. The first assumption implies that identical subproblems may arise at different nodes of the search tree and will need to be solved anew whenever one of them is encountered. Modifications needed for searching a general AND/OR graph are discussed in Nilsson (1971). A way of eliminating the second assumption, that all subproblems are independent, is discussed in Article C4, Heuristic Search of an AND/OR Graph.



### Breadth-first Search of an AND/OR Tree

The following algorithm describes the breadth-first search of an AND/OR tree. If a solution tree exists, this algorithm finds a solution tree of minimum depth, provided that intermediate OR nodes are ignored in calculating the depth of the tree. The start node is assumed not to be a terminal node.

- (1) Put the start node on a list, OPEN, of unexpanded nodes.
- (2) Remove the first node,  $n$ , from OPEN.
- (3) Expand node  $n$ --generating all its immediate successors and, for each successor  $m$ , if  $m$  represents a set of more than one subproblem, generating successors of  $m$  corresponding to the individual subproblems. Attach, to each newly generated node, a pointer back to its immediate predecessor. Place all the new nodes that do not yet have descendants at the end of OPEN.
- (4) If no successors were generated in (3), then
  - (a) Label node  $n$  unsolvable.
  - (b) If the unsolvability of  $n$  makes any of its ancestors unsolvable, label these ancestors unsolvable.
  - (c) If the start node is labeled unsolvable, exit with failure.
  - (d) Remove from OPEN any nodes with an unsolvable ancestor.
- (5) Otherwise, if any terminal nodes were generated in (3), then
  - (a) Label these terminal nodes solved.
  - (b) If the solution of these terminal nodes makes any of their ancestors solved, label these ancestors solved.
  - (c) If the start node is labeled solved, exit with success.
  - (d) Remove from OPEN any nodes that are labeled solved or that have a solved ancestor.
- (6) Go to step 2.

### Depth-first Search of an AND/OR Tree

A bounded depth-first search can be obtained by changing only step 3 of the breadth-first algorithm. The revised step 3 is as follows:

- (3') *If the depth of  $n$  is less than the depth bound, then:* Expand node  $n$ , generating all its immediate successors and, for each successor  $m$ , if  $m$  represents a set of more than one subproblem, generating successors of  $m$  corresponding to the individual subproblems. Attach, to each newly generated node, a pointer back to its immediate predecessor. Place all the new nodes that do not yet have descendants at the *beginning* of OPEN.

The depth-first search will find a solution tree, provided one exists within the depth bound. As with breadth-first search, the notion of depth is more meaningful if intermediate OR nodes are not counted. For this purpose one might add the following to the end of step 3':

For each node  $x$  added to OPEN, set the depth of  $x$  to be the depth of node  $n$ , plus 1.

Given that the start node has depth 0, the depth of any node  $x$  will then be the length of the operator sequence that must be applied to reach node  $x$  from the start node.

#### References

See Nilsson (1971).

### C3. Heuristic State-space Search

#### C3a. Basic Concepts in Heuristic Search

In the blind search of a state-space (Article C1) or an AND/OR graph (Article C2), the number of nodes expanded before reaching a solution is likely to be prohibitively large. Because the order of expanding the nodes is purely arbitrary and does not use any properties of the problem being solved, one usually runs out of space or time (or both) in any but the simplest problems. This result is a manifestation of the *combinatorial explosion*.

Information about the particular problem domain can often be brought to bear to help reduce the search. In this section, it is assumed that the definitions of initial states, operators, and goal states all are fixed, thus determining a search space; the question, then, is how to search the given space efficiently. The techniques for doing so usually require additional information about the properties of the specific problem domain beyond that which is built into the state and operator definitions. Information of this sort will be called *heuristic information*, and a search method using it (whether or not the method is foolproof) will be called a *heuristic search method* (Nilsson, 1971).

#### The Importance of Heuristic Search Theory

Heuristic search methods were employed by nearly all early problem-solving programs. Most of these programs, though, were written to solve problems from a single domain, and the domain-specific information they used was closely intertwined with the techniques for using it. Thus the heuristic techniques themselves were not easily accessible for study and adaptation to new problems, and there was some likelihood that substantially similar techniques would have to be reinvented repeatedly. Consequently, an interest arose in developing generalized heuristic search algorithms, whose properties could be studied independently of the particular programs that might use them. (See Newell & Ernst, 1965; Feigenbaum, 1969; Sandewall, 1971.) This task, in turn, required a way of describing problems that generalized across many different domains. Such generalized problem formulations have been discussed in Section B, Problem Representation, in an approach generally following Nilsson (1971). Given a generalized problem representation, the most basic heuristic search techniques can be studied as variations on blind search methods for the same type of problem representation.

The current state of heuristic search theory has been diversely judged. One of the best known students of the subject has remarked, "The problem of efficiently searching a graph has essentially been solved and thus no longer occupies AI researchers" (Nilsson, 1974). Other work makes it clear, however, that the theory is far from complete (e.g., Gaschnig, 1977; Simon & Kadane, 1975). Its kinship with complexity theory now tends to be emphasized (see Pohl, 1977).

#### Ways of Using Heuristic Information

The points at which heuristic information can be applied in a search include

- (a) deciding which node to expand next, instead of doing the expansions in a strictly breadth-first or depth-first order;

- (b) in the course of expanding a node, deciding which successor or successors to generate--instead of blindly generating all possible successors at one time; and
- (c) deciding that certain nodes should be discarded, or *pruned*, from the search tree.

A state-space search algorithm is presented below that uses heuristic information only at the first of these points, deciding which node to expand next, on the assumption that nodes are to be expanded fully or not at all. The general idea is always to expand the node that seems "most promising." A search that implements this idea is called an *ordered search* or *best-first search*. Ordered search has been the subject of considerable theoretical study, and several variations on the basic algorithm below are reviewed in articles IIC3b through IIC3d (ordered state-space search) and article IIC4 (ordered AND/OR graph search).

The other two uses of heuristic information can be discussed more briefly. Decisions of the second kind--determining which successors to generate--are often decisions of operator selection, determining which operator to apply next to a given node. A node to which some but not all applicable operators have been applied is said to have been *partially developed* or *partially expanded*. The use of heuristic information to develop nodes partially, reserving the possibility of fuller expansion at a later point in the search, has been investigated by Michie (1967) and by Michie and Ross (1970). Other applications of the idea of limiting the successors of a given node occur in game-playing programs (see C5c). Another important variant of the idea is *means-ends analysis*, which, instead of deciding on an applicable operator, chooses an operator most likely to advance the search whether or not it is immediately applicable. The problem of making the operator applicable, if necessary, is addressed secondarily. (See D2, GPS; and D5, STRIPS.)

The third use of heuristic information, for *pruning*, amounts to deciding that some nodes should never be expanded. In some cases, it can be definitely determined that a node is not part of a solution, and the node may then be safely discarded, or *pruned*, from the search tree. In other cases pruning may be desirable even though the nodes pruned cannot be guaranteed inessential to a solution. One reason, in conjunction with a best-first search, is simply to save the space that would be required to retain a large number of apparently unpromising nodes on a list of candidates for possible future expansion. For examples, see Doran (1967) and Harris's *bandwidth search* (article IIC3c). Another reason for pruning is as a restriction on a search that is otherwise blind. For example, a breadth-first search could be modified to choose between expansion and pruning for each node it considers. This pruning to control the search is also very important for problems in which all solutions, rather than just a single solution, must be found; for finding all solutions implies an exhaustive exploration of all unpruned parts of the search space. An example of a search for all solutions is the DENDRAL program (see Applications.Dendral).

#### Ordered State-space Search

An *ordered* or *best-first search*, as mentioned above, is one that always selects the most promising node as the next node to expand. The choice is ordinarily assumed to be global,

that is, to operate on the set of all nodes generated but not yet expanded. A local choice would also be possible, however; for example, an *ordered depth-first search* would be one that always expands the most promising successor of the node last expanded.

The promise of a node can be defined in various ways. One way, in a state-space problem, is to estimate its distance from a goal node; another is to assume that the solution path includes the node being evaluated and estimate the length or difficulty of the entire path. Along a different dimension, the evaluation may consider only certain predetermined features of the node in question, or it may determine the relevant features by comparing the given node with the goal. In all these cases, the measure by which the promise of a node is estimated is called an *evaluation function*.

A basic algorithm for ordered state-space search is given by Nilsson (1971). The evaluation function is  $f^*$ ; it is defined so that the more promising a node is, the smaller is the value of  $f^*$ . The node selected for expansion is one at which  $f^*$  is minimum. The state space is assumed to be a general graph.

The algorithm is as follows:

- (1) Put the start node  $s$  on a list, called OPEN, of unexpanded nodes. Calculate  $f^*(s)$  and associate its value with node  $s$ .
- (2) If OPEN is empty, exit with failure; no solution exists.
- (3) Select from OPEN a node  $i$  at which  $f^*$  is minimum. If several nodes qualify, choose a goal node if there is one, and otherwise choose among them arbitrarily.
- (4) Remove node  $i$  from OPEN and place it on a list, called CLOSED, of expanded nodes.
- (5) If  $i$  is a goal node, exit with success; a solution has been found.
- (6) Expand node  $i$ , creating nodes for all its successors. For every successor node  $j$  of  $i$ :
  - (a) Calculate  $f^*(j)$ .
  - (b) If  $j$  is neither in list OPEN nor in CLOSED, then add it to OPEN, with its  $f^*$  value. Attach a pointer from  $j$  back to its predecessor  $i$  (in order to trace back a solution path once a goal node is found).
  - (c) If  $j$  was already on either OPEN or CLOSED, compare the  $f^*$  value just calculated for  $j$  with the value previously associated with the node. If the new value is lower, then (i) substitute it for the old value, (ii) point  $j$  back to  $i$  instead of to its previously found predecessor, and (iii) if node  $j$  was on the CLOSED list, move it back to OPEN.
- (7) Go to 2.

Step 6c is necessary for general graphs, in which a node can have more than one predecessor. The predecessor yielding the smaller value of  $f^*(j)$  is chosen. For trees, in which a node has at most one predecessor, step 6c can be ignored. Note that even if the search space is a general graph, the subgraph that is made explicit is always a tree since node  $j$  never records more than one predecessor at a time.

Breadth-first, uniform-cost, and depth-first search (Article C1, Blind State-space Search) are all special cases of the ordered search technique. For breadth-first search, we choose  $f^*(i)$  to be the depth of node  $i$ . For uniform-cost search,  $f^*(i)$  is the cost of the path

from the start node to node  $i$ . A depth-first search (without a depth bound) can be obtained by taking  $f^*(i)$  to be the negative of the depth of the node.

The purpose of ordered search, of course, is to reduce the number of nodes expanded as compared to blind-search algorithms. Its effectiveness in doing this depends directly on the choice of  $f^*$ , which should discriminate sharply between promising and unpromising nodes. If the discrimination is inaccurate, however, the ordered search may miss an optimal solution or all solutions. If no exact measure of promise is available, therefore, the choice of  $f^*$  involves a trade-off between time and space on the one hand and the guarantee of an optimal solution, or any solution, on the other.

### Problem Types and the Choice of $f^*$

The measure of a node's promise--and consequently, the appropriateness of a particular evaluation function--depends on the problem at hand. Several cases can be distinguished by the type of solution they require. In one, it is assumed that the state space contains multiple solution paths with different costs; the problem is to find the optimal (i.e., minimum cost) solution. This first case is well understood; see Article C3b on the A\* algorithm.

The second situation is similar to the first but with an added condition: The problem is hard enough that, if it is treated as an instance of case one, the search will probably exceed bounds of time and space before finding a solution. The key questions for case two are (a) how to find good (but not optimal) solutions with reasonable amounts of search effort and (b) how to bound both the search effort and the extent to which the solution produced is less than optimal.

A third kind of problem is one in which there is no concern for the optimality of the solution; perhaps only one solution exists, or any solution is as good as any other. The question here is how to minimize the search effort--instead of, as in case two, trying to minimize some combination of search effort and solution cost.

An example of case three comes from theorem proving, where one may well be satisfied with the most easily found proof, however inelegant. A clear example of case two is the traveling-salesman problem, in which finding some circuit through a set of cities is trivial, and the difficulty, which is very great, is entirely in finding a shortest or close-to-shortest path. Most treatments, however, do not clearly distinguish between the two cases. A popular test problem, the 8-puzzle, can be treated as being in either class. For further discussion of cases two and three, see Article C3c, Relaxing the Optimality Requirement.

### References

See Doran (1967), Feigenbaum (1969), Gaschnig (1977), Michie (1967), Michie & Ross (1970), Newell & Ernst (1965), Newell & Simon (1972), Nilsson (1971), Nilsson (1974), Pohl (1977), Sandewall (1971), and Simon & Kadane (1975).

### C3b. A\*--Optimal Search for an Optimal Solution

The A\* algorithm, described by Hart, Nilsson, and Raphael (1968), addresses the problem of finding a minimal cost path joining the start node and a goal node in a state-space graph. This problem subsumes the problem of finding the path between such nodes containing the smallest number of arcs. In the latter problem, each arc (representing the application of an operator) has cost 1; in the minimal cost path problem, the costs associated with arcs can be arbitrary. Historically, the predecessors of A\* include Dijkstra's algorithm (1959) and Moore's algorithm (1959). A class of algorithms similar to A\* is used in operations research under the name *branch-and-bound* algorithms (see Hall, 1971; Hillier & Lieberman, 1974; Lawler & Wood, 1966; and Reingold, Nievergelt, & Deo, 1977).

The algorithm used by A\* is an *ordered state-space search* (Article C3a). Its distinctive feature is its definition of the *evaluation function*  $f^*$ . As in the usual ordered search, the node chosen for expansion is always one at which  $f^*$  is minimum.

Since  $f^*$  evaluates nodes in light of the need to find a minimal cost solution, it considers the value of each node  $n$  as having two components: the cost of reaching  $n$  from the start node, and the cost of reaching a goal from node  $n$ . Accordingly,  $f^*$  is defined by

$$f^*(n) = g^*(n) + h^*(n)$$

where  $g^*$  estimates the minimum cost of a path from the start node to node  $n$ , and  $h^*$  estimates the minimum cost from node  $n$  to a goal. The value  $f^*(n)$  thus estimates the minimal cost of a solution path passing through node  $n$ . The actual costs, which  $f^*$ ,  $g^*$ , and  $h^*$  only estimate, are denoted by  $f$ ,  $g$ , and  $h$ , respectively. It is assumed that all arc costs are positive.

The function  $g^*$ , applied to a node  $n$  being considered for expansion, is calculated as the actual cost from the start node  $s$  to  $n$  along the cheapest path found so far by the algorithm. If the state space is a tree, then  $g^*$  gives a perfect estimate since only one path from  $s$  to  $n$  exists. In a general state-space graph,  $g^*$  can err only in the direction of overestimating the minimal cost; its value is adjusted downward if a shorter path to  $n$  is found. Even in a general graph, there are certain conditions (mentioned below) under which  $g^*(n)$  can be shown to be a perfect estimate by the time node  $n$  is chosen for expansion.

The function  $h^*$  is the carrier of *heuristic information* and can be defined in any way appropriate to the problem domain. For the interesting properties of the A\* algorithm to hold, however,  $h^*$  should be nonnegative, and it should never overestimate the cost of reaching a goal node from the node being evaluated. That is, for any such node  $n$  it should always hold that  $h^*(n)$  is less than or equal to  $h(n)$ , the actual cost of an optimal path from  $n$  to a goal node. This last condition is called the *admissibility condition*.

#### Admissibility and Optimality of A\*

It can be shown that if  $h^*$  satisfies the admissibility condition and if, in addition, all arc costs are positive and can be bounded from below by a positive number, then A\* is guaranteed to find a solution path of minimal cost if any solution path exists. This property is called the property of *admissibility*.

Although the admissibility condition requires  $h^*$  to be a lower bound on  $h$ , it is to be expected that the more nearly  $h^*$  approximates  $h$ , the better the algorithm will perform. If  $h^*$  were identically equal to  $h$ , an optimal solution path would be found without ever expanding a node off the path (assuming only one optimal solution exists). If  $h^*$  is identically zero,  $A^*$  reduces to the blind uniform-cost algorithm (Article C1). Two otherwise similar algorithms, say  $A_1$  and  $A_2$ , can be compared with respect to their choices of the  $h^*$  function, say  $h_1^*$  and  $h_2^*$ . Algorithm  $A_1$  is said to be *more informed than*  $A_2$  if, whenever a node  $n$  (other than a goal node) is evaluated,

$$h_1^*(n) > h_2^*(n).$$

On this basis an *optimality* result for  $A^*$  can be stated: If  $A$  and  $A^*$  are admissible algorithms such that  $A^*$  is more informed than  $A$ , then  $A^*$  never expands a node that is not also expanded by  $A$ . A proof (correcting the proof given in Nilsson, 1971) appears in Gelperin (1977).

#### Optimality and Heuristic Power

The sense in which  $A^*$  yields an optimal search has to do only with the number of nodes it expands in the course of finding a minimal-cost solution. But there are other relevant considerations. First, the difficulty of computing  $h^*$  also affects the total computational effort. Second, it may be less important to find a solution whose cost is absolutely minimum than to find a solution of reasonable cost within a search of moderate length. In such a case one might prefer an  $h^*$  that evaluates nodes more accurately in most cases but sometimes overestimates the distance to a goal, thus yielding an inadmissible algorithm. (See Article C3c.) The choice of  $h^*$  and the resulting *heuristic power* of the algorithm depend upon a compromise among these considerations.

A final question one might consider is the number of node expansions, as opposed to the number of distinct nodes expanded by  $A^*$ . The two totals will be the same provided that whenever a node  $n$  is expanded (moved to the CLOSED list), an optimal path to  $n$  has already been found. This condition is always satisfied in a state-space tree, where  $g^*(n) = g(n)$  necessarily. It will also be satisfied in a general state-space graph if a condition called the *consistency assumption* holds (see Hart, Nilsson, & Raphael, 1968). The general idea of the assumption is that a form of the triangle inequality holds throughout the search space. Specifically, the assumption is that for any nodes  $m$  and  $n$ , the estimated distance  $h^*(m)$  from  $m$  to a goal should always be less than or equal to the actual distance from  $m$  to  $n$  plus the estimated remaining distance,  $h^*(n)$ , from  $n$  to a goal. For an  $h^*$  not satisfying the consistency assumption on a general state-space graph, Martelli (1977) has shown that  $A^*$  is not optimal with respect to the number of expansions and has given an algorithm that runs more efficiently under these circumstances.

#### References

See Dijkstra (1959), Gelperin (1977), Hall (1971), Hart, Nilsson, & Raphael (1968), Hart, Nilsson, & Raphael (1972), Hillier & Lieberman (1974), Lawler & Wood (1966), Martelli (1977), Moore (1959), and Reingold, Nievergelt, & Deo (1977).



### C3c. Relaxing the Optimality Requirement

The A\* algorithm (C3b) is an ordered state-space search using the evaluation function  $f^* = g^* + h^*$ . If the appropriate conditions are met, including most importantly the *admissibility condition*, that the estimate  $h^*(n)$  is always less than or equal to  $h(n)$ , then A\* is guaranteed to find an optimal solution path if one exists. Again under suitable conditions, the performance of A\* is optimal in comparison with other similarly defined admissible algorithms. Still, several questions remain:

- (1) One may be more concerned with minimizing search effort than with minimizing solution cost. Is  $f^* = g^* + h^*$  an appropriate evaluation function in this case?
- (2) Even if solution cost is important, the combinatorics of the problem may be such that an admissible A\* cannot run to termination. Can speed be gained at the cost of a bounded decrease in solution quality?
- (3) It may be hard to find a good heuristic function  $h^*$  that satisfies the admissibility condition; with a poor but admissible heuristic function, A\* deteriorates into blind search. How is the search affected by an inadmissible heuristic function?

#### Minimizing Search Effort

An approach to the first question can be stated as follows. The reason for including  $g^*$  in the evaluation function is to add a breadth-first component to the search; without  $g^*$ , the evaluation function would estimate, at any node  $n$ , the remaining distance to a goal and would ignore the distance already covered in reaching  $n$ . If the object is to minimize search effort instead of solution cost, one might conclude that  $g^*$  should be omitted from the evaluation function. An early heuristic search algorithm that did just this was Doran and Michie's Graph Traverser (Doran & Michie, 1966; Doran, 1967); the evaluation function used was of the form  $f^* = h^*$ , and the object was to minimize total search effort in finding solutions to the 8-puzzle and other problems. A generalization covering the Graph Traverser algorithm, A\*, and others has been defined by Pohl (1969, 1970a, 1970b) as the Heuristic Path Algorithm (HPA). This algorithm gives an ordered state-space search with an evaluation function of the form

$$f^* = (1 - w)g^* + wh^*$$

where  $w$  is a constant in  $[0, 1]$  giving the relative importance to be attached to  $g$  and  $h$ . Choosing  $w = 1$  gives the Graph Traverser algorithm;  $w = 0$  gives breadth-first search; and  $w = .5$  is equivalent to the A\* function  $f^* = g^* + h^*$ .

Pohl's results concerning HPA indicate that, at least in special cases, omitting  $g^*$  from the evaluation function is a mistake. One case is that in which  $h^*$  is the most accurate heuristic function possible: If  $h^*(n) = h(n)$  at every node  $n$ , the evaluation function  $f^* = h^*$  still expands no fewer nodes than  $f^* = g^* + h^*$ . The other case assumes a simplified state space, whose graph is an infinite  $m$ -ary tree, and assumes that the error in  $h^*$ --which may underestimate or overestimate  $h$ --is bounded by a nonnegative integer  $e$ . In this situation it is shown that the maximum number of nodes expanded with  $f^* = h^*$  is greater than the

maximum number expanded with  $f^* = g^* + h^*$ , and that the difference between the maxima is exponential in the error bound  $\epsilon$ . This analysis by Pohl is one of the earliest applications of oracle or adversary analysis for discovering worst-case algorithmic efficiency. As such it is an important precursor to work on NP-complete problems and their attempted solution by heuristics. (For a general introduction to NP-completeness see Aho, Hopcroft, & Ullman, 1974.)

The two functions  $f^* = h^*$  and  $f^* = g^* + h^*$  have not been analyzed with respect to their average-case, as opposed to worst-case, behavior. Pohl's empirical results suggest that ordered search may typically expand the fewest nodes, provided the  $h^*$  function is fairly good, if  $g^*$  is included but given less weight than  $h^*$ --that is, with  $w$  greater than .5 but less than 1. These results were obtained for the 15-puzzle, a task exactly like the 8-puzzle except that it uses 15 tiles in a  $4 \times 4$  array.

For problems that differ from the 15-puzzle, in that some states lead to dead ends rather than only to longer solutions, a somewhat different approach has been taken recently by Simon and Kadane (1975). Whereas the evaluation functions  $f^* = g^* + h^*$  and  $f^* = h^*$  are based on the estimated solution cost at a given node, Simon and Kadane propose that the function should also take explicit account of the probability that the node is in fact on a solution path. With such a function, an expected long search with high probability of success could readily rate just as favorably as one that is potentially shorter but which has a higher chance of failing.

#### Solution Quality and Heuristic Error

The second question, of speed vs. solution quality, has been studied by Pohl (1973, 1977) and Harris (1973, 1974). Harris's work concerns the third question (inadmissible heuristic functions) as well, as do Pohl's results summarized above. Both Harris and Pohl consider the *traveling-salesman problem*, which is NP-complete (Karp, 1972).

Pohl's approach is a further generalization of the HPA evaluation function: Now  $f^*(n) = g^*(n) + w(n)h^*(n)$ . That is, the relative weight  $w$  to be attached to  $g^*$  and  $h^*$  is no longer constant; the function  $w(n)$ , which may be greater than or equal to 1, is defined to vary with the depth of node  $n$ . This approach is called *dynamic weighting*. With a definition of  $w$  that weights  $h^*$  less heavily as the search goes deeper, and with the assumption that  $h^*$  is a lower bound on  $h$ , Pohl shows that HPA will find a solution to the traveling-salesman problem whose cost is bounded by the ratio

$$\frac{\text{cost of tour found}}{\text{cost of optimal solution}} < 1 + \epsilon$$

where  $\epsilon$  is a constant in  $[0,1)$  which appears in the definition of  $w$ .

Dynamic weighting was tested on an instance of the traveling-salesman problem, known as the Croes problem, which involves 20 cities and has a known optimal solution cost of 246. An admissible  $A^*$ --which produces an optimal solution if it produces any--had still not terminated after expanding 500 nodes. With dynamic weighting, however, together with an

appropriate choice of  $\epsilon$  and the same  $h^*$  function, a solution with cost 260 was found by expanding only 53 nodes.

Harris's approach, called *bandwidth search*, is somewhat different from Pohl's. It assumes that no good  $h^*$  function satisfying the admissibility condition is available. In its place, he introduces the *bandwidth condition*, which requires that for all non-goal nodes  $n$ ,

$$(1) h^*(n) \leq h(n) + \epsilon$$

and

$$(2) h(n) - d \leq h^*(n) .$$

It is assumed that  $h^*$  satisfies the *consistency assumption* (see Article C3b).

With respect to the first part of the condition, it can be shown that if  $h^*$  never overestimates the distance to a goal by more than  $\epsilon$ , the cost of a solution found by  $A^*$  will not exceed the cost of an optimal solution by more than  $\epsilon$ . With such an  $h^*$ , the algorithm is said to be  *$\epsilon$ -admissible*; and the goal it finds,  *$\epsilon$ -optimal*.

Once the bandwidth search finds some solution, a further application of condition (1) may show that the cost of the solution found is in fact closer than  $\epsilon$  to an optimal solution. This is possible because (a) the cost of the solution found is known, and (b) a lower bound on the cost of every other solution is the minimum, over all nodes  $n$  remaining on the OPEN list, of  $f^*(n) - \epsilon$ . If the difference between these two quantities is too big, the search can be continued until it finds a solution that is acceptably close to the optimum.

The second part of the bandwidth condition, condition (2), can be used to save storage space by dropping nodes from the OPEN list, without any risk of dropping a node that is in fact on an optimal path to a goal. Let node  $q$  be a node that, having a minimum value of  $f^*$ , has been selected for expansion. Then any node  $m$  may safely be dropped from OPEN if  $f^*(m)$  is hopelessly big compared to  $f^*(q)$ . Specifically, it can be shown that all nodes  $m$  can be dropped if there is a node  $q$  such that

$$f^*(m) - (\epsilon + d) > f^*(q) .$$

Harris notes that it may be difficult to find a heuristic function  $h^*$  that satisfies both parts of the bandwidth condition. One may instead define two heuristic functions, one to order the search and one to determine which nodes can be dropped. Such functions, say  $h_1^*$  and  $h_2^*$ , should then satisfy

$$(1') h_1^*(n) \leq h(n) + \epsilon$$

and

$$(2') h(n) - d \leq h_2^*(n) .$$

Using two such heuristic functions, Harris tested the bandwidth search on several instances of the traveling-salesman problem including the 20-city Croes problem mentioned above. Harris's results, including a comparison with  $A^*$  using an admissible heuristic function, are summarized below. The OPEN list was limited to 500 nodes.

No. of cities	BANDWIDTH SEARCH		ADMISSIBLE SEARCH	
	Quality of solution	Nodes expanded	Quality of solution	Nodes expanded
6	5-optimal	6		
6	optimal	14	optimal	18
11	optimal	14	none	500 open nodes
20	4-optimal	42	none	500 open nodes

Figure 1. Comparison of bandwidth search and admissible search.

#### References

See Aho, Hopcroft, & Ullman (1974), Doran & Michie (1966), Doran (1967), Harris (1973), Harris (1974), Karp (1972), Nilsson (1971), Pohl (1969), Pohl (1970a), Pohl (1970b), Pohl (1973), Pohl (1977), and Simon & Kadane (1975).

### C3d. Bidirectional Search

Earlier articles in this chapter describe (a) heuristic state-space search methods using forward reasoning and (b) a blind state-space search combining forward and backward reasoning into a *bidirectional* algorithm. The kinds of problems to which a bidirectional state-space method applies are considered in Article C1; in general, it must be possible in these problems to search either forward, from the initial state toward the goal, or backward, from the goal toward the initial state. A bidirectional search pursues both lines of reasoning in parallel, growing two search trees and terminating when they meet. The motivation is that, in many cases, the number of nodes in a search tree grows exponentially with its depth; if a solution can be found by using two trees of half the depth, the search effort should be reduced significantly. Blind bidirectional search was in fact found to expand far fewer nodes than its unidirectional counterpart. A natural next question is whether heuristic bidirectional search can give still greater improvements in efficiency.

This question was investigated by Pohl (1969, 1971). Whereas his blind bidirectional algorithm used forward and backward uniform-cost search, his heuristic algorithm used forward and backward *ordered search*. Otherwise, the two algorithms differed mainly in their termination conditions. In both cases the termination condition was complicated by the fact that the algorithms were designed to find an *optimal* path between the start and goal nodes; they could be simplified if any path would do.

As *evaluation functions*, Pohl's heuristic bidirectional algorithm used functions parallel to those of  $A^*$ . For a node  $x$  in the forward search tree:

$gs(x)$  measured the shortest path found so far from the start node,  $s$ , to  $x$ ;

$hs(x)$  estimated the minimum remaining distance from  $x$  to the terminal node,  $t$ ; and

$fs(x) = gs(x) + hs(x)$  was the evaluation function.

Similarly, for a node  $x$  generated in the backward search:

$gt(x)$  measured the shortest path found so far from  $x$  to  $t$ ;

$ht(x)$  estimated the minimum distance from  $s$  to  $x$ ; and

$ft(x) = gt(x) + ht(x)$  was the evaluation function.

Constraints were placed on the heuristic functions  $hs$  and  $ht$ , corresponding to the admissibility condition and the consistency assumption of  $A^*$ , in order to guarantee the optimality of the solution.

Pohl's results, in experiments using bidirectional heuristic search on the 15-puzzle, were disappointing. It was hoped that the search trees rooted at the start and goal nodes would meet near the middle of the solution path. In blind search, this had happened necessarily because both trees were expanded breadth-first. (Recall that uniform-cost search is a generalization of the breadth-first algorithm.) In the heuristic case, however, the

search in each direction was narrowed. Since each problem had many alternate solutions, the typical outcome was that both search trees grew to include nearly complete, but different, solution paths before intersecting.

Several ideas have been advanced for forcing the trees to meet earlier while retaining the benefit of heuristic information (Pohl, 1971; Kowalski, 1972; de Champeaux & Sint, 1975, 1977; Pohl, 1977). One that has been tested is that of Champeaux and Sint, which redefines the heuristic functions  $h_s$  and  $h_t$  as follows:

Let T-OPEN be the list of unexpanded nodes of the backward search tree. For a node  $x$  in the forward search tree,  $h_s(x)$  estimates the minimum distance from  $x$  to the goal  $t$  by way of some node  $y$  in T-OPEN. That is,  $h_s(x)$  is the minimum, over all nodes  $y$  on T-OPEN, of the estimated distance from  $x$  to  $y$  plus  $g_t(y)$ , the length of the shortest known path from  $y$  to the goal.

The function  $h_t$  is defined analogously. The authors reported, for the same problems Pohl had used, that the algorithm generally produced shorter solution paths, with fewer nodes expanded, and that the search graphs now did meet near the middle of the search space. Unfortunately, however,  $h_s$  and  $h_t$  were so expensive to compute--since for each node  $x$  to be expanded, its distance must be estimated to every node  $y$  on the opposite OPEN list--that the algorithm still ran much more slowly than unidirectional heuristic search.

#### References

See de Champeaux & Sint (1975), de Champeaux & Sint (1977), Kowalski (1972), Pohl (1969), Pohl (1971), and Pohl (1977).

#### C4. Heuristic Search of an AND/OR Graph

This article returns to the problem of searching an AND/OR graph, as opposed to an ordinary state-space graph. The distinction between the two is the presence of AND nodes, which add conceptual complications to the search problem. Each node of the AND/OR graph represents a goal to be achieved. It will be assumed throughout that reasoning is backward, from an initial goal (the root) toward an equivalent set of subgoals, all of which have immediate solutions. On this assumption, an AND/OR graph constitutes (in the terminology of this chapter) a *problem-reduction representation*. This identification gives another way of stating the distinction between problem-reduction and state-space representations: State-space operators always take exactly one input and produce exactly one output; a problem-reduction operator also takes a single input but may produce multiple outputs (see Section B).

To put the matter further into perspective, one may also conceive of searching an AND/OR graph in the forward direction--from the primitive problems, whose solutions are already known, toward the problem one actually wishes to solve. Just such a graph search is that typically conducted by a resolution theorem-prover, as it brings together two or more axioms or previous conclusions and applies to them an operator yielding one new deduction as its result. (See Theorem Proving.) *Forward reasoning* in an AND/OR graph, then, would be distinguished from a state-space search by the presence of multiple-input, single-output operators. For further discussion, including an algorithm for *bidirectional search* of an AND/OR graph, see Kowalski (1972); see also Martelli and Montanari (1973).

The search of an AND/OR graph using backward reasoning raises numerous problems. Previous articles (B2 and C2) have considered

- (a) what constitutes a solution subgraph of an AND/OR graph, and
- (b) blind search algorithms for finding a solution subgraph.

This article considers three additional problems:

- (c) What might one mean by an optimal solution subgraph?
- (d) How can *heuristic information* be brought to bear on the search for an optimal solution?
- (e) What limitations are there on AND/OR graphs and the associated search algorithms as general tools for problem solving?

#### The Definition of an Optimal Solution

A solution of an AND/OR graph is a subgraph demonstrating that the start node is solved. As in a state-space search, one may ask for a solution of minimal cost. The cost of a solution tree can be defined in either of two ways (Nilsson, 1971):

The *sum cost* of a solution tree is the sum of all arc costs in the tree.

The *max cost* of a solution tree is the sum of arc costs along the most expensive path from the root to a terminal node.

For example, if every arc in the solution tree has cost 1, then the sum cost is the number of arcs in the tree, and the max cost is the depth of the deepest node.

If the entire search space had been explored, then an optimal solution tree could be constructed and its cost measured as follows. Let  $c(n,m)$  be the cost of the arc from node  $n$  to a successor node  $m$ . Define a function  $h(n)$  by:

If  $n$  is a terminal node (a primitive problem), then  $h(n) = 0$ .

If  $n$  has OR successors, then  $h(n)$  is the minimum, over all its successors  $m$ , of  $c(n,m) + h(m)$ .

If  $n$  has AND successors and sum costs are used, then  $h(n)$  is the summation, over all successors  $m$ , of  $c(n,m) + h(m)$ .

If  $n$  has AND successors and max costs are used, then  $h(n)$  is the maximum, over all successors  $m$ , of  $c(n,m) + h(m)$ .

If  $n$  is a nonterminal node with no successors, then  $h(n)$  is infinite.

According to this definition,  $h(n)$  is finite if and only if the problem represented by node  $n$  is solvable. For each solvable node  $n$ ,  $h(n)$  gives the cost of an optimal solution tree for the problem represented by node  $n$ . If  $s$  is the start node, then  $h(s)$  is the cost of an optimal solution to the initial problem.

Consider, for example, the AND/OR tree of Figure 1, with arc costs as indicated. Each node without successors is marked  $t$  or  $u$  according to whether it is terminal or unsolvable.

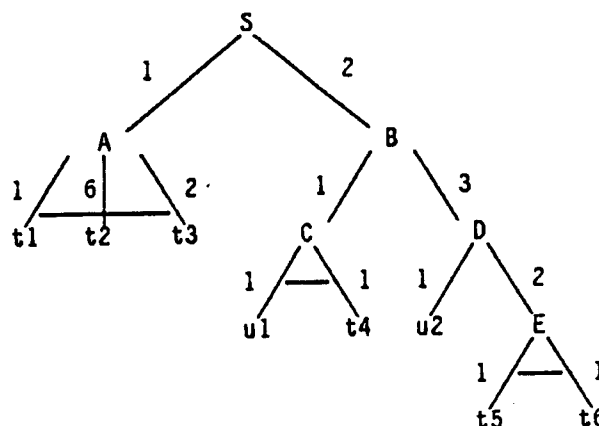


Figure 1. An AND/OR tree.



If *sum costs* are used, the values of  $h$  are as shown in Figure 2, and the optimal solution is the subgraph comprising nodes S, B, D, E, t5, and t6. The abbreviation *inf* denotes infinity.

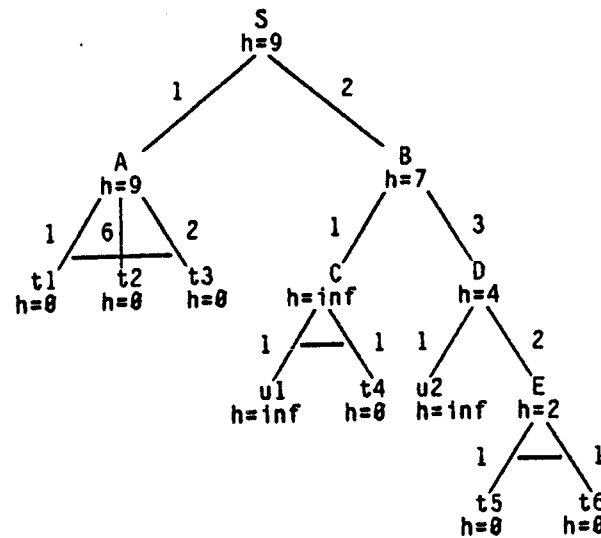


Figure 2. Sum costs.

If *max costs* are used, then the values of  $h$  are as shown in Figure 3, and the optimal solution is the subgraph comprising nodes S, A, t1, t2, and t3.

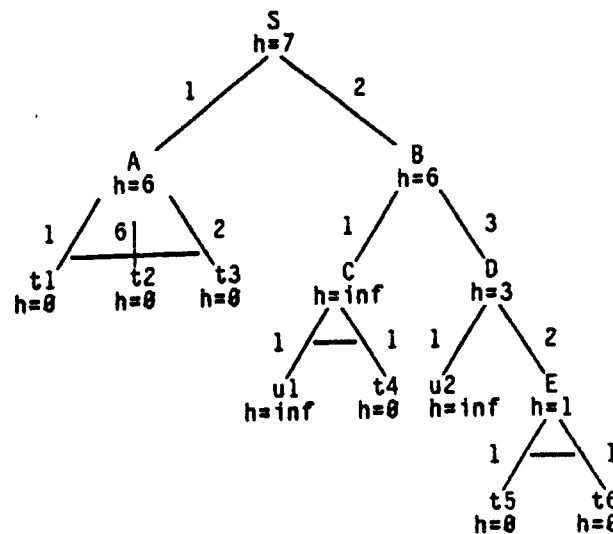


Figure 3. Max costs.

### Ordered Search Algorithms for an AND/OR Graph

In an ordered state-space search, one may use an *evaluation function*  $f^*$  that, applied to node  $n$ , returns the estimated minimum cost of a solution path passing through node  $n$ . The next node expanded is always one at which  $f^*$  is minimum--that is, one extends the most promising potential solution path. The successors of node  $n$  are new nodes, but one could just as well think of them as new potential solution paths, each differing from a parent (potential solution path) by the inclusion of one more step.

In the extension of heuristic search to AND/OR graphs, there is no longer a one-to-one correspondence between the choice of a node to expand and the choice of a potential solution to be extended. Consider, for example, the search graph of Figure 4.

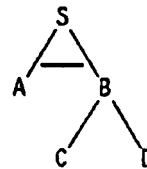
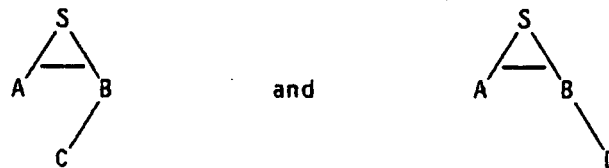


Figure 4. An AND/OR graph containing two potential solution trees.

Since C and D are OR nodes, an actual solution of node S will contain only one of them. To expand node A is thus to extend two *potential solution trees*,



Conversely, a decision to extend the potential solution tree on the left can be carried out by expanding either node A or node C. One must be clear, therefore, about what kind of object the expansion process is to apply to. This decision will affect the definition of the evaluation function.

**Nilsson's algorithm.** An approach taken by Nilsson (1969, 1971) selects individual nodes to expand by a two-step process: First, identify the most promising potential solution tree; then choose a node within that tree for expansion. To accomplish the first step, an evaluation function  $h^*$  is defined at every node  $n$  of the tree that has not been shown to be unsolvable. This function is an estimate of  $h(n)$ ; that is, it estimates the cost of an optimal solution to the problem at node  $n$ . If  $n$  is known to be a terminal node, then by definition  $h^*(n) = h(n) = 0$ . Otherwise, if  $n$  has not yet been expanded, then the estimate must be based on whatever heuristic information is available from the problem domain. For example, in the search tree of Figure 4,  $h^*$  would provide heuristic estimates of the cost of solving nodes A, C, and D. The following rule then permits  $h^*$  to be computed for each node whose successors have already been generated (and to be recomputed as the search tree is expanded):

If  $n$  has OR successors  $m$ , then  $h^*(n)$  is the minimum, over these successors, of  $c(n,m) + h^*(m)$ .

If  $n$  has AND successors  $m$  and sum costs are used, then  $h^*(n)$  is the summation, over these successors, of  $c(n,m) + h^*(m)$ .

If  $n$  has AND successors  $m$  and max costs are used, then  $h^*(n)$  is the maximum, over these successors, of  $c(n,m) + h^*(m)$ .

Finally, the most promising potential solution tree,  $T$ , is defined in terms of  $h^*$ :

The start node  $s$  is in  $T$ .

If the search tree (the part of the search space generated so far) contains a node  $n$  and AND successors of  $n$ , then all these successors are in  $T$ .

If the search tree contains a node  $n$  and OR successors  $m$  of  $n$ , then one successor  $m$  is in  $T$  such that  $c(n,m) + h^*(m)$  is minimal.

The estimated cost of  $T$  is  $h^*(s)$ . If all the other potential solution trees for the same search tree were constructed, it would be found that  $T$  is one for which  $h^*(s)$  is minimal.

An ordered-search algorithm for an AND/OR tree can now be stated as follows:

- (1) Put the start node,  $s$ , on a list, OPEN, of unexpanded nodes.
- (2) From the search tree constructed so far (initially, just  $s$ ), compute the most promising potential solution tree  $T$ .
- (3) Select a node  $n$  that is on OPEN and in  $T$ . Remove node  $n$  from OPEN and place it on a list called CLOSED.
- (4) If  $n$  is a terminal node, then
  - (a) Label node  $n$  solved.
  - (b) If the solution of  $n$  makes any of its ancestors solved, label these ancestors solved.
  - (c) If the start node is solved, exit with  $T$  as the solution tree.
  - (d) Remove from OPEN any nodes with a solved ancestor.
- (5) Otherwise, if node  $n$  has no successors (i.e., if no operator can be applied), then
  - (a) Label node  $n$  unsolvable.
  - (b) If the unsolvability of  $n$  makes any of its ancestors unsolvable, label all such ancestors unsolvable as well.
  - (c) If the start node is labeled unsolvable, exit with failure.
  - (d) Remove from OPEN any nodes with an unsolvable ancestor.
- (6) Otherwise, expand node  $n$ , generating all its immediate successors and, for each successor  $m$  representing a set of more than one subproblem, generating successors of  $m$  corresponding to the individual subproblems. Attach, to each newly generated node, a pointer back to its immediate predecessor, and compute  $h^*$  for each newly generated node. Place all the new nodes that do not yet have descendants on OPEN. Finally, recompute  $h^*(n)$  and  $h^*$  at each ancestor of  $n$ .
- (7) Go to (2).

The ordered-search algorithm can be shown to be *admissible*--that is, it will find a minimum-cost solution tree if any solution exists--provided that: (a)  $h^*(n)$  is less than or

equal to  $h(n)$  for each open node  $n$ , and (b) all arc costs are greater than some small positive number  $d$ . The efficiency of the algorithm, however, depends both on the accuracy of  $h^*$  and on the implementation of step 3, in which, having found the most promising potential solution tree to expand, one must decide to expand a specific node within that tree. If the partial tree  $T$  is in fact part of an optimum solution, the choice is immaterial. If it is not, however, then the best node to expand would be the one that will earliest reveal the error.

**Chang and Slagle's algorithm.** A different approach has been taken by Chang and Slagle (1971). Here the objects expanded are potential solution graphs. A *tip node* in such a graph is any node that does not yet have successors. To expand the potential solution graph, one expands all its nonterminal tip nodes at once and then forms all the new potential solution graphs that result. Each graph is represented on the OPEN list by the conjunction of its tip nodes, representing a set of subproblems to which the start node can be reduced.

For example, suppose that expansion of the initial graph, consisting of only the start node  $S$ , shows that  $S$  can be reduced to problems  $A$  and  $B$  or to problem  $C$ . The OPEN list then becomes  $(A\&B, C)$ . Assume that  $A\&B$  is selected for expansion, that  $A$  can be reduced to  $D$  or  $E$ , and that  $B$  can be reduced to  $F$  or  $G$ . There are four new potential solution trees, and the OPEN list is now  $(D\&F, D\&G, E\&F, E\&G, C)$ . The search succeeds when it selects for expansion a potential solution graph represented by a conjunction of nodes all of which are terminal.

The Chang and Slagle approach assimilates AND/OR graph search to the problem of state-space search. Each distinct conjunction of problems to be solved corresponds to a distinct state of a state-space graph. The evaluation function used,  $f^*$ , is also parallel to the function used in  $A^*$ : It is defined by  $f^* = g^* + h^*$ , where  $g^*$  measures the cheapest way found so far to reduce the start node to a given conjunction of subproblems and  $h^*$  estimates the minimum remaining cost of a graph sufficient to solve all those subproblems.

The treatment of AND/OR graph search as an instance of state-space search has several consequences. One is that the search of a general AND/OR graph, as opposed to an AND/OR tree, now raises no special problems. Another is that the algorithm can be shown (Chang & Slagle, 1971), under appropriate conditions, to be not only admissible but also optimal with respect to the number of potential solution graphs expanded. It does not, however, appear to be optimal (in some reasonable sense of that term) in comparison with algorithms that expand only one node at a time (see Kowalski, 1972).

### Interdependent Subproblems

The discussion so far has assumed that whenever the start node is reduced to a conjunction of subproblems, all subproblems can be solved independently, so that the solution to one has no effect on the solution to any other. This assumption is frequently unjustified, and much of the chapter on Planning explores ways of dealing with interacting subproblems. Two kinds of examples, given by Levi and Sirovich (1975, 1976) with explicit reference to the AND/OR graph formalism, are: (a) problems requiring consistent binding of variables and (b) problems involving the expenditure of scarce resources.

An illustration of the former is the well-known problem of showing that there exists a fallible Greek, given that the entire search space is as follows:

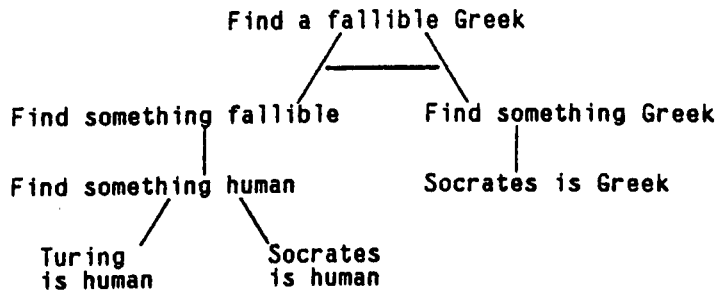


Figure 5. An AND/OR graph requiring consistent binding of the variable "something."

An algorithm like Nilsson's fails here for two reasons. First, it has no mechanism for discovering that "Turing is human" and "Socrates is Greek" fail to constitute a solution. Second, even if such a mechanism were introduced, the algorithm has no means for undoing the solution to a subproblem once it has been solved. If "Turing is human" is the first problem found to be primitive, then "Find something human" and "Find something fallible" are marked solved; "Socrates is human" is removed from the OPEN list as no longer in need of consideration; and "Find something Greek," using the previous value of "something," then becomes unsolvable.

An example of the second type of problem is the following: Show that John can seduce the actress, given that seducing the actress can be reduced to getting a car and getting a yacht; and that John has \$5000, a car costs \$5000, and a yacht costs \$5000. Here either of the algorithms given above would wrongly conclude that John can seduce the actress. A variant of the scarce resource problem arises in robot planning tasks (such as those performed by STRIPS, Article 05), where application of an operator representing a robot action solving one subproblem may make inapplicable the operator needed to solve another subproblem.

To handle problems of these kinds, Levi and Sirovich define a *generalized AND/OR graph*, which differs most importantly from an ordinary AND/OR graph in that reduction operators are permitted to take two or more nodes as input. For example, let R be a resource that can be used only once. Then if, in the standard formulation, the original problem is to accomplish P1 and P2, the problem is reformulated as P1 & P2 & R. Suppose the following reduction operators are available (where  $\rightarrow$  means "can be reduced to" and T denotes a trivial problem):

- 1)  $S \rightarrow P1 \ \& \ P2 \ \& \ R$
- 2)  $P1 \ \& \ R \rightarrow T$
- 3)  $P1 \rightarrow P3$
- 4)  $P2 \ \& \ R \rightarrow P3$
- 5)  $P3 \rightarrow T$
- 6)  $R \rightarrow T$

Then there is only one solution, which is achieved using operators 1, 3, 4, and 5.

In the ordered search of a generalized AND/OR graph, the objects placed on the OPEN list are potential solution graphs, not individual nodes. Expansion of a potential solution graph (PSG) consists of applying all possible operators to obtain a new set of PSGs, each differing from its parent by virtue of one additional operator application. If the same subproblem occurs more than once within a PSG, each occurrence is represented by a separate node. If the same PSG is generated more than once, later occurrences are simply discarded. Since distinct PSGs are retained, alternate solutions to the same subproblem are available.

As in the usual ordered search, the object chosen for expansion next is always one where the evaluation function is minimum. The evaluation function is  $h^*$ ; for each PSG, it is computed similarly to the  $h^*$  of Nilsson's algorithm. The value of each potential solution graph is then the evaluation of the start node,  $h^*(s)$ , as computed for that graph. Both *admissibility* and *optimality*--the latter with respect to the number of PSGs expanded--can be shown.

#### References

See Chang & Slagle (1971), Kowalski (1972), Levi & Sirovich (1975), Levi & Sirovich (1976), Martelli & Montanari (1973), Nilsson (1969), and Nilsson (1971).

## C5. Game Tree Search

## C5a. Minimax Procedure

## The Minimax Formalism

The minimax procedure is a technique for searching game trees (Article B3). As a first example, Figure 1 gives a simple game tree to which the procedure may be applied. Each node represents a position in the game. Nonterminal nodes are labeled with the name of the player, A or B, who is to move from that position. It is A's turn, and the problem is to find his best move from position 1. Exactly three moves remain in the game. Terminal nodes are marked with their value to player A by the words "win," "lose," or "draw."

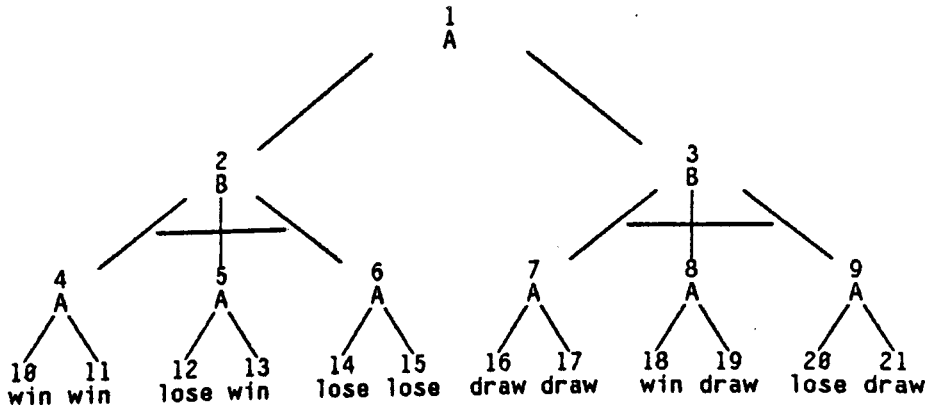


Figure 1. A game tree from the standpoint of player A, who is to move next.

According to the minimax technique, player A should move to whichever one of positions 2 or 3 has the greater value to him. Given the values of the terminal positions, the value of a nonterminal position is computed, by backing up from the terminals, as follows:

The value to player A of a node with OR successors (a node from which A chooses the next move) is the maximum value of any of its successors.

(1)

The value to A of a node with AND successors (a node from which B chooses the next move) is the minimum value of any of its successors.

In the example, node 2 evaluates to a loss for A (since B can then force a loss by moving to node 6), and node 3 evaluates to a draw (since the best B can then do is move to node 7 or 9). It will be noted that the prediction of the opponent's behavior assumes he is also using minimax: In evaluating a node with AND successors, A must assume that B will make his best possible move. The technique ignores the possibility that B might overlook his chance for a

sure win if A goes to node 2. Similarly, it supplies no basis on which B might choose to move to node 9 in preference to node 7.

Because of the way in which nodes are evaluated, player A (whose viewpoint the tree represents) is often called MAX, and player B, MIN. The names PLUS and MINUS are also sometimes used. If the tree of Figure 1 were to be evaluated from MIN's standpoint instead of MAX's, it would appear as in Figure 2. The AND and OR nodes are reversed, and the value of each node to MIN is the opposite of its value to MAX.

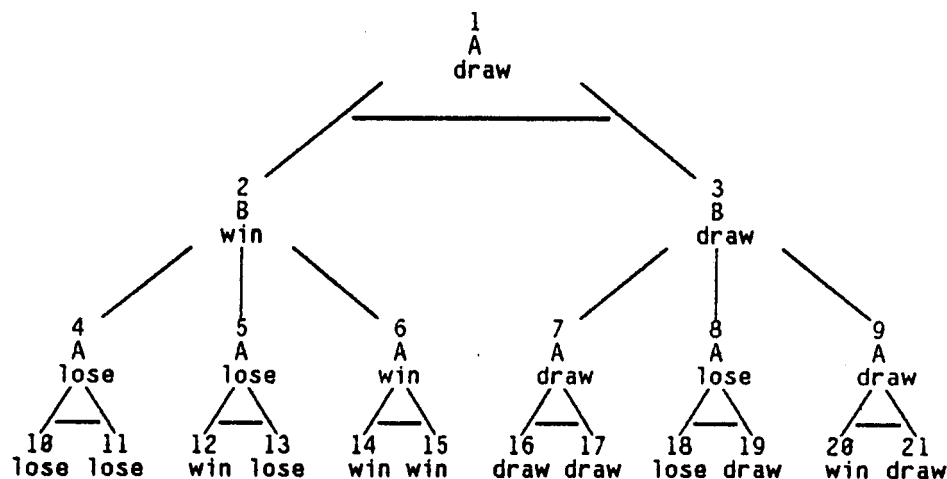


Figure 2. The game tree of Figure 1 from B's standpoint.

### The Negmax Formalism

Knuth and Moore (1975) have given a game-tree representation that unifies Figures 1 and 2 and conveniently permits a single procedure to return optimal moves for both players A and B. In this representation, the value given each node is its value to the player whose turn it would be to move at that node. If  $n$  is a terminal node, its value is an integer denoted  $f(n)$ . (The value of  $n$  to the other player is  $-f(n)$ .) The value of every node is then returned by a function  $F$  defined as follows:

$$F(n) = f(n), \text{ if } n \text{ has no successors;} \\ F(n) = \max \{-F(n_1), \dots, -F(n_k)\}, \text{ if } n \text{ has successors } n_1, \dots, n_k.$$

The best move for either player is then to a node with maximum value; that is, the player whose turn it is at node  $n$  should move from node  $n$  to a node  $n_i$  with  $-F(n_i) = F(n)$ . This formulation, which is equivalent to minimax, is called *negmax*. The tree it produces for the game of Figures 1 and 2 is shown in Figure 3. The numerical value of a win is assumed to be +1; of a loss, -1; and of a draw, 0.



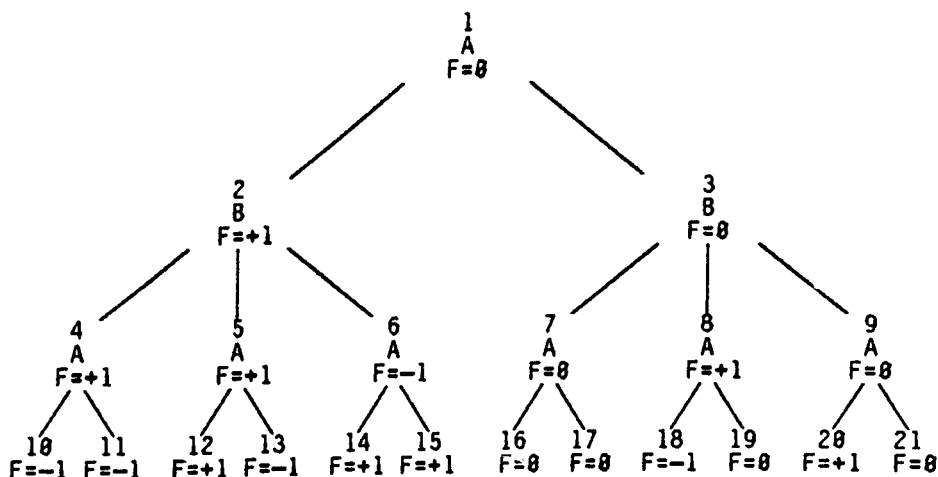


Figure 3. The game tree of Figure 1 in NEGMAX notation.

### Searching a Partial Game Tree

In the above descriptions of the minimax and negmax algorithms, it was assumed that a complete game tree had already been generated. For most games, however, the tree of possibilities is far too large to be generated fully and searched backward from the terminal nodes for an optimal move. An alternative is to generate a reasonable portion of the tree, starting from the current position; make a move on the basis of this partial knowledge; let the opponent reply; and then repeat the process beginning from the new position. A "reasonable portion of the tree" might be taken to mean all legal moves within a fixed limit of depth, time, or storage, or it might be refined in various ways. For discussion of the refinements, see article C5c.

Once the partial tree exists, minimaxing requires a means for estimating the value of its *tip nodes*, that is, the nodes of the partial tree without successors. A function assigning such a value is called a *static evaluation function*; it serves a purpose comparable to that of the heuristic function  $h^*$  used in Nilsson's ordered search of an AND/OR tree (Article C4). If the partial game tree contains any nodes that are terminal for the entire tree, the static evaluation function conventionally returns positive infinity for a win, negative infinity for a loss, and zero for a draw. At other tip nodes, the function has a finite value which, in the minimax formulation, is positive for positions favorable to MAX and negative at positions favorable to MIN. The minimax procedure then assigns *backed-up values* to the ancestors of the tip nodes in accordance with the rules given in (1) above. It is assumed that the backed-up evaluations give a more accurate estimate of the true value of MAX's possible moves than would be obtained by applying the static evaluation function directly to those moves and not looking ahead to their consequences.

### References

See Knuth & Moore (1975), Nilsson (1971), Slagle (1971), and Winston (1977).

## C5b. Alpha-beta Pruning

The minimax procedure described in Article C5a decides on a best move from node  $n$ , in a full or partial game tree, by evaluating every node in the tree that descends from node  $n$ . Frequently, this exhaustive evaluation is a waste of time. Two examples are shown in Figures 1 and 2. Each node is marked with the name of the player who is to move from that position.

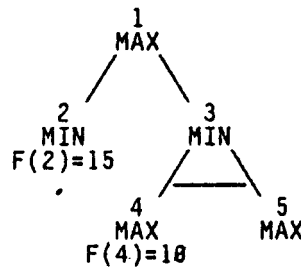


Figure 1. An alpha cutoff.

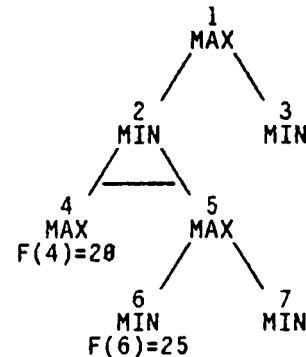


Figure 2. A beta cutoff.

In Figure 1, nodes 2 and 4 have been evaluated either by the static evaluation function or by backing up from descendants omitted from the figure. If MAX moves to node 2, he achieves a position whose estimated value is 15. If he moves to node 3, MIN can hold him to 10. Therefore, the value of node 3 is at most 10, so MAX should decide to move to node 2. The important point is that this decision can be made without evaluating node 5 or any of its possible descendants.

In Figure 2, node 4 has an estimated value to MAX of 20. When node 6 is evaluated at 25, it becomes clear that MIN should avoid moving to node 5. Node 2 can therefore be assigned a value of 20 without any need to evaluate node 7 or any of its descendants.

The *alpha-beta technique* for evaluating nodes of a game tree eliminates these unnecessary evaluations. If, as is usual, the generation of nodes is interleaved with their evaluation, then nodes such as the descendants of node 5 in Figure 1 and of node 7 in Figure 2 need never even be generated. The technique uses two parameters, alpha and beta. In Figure 1, the parameter alpha carries the lower bound of 15 on MAX's achievement from node 1; the elimination of node 5 is an *alpha cutoff*. In Figure 2, the parameter beta is set to 20 at node 4, representing an upper bound on the value to MAX of node 2; the elimination of node 7 is a *beta cutoff*. The procedure guarantees that the root node of the tree will have the same final value as if exhaustive minimaxing were employed.

A concise statement of the alpha-beta procedure has been given by Knuth and Moore (1975). It uses their *negmax* representation in which both players are treated as wishing to maximize (see Article C5a). Figure 3 shows how Figures 1 and 2 are transformed in the *negmax* representation.

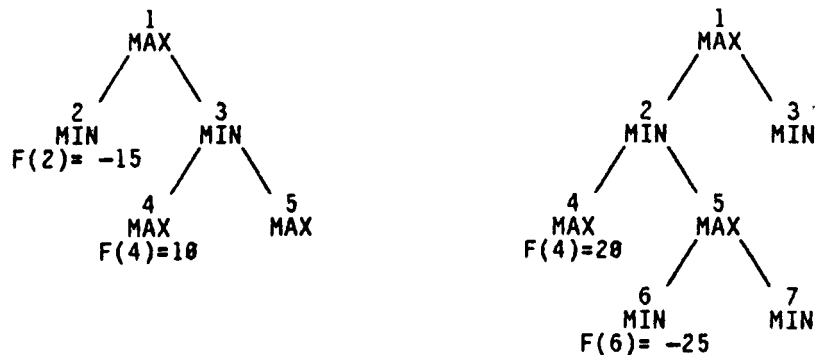


Figure 3. The NEGMAX representation of Figures 1 and 2.

To evaluate node 1 of either tree, the procedure is called with the parameters POSITION = node 1, ALPHA = negative infinity, and BETA = positive infinity. The static evaluation function is called  $f$ . The procedure, here called VALUE, is as follows:

```

INTEGER PROCEDURE value(POSITION p, INTEGER alpha, INTEGER beta)
BEGIN
  INTEGER m, i, t, d
  determine the successor positions  $p_1, p_2, \dots, p_d$ 
  of position p;
  IF d = 0 THEN value :=  $f(p)$  ELSE
  BEGIN
    m := alpha;
    FOR i := 1 STEP 1 UNTIL d DO
      BEGIN t := -value( $p_i, -beta, -m$ );
        IF t > m THEN m := t;
        IF m > beta or m = beta THEN GO TO done;
      END;
    done: value := m;
  END;
END;

```

For an intuitively developed LISP version of the alpha-beta procedure, see Winston (1977). An excellent review of the historical development of the technique appears in Knuth and Moore (1975).

### Ordering of Successors

The degree to which the alpha-beta procedure represents an improvement in efficiency over straight minimaxing varies with the order in which successor nodes are evaluated. For example, no cutoff would occur in Figure 1 if node 3 were considered before node 2.

In general, it is desirable that the best successor of each node be the first one evaluated--that is, that the first move MAX considers be his best move, and that the first

reply considered for MIN be the move that is best for MIN and worst for MAX. Several schemes for ordering the successors of a node have been described to try to achieve this state of affairs. One possibility, an example of *fixed ordering*, is to apply the static evaluation function to the successors, taking the results of this preliminary evaluation as an approximation of their expected backed-up values. A method of this sort will result in depth-first generation and evaluation of the partial game tree, subject to the depth bound or other criteria for terminating generation. For some other possibilities, see Article C5c.

### Efficiency in Uniform Game Trees

Since the alpha-beta procedure is more complicated than minimaxing, although it yields the same result, one may inquire how great an increase it produces in search efficiency. Most theoretical results on this question deal with *uniform* game trees: A tree is said to be uniform if every tip node has depth  $d$  and every nontip node has exactly  $b$  successors. Here  $b$  is called the *branching factor* or *degree* of the tree.

The results reviewed below come from Knuth and Moore (1975) and, for the best case, Slagle and Dixon (1969). For other related work, see Fuller et al. (1973), Newborn (1977), and Baudet (1978).

**The best case.** A uniform game tree of depth  $d$  and degree  $b$  contains exactly  $b^d$  tip nodes, all of which must be examined by minimax. In the worst case, alpha-beta also must examine every tip node. In the best case, alpha-beta examines only about twice the square root of the number of tip nodes. More precisely, assuming the value of the root is not infinite, the number of tip nodes examined in the best case is

$$b^{\lceil (d+1)/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$$

(where square brackets represent the greatest integer function); and the nodes examined in the tree as a whole are precisely the *critical nodes*, defined as follows:

Type 1 critical nodes are the root node and all first successors of type 1 nodes.

Type 2 critical nodes are all further successors (except the first) of type 1 nodes and all successors of type 3 nodes.

Type 3 critical nodes are the first successors of type 2 nodes.

Figure 4 illustrates the distribution of critical nodes in a uniform tree of degree 3 and depth 3.

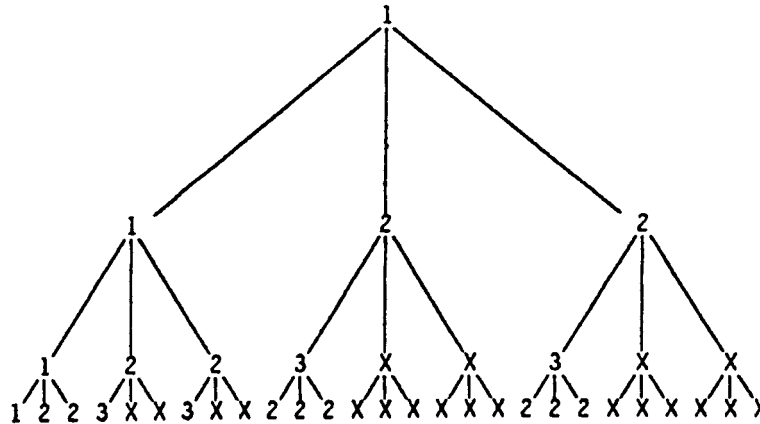


Figure 4. Distribution of critical nodes.

Knuth and Moore have shown that the best case occurs for a uniform tree if the best move is considered first at each critical node of types 1 and 2. Attempts to order the successors of type 3 positions contribute nothing to efficiency, since these successors are type 2 nodes, which must all be examined anyway.

**Random uniform game trees.** Knuth and Moore also show that the alpha-beta technique is optimal in the sense that no algorithm can evaluate any game tree by examining fewer nodes than alpha-beta does with an appropriate ordering of successors. Realistically, of course, one cannot expect to achieve the optimal successor ordering, since this would imply full knowledge of the game tree before it is generated. Assuming, therefore, that the tip nodes of the tree have distinct random values, Knuth and Moore show that the expected number of tip nodes examined, in evaluation of a uniform tree with branching factor  $b$  and depth  $d$ , has an asymptotic upper bound of

$$(b/(\log b))^d$$

as  $d$  goes to infinity.

**Totally dependent uniform game trees.** One other type of tree considered by Knuth and Moore, perhaps more realistic than the one in which tip nodes have random values, corresponds to games in which each move is critical: If a poor move is ever chosen, there is no way to recoup. The model is a uniform game tree that is *totally dependent*: For any two successors of node  $p$ , these successors can be labeled  $q$  and  $r$  so that every tip node descended from node  $q$  has greater value than any tip node descended from node  $r$ . In this type of tree, if the degree is at least 3, the expected number of tip positions examined is bounded by a constant (depending on the degree) multiplied by the number of tip nodes examined by the alpha-beta method in the best case.

#### References

See Baudet (1978), Fuller, Gaschnig & Gillogly (1973), Knuth & Moore (1975), Newborn (1977), Nilsson (1971), Slagle & Dixon (1969), Slagle (1971), and Winston (1977).

### C5c. Heuristics in Game Tree Search

In the search of a game tree (Article B3), as in other kinds of search, there are various points at which heuristic information may be applied. The parallel is not exact, however. In one-person problem solving, the main uses for heuristic information are to decide which node to expand next, which operator to apply next, and, in some algorithms, which nodes to prune from the search tree. (See Article C3a.) In game-playing programs, these questions also exist, but with a shift in emphasis. In addition, some new questions arise: When should the search be terminated? How should a move be chosen on the basis of the search that has been made?

The simplest answers to these questions were described in Article C5a: Expand every node completely, in any convenient order and with no pruning, until every tip node represents a termination of the game. Then, working back from the end of the game, use the minimax procedure to find a winning line of play (if one exists), and follow this line of play throughout the game. Article C5b, Alpha-beta Pruning, described an improvement on this approach that yields the same final result with greater efficiency.

A program using only these basic techniques would play a theoretically perfect game; its task would be like searching an AND/OR tree for a solution to a one-person problem. For a simple game like tic-tac-toe (see Article B3), such a program would no doubt be feasible. For complex games, however, it has been recognized from the beginning that searching from the start of the game to its end would be impossible. In chess, for example, with around 30 legal moves from each position and about 40 moves for each player in a typical game, there are some  $(30^2)^{40}$  or  $10^{120}$  different plays of the game (Shannon, 1950).

Because of the magnitude of the search space in chess, checkers, and other nontrivial games, there is a major difference between programs that play such games and programs that use the methods of this chapter to solve nonadversary problems. The latter either find a solution or fail, having run out of time or space; much of the research assumes that some solution can be found and deals with how to guarantee that it is optimal or nearly optimal (see Section C3, Heuristic State-space Search). The question for a chess program, in contrast, is how to play a good game even though it has not found a solution to the problem of winning. Repeatedly the program must become committed to its next move long before the end of the game comes into view. Whether the move chosen is in fact part of a winning strategy is unknown until later in the game.

For a nontrivial game playing program, then, the issues listed at the beginning of this article are all aspects of a broader question: Can the basic search techniques, designed for seeking a guaranteed win, be successfully adapted to the problem of simply choosing the next move? In addition, one might well ask whether there are alternatives to search as the basis for move selection. Most of the work exploring these questions has been done in the specific domain of chess. In general, the discussion below is limited to chess programs and Samuel's checkers program (1963, 1967).

### Alternatives to Search

An example of choosing a move on a basis other than search is the use of "book moves" in the opening of a chess game (see Frey, 1977, pp. 77-79). More generally, there

is an emphasis in the recent computer chess literature on treating the problem of move choice as a problem of recognizing patterns on the board and associating appropriate playing methods with each pattern (e.g., Charness, 1977, p. 52; Bratko et al., 1978; Wilkins, 1979).

It is not expected, however, that search can be eliminated entirely from chess programs; even human players do some searching. Rather, the choice-of-move problem is seen as involving a tradeoff between the amount of specialized chess knowledge a program has and the amount of search it needs to do. (See, e.g., Berliner 1977c; Michie, 1977.) Another there are limits on the amount of knowledge a program can be given: The combinatorics of chess preclude storing an exhaustive representation of the game; and even the knowledge possessed by chess masters, which greatly restricts search in human play, also remains very far from complete formalization.

The last section of this article reviews several programs that attempt to use human-like knowledge to eliminate most searching. The sections preceding it concern techniques used in programs in which search rather than knowledge is predominant.

### Search-based Programs

The most successful game-playing programs so far have made search rather than knowledge their main ingredient. These include, among the earlier programs, Samuel's checkers program (1963, 1967), which came close to expert play; and Greenblatt's chess program (1967), which was the first to compete in tournaments and which earned a rating of 1400-1450, making it a Class C player. (Current classes of the United States Chess Federation are E through A, Expert, Master, and Senior Master. See Hearst, 1977, p. 171.) Notable later programs include the Soviet program KAISSA (Adelson-Velskiy et al., 1975), which won the first world computer chess championship in 1974, and Slate and Atkin's CHESS 4.5 (1977), whose current standing is mentioned below. (For general reviews of computer chess competition, see Berliner, 1978a; Mittman, 1977; and Newborn, 1975.)

All the programs referred to above follow the basic search paradigm formulated by Shannon in 1950. In its simplest form, which was called a Type A program, Shannon's paradigm made just two changes to the procedure mentioned above that calls for searching exhaustively all the way to the end of the game. First, the game tree was to be generated only to a fixed depth. Second, since the nodes at the depth limit would normally be nonterminal, a means of estimating the promise of these nodes was required. The estimate was to be given by a *static evaluation function*, whose values could then be backed up by minimaxing to determine the next move. After this move was made and the opponent had replied, the search process would be repeated beginning from the new position.

Shannon noted that a simple Type A program would play chess both badly and slowly. He suggested two directions for improvement in a Type A program, with which the program would become Type B. The general objectives were, first, to let the exploration of a line of play continue to a reasonable stopping point instead of invariably cutting it off at an arbitrary depth; and, second, to provide some selectivity about the lines of play considered, so that more time could be spent investigating strong moves and less on pointless ones.

Even a Type B program, Shannon concluded, seemed to rely too much on brute-force calculation rather than on knowledgeable analysis of the situation to choose a move.

Nevertheless, his proposals established a framework that most competitive game-playing programs have adopted. The framework raises a large number of interrelated issues, which are discussed in the following sections.

### Static Evaluation

A *static evaluation function*, by definition, is one that estimates the value of a board position without looking at any of that position's successors. An ideal function would be one that reports whether the position leads to a win, a loss, or a draw (provided neither side makes a mistake). Even more informatively, the function might report the number of moves required to win, with an arbitrarily large value if no win is possible. But functions that can distinguish between winning and losing positions are known only for simple games; an example of such a function for the game Nim is given in Shannon (1950).

Where perfect evaluation functions are unavailable, the actual static evaluator must return an estimate. Unlike the *evaluation function* used in an ordinary state-space or AND/OR graph search (C3a, C4), the static evaluation function of a game-playing program does not normally attempt directly to estimate the distance to a win from the position evaluated. (For a proposal that the function should do just this, see Harris, 1974.) Instead, the function is usually a linear polynomial whose terms represent various features of the position, high values being given for features favorable to the program and low ones for those favoring the opponent. In chess, the most important feature is material, the relative value of each side's pieces on the board. Other typical features, familiar to chess players, include king safety, mobility, center control, and pawn structure.

The most extended treatment of evaluation functions in the literature is provided by Samuel (1963, 1967). For checkers, he concluded (1967, p. 611) that the optimal number of features to be used in the evaluation function was between twenty and thirty. Samuel's main interest was in machine learning; one approach he took was to provide his checkers program with a large set of features for possible use in the evaluation function and to let the program determine, as it gained playing experience, both which of these features should be included and what their relative weights should be. In a later version of the program, the emphasis was shifted to taking the interactions among features into account in evaluating positions. With this change, the evaluation function became nonlinear, and considerable improvement was reported in its quality as measured by the correlation with moves chosen in master play (Samuel, 1967; see also Griffith, 1974). For further discussion of Samuel's work, see Learning.

Reasonably accurate static evaluation, then, requires a rather complex function. But there is an important limit on the complexity that is feasible, especially for a program that plays in tournaments, under time limitations. As the total number of tip nodes in the search tree increases, the time available for evaluating any single tip node goes down. Thus Gillogly's chess program TECH (1972), which was intended as an experiment in how much could be accomplished on advanced machines by simple brute force search, and which generates up to 500,000 tip nodes even with alpha-beta pruning, uses material as the only factor in its static evaluations.



### Backed-up Evaluation

The Shannon paradigm assumes that the step between static evaluation and the choice of a move is simply minimaxing: The program moves to any position with the best backed-up minimax value. This step is indeed very commonly used. But it is worth noting that, since the static evaluation function may be wrong, the minimax procedure no longer serves its original purpose of defining and identifying a move that is theoretically correct. Instead, minimaxing has itself become a heuristic for the choice of move. Several programs have therefore experimented with varying or supplementing the minimax procedure. Slagle and Dixon (1970), for example, in experiments with the game of kalah, compute the backed-up value of a node by taking into account not only the value of its best successor but also whether the node has several good successors or just one. Gillogly's TECH (1972), having computed minimax values on the basis of an extremely simple static evaluation, breaks ties between moves with equal minimax values by an analysis of features not considered by the evaluation function. Newell, Shaw, and Simon (1963a) set a value in advance that the search is expected to achieve; the first move found that meets this standard is made, and only if no move is good enough is the best minimax value used to determine the move (see also Newell & Simon, 1972).

### Depth of Search

If perfect evaluation functions were available, a game-playing program could proceed at each turn by generating all legal moves, evaluating each of the resulting positions, and choosing the move leading to the best value. The reason for looking farther ahead is to compensate for errors in the static evaluation. The assumption is that, since static evaluation has a predictive aspect, there will be less room for mistaken prediction if a deep tree is generated before the evaluation function is applied.

The controlling fact about search depth is the combinatorial explosion. If the average number of legal moves from a position, the *branching factor*, is  $b$ , the game tree will have about  $b^d$  nodes at depth  $d$ . According to Shannon's estimate for chess, a complete tree carried to depth 6--3 moves for each player--would already have about one billion tip nodes. At the same time, Shannon noted, a world champion may occasionally look ahead, along a single line of play, to a depth as great as 15 or 20. More recently Hans Berliner, a former World Correspondence Chess Champion, has said he finds it necessary at least once in a game to look ahead to a depth of 14 or more (1974, p. 1-8). The question, then, is how to get the needed depth, in the right places, without succumbing to the combinatorial explosion. An alternative question would be how to avoid the need for so deep a search. The remainder of this article concerns attempts to solve or at least alleviate these problems. First, however, experience with the use of depth bounds as such will be reviewed.

**Fixed-depth search with extensions for quiescence.** The simplest lookahead procedure, which was called for by Shannon's Type A strategy, is to set a fixed depth, or *ply*, to which the game tree is to be generated and to apply the static evaluation function only to nodes at this depth. Thus a 4-ply search would statically evaluate the positions reached after exactly two turns for each player. There are serious drawbacks in this procedure, as Shannon observed, and it was used only in very early programs (Kister et al., 1957; Bernstein et al., 1959). For example, a chess evaluation function based mainly on material cannot return a realistic value if at the depth limit the players happen to be halfway

through an exchange of pieces. The concept of a *quiescent* or *dead* position was introduced to get around such difficulties (Shannon, 1950; see also Turing, 1953): Search would be extended beyond the normal limit, from nonquiescent positions only, until all tip nodes were relatively stable or perhaps until some absolute depth-bound had been reached.

This introduction of a quiescence search was one of the two features that changed a program, in Shannon's terminology, from Type A to Type B. On Shannon's suggested definition, a position was considered nonquiescent if "any piece is attacked by a piece of lower value, or by more pieces than defences or if any check exists on a square controlled by opponent" (1950, p. 271). Many programs have adopted a similar definition, with the result that the only moves examined beyond the normal limit are checks and immediate captures (e.g., Gillogly, 1972; Adelson-Velskiy et al., 1975; Slate & Atkin, 1977). If such a quiescence search is combined with considering all legal moves down to the normal depth limit, the program is still called Type A in current terminology (e.g., Berliner, 1978a).

**The horizon effect.** Searching to an arbitrarily limited depth, even with extensions for checks and captures, creates a phenomenon that Berliner (1973, 1974) has called the *horizon effect*. Berliner's general observation is that, whenever search is terminated (short of the end of the game) and a static evaluation function is applied, the program's "reality exists in terms of the output of the static evaluation function, and anything that is not detectable at evaluation time does not exist as far as the program is concerned" (1974, p. 1-1).

Two kinds of errors ensue. The first is called the negative horizon effect: The program manipulates the timing of moves to force certain positions to appear at the search horizon, and it thus may conclude that it has avoided some undesirable effect when in fact the effect has only been delayed to a point beyond the horizon. A second kind of error, the positive horizon effect, involves reaching for a desirable consequence: Either the program wrongly concludes that the consequence is achievable, or it fails to realize that the same consequence could also be achieved later in the game in a more effective form. This last problem, Berliner believes, can be met only by finding ways to represent and use more chess knowledge than traditional programs have included (1974, p. 1-7).

For most of the errors coming from the horizon effect, however, the diagnosis is that the typical definitions of quiescence are highly oversimplified. Ideally a position would be considered quiescent only when the static evaluation function, applied to that position, could return a realistic value, that is, when the value of every term included in the function had become stable. A quiescence search that pursues only captures and checking moves, however, considers only changes in the material term. The material term itself, moreover, usually reflects only the presence of the pieces on the board; its value will be unchanged by a move that guarantees a capture later instead of making a capture now.

To get around the problems arising from inadequate quiescence analysis, a first approach called *secondary search* was developed by Greenblatt (1967): Whenever a move appeared, on the basis of the regular search (including quiescence), to be the best move considered so far, the predicted line of play was extended by searching another two ply (plus quiescence) to test the evaluation. Berliner points out, however: "The horizon effect cannot be dealt with adequately by merely shifting the horizon" (1974, p. 1-4). One direction in current work, therefore, looks toward a much fuller quiescence analysis as a substitute for arbitrary depth bounds. (See Harris, 1975, 1977; Slate & Atkin, 1977, pp. 115-117; and,

for an early example, Newell & Simon, 1972, pp. 678-698.) Berliner meanwhile is developing a general algorithm, not limited to chess, for causing tree search to terminate with a best move, even though no depth limit has been set and no full path to a win has been found (Berliner, 1977c, 1978b).

**Iterative deepening.** Despite its drawbacks, most current programs still use a fixed-depth search, extended for checks and capture sequences. A variation used by CHESS 4.5 (Slate & Atkin, 1977) is called *iterative deepening*: A complete search, investigating all legal moves (subject to alpha-beta pruning), is done to depth 2, returning a move. The search is then redone to depth 3, again to depth 4, and so on until a preset time limit is exceeded. For efficiency, information from earlier iterations is saved for use in later ones. Running on the very fast CDC Cyber 176, the program searches to an average depth of 6 plies in tournament play, with search trees averaging 500,000 nodes (Newborn, 1978). It is the first program to have achieved an Expert rating in human play. In the fall of 1978 a new version, CHESS 4.7, was reportedly rated 2160 (Levy, 1979); Master ratings begin at 2200. It remains an open question how much stronger the program can become.

### Ordering of Search

The Shannon paradigm did not specify any particular order in which the nodes of the search tree were to be explored or in which moves from a given node were to be considered. For efficient use of space, the order of node expansion is usually depth-first; a depth-first algorithm needs to store explicitly only those nodes on the path it is currently investigating and not the parts of the tree where search has been completed.

With the invention of alpha-beta pruning, the order of considering moves within a depth-first search became highly significant. If the order is ideal, then in a tree with branching factor  $b$  the number of nodes that must be examined at depth  $d$  is reduced from  $b^d$  to only about  $2b^{d/2}$ . (See Article C5b.) For example, Shannon's estimated  $10^9$  chess positions at depth 6 would be reduced to around 50,000. It also follows that, for a constant number of tip nodes examined, correct ordering of the moves for alpha-beta cutoffs would allow the search depth to be roughly doubled. In general, the desired ordering is one in which the first move considered at a position is the best move for the player whose turn it is. Usually, of course, there is no method guaranteed to achieve this ordering, for if there were, it would enable moves to be chosen with no search at all. Several heuristics have been used, however, to try to approximate optimal ordering.

Perhaps the simplest idea for move ordering is the *fixed-ordering* method mentioned in Article C5b: For each move from a node, generate a new node for the resulting position, apply the static evaluation function to the position, and order the nodes according to this preliminary estimate. For greater efficiency, several programs have used a separate function for move ordering, which applies to the move itself instead of to the position that results from it (Greenblatt, 1967; Berliner, 1974, p. 11-16; Adelson-Velskiy, 1975). In either case the game tree is explored by an *ordered depth-first search* (Article C3a).

A fuller basis for choosing which move to consider first is provided by Slate and Atkin's iterative deepening technique, which makes repeated depth-first searches. Each iteration constructs a line of play, down to its depth limit, consisting of apparently best moves. The following iteration, going one ply deeper, thus has available an estimated best move from each position along this line of play. (See Slate & Atkin, 1977, pp. 102-103.)

A further approach to move ordering makes explicit the idea of a *refutation move*: For each move that is not a best move, it should be shown as quickly as possible that the move is bad. To do this, strong replies should be considered first, which may refute the move proposed. Typical implementations consider all capturing moves first, and then consider *killer moves*. The idea here, called the *killer heuristic*, is that if a move has served as a refutation in some previously examined position that is similar to the current one, it is likely to be a refutation in the current position too. For more on the killer heuristic and other refutation techniques, see Slate and Atkin (1977), Adelson-Velskiy (1975), Gillogly (1972), and Frey (1977).

Once the moves have been ordered at a given node and the search has moved downward, following the move that seemed best, it may turn out that this move is actually a very bad one for reasons that were not apparent earlier. Since accurate move ordering is important to maximizing alpha-beta cutoffs, it might be worthwhile at this point to go back, reorder the moves, and start again with a different estimated best move. Such a procedure, called *dynamic ordering*, was investigated by Slagle and Dixon (1969), using the game of kalah. They reported a modest improvement over fixed ordering for trees of depth at least 6. On the other hand, Berliner's chess program experienced a serious increase in running time when dynamic ordering was used (1974, p. IV-14). A procedure somewhat similar to dynamic ordering was also used by Samuel (1967).

If dynamic ordering is carried to its limit, so that reordering is considered every time a node is expanded instead of only under more limited conditions, the search procedure in effect changes from depth-first to *best-first*. That is, the move considered next (or the position to which it leads) is on some estimate the most promising in the entire search tree generated so far, subject to whatever depth limit exists. Nilsson (1968, 1971) implements this idea by adapting his algorithm for best-first AND/OR tree search (C4) to game trees. Harris (1975, 1977) suggests another adaptation, in which the motivation of maximizing alpha-beta pruning no longer plays a role and instead the objective is to expand the most active positions first, using a thorough quiescence analysis rather than a depth limit as the criterion for search termination.

### Width of Search

The techniques discussed so far are consistent with the idea that all legal moves from a position must be examined, at least sufficiently to establish that they can be safely pruned by alpha-beta. This consideration of all legal moves is referred to as *full-width searching*. Some of the earliest programs used a full-width search for simplicity; strong current programs use it because of the great difficulty in determining, without search, which moves can be safely ignored (Turing, 1953; Klster et al., 1957; Gillogly, 1972; Adelson-Velskiy et al., 1975; Slate & Atkin, 1977). The problem, of course, is that an excellent move may look very poor at first sight.

Yet the average number of legal moves from a chess position is at least 30, and even with a maximum of alpha-beta pruning the tree grows exponentially. Making the search more selective was Shannon's second requirement to change a program from Type A to Type B. Many people have been convinced that such selectivity is essential to a strong chess program, both in order to increase search depth and to permit more sophisticated evaluation of the nodes remaining in the search tree. Berliner, for example, has advocated reducing the

total search tree size to at most 5000 nodes, with a branching factor of less than 1.9 (1974, p. 1-16). Although some reconsideration of these ideas has been prompted by the success of CHESS 4.7 using full-width search, it appears that that program is still weak at long endgame sequences (see Berliner, 1978a; Michie & Bratko, 1978). Moreover, there are other games for which it is even clearer that full-width search is not the answer. For the game of *go*, for example, the average branching factor has been estimated at perhaps 200 (Thorp & Walden, 1970), and for backgammon, where legal moves depend on the throw of the dice as well as the board position, the factor is over 800 (Berliner, 1977a).

Various devices have been tried in the effort to increase the selectivity of the search without missing good moves. Some are conceptually simple, introducing little or no new chess-specific knowledge into the program. Others attempt to formulate and use chess concepts as sophisticated as those a chess master might employ. The remainder of this section reviews mainly the earlier search-controlling devices. The following section mentions work, some of which moves outside the Shannon paradigm, in which the effort to capture expert chess knowledge becomes primary.

**Forward pruning.** One way of limiting the number of moves to be considered introduces no new complications: Simply generate all legal moves at a position, use a fixed-ordering scheme to sort them according to their apparent goodness, or *plausibility*, and then discard all but the best few moves. Such a technique, called *plausible-move generation* or *forward pruning*, was used by Kotok (1962) and Greenblatt (1967); see also Samuel (1967). A further feature of these programs, sometimes called *tapered forward pruning*, was that the number of moves retained was a function of the depth at which they were generated. For example, Greenblatt's program in tournament play retained 15 moves from a position at either of the top two levels of the tree, 9 moves at the next two levels, and 7 moves thereafter. These figures could be increased in special cases--for example, to be sure that moves of more than a single piece were considered.

Another form of forward pruning, distinct from plausible move generation, operates not at the time when moves are originally generated but later, when one of these moves (or the position to which it leads) is being selected for further exploration. At this point a preliminary estimate of the value of the move or position may already have been made by the move-ordering scheme. If this estimate is outside the limits alpha and beta, the currently known bounds on the outcome of the entire search (see C5b), the node is pruned without further investigation. It is possible, of course, that the actual backed-up value of the node would have turned out to be between alpha and beta. In that case a good move may have been missed. (See Samuel, 1967; Berliner, 1974, p. IV-13.)

Still another basis for forward pruning has been explored by Adelson-Velskiy et al. (1975). They observe that KAISSA's search trees include many lines of play that a human would consider absurd, not necessarily because the moves are bad a priori, but because the human player has already considered and rejected the same moves in an analogous position. The proposal, then, is to remember moves that have been found to be absurd (on some definition) and to reject them in other positions too unless there has been an appropriate change of circumstances. In effect, this *method of analogies* involves trying to establish conditions under which a refutation is guaranteed to be effective. Then the line of play constituting the refutation would not need to be explored separately every time it is applicable. (See Frey, 1977, p. 68.)

**Goal-directed move generation.** Returning to the initial generation of moves, there is another kind of plausible move generator that comes closer to mimicking the way that humans might decide which moves are worth considering. Instead of generating all legal moves and discarding some, this approach does not generate moves at all unless they seem relevant to some *goal*. The earliest step in this direction was Bernstein's program (1959), which contained a sequence of board features to be tested for and a procedure for generating moves in response to each feature that was present. The first few tests in the sequence were (1) is the king in check? (2) can material be gained, lost, or exchanged? and (3) is castling possible? A maximum of 7 plausible moves was returned. Questions later in the sequence were not asked if earlier questions caused the maximum to be reached. Searching to a fixed depth of 4 ply, the program generated trees with about 2400 tip nodes.

More explicitly goal-directed move generation was included in Newell, Shaw, and Simon's 1958 chess program (Newell, Shaw, & Simon, 1963a; Newell & Simon, 1972). Indeed, the entire program was organized in terms of goals, although only three--material, center control, and piece development--were actually implemented. At each turn, the program began by making a preliminary analysis to decide which of the goals were relevant to the situation; these were entered, in order of importance, on a current goal-list. It was intended, in a more fully developed program, that as the game progressed the goals of center control and development would drop out, since they are important mainly in the opening, and would be replaced by others more appropriate to later phases of the game.

Each active goal in the Newell, Shaw, and Simon program was responsible for generating relevant moves at the first level of the tree. In addition, each goal contained its own separate generator for moves at deeper levels, its own criteria for whether a position was dead, and its own static evaluation function. The search proceeded, in a highly selective manner, until the tip nodes were dead with respect to all active goals. Static evaluations with respect to the various goals were combined lexicographically, so that the highest priority goal was dominant and the others served only as tiebreakers. Newell and Simon report that the program's average search tree contained only 13 nodes--with no apparent loss in playing power compared to other programs up to that time (1972, p. 694).

#### **Knowledge-based Programs**

The Bernstein and Newell, Shaw, and Simon programs were early efforts to introduce significant chess knowledge, organized in human terms, to limit brute-force search. The actual knowledge was very sketchy; apparently neither program ever won a game (see Newell & Simon, 1972, pp. 677, 690).

An attempt at fuller use of chess knowledge was made in Berliner's program, CAPS-II (1974, 1977b). Much of the work involved developing a representation suitable for use in selectively generating moves, making preliminary evaluations of the moves so proposed, and describing the actual consequences discovered when a move was tried. The moves generated depend on the current goal state, which may be King in Check, Aggressive, Preventive Defense, Nominal Defense, Dynamic Defense, or Strategy. In contrast to the Newell, Shaw, and Simon program, the goal states are mutually exclusive, and state transitions occur dynamically as the tree is searched, in accordance with a complex flowchart. An important feature of the program, the Causality Facility, relates to both move generation and move ordering, as well as to pruning in some cases. The problem it attacks is

a general one in tree searching: When a path has been explored and found unsatisfactory, most programs have no way to diagnose what went wrong and use this information in deciding where to search next.

The basic search algorithm in CAPS-II is depth-first, with minimaxing and alpha-beta pruning. The Causality Facility operates as a refinement on this search. A first new feature is that, whenever a value is backed up in the search tree as a tentative minimax value, certain information is accumulated about the consequences of the move or moves that produced the value. The data structure in which the information is stored is called a Refutation Description. As the basis for making use of the Refutation Description, the program uses a variable representing the expected value of the position at the root of the search tree; this value, which may be updated during the search, lies somewhere between the bounds given by alpha and beta. Now, the tentative value newly backed up to a node can be compared with the expected value. If the comparison is unsatisfactory, the Causality Facility uses the Refutation Description to decide whether the last move tried from the node could have been responsible. It generates a list of alternative moves from the node, with the aim of avoiding the unsatisfactory result. These moves are compared with the list of moves from the node that had been generated earlier but which have not yet been tried. The comparison is used to reorder moves already on the untried list and, depending on the state the program is in, to add new moves to the list and to prune old ones.

Whereas Berliner's program plays the full game of chess, there are several other recent programs which, in their emphasis on representing chess knowledge, limit their task to solving problems that involve only selected aspects of the game. Two of these are the programs of Pitrat (1977) and Wilkins (1979). In each, the task is to find a line of play that wins material, beginning from a given middle-game position. The approach in both programs is to work backward from the goal of winning material to a structure of subgoals that constitutes a *plan*. (See Planning.) Wilkins's program, PARADISE, for example, has as a main theme the expression of chess concepts, like making a square safe for a piece or safely capturing a piece, in terms that can be used as subgoals and eventually reduced to specific moves. Initially, a plan is based not on search but on an extensive analysis of the originally given position; it may contain conditional branches depending on general categories of moves with which the opponent might reply. The general plan is then used to guide search, generating a very small tree. Moves considered for the program to make are only those relevant to the current subgoal; for the simulated opponent, all reasonable defensive moves are considered. If search shows that the plan has failed, a causality facility similar to Berliner's is used to analyze the difficulty and suggest a new plan.

Both the Pitrat and the Wilkins programs have succeeded in solving problems where the winning line of play goes to a depth of around 20 ply. Pitrat reports, for a set of 11 problems, that search tree sizes ranged from about 200 to 22,000 nodes; computation time varied from under 3 seconds to about 7.5 minutes. Wilkins's PARADISE generates smaller trees but uses more time; for 89 problems solved, the number of nodes in the search tree ran from a minimum of 3 to a maximum of 215, and time to find the solution varied from 19 seconds to 33 minutes. Wilkins also reports a good success rate compared to previous programs tested on the same set of problems, including Berliner's program, Gillogly's TECH, and an earlier version of CHESS 4.5. The programs other than PARADISE, however, were tested with a time limit of only 5 minutes per problem.

A final example of the use of chess knowledge to solve a class of problems is the work

of Donald Michie and his colleagues on chess endgames (e.g., Bratko, Kopec, & Michie, 1978; Michie & Bratko, 1978). Here each combination of pieces with which the endgame may be played is treated as posing a separate problem. One problem, denoted KNKR, is to defend with king and knight against king and rook, starting from any of some 3 million legal positions involving only those pieces. The objective is to provide the program with enough knowledge about this specific class of chess problems to achieve theoretically correct play, even in situations where chess masters sometimes err, and to accomplish this using only a moderate amount of search.

The program's knowledge is encoded in a data structure called an Advice Table, within which patterns occurring on the board may be described. Each pattern has an associated list of goals, or "pieces of advice," in the order in which they should be attempted. The object then becomes to find a solution--in the sense of a solution subtree of an AND/OR tree (C2)--to the problem of satisfying one of the goals. Unlike a standard AND/OR tree search, however, the "advice" includes not only a definition of when tip nodes should be considered terminal, but also constraints that every intermediate node in the solution tree must satisfy.

The amount of search required to find a solution using an Advice Table depends on how much knowledge the table contains. If the only goal provided were avoidance of mate, a search to the impossible depth of 85 ply would be needed to find the best defense from some positions. With the additional advice not to lose the knight and to keep king and knight together, search to about 10 ply is sufficient. With the further refinements included in the actual Advice Table, the program is reported to play the KNKR endgame at master level using only a 4-ply search.

### References

See Adelson-Velskiy, Arlazarov, & Donskoy (1975), Berliner (1973), Berliner (1974), Berliner (1977a), Berliner (1977b), Berliner (1977c), Berliner (1978a), Berliner (1978b), Bernstein et al. (1959), Bratko, Kopec, & Michie (1978), Charness (1977), Frey (1977), Gillogly (1972), Greenblatt, Eastlake, & Crocker (1967), Griffith (1974), Harris (1974), Harris (1975), Harris (1977), Hearst (1977), Kister et al. (1957), Kotok (1962), Levy (1979), Michie (1977), Michie & Bratko (1978), Mittman (1977), Newborn (1975), Newborn (1978), Newell, Shaw, & Simon (1963a), Newell & Simon (1972), Nilsson (1969), Nilsson (1971), Plitrat (1977), Samuel (1963), Samuel (1967), Shannon (1950), Slagle & Dixon (1969), Slagle & Dixon (1970), Slate & Atkin (1977), Thorp & Walden (1970), Turing (1953), and Wilkins (1979).



## D. Example Search Programs

### D1. Logic Theorist

The Logic Theorist (LT) was a program written by Allen Newell, J. C. Shaw, and H. A. Simon in 1956, as a joint project of the RAND Corporation and the Carnegie Institute of Technology. It was one of the earliest programs to investigate the use of heuristics in problem solving. The term *heuristics*, as used by Newell, Shaw and Simon, referred to "the complex processes . . . that are effective in problem-solving." They stated,

We are not interested in methods that guarantee solutions, but which require vast amounts of computation. Rather, we wish to understand how a mathematician, for example, is able to prove a theorem even though he does not know when he starts how, or if, he is going to succeed. (Newell, Shaw, & Simon, 1963b, p. 109)

Heuristics were thus identified with processes "that may solve a given problem, but offer no guarantee of doing so" (p. 114; see also Overview).

In descriptions of the Logic Theorist program, the heuristics discussed by Newell, Shaw, and Simon relate principally to limiting the search space by means of an apt problem formulation. Within the defined space, the search was blind except for some minor selectivity in the selection of operators (see C3a).

The problem domain of the Logic Theorist is the proof of theorems in the propositional calculus (see Representation.Logic). The basis is Whitehead and Russell's *Principia Mathematica*, from which both axioms and theorems to be proved were taken. There are five axioms, as follows:

1.  $(p \vee p) \supset p$
2.  $p \supset (q \vee p)$
3.  $(p \vee q) \supset (q \vee p)$
4.  $[p \vee (q \vee r)] \supset [q \vee (p \vee r)]$
5.  $(p \supset q) \supset [(r \vee p) \supset (r \vee q)]$

Some typical theorems that LT was given to prove include:

- 2.01.  $(p \supset \sim p) \supset \sim p$
- 2.45.  $\sim(p \vee q) \supset \sim p$
- 2.31.  $[p \vee (q \vee r)] \supset [(p \vee q) \vee r]$

The numbering of the theorems is taken from Whitehead and Russell. In some cases, the data given the program included not only the axioms but also previously proved theorems from that work. When all earlier theorems were included with the axioms, the program succeeded in proving 38 of the first 52 theorems in Chapter 2 of *Principia Mathematica*, in the sequence given there.

The program operates by *reasoning backward*, from the theorem to be established, to the axioms and given theorems. Three operators were provided for reducing the theorem to be proved, let us say X, to an axiom or theorem. These operators were:

**Detachment:** To show  $X$ , find an axiom or theorem of the form  $A \supset X$ , and transform the problem to the problem of showing  $A$ .

**Forward chaining:** To show  $X$  where  $X$  has the form  $A \supset C$ , find an axiom or theorem of the form  $A \supset B$ , and transform the problem to the problem of showing  $B \supset C$ .

**Backward chaining:** To show  $X$  where  $X$  has the form  $A \supset C$ , find an axiom or theorem of the form  $B \supset C$ , and transform the problem to the problem of showing  $A \supset B$ .

Since the axioms and given theorems contain variables, consideration must be given to the means for deciding whether a problem has in fact been reduced to something known. The question is whether a current problem expression  $X$  is an instance of an axiom or known theorem. The test, called the Substitution Test, uses two rules of inference distinct from those reflected in the operators:

**Substitution:** A variable in a theorem may be replaced, in all its occurrences throughout the theorem, by an expression. For example, substituting the expression " $p \vee q$ " for the variable " $p$ " transforms

$p \supset (q \vee p)$

into

$(p \vee q) \supset [q \vee (p \vee q)].$

**Replacement:** The connective " $\supset$ " is interchangeable with its definition. That is, if  $p$  and  $q$  are expressions, then

$p \supset q$

can be replaced by

$\sim p \vee q$

and vice versa.

As well as being used to determine whether a proof is complete, the substitution test is also essential for determining what applications of the three operators are possible with respect to a given problem expression.

The general algorithm used by the Logic Theorist is a blind, breadth-first state-space search using backward reasoning. The initial state corresponds to the original theorem to be proved. To test whether an expression has been proved, the program applies the substitution test, pairing the problem expression with each axiom and assumed theorem, in turn. If substitution fails, the expression is placed on a list of open problems; problems are selected from this list to become the current problem in first-in, first-out order.

To a problem selected from the list, each of the three operators is applied, in fixed order and in all possible ways, to generate new open problems. The search terminates with success as soon as a single problem is generated that passes the substitution test, since this means that a path has been completed between an axiom and the original problem. The search fails if it exceeds time or space limits, or if it runs out of open problems.

An example of a case in which the latter occurs is the attempted proof of the theorem

$p$  or  $\sim\sim p$ .

To succeed with this proof, LT would have needed more powerful operators; this particular problem required the ability, which LT lacked, to transform a problem to a set of subproblems, or conjunctive subgoals, which *all* had to be solved in order to solve the original problem.

There are some qualifications to the preceding general description of LT. One concerns the statement that each operator is applied to the current problem in every possible way, that is, that the current problem expression is matched against every axiom and assumed theorem to determine the applicability of any of the operators to that expression-axiom pair. In fact, the program attempted a match for the purpose of discovering an appropriate substitution only if the pair had passed a test indicating equality of certain gross features, such as the number of distinct variables in each. This test for similarity occasionally rejected a pair for which a substitution in fact would have been possible, thus excluding a proof the program would otherwise have found. Overall, the utility of this similarity test was considered rather marginal.

Some other additions, apparently made in a later version of the program (see Newell & Simon, 1972, pp. 125-128), included (a) ordering the open problems, taking up those involving simpler expressions first instead of proceeding in a strictly breadth-first order, and (b) rejecting some subproblems entirely as too complicated or apparently unprovable. In the implementation of these features, the latter appeared to be the more effective measure in reducing search effort. There was also experimentation, as mentioned previously, with the number of theorems that could be assumed as *given* in addition to the basic axioms. The conclusion on this point was that "a problem solver may be encumbered by too much information, just as he may be handicapped by too little" (Newell & Simon, 1972, p. 127).

## References

See Newell, Shaw, & Simon (1963b), Newell & Simon (1972), and Whitehead & Russell (1925).

## D2. General Problem Solver

The General Problem Solver (GPS) was developed by Allen Newell, J. C. Shaw, and H. A. Simon beginning in 1957. The research had a dual intention: It was aimed both at getting machines to solve problems requiring intelligence and at developing a theory of how human beings solve such problems. GPS was the successor of the authors' earlier Logic Theorist program (Article D1), whose methods had only a slight resemblance to those used by humans working on similar problems. Development of GPS continued through at least ten years and numerous versions of the program. The final version, described in detail in Ernst and Newell (1969), was concerned with extending the generality of the program, not with the psychological theory.

The name "General Problem Solver" came from the fact that GPS was the first problem-solving program to separate its general problem-solving methods from knowledge specific to the type of task at hand. That is, the problem-solving part of the system gave no information about the kind of task being worked on; task-dependent knowledge was collected in data structures forming a *task environment*. Among the data structures were *objects* and *operators* for transforming objects. A task was normally given to GPS as an initial object and a desired object, into which the initial object was to be transformed. GPS objects and operators were similar to the states and operators of a state-space problem representation (Article B1).

The general problem-solving technique introduced by GPS, however, does not fit neatly into either the state-space or the problem-reduction representation formalisms. It differs from a standard state-space search (e.g., Article IIC1) in the way it decides what path to try next. This technique, called *means-ends analysis*, is a major theoretical contribution of the program. It assumes that the *differences* between a current object and a desired object can be defined and classified into types and that the operators can be classified according to the kinds of difference they might reduce. At each stage, GPS selects a single relevant operator to try to apply to the current object. The search for a successful operator sequence proceeds depth first as long as the chosen operators are applicable and the path shows promise. Backup is possible if the current path becomes unpromising--for example, if eliminating one difference has introduced a new one that is harder to get rid of.

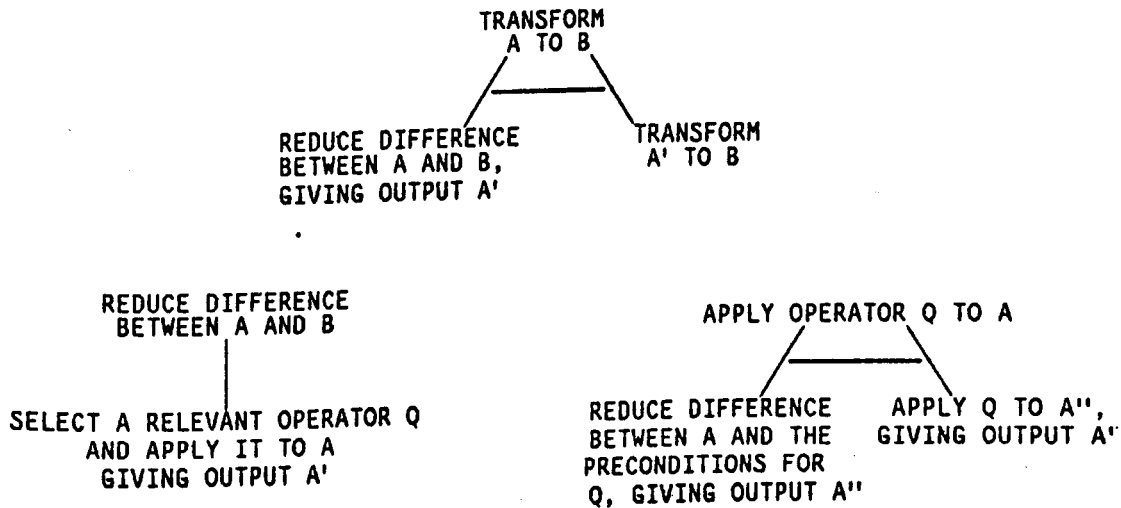
An important feature of means-ends analysis is the fact that the operator selected as relevant to reducing a difference may in fact be inapplicable to the current object. Rather than rejecting the operator for this reason, GPS attempts to change the current object into an object appropriate as input to the chosen operator. The result of this strategy is a recursive, goal-directed program that records the search history in an AND/OR graph (Article B2) with partial development of nodes (Article C3a).

### Goals and Methods

The most important data structure used by GPS is the *goal*. The goal is an encoding of the current situation (an object or list of objects), the desired situation, and a history of the attempts so far to change the current situation into the desired one. Three main types of goals are provided:

1. Transform object A into object B.
2. Reduce a difference between object A and object B by modifying object A.
3. Apply operator Q to object A.

Associated with the goal types are *methods*, or procedures, for achieving them. These methods, shown in a simplified version in Figure 1, can be understood as problem-reduction operators that give rise either to AND nodes, in the case of transform or apply, or to OR nodes in the case of a reduce goal. The initial task presented to GPS is represented as a transform goal, in which A is the initial object and B the desired object.



The recursion stops if the goal is primitive--that is, if for a transform goal there is no difference between A and B; and if for an apply goal the operator Q is immediately applicable. For a reduce goal, the recursion may stop, with failure, when all relevant operators have been tried and have failed.

### Selection of Operators

In trying to transform object A to object B, the transform method uses a matching process to discover the differences between the two objects. The possible types of difference are predefined and ordered by estimated difficulty, for each kind of task. The most difficult difference found is the one chosen for reduction. A domain-dependent data structure called the *Table of Connections* lists the operators relevant to reducing each difference type.

### Depth Bounds

Several heuristics are provided to prevent GPS from following a false path indefinitely. Some of the bases on which a current goal may be abandoned, at least temporarily, are the following:

1. Each goal should be easier than its parent goal.

2. Of a pair of AND nodes representing subgoals generated by transform or apply, the second subgoal attempted should be easier than the first.
3. A newly generated object should not be much larger than the objects occurring in the topmost goal.
4. Once a goal has been generated, the identical goal should not be generated again.

### An Example

The first task environment to which GPS was applied was the domain of the Logic Theorist: proving theorems in the propositional calculus. The initial and desired objects were expressions, one to be transformed into the other by means of operators representing rules of inference. There were twelve operators altogether, including the following rules. (The symbol " $\Rightarrow$ " means "may be rewritten as.")

Rule 1.  $A \vee B \Rightarrow B \vee A$   
 $A \wedge B \Rightarrow B \wedge A$

...

Rule 5.  $A \vee B \Leftrightarrow \sim(\sim A \wedge \sim B)$

Rule 6.  $A \supset B \Leftrightarrow \sim A \vee B$

Six possible difference types were recognized:

- (a) occurrence of a variable in one expression but not the other,
- (b) occurrence of a variable a different number of times in the two expressions,
- (c) difference in sign,
- (d) difference in binary connective,
- (e) difference in grouping, and
- (f) difference in position of components.

The list just given is in decreasing order of assumed difficulty. Every difference between main expressions, however, was considered more difficult than any difference between subexpressions.

With this background, a trace (slightly simplified) of GPS's performance on a simple example can be given. The problem is to transform the initial expression

$$R \wedge (\sim P \supset Q),$$

denoted L1, into the desired expression

$$(Q \vee P) \wedge R,$$

denoted L0. The trace is shown below.

Goal 1: Transform L1 into L0.

Goal 2: Reduce positional difference between L1 and L0.

Goal 3: Apply Rule 1 to L1.

Return L2:  $(\sim P \supset Q) \wedge R$

Goal 4: Transform L2 into L0.

Goal 5: Reduce difference in connective between left subexpressions of L2 and L0.

Goal 6: Apply Rule 5 to left part of L2.

Goal 7: Reduce difference in connective between left part of L2 and precondition for Rule 5.

Reject goal 7 as no easier than goal 5.

Goal 8: Apply Rule 6 to left part of L2.

Return L3:  $(P \vee Q) \wedge R$

Goal 9: Transform L3 into L0.

Goal 10: Reduce positional difference between left parts of L3 and L0.

Goal 11: Apply Rule 1 to left part of L3.

Return L4:  $(Q \vee P) \wedge R$

Goal 12: Transform L4 to L0.

No difference exists, so problem is solved.

### The Problem of Generality

GPS was intended to model generality in problem solving through use of the broadly applicable techniques of heuristic search, and the strategy of means-ends analysis in particular. The implementation of these techniques was dependent on the internal representation of objects and operators. These representations, in early versions of GPS, were nicely suited to logic tasks like the example above. But they were inadequate for many other kinds of heuristic search problems. Before Ernst's extensions to the program (Ernst & Newell, 1969), GPS had in fact solved only two problems outside the logic domain.

The object of Ernst's work was to extend the number of kinds of problems that GPS could handle while holding its power at a constant level. One of his generalizations was in the representation of objects. Earlier, a desired object had had to be specified by giving its exact form. Forms containing variables and lists of forms could be used if necessary. But

these too were inadequate for representing symbolic integration problems, in which the desired object is any form whatever that does not contain an integral sign. Hence the description of a desired object by a list of constraints was introduced.

Another change was in the representation of operators, originally specified by giving the form of the input object and the form of the resulting output object. For some kinds of problems, it was desirable to have other tests of applicability besides the form of the input object, and to be able to describe the output object as a function of the input. A third change enabled GPS to deal with unordered sets of symbols, eliminating the need for special operators to permute their elements.

The generalized program succeeded in solving problems of 11 different kinds including symbolic integration, resolution theorem proving, and a variety of puzzles. Each generalization, however, entailed changes in the ways the problem representations could be processed, and these led in turn to deterioration with respect to the kinds of differences that could be detected. The only representable differences became "local" ones. An example of a global difference, which GPS could no longer recognize, was the total number of times a variable occurred in a logic formula. Consequently, theorem proving in the propositional calculus was not among the eleven tasks that the final version of GPS could do.

In the task domains in which GPS did succeed, it could solve only simple problems; and those, less efficiently than special-purpose problem solvers. If a long search was required, it ran out of memory space; and even easy problems, if they needed objects as complex as a chess position, quickly exhausted memory on a machine with 65K words. But GPS was not expected to be a performance program. What it yielded, in its authors' view, was "a series of lessons that give a more perfect view of the nature of problem solving and what is required to construct processes that accomplish it" (Ernst & Newell, 1969, p. 2). Although additional generalizations, such as game playing, were considered feasible, the authors concluded that GPS needed no further programming accretions and recommended that it be laid to rest.

#### References

See Ernst & Newell (1969), Newell & Ernst (1965), Newell, Shaw, & Simon (1960), Newell & Simon (1963), and Newell & Simon (1972).



### D3. Gelernter's Geometry Theorem-proving Machine

Herbert Gelernter's geometry theorem-proving machine was a program written in 1959 at the IBM Research Center in New York. The program was written in an extended FORTRAN, the FORTRAN List Processing Language, and implemented on an IBM 704 computer. The purpose of the program was to solve problems taken from high-school textbooks and final examinations in plane geometry. As with Newell, Shaw, and Simon's Logic Theorist, which proved theorems in the propositional calculus, the fact that there were algorithms for solving problems in these domains was considered irrelevant, since the object was to explore the use of heuristic methods in problem solving.

The formal system within which the geometry program worked contained axioms on parallel lines, congruence, and equality of segments and angles. This set of axioms, which was not meant to be either complete or nonredundant, was along the lines of an elementary textbook. The axioms played the role of *problem-reduction* operators. Some examples are: (a) To show that two line segments are equal, show that they are corresponding elements of congruent triangles; (b) to show that two angles are equal, show that they are both right angles; and (c) to show that two triangles are congruent, show the equality of a side and an angle in corresponding positions, or of an angle and two sides. The operators for establishing congruence split the problem into three subproblems, each to be solved separately by showing equality for one pair of elements. Newell and Simon (1972, p. 138) indicate that the geometry machine was the first program that was able to handle *conjunctive subgoals*. The program works backwards from the theorem to be proved, recording its progress in what amounted to an AND/OR tree (Article B2).

Some examples of problems solved by the program were the following:

1. Given that angle ABD equals angle DBC, that segment AD is perpendicular to segment AB, and that segment DC is perpendicular to segment BC, show that AD equals CD.

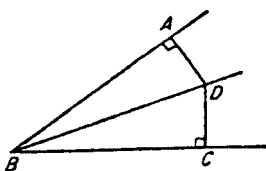


Figure 1. Diagram for problem 1.

2. Given that ABCD is a quadrilateral, with segment BC parallel to segment AD and with BC equal to AD, show that segment AB equals segment CD.

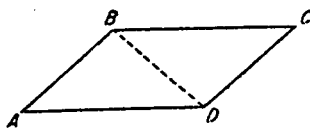


Figure 2. Diagram for problem 2.

A problem was given to the program in the form of a statement describing the premises and the goal. A proof was a sequence of statements giving the reduction of the goal to trivial goals--ordinarily, goals to establish an already established formula. One feature used to reduce the search effort needed to find a proof was the recognition of *syntactic symmetry*. Some examples of symmetric pairs of goals are the following:

- a. If  $d(x,y)$  is the distance from point  $x$  to point  $y$ , then  $d(A,B) = d(C,D)$  is symmetric with  $d(D,C) = d(A,B)$ .
- b. If ABCD is a parallelogram and point E is the intersection of its diagonals AC and BD, then  $d(A,E) = d(E,C)$  is symmetric with  $d(B,E) = d(E,D)$ .

The recognition of symmetry was used in two ways. First, if a given goal was ever reduced to a subgoal symmetric with it, the subgoal could be rejected as representing circular reasoning. Second, if parallel goals A and B were syntactically symmetric and goal A had been established, then goal B could be established by symmetry--in effect by saying, for the second half of the proof, "Similarly, B."

The most notable feature of the program, however, was an additional part of the problem statement used to avoid attempting proofs by blind syntactic manipulation alone. This input was a diagram, similar to Figures 1 and 2 (although specified by lists of coordinates), of the points and line segments mentioned in the theorem. The particular input figure was chosen to avoid spurious coincidences and reflect the greatest possible generality. Whenever a subgoal was generated, it was checked for consistency with the diagram. If false in the diagram, the subgoal could not possibly be a theorem and therefore could be *pruned* from the search tree. A slight qualification is that finite precision arithmetic, applied to the diagram, occasionally caused a provable subgoal to be pruned erroneously; but it was reported that the program had found other paths to the solution in such cases. It was estimated that the use of a diagram, together with the discard of subgoals representing circular reasoning, eliminated about 995 out of every thousand subgoals.

The diagram also served a second purpose: It provided an additional criterion by which a problem could be considered *primitive*. For example, a rigorous proof of the theorem in problem 1 would require showing that DB is a line segment and that BCD and BAD are triangles. The axioms needed would have been (a) if X and Y are distinct points, then XY is a line segment; and (b) if X, Y, and Z are three distinct non-collinear points, then XYZ is a triangle. For a limited class of such properties, the program did not require formal proof but rather considered them established if they were true in the diagram. It recorded explicitly the assumptions that had been made based on the diagram.

The central loop of the program repeatedly selects the next goal to try. Two heuristics were included for goal selection. One gave highest priority to classes of goals, such as identities, that could usually be established in one step. The second assigned a "distance" between the goal statement and the set of premise statements; after the one-step goals had been developed, the remaining goals were selected in order of increasing distance from the premise set.

Once a goal was chosen for development, the action taken depended on its status. Ordinarily, it would be reduced to subgoals and the subgoals, if consistent with the diagram but not sufficient to establish the current goal immediately, would be added to the list of

goals to try. If no new acceptable subgoals were generated, the program checked whether a *construction* was possible--a construction being the addition to the premises of the problem of a line segment between two existing but previously unconnected points. The new segment would be extended to its intersections with other segments in the figure. New points could be added to the premises only if generated by such intersections.

A goal for which a construction was found possible was saved--to be tried again later if all goals not requiring construction should be exhausted. If the goal was later selected for a second try, a construction would be made and the problem started over with an expanded premise set. An example of the use of this technique occurs in problem 2, where in considering the goal  $AB = CD$ , the program generated a subgoal of showing that triangles ABD and CDB were congruent. The subgoal makes sense only if a line segment BD exists, so the segment is constructed, and the proof eventually succeeds.

### References

See Elcock (1977), Gelernter (1959), Gelernter (1963), Gelernter, Hansen, & Gerberich (1960), Gelernter, Hansen, & Loveland (1963), Gelernter & Rochester (1958), and Gilmore (1970).

#### D4. Symbolic Integration Programs

##### Slagle's SAINT

James Slagle's SAINT program (Symbolic Automatic INTEgrator) was written as a 1961 doctoral dissertation at MIT. The program solves elementary symbolic integration problems--mainly indefinite integration--at about the level of a good college freshman. SAINT was written in LISP and run interpretively on the IBM 7090.

The kinds of questions Slagle intended his thesis to address were some of the earliest questions for AI. They included, for example, "Can a computer recognize the kinds of patterns that occur in symbolic expressions? Just how important is pattern recognition? . . . Can intelligent problem solving behavior really be manifested by a machine?" (Slagle, 1961, p. 9). The domain of symbolic integration was chosen as a source of well-defined, familiar, but nontrivial problems requiring the manipulation of symbolic rather than numerical expressions.

The integration problems that SAINT could handle could have only elementary functions as integrands. These functions were defined recursively to comprise the following:

1. Constant functions.
2. The identity function.
3. Sum, product, and power of elementary functions.
4. Trigonometric, logarithmic, and inverse trigonometric functions of elementary functions.

Three kinds of operations on an integrand were available:

1. Recognize the integrand as an instance of a standard form, thus obtaining the result immediately by substitution. Twenty-six such standard forms were used. A typical one indicated that if the integrand has the form  $c^v dv$ , the form of the solution is  $(c^v)/(\ln c)$ .
2. Apply an "algorithm-like transformation" to the integral--that is, a transformation that is almost guaranteed to be helpful whenever it can be applied. Eight such transformations were provided, including (a) factoring out a constant and (b) decomposing the integral of a sum into a sum of integrals.
3. Apply a "heuristic transformation"--that is, a transformation carrying significant risk such that, although applicable, it might not be the best next step. The 10 heuristic transformations included certain substitutions and the technique of integration by parts. One technique that was not implemented was the method of partial fractions.

The program starts with the original problem as a goal, specified as an integrand and a variable of integration. For any particular goal, the strategy is first to try for an immediate solution by substitution into a standard form; failing that, to transform it by any applicable algorithm-like transformation; and finally to apply each applicable heuristic transformation in

turn. Both the algorithm-like and the heuristic transformations, however, generate new goals, to which the same strategy may be applied. The result is an AND/OR graph of goals (Article B2).

The order in which goals are pursued by SAINT depends heavily on what operations can be applied to them. At the level of heuristic transformations, the algorithm is an *ordered search*: A list, called the Heuristic Goal List, keeps track of goals on which progress can be made only by applying heuristic transformations--that is, integrands that are not of standard form nor amenable to any algorithm-like transformation. To each goal on this list is attached an estimate of the difficulty of achieving it. The measure of difficulty used is the maximum level of function composition in the integrand. Other characteristics of the goal, such as whether it is a rational function, an algebraic function, a rational function of sines and cosines, and the like, are also stored as an aid to determining which heuristic transformations will in fact apply. The outer loop of the program repeatedly selects the goal that looks the easiest from the Heuristic Goal List, expands it by applying all applicable heuristic transformations, and possibly, as a result of the expansion, adds new elements to the Heuristic Goal List. The program terminates with failure if it runs out of heuristic goals to work on or if it exceeds a pre-set amount of working space.

An important qualification to this process concerns the use of standard forms and algorithm-like transformations. As soon as any new goal is generated (or the original goal read in), an immediate solution of it is attempted. The attempt consists of, first, checking whether the integrand is a standard form; if it is not, checking whether an algorithm-like transformation applies; and if one does, applying it and calling the immediate solution procedure recursively on each goal resulting from that transformation. When the recursion terminates, either the generated goal has been achieved or there is a set of goals--the generated goal itself or some of its subgoals--to be added to the Heuristic Goal List. During expansion of a node (one iteration of the outer loop), new heuristic goals are accumulated in a temporary goal list; only after expansion is complete are their characteristics computed and the additions made to the Heuristic Goal List.

Whenever a goal is achieved, the implications of its achievement are immediately checked. If it is the original goal, the program terminates successfully. Otherwise, if it was achieved by substitution into a standard form, it may cause the achievement of one or more parent goals as well. If it was achieved by solution of a sufficient number of its subproblems, it may not only cause its parent or parents to be achieved in turn, but may also make others of its subproblems, which have not yet been solved, superfluous. These checks are implemented in a recursive process, referred to as "pruning the goal tree," that is initiated as soon as any goal is achieved. Thus a heuristic goal can be achieved without having been fully expanded.

### Moses's SIN

A second important symbolic integration program, SIN (Symbolic INtegrator), was written by Joel Moses in 1969, also as a doctoral dissertation at MIT. Its motivation and its strategy as an AI effort were quite different from those of SAINT. Whereas Slagle had compared the behavior of SAINT to that of freshman calculus students, Moses aimed at behavior comparable to expert human performance. He viewed SAINT as emphasizing generality in that it examined mechanisms, like heuristic search, that are useful in many diverse problem

domains. SIN, in contrast, was to emphasize expertise in a particular, complex domain. To do this, it concentrated on problem analysis, using more knowledge about integration, than SAINT had employed, to minimize the need for search. In fact, Moses did not view SIN as a heuristic search program. Hence, the program will be described only briefly here; and a second part of Moses's dissertation, a differential equation solver called SOLDIER, will not be described.

SIN worked in three stages, each stage being capable of solving harder problems than the stage before. Stage 1 corresponded roughly to Slagle's immediate solution procedure but was more powerful. It used a table of standard forms; two of Slagle's algorithm-like transformations; and, most importantly, a method similar to one of Slagle's heuristic transformations, referred to as the Derivative-divides method. The idea behind this grouping of methods was that they alone would be sufficient to solve the most commonly occurring problems, without invoking the computationally more expensive machinery of the later stages.

A problem that stage 1 could not solve was passed on to stage 2. This stage consisted of a central routine, called FORM, and 11 highly specific methods of integration. (One of these methods was a program for integrating rational functions that had been written by Manove, Bloom, and Engelman, of the MITRE Corporation, in 1964.) In general, the task of FORM was to form a hypothesis, usually based on local clues in the integrand, about which method, if any, was applicable to the problem. Only rarely did more than one method apply. The routine chosen first tried to verify its applicability; if it could not, it returned to let FORM try again. If the routine did verify the hypothesis, however, SIN then became committed to solving the problem by that method or not at all. The method chosen either solved the problem using mechanisms internal to it or transformed the problem and called SIN recursively to solve the transformed problem.

Stage 3 of SIN was invoked, as a last resort, only if no stage 2 method was applicable. Two general methods were programmed here. One method was integration-by-parts, which used blind search, subject to certain constraints, to find the appropriate way to factor the integrand. The other was a nontraditional method based on the Liouville theory of integration and called the EDGE (EDUCATED GuEss) heuristic. This method involved guessing the form of the integral. The EDGE heuristic was characterized as using a technique similar to *means-ends analysis*, if its guess did not lead directly to a solution.

#### Performance of SAINT and SIN

SAINT was tested on a total of 86 problems, 54 of them chosen from MIT final examinations in freshman calculus. It succeeded in solving all but two. The most difficult problem it solved, both in terms of time and the number of heuristic transformations occurring in the solution tree (four), was the integral of

$$\frac{(\sec t)^2}{1 + (\sec t)^2 - 3(\tan t)} dt.$$

Slagle proposed additional transformations that would have handled the two failures, which were the integrals of

$$x(1+x)^{1/2} dx \quad \text{and} \quad \cos(x^{1/2}) dx.$$

SIN, in contrast, was intended to model the behavior of an expert human integrator. The results of running SIN on all of Slagle's test problems were that more than half were solved in stage 1, and all but two of the rest (both of which used integration by parts) were solved in stage 2. After adjusting for the facts that SAINT and SIN ran on different machines and that one was interpreted and the other compiled, and for other factors making the programs difficult to compare, Moses estimated that SIN would run on the average about three times faster than SAINT. Taking into account a test on more difficult problems as well, he expressed the opinion that SIN was "capable of solving integration problems as difficult as ones found in the largest tables" (p. 140) and that it was fast and powerful enough for use in "a practical on-line algebraic manipulation system" (p. 6). For later developments in this direction, see Applications.Macsyma.

#### References

See Manove, Bloom, & Engelman (1968), Moses (1967), Slagle (1961), and Slagle (1963).

## D5. STRIPS

STRIPS is a problem-solving program written by Richard Fikes and Nils Nilsson (1971) at SRI International. Each problem for STRIPS is a goal to be achieved by a robot operating in a simple world of rooms, doors, and boxes. The solution is a sequence of operators, called a *plan*, for achieving the goal. (For a review of the various senses of the word *plan*, see Planning). The robot's actual execution of the plan is carried out by a separate program, distinct from STRIPS. A later (1972) addition to the basic STRIPS system permits plans to be generalized and used again, giving the system some capacity for *learning*.

### The Basic STRIPS System

**The world model.** The world in which the STRIPS robot works consists of several rooms connected by doors, along with some boxes and other objects that the robot can manipulate. STRIPS represents this world by a set of well-formed formulas in the first-order predicate calculus (see Representation.Logic). Some formulas in the world model are static facts, such as which objects are pushable and which rooms are connected. Other facts, such as the current location of objects, must be changed to reflect the actions of the robot.

**Operators.** The primitive actions available to the robot are represented by *operators*. Typical operators include going somewhere and pushing an object somewhere, the locations being given as parameters. Each operator has *preconditions* to its applicability; to push a box, for example, the robot must first be next to the box. The application of an operator is realized by making changes in the world model. The appropriate changes are given by a *delete list* and an *add list*, specifying the formulas to be removed from and added to the world model as a result of the operation. Thus, each operator explicitly describes what it changes in the world model.

A typical operator is GOTOB, which denotes the robot's going up to an object in the same room:

```
GOTOB (bx) "go to object bx"
Preconditions: TYPE(bx,OBJECT) and
              THERE EXISTS (rx) [INROOM(bx,rx) and INROOM(ROBOT,rx)]
Delete list:  AT(ROBOT,*,*) , NEXTTO(ROBOT,*)
Add list:     NEXTTO(ROBOT,bx) .
```

The precondition statement requires that *bx* be an object and that both *bx* and the robot be in the same room, *rx*. The asterisks in the delete list represent arguments with any values whatever.

**Method of operation.** STRIPS operates by searching a space of world models to find one in which the given goal is achieved. It uses a *state-space representation* in which each state is a pair (world model, list of goals to be achieved). The initial state is (MO, (GO)), where MO is the initial world model and GO the given goal. A terminal state gives a world model in which no unsatisfied goals remain.

Given a goal *G* (stated as a formula in the predicate calculus), STRIPS first tries to prove that *G* is satisfied by the current world model. To do this, the program uses a modified



version of the resolution-based theorem prover QA3 (Garvey & Kling, 1969). Typically the proof fails, within a pre-specified resource limit, because no more resolvents can be formed (see Theorem Proving, Resolution). At this point, STRIPS needs to find a different world model which the robot can achieve and which satisfies the goal. Because this task is complicated for a simple theorem prover, the system switches to a *means-ends analysis* similar to that of GPS (Article D2).

To do the means-ends analysis, the program extracts a *difference* between the goal and the current model and selects a "relevant" operator to reduce the difference. The difference consists of any formulas from the goal that remain outstanding when the proof attempt is abandoned (pruned, if this set is large). A relevant operator is one whose add list contains formulas that would remove some part of the difference, thereby allowing the proof to continue.

If the operator is applicable, the program applies it and tries to achieve the goal in the resulting model; otherwise, the operator's precondition becomes a new subgoal to be achieved. Since there may be several relevant operators at each step, this procedure generates a tree of models and subgoals. STRIPS uses a number of heuristics to control the search through this tree.

#### An Example of the Basic System's Performance

As a simple example, suppose the robot is in ROOM1 and the goal is for it to be next to BOX1, which is in adjacent ROOM2. The initial world model MO contains such clauses as

```
INROOM (ROBOT,ROOM1),
INROOM (BOX1,ROOM2),
TYPE (BOX1,OBJECT),
CONNECTS (DOOR12,ROOM1,ROOM2),
STATUS (DOOR12,OPEN), ...
```

and the goal GO is

```
G0 = NEXTTO (ROBOT,BOX1) .
```

GO is not satisfied, and the difference between it and the initial model is ~NEXTTO (ROBOT,BOX1). STRIPS determines that GOTOB (bx), defined above, is a relevant operator, with bx instantiated as BOX1. The operator instance GOTOB (BOX1), denoted OP1, is not immediately applicable (because the robot is in the wrong room), so its precondition G1,

```
G1 = TYPE (BOX1,OBJECT) and
      THERE EXISTS (rx) [INROOM (BOX1,rx) and INROOM (ROBOT,rx)]
```

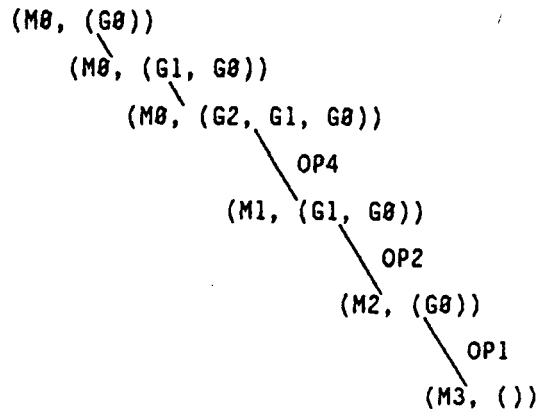
becomes a new subgoal. Relevant operators for reducing the difference between G1 and the initial model MO are: OP2 = GOTHRUDOOR (dx,ROOM2) and OP3 = PUSHTHRUDOOR (BOX1,dx,ROOM1) (i.e., move the robot to the room with the box, or move the box to the room with the robot). If the former course (the better one, obviously) is selected, the precondition

G2 = STATUS (dx, OPEN) and NEXTTO (ROBOT, dx) and  
 THERE EXISTS (rx) [INROOM (ROBOT, rx) and CONNECTS (dx, rx, ROOM2)]

is the new subgoal. The difference  $\sim$ NEXTTO (ROBOT, DOOR12) can be reduced by the operator OP4 = GOTODOOR (DOOR12), which is applicable immediately. Applying OP4 adds the clause NEXTTO (ROBOT, DOOR12) to the model, creating a new world model M1. G2 is now satisfied with dx = DOOR12, so OP2 can be instantiated as GOTHRUDOOR (DOOR12, ROOM2) and applied. This deletes the clause INROOM (ROBOT, ROOM1) and adds INROOM (ROBOT, ROOM2). G1 is now satisfied, so OP1 is applied, deleting NEXTTO (ROBOT, DOOR12) and adding NEXTTO (ROBOT, BOX1), the desired goal. The final plan is thus:

OP4: GOTODOOR (DOOR12)  
 OP2: GOTHRUDOOR (DOOR12, ROOM2)  
 OP1: GOTOB (BOX1)

The corresponding solution path through the state space tree is as follows:



### Generalization of Plans

In the basic STRIPS system, each new problem was solved from scratch. Even if the system had produced a plan for solving a similar problem previously, it was not able to make any use of this fact. A later version of STRIPS provides for generalizing plans and saving them, to assist both in the solution of subsequent problems and also in the intelligent monitoring of the robot's execution of the particular plan.

**Triangle tables.** A specific plan to be generalized, say (OP1, OP2, ..., OPn), is first stored in a data structure called a *triangle table*. This is a lower triangular array representing the preconditions for and effects of each operator in the plan. Some of its properties are the following:

1. Cell (i,0) contains clauses from the original model that are still true when operator i is to be applied and that are preconditions for operator i, OPi.

2. Marked (starred) clauses elsewhere in row  $i$  are preconditions for operator  $i$  added to the model by previous operators.
3. The effects of applying operator  $i$  are shown in row  $i+1$ . The operator's add list appears in cell  $(i+1, i)$ . For each previous operator, say operator  $j$ , clauses added by operator  $j$  and not yet deleted are copied into cell  $(i+1, j)$ .
4. The add list for a sequence of operators 1 through  $i$ , taken as a whole, is given by the clauses in row  $i+1$  (excluding column 0).
5. The preconditions for a sequence of operators  $i$  through  $n$ , taken as a whole, are given by the marked clauses in the rectangular sub-array containing row  $i$  and cell  $(n+1, 0)$ . This rectangle is called the  $i$ -th *kernel* of the plan.

The triangle table for the previous example is shown below. Operators have been renumbered in the order of their use.

1	*INROOM(ROBOT, ROOM1) *CONNECTS(D12, ROOM1,ROOM2)	OP1 GOTODOOR(D12)		
2	*INROOM(ROBOT, ROOM1) *CONNECTS(D12, ROOM1,ROOM2) *STATUS(D12, OPEN)	*NEXTTO (ROBOT,D12)	OP2 GOTHRUDOOR (D12,ROOM2)	
3	*INROOM(BOX1, ROOM2) *TYPE(BOX1, OBJECT)	NEXTTO (ROBOT,D12)	*INROOM (ROBOT,ROOM2)	OP3 GOTOB(BOX1)
4			INROOM (ROBOT,ROOM2)	NEXTTO (ROBOT,BOX1)
	0	1	2	3

Figure 1. A triangle table.

**Method of Generalization.** The plan is generalized by replacing all constants in each of the clauses in column 0 by distinct parameters and the rest of the table with clauses that assume that no argument to an operator has been instantiated. The result may be too general, so the proof of the preconditions for each operator is run again, noting any substitutions for parameters that constrain the generality of the plan. Some further corrections are made for remaining overgeneralization, which might make the plan either inconsistent or inefficient in use. Finally, the generalized plan, termed a MACROP, is stored away for future use.

In the example above, the generalized plan would be

GOTODOOR (dx)  
 GOTHRUDOOR (dx,rx1)  
 GOTOB (bx)

with preconditions:

INROOM (ROBOT,rx2)  
 CONNECTS (dx,rx2,rx1)  
 STATUS (dx,OPEN)  
 INROOM (bx,rx1)  
 TYPE (bx,OBJECT)

and add list:

NEXTTO (ROBOT,bx)  
 INROOM (ROBOT,rx1)

That is, the generalized plan sends the robot from any room through a connecting door to an object in the adjacent room.

Using the MACROP to guide execution. When STRIPS produces a detailed plan to achieve a goal, it does not necessarily follow that the robot should execute the plan exactly as given. One possibility is that some action fails to achieve its expected effect, so that the corresponding step of the plan needs to be repeated. Another is that the plan found is less than optimal and would be improved by omitting some steps entirely. The necessary flexibility during execution is provided by using the MACROP rather than the detailed plan in monitoring the robot's actions.

At the beginning of execution, the parameters of the MACROP are partially instantiated to the case at hand. The robot then attempts, at each stage, to execute the highest numbered step of the plan whose preconditions are satisfied. This procedure omits unnecessary steps and allows repeated execution, possibly with changed parameters, of a step that has failed. If there is no step whose preconditions are satisfied, replanning occurs. Determining which step can be done next is accomplished by a scan that exploits the design of the triangle table.

Using MACROPs in planning. When STRIPS is given a new problem, the time it takes to produce an answer can be reduced very considerably if there exists a MACROP that can be incorporated into its solution. The MACROP given above, for example, could be used as the first part of a plan to fetch a box from an adjacent room. The part of the MACROP consisting of its first two suboperators, if used alone, would also give a ready-made solution to the problem "Go to an adjacent room"; or it could be used repeatedly in solving "Go to a distant room."

The triangle table provides the means of determining whether a relevant macro operator exists. To determine whether the sequence of operators 1 through  $i$  of the MACROP is relevant, STRIPS checks the add list of this sequence as given by the  $i+1$ th row of the table. Once a MACROP is selected, irrelevant operators are edited out by a straightforward algorithm, leaving an economical, possibly parameterized operator for achieving the desired add list. The operator's preconditions are taken from the appropriate cells of column 0. Thus, almost any sub-sequence of operators from a MACROP can become a macro operator in a new plan. To keep new MACROPs from producing an overwhelming number of different

operators that must be considered during planning, the system contains provisions for preventing consideration of redundant parts of overlapping MACROPs and for deleting MACROPs that have been completely subsumed by new ones.

In a sequence of problems given to STRIPS, the use of MACROPs in some cases reduced planning time by as much as two-thirds. The longest plan so formed, consisting of 11 primitive operations, took the robot from one room to a second room, opened a door leading to a third room, took the robot through the third room to a fourth room, and then pushed two pairs of boxes together. One drawback noted by the authors was that, however long the solution sequence, STRIPS at each stage of its search dealt with every operation in complete detail. A later program, Sacerdoti's ABSTRIPS (Article D6), provides the mechanism for deferring the details of the solution until after its main outline has been completed.

#### References

See Fikes & Nilsson (1971), Fikes, Hart, & Nilsson (1972), and Garvey & Kling (1969).

## D6. ABSTRIPS

A combinatorial explosion faces all problem solvers that attempt to use heuristic search in a sufficiently complex problem domain. A technique called *hierarchical search* or *hierarchical planning*, implemented in Earl Sacerdoti's ABSTRIPS (1974), is an attempt to reduce the combinatorial problem. The idea is to use an approach to problem solving that can recognize the most significant features of a problem, develop an outline of a solution in terms of those features, and deal with the less important details of the problem only after the outline has proved adequate.

The implementation of this approach involves using distinct levels of problem representation. A simplified version of the problem, from which details have been omitted, occurs in a *higher level problem space* or *abstraction space*; the detailed version, in a *ground space*. By a slight extension, providing for several levels of detail instead of just two, a hierarchy of problem spaces is obtained. In general, each space in the hierarchy serves both as an abstraction space for the more detailed space just below it and as a ground space with respect to the less detailed space just above.

### Background--The STRIPS System

ABSTRIPS is a modification of the STRIPS system, described in Article D5. The problem domain for both programs is a world of robot planning. In both, the program is given an initial state of the world, or *world model*, consisting of a set of formulas that describe the floor plan of a group of rooms and other facts such as the location of the robot and other objects within these rooms. The goal state to be achieved is also given. The elements of a solution sequence are *operators* representing robot actions; examples are operators for going up to an object, pushing an object, and going through a door. The definition of each operator contains three kinds of formulas: (a) its *preconditions*, representing statements that must be true of a world model in order for the operator to be applicable; (b) its *add list*, a list of formulas that will become true and should be added to the world model when the operator is applied; and (c) its *delete list*, a corresponding list of formulas to be deleted from the model upon application of the operator. The search for a sequence of operators producing the desired world model is guided by a *means-ends analysis* similar to that of GPS (Article D2).

### Abstraction Spaces

Given the world models and operator descriptions of the basic STRIPS system, the first question is how to define the "details" that are to be ignored in the first pass at a solution. Sacerdoti's answer was to treat as details certain parts of the operator preconditions. At all levels of abstraction, the world models and the add and delete lists of operators remain exactly the same. Such a definition of "details" was found to be strong enough to produce real improvements in problem-solving efficiency, while keeping a desirable simplicity in the relationship between each abstraction space and its adjacent ground space.

The preconditions for an operator are stated as a list of predications, or *literals*, concerning the world model to which the operator is to be applied. The relative importance of literals is indicated by attaching to each a number called its *criticality value*. The hierarchy of problem spaces is then defined in terms of levels of criticality: In the space of criticality  $\pi$ , all operator preconditions with criticality less than  $\pi$  are ignored.

The assignment of criticality values is done just once for a given definition of the problem domain. The general ideas to be reflected in the assignment are the following:

1. If the truth value of a literal cannot be changed by any operator in the problem domain, it should have the highest criticality.
2. If the preconditions for an operator include a literal L that can be readily achieved once other preconditions for the same operator are satisfied, then L should be less critical than those other preconditions.
3. If the possibility of satisfying literal L depends on additional preconditions besides those referred to in (2), then L should have high but less than maximum criticality.

The actual assignment of criticalities is done by a combination of manual and automatic means. First, the programmer supplies a partial ordering of all predicates that can appear in operator preconditions. The partial ordering serves two purposes: It supplies a tentative criticality value for all instances of each predicate, and it governs the order in which the program will consider literals for possible increases (but not decreases) in criticality.

As an example, consider an operator TURN-ON-LAMP (x), with preconditions

TYPE (x,LAMP) and THERE EXISTS (r) [INROOM (ROBOT,r) and  
INROOM (x,r)] and PLUGGED-IN (x) and NEXTTO (ROBOT,x) .

The partial ordering of predicates, reflecting an intuitive view of their relative importance, might be as follows:

Predicate	Rank
TYPE	4
INROOM	3
PLUGGED-IN	2
NEXTTO	1

Figure 1. Initial ranking of predicates.

The assignment algorithm, whose output is summarized in the figure below, would first find that the truth of TYPE (x,LAMP) is beyond the power of any operator to change and therefore would set its criticality to the maximum; in this case, 6. Then it would find that TYPE (x,LAMP) is an insufficient basis for achieving INROOM (ROBOT,r) or INROOM (x,r); so these two literals would have their criticality raised to the next highest value, 5. Next PLUGGED-IN (x) is considered, and a plan to achieve PLUGGED-IN (x) is found using only the literals already processed as a starting point. Hence, the PLUGGED-IN literal retains its tentative criticality of 2, and, similarly, NEXTTO (ROBOT,x) is given criticality 1. The result, after similar processing of the preconditions of the other operators in the domain, is a hierarchy of at least four, and possibly six, distinct problem spaces.

Literal	Criticality Value
TYPE (x, LAMP)	6
INROOM (ROBOT, r)	5
INROOM (x, r)	5
PLUGGED-IN (x)	2
NEXTTO (ROBOT, x)	1

Figure 2. Final criticality of literals.

### Control Structure

A problem statement for ABSTRIPS, as for STRIPS, consists of a description of the state of the world to be achieved. A solution is a plan, or sequence of operators, for achieving it. ABSTRIPS proceeds by forming a crude plan at the highest level of abstraction and successively refining it. The executive is a recursive program taking two parameters: the current level of criticality, defining the abstraction space in which planning is to take place, and a list of nodes representing the plan to be refined. Before the initial call, criticality is set to the maximum, and the skeleton plan is initialized to a single operator--a dummy--whose preconditions are precisely the goal to be achieved. ABSTRIPS computes the difference between the preconditions and the current world model, finds operators relevant to reducing the difference and, if necessary, pursues subgoals of satisfying the preconditions of the selected operators. During this process, any preconditions of less than the current criticality are ignored. A search tree is built from which, if the process succeeds, a fuller operator sequence leading from the initial world model to the goal can be reconstructed. This new skeleton plan, together with the next lower criticality level, are passed recursively to the executive for the next round of planning.

The search strategy used by ABSTRIPS can be called *length-first*, in that the executive forms a complete plan for reaching the goal in each abstraction space before considering plans in any lower level space. This approach has the advantage that it permits early recognition of dead ends, thus reducing the work wasted in extending the search tree along fruitless paths involving detailed preconditions. If a subproblem in any particular space cannot be solved, control is returned to its abstraction space, and the search tree is restored to its previous state in that space. The node that caused the failure in the lower level space is eliminated from consideration and the search is continued in the higher level space. This mechanism, an example of *backtracking*, suffers from the problem that no information is available at the higher level on what caused the plan to fail.

Because backtracking can be inefficient and also because each operator in an abstraction space may be expanded to several operators in the ground space, it is important for ABSTRIPS to produce good plans at the highest level. Two modifications to STRIPS were made to try to insure that it would do so.

First, whereas a STRIPS search tended to be *depth-first* and therefore sometimes found non-optimal solutions, ABSTRIPS makes the order of expanding nodes in the search tree dependent on the level of abstraction. At the highest level it uses an *evaluation function* that may increase the search effort but which insures that the shortest possible solution sequence will be found. (See Article C3b on A\*.)



The second change relates to the instantiation of operator parameters, in cases where two or more choices seem equally good. While STRIPS made a choice arbitrarily, ABSTRIPS defers the choice until a greater level of detail indicates one to be preferable. Backtracking can still occur should the choice be mistaken.

### Performance

ABSTRIPS and STRIPS were compared on a sequence of problems. One of the longest, needing 11 operators for its solution, required the robot to open a door, go through the adjacent room to another room, push two boxes together, and then go through two more doors to reach the room where it was to stop. The basic STRIPS system required over thirty minutes of computer time to find the solution; ABSTRIPS used 5:28 minutes and generated only half the number of search-tree nodes. It was noted that by the time ABSTRIPS reached the most detailed level, it had in effect replaced the original large problem by a sequence of 7 easy subproblems.

### References

See Sacerdoti (1974).

## References

- Adelson-Velskiy, G. M., Arlazarov, V. L., & Donskoy, M. V. Some methods of controlling the tree search in chess programs. *Artificial Intelligence*, 1975, 6, 361-371.
- Aho, A. V., Hopcroft, J. E., & Ullman, J. D. *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, 1974.
- Amarel, S. On representations of problems of reasoning about actions. In D. Michie (Ed.), *Machine Intelligence 3*. New York: American Elsevier, 1968. Pp. 131-171.
- Baudet, G. M. On the branching factor of the alpha-beta pruning algorithm. *Artificial Intelligence*, 1978, 10, 173-199.
- Berliner, H. J. Some necessary conditions for a master chess program. In *IJCAI 3*, 1973. Pp. 77-85.
- Berliner, H. J. Chess as problem solving: the development of a tactics analyzer. Dept. of Computer Science, Carnegie-Mellon University, 1974.
- Berliner, H. J. Experiences in evaluation with BKG--a program that plays backgammon. In *IJCAI 5*, 1977. Pp. 428-433. (a)
- Berliner, H. J. A representation and some mechanisms for a problem-solving chess program. In M. R. B. Clarke (Ed.), *Advances in Computer Chess 1*. Edinburgh: Edinburgh University Press, 1977. Pp. 7-29. (b)
- Berliner, H. J. Search and knowledge. In *IJCAI 5*, 1977. Pp. 975-979. (c)
- Berliner, H. J. A chronology of computer chess and its literature. *Artificial Intelligence*, 1978, 10, 201-214. (a)
- Berliner, H. J. The B\* search algorithm: a best-first proof procedure. CMU-CS-78-112, Dept. of Computer Science, Carnegie-Mellon University, 1978. (b)
- Bernstein, A., Arbuckle, T., Roberts, M. de V., & Belsky, M. A. A chess playing program for the IBM 704. In *Proc. Western Joint Computer Conference*, 1958. New York: American Institute of Electrical Engineers, 1959. Pp. 157-159.
- Bratko, I., Kopec, D., & Michie, D. Pattern-based representation of chess end-game knowledge. *Computer J.*, 1978, 21, 149-153.
- Chang, C. L., & Slagle, J. R. An admissible and optimal algorithm for searching AND/OR graphs. *Artificial Intelligence*, 1971, 2, 117-128.
- Charness, N. Human chess skill. In P. W. Frey (Ed.), *Chess Skill in Man and Machine*. New York: Springer-Verlag, 1977. Pp. 34-53.

- de Champeaux, D., & Sint, L. An improved bi-directional search algorithm. In *IJCAI 4*, 1975. Pp. 309-314.
- de Champeaux, D., & Sint, L. An improved bi-directional heuristic search algorithm. *J. ACM*, 1977, 24, 177-191.
- Dijkstra, E. W. A note on two problems in connection with graphs. *Numerische Mathematik*, 1959, 1, 269-271.
- Doran, J. An approach to automatic problem-solving. In N. L. Collins & D. Michie (Eds.), *Machine Intelligence 1*. New York: American Elsevier, 1967. Pp. 105-123.
- Doran, J. E., & Michie, D. Experiments with the graph traverser program. *Proceedings of the Royal Society of London*, 1966, 294 (series A), 235-259.
- Elcock, E. W. Representation of knowledge in a geometry machine. In E. W. Elcock & D. Michie (Eds.), *Machine Intelligence 8*. New York: John Wiley & Sons, 1977. Pp. 11-29.
- Ernst, G., & Newell, A. *GPS: A Case Study in Generality and Problem Solving*. New York: Academic Press, 1969.
- Feigenbaum, E. A. Artificial intelligence: Themes in the second decade. In A. J. H. Morrell (Ed.), *Information Processing 68: Proc. IFIP Congress 1968 (Vol. 2)*. Amsterdam: North-Holland, 1969. Pp. 1008-1024.
- Feigenbaum, E. A., & Feldman, J. (Eds.) *Computers and Thought*. New York: McGraw-Hill, 1963.
- Fikes, R. E., Hart, P., & Nilsson, N. J. Learning and executing generalized robot plans. *Artificial Intelligence*, 1972, 3, 251-288.
- Fikes, R. E., & Nilsson, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 1971, 2, 189-208.
- Frey, P. W. An introduction to computer chess. In P. W. Frey (Ed.), *Chess Skill in Man and Machine*. New York: Springer-Verlag, 1977. Pp. 54-81.
- Fuller, S. H., Gaschnig, J. G., & Gillogly, J. J. Analysis of the alpha-beta pruning algorithm. Department of Computer Science, Carnegie-Mellon University, 1973.
- Garvey, T., & Kling, R. User's guide to QA3.5 question-answering system. Technical Note 15, AI Group, Stanford Research Institute, Menlo Park, Calif., 1969.
- Gaschnig, J. Exactly how good are heuristics? Toward a realistic predictive theory of best-first search. In *IJCAI 5*, 1977. Pp. 434-441.
- Gelernter, H. A note on syntactic symmetry and the manipulation of formal systems by machine. *Information and Control*, 1959, 2, 80-89.

- Gelernter, H. Realization of a geometry-theorem proving machine. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963. Pp. 134-152.
- Gelernter, H., Hansen, J. R., & Gerberich, C. L. A Fortran-compiled list processing language. *J. ACM*, 1960, 7, 87-101.
- Gelernter, H., Hansen, J. R., & Loveland, D. W. Empirical explorations of the geometry-theorem proving machine. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963. Pp. 153-163.
- Gelernter, H., & Rochester, N. Intelligent behavior in problem-solving machines. *IBM J. R&D*, 1968, 2, 336-345.
- Gelperin, D. On the optimality of A\*. *Artificial Intelligence*, 1977, 8, 69-76.
- Gillogly, J. J. The technology chess program. *Artificial Intelligence*, 1972, 3, 145-163.
- Gilmore, P. C. An examination of the geometry theorem machine. *Artificial Intelligence*, 1970, 2, 171-187.
- Good, I. J. A five-year plan for automatic chess. In E. Dale & D. Michie (Eds.), *Machine Intelligence 2*. New York: American Elsevier, 1968. Pp. 89-118.
- Greenblatt, R. D., Eastlake, D. E., & Crocker, S. D. The Greenblatt chess program. In *AFIPS Conference Proc., Fall Joint Computer Conference, 1967*. Washington, D. C.: Thompson Books, 1967. Pp. 801-810.
- Griffith, A. K. A comparison and evaluation of three machine learning procedures as applied to the game of checkers. *Artificial Intelligence*, 1974, 5, 137-148.
- Hall, P. A. V. Branch-and-bound and beyond. In *IJCAI 2*, 1971. Pp. 641-650.
- Harris, L. R. The bandwidth heuristic search. In *IJCAI 3*, 1973. Pp. 23-29.
- Harris, L. R. The heuristic search under conditions of error. *Artificial Intelligence*, 1974, 5, 217-234.
- Harris, L. R. The heuristic search and the game of chess: a study of quiescence, sacrifices, and plan oriented play. In *IJCAI 4*, 1975. Pp. 334-339.
- Harris, L. R. The heuristic search: an alternative to the alpha-beta minimax procedure. In P. W. Frey (Ed.), *Chess Skill in Man and Machine*. New York: Springer-Verlag, 1977. Pp. 157-166.
- Hart, P. E., Nilsson, N. J., & Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. SSC*, 1968, SSC-4, 100-107.
- Hart, P. E., Nilsson, N. J., & Raphael, B. Correction to *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. *SIGART Newsletter*, No. 37, December 1972, pp. 28-29.

- Hearst, E. Man and machine: chess achievements and chess thinking. In P. W. Frey (Ed.), *Chess Skill in Man and Machine*. New York: Springer-Verlag, 1977. Pp. 167-200.
- Hillier, F. S., & Lieberman, G. J. *Operations Research* (2nd ed.). San Francisco: Holden-Day, 1974.
- Jackson, P. C. *Introduction to Artificial Intelligence*. New York: Petrocelli, 1974.
- Karp, R. M. Reducibility among combinatorial problems. In R. E. Miller & J. W. Thatcher (Eds.), *Complexity of Computer Computations*. New York: Plenum Press, 1972. Pp. 85-103.
- Kister, J., Stein, P., Ulam, S., Walden, W., & Wells, M. Experiments in chess. *J. ACM*, 1957, 4, 174-177.
- Knuth, D. E., & Moore, R. W. An analysis of alpha-beta pruning. *Artificial Intelligence*, 1975, 6, 293-326.
- Kotok, A. A chess playing program. RLE and MIT Computation Center Memo 41, Artificial Intelligence Project, Massachusetts Institute of Technology, 1962.
- Kowalski, R. And-or graphs, theorem-proving graphs, and bi-directional search. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 7*. New York: John Wiley & Sons, 1972. Pp. 167-194.
- Lawler, E. W., & Wood, D. E. Branch-and-bound methods: A survey. *Operations Research*, 1966, 14, 699-719.
- Levi, G., & Sirovich, F. A problem reduction model for non-independent subproblems. In *IJCAI* 4, 1975. Pp. 340-344.
- Levi, G., & Sirovich, F. Generalized AND/OR graphs. *Artificial Intelligence*, 1976, 7, 243-259.
- Levy, D. The computer chess revolution. *Chess Life & Review*, February 1979, 84-85.
- Manove, M., Bloom, S., & Engelman, E. Rational functions in MATHLAB. In D. G. Bobrow (Ed.), *Symbol Manipulation Languages and Techniques*. Amsterdam: North-Holland, 1968. Pp. 86-102.
- Martelli, A. On the complexity of admissible search algorithms. *Artificial Intelligence*, 1977, 8, 1-13.
- Martelli, A., & Montanari, U. Additive AND/OR graphs. In *IJCAI* 3, 1973. Pp. 1-11.
- Michie, D. Strategy building with the graph traverser. In N. L. Collins & D. Michie (Eds.), *Machine Intelligence 1*. New York: American Elsevier, 1967. Pp. 135-152.
- Michie, D. A theory of advice. In E. W. Elcock & D. Michie (Eds.), *Machine Intelligence 8*. New York: John Wiley & Sons, 1977. Pp. 151-168.

- Michie, D., & Bratko, I. Advice table representations of chess end-game knowledge. In *Proc. AISB/GI Conference on Artificial Intelligence*, 1978. Pp. 194-200.
- Michie, D., & Ross, R. Experiments with the adaptive graph traverser. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 5*. New York: American Elsevier, 1970. Pp. 301-318.
- Minsky, M. Steps toward artificial intelligence. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963. Pp. 406-450.
- Mittman, B. A brief history of the computer chess tournaments: 1970-1975. In P. W. Frey (Ed.), *Chess Skill in Man and Machine*. New York: Springer-Verlag, 1977. Pp. 1-33.
- Moore, E. F. The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching, Part II*. Cambridge: Harvard University Press, 1959. Pp. 285-292.
- Moses, J. Symbolic integration. MAC-TR-47, Project MAC, Massachusetts Institute of Technology, 1967.
- Newborn, M. *Computer Chess*. New York: Academic Press, 1975.
- Newborn, M. The efficiency of the alpha-beta search on trees with branch-dependent terminal node scores. *Artificial Intelligence*, 1977, 8, 137-153.
- Newborn, M. Computers and chess news: recent tournaments. *SIGART Newsletter*, No. 65, April 1978, p. 11.
- Newell, A., & Ernst, G. The search for generality. In W. A. Kalenich (Ed.), *Information Processing 1965: Proc. IFIP Congress 65*. Washington: Spartan Books, 1965. Pp. 17-24.
- Newell, A., Shaw, J. C., & Simon, H. A. A variety of intelligent learning in a general problem-solver. In M. C. Yovits & S. Cameron (Eds.), *Self-organizing Systems*. New York: Pergamon Press, 1960. Pp. 153-189.
- Newell, A., Shaw, J. C., & Simon, H. A. Chess-playing programs and the problem of complexity. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963. Pp. 39-70. (a)
- Newell, A., Shaw, J. C., & Simon, H. A. Empirical explorations with the logic theory machine: A case history in heuristics. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963. Pp. 109-133. (b)
- Newell, A., & Simon, H. A. GPS, a program that simulates human thought. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963. Pp. 279-293.
- Newell, A., & Simon, H. A. *Human Problem Solving*. Englewood Cliffs, N. J.: Prentice-Hall, 1972.

- Newell, A., & Simon, H. A. Computer science as empirical inquiry: Symbols and search. The 1976 ACM Turing Lecture. *Comm. ACM*, 1976, 19, 113-126.
- Nilsson, N. J. Searching problem-solving and game-playing trees for minimal cost solutions. In A. J. H. Morrell (Ed.), *Information Processing 68: Proc. IFIP Congress 1968* (Vol. 2). Amsterdam: North-Holland, 1969. Pp. 1556-1562.
- Nilsson, N. J. *Problem-Solving Methods in Artificial Intelligence*. New York: McGraw-Hill, 1971.
- Nilsson, N. J. Artificial intelligence. In J. L. Rosenfeld (Ed.), *Information Processing 74: Proc. IFIP Congress 74*. Amsterdam: North-Holland, 1974. Pp. 778-801.
- Pitrat, J. A chess combination program which uses plans. *Artificial Intelligence*, 1977, 8, 275-321.
- Pohl, I. Bi-directional and heuristic search in path problems. SLAC Report No. 104, Stanford Linear Accelerator Center, Stanford, 1969.
- Pohl, I. First results on the effect of error in heuristic search. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 5*. New York: American Elsevier, 1970. Pp. 219-236. (a)
- Pohl, I. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1970, 1, 193-204. (b)
- Pohl, I. Bi-directional search. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 6*. New York: American Elsevier, 1971. Pp. 127-140.
- Pohl, I. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *IJCAI 3*, 1973. Pp. 12-17.
- Pohl, I. Practical and theoretical considerations in heuristic search algorithms. In E. W. Elcock & D. Michie (Eds.), *Machine Intelligence 8*. New York: John Wiley & Sons, 1977. Pp. 55-72.
- Polya, G. *How to Solve It* (2nd ed.). New York: Doubleday Anchor, 1957.
- Raphael, B. *The Thinking Computer*. San Francisco: W. H. Freeman, 1976.
- Reingold, E. M., Nievergelt, J., & Deo, N. *Combinatorial Algorithms: Theory and Practice*. Englewood Cliffs, N. J.: Prentice-Hall, 1977.
- Sacerdoti, E. D. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 1974, 5, 115-135.
- Samuel, A. L. Some studies in machine learning using the game of checkers. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963. Pp. 71-105.

- Samuel, A. L. Some studies in machine learning using the game of checkers. II--recent progress. *IBM J. R&D*, 1967, 11, 601-617.
- Sandewall, E. J. Heuristic search: Concepts and methods. In N. V. Findler & B. Meltzer (Eds.), *Artificial Intelligence and Heuristic Programming*. New York: American Elsevier, 1971. Pp. 81-100.
- Shannon, C. E. Programming a computer for playing chess. *Philosophical Magazine (Series 7)*, 1950, 41, 256-275.
- Shannon, C. E. A chess-playing machine. In J. R. Newman, *The World of Mathematics* (vol. 4). New York: Simon & Schuster, 1956. Pp. 2124-2133.
- Simon, H. A., & Kadane, J. B. Optimal problem-solving search: All-or-none solutions. *Artificial Intelligence*, 1975, 6, 235-247.
- Slagle, J. R. A heuristic program that solves symbolic integration problems in freshman calculus: Symbolic Automatic Integrator (SAINT). 5G-0001, Lincoln Laboratory, Massachusetts Institute of Technology, 1961.
- Slagle, J. R. A heuristic program that solves symbolic integration problems in freshman calculus. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963. Pp. 191-203. (Also in *J. ACM*, 1963, 10, 507-520.)
- Slagle, J. R. *Artificial Intelligence: The Heuristic Programming Approach*. New York: McGraw-Hill, 1971.
- Slagle, J. R., & Dixon, J. K. Experiments with some programs that search game trees. *J. ACM*, 1969, 16, 189-207.
- Slagle, J. R., & Dixon, J. K. Experiments with the M & N tree-searching program. *Comm. ACM*, 1970, 13, 147-154.
- Slate, D. J., & Atkin, L. R. CHESS 4.5--the Northwestern University chess program. In P. W. Frey (Ed.), *Chess Skill in Man and Machine*. New York: Springer-Verlag, 1977. Pp. 82-118.
- Thorp, E., & Walden, W. E. A computer-assisted study of Go on  $m \times n$  boards. In R. B. Banerji & M. D. Mesarovic (Eds.), *Theoretical Approaches to Non-Numerical Problem Solving*. Berlin: Springer-Verlag, 1970. Pp. 303-343.
- Turing, A. M., et al. Digital computers applied to games. In B. V. Bowden (Ed.), *Faster Than Thought*. London: Pitman, 1953. Pp. 286-310.
- Vanderbrug, G., & Minker, J. State-space, problem-reduction, and theorem proving--some relationships. *Comm. ACM*, 1975, 18, 107-115.
- Whitehead, A. N., & Russell, B. *Principia Mathematica* (2nd ed., Vol. 1). Cambridge: The University Press, 1925.
- Wilkins, D. Using plans in chess. To appear in *IJCAI* 6, 1979.



Winston, P. H. **Artificial Intelligence**. Reading, Mass.: Addison-Wesley, 1977.

## Index

- 15-puzzle 52, 55
- 8-puzzle 23, 38, 48, 51, 52
- A\* 49-50, 51, 52, 53-54, 62  
 abstraction space 104  
 ABSTRIPS 1, 20, 103, 104-107  
 add list 98, 99, 101, 102, 104  
 Adelson-Velskiy, G. M. 73, 79  
 admissibility 49, 61, 62, 64  
 admissibility condition 49, 51, 55  
 alpha-beta pruning 68-72, 77  
 Amarel, Saul 20  
 AND/OR graph 18, 27-29, 32, 57, 86, 91, 95  
 AND/OR graph search 41-45, 57-64  
 AND/OR tree 28, 42, 72  
 Atkin, L. R. 73, 77
- backed-up values 67  
 backgammon 79  
 backtracking 16, 106  
 backward reasoning 16-18, 26, 38, 42, 57, 83, 84  
 bandwidth condition 53  
 bandwidth search 46, 53-54  
 Berliner, H. J. 75, 76-77, 78, 80-81  
 Bernstein, A. 80  
 best-first search 46, 78  
 bidirectional search 17, 38-40, 55-56, 57  
 blind search 1, 21-22, 34-45, 47-48, 55, 84  
 bottom-up reasoning 17  
 branch-and-bound 49  
 branching factor 70, 75  
 breadth-first search 34-36, 43, 47, 51, 56, 84
- Chang, C. L. 62  
 checkers 19, 32, 72, 73, 74
- chess 16, 19, 32, 72-82  
 combinatorial explosion 19, 20, 45, 75  
 conjunctive subgoals 85, 91  
 consistency assumption 50, 53, 55  
 construction 93  
 control strategy 16  
 critical nodes 70  
 criticality value 104
- data-driven reasoning 17  
 database 1  
 de Champeaux, D. 56  
 dead position 67, 76  
 degree of a tree 70  
 delete list 98, 104  
 depth bound 37, 43, 75, 87  
 depth of a node 37  
 depth-first search 37-38, 43-44, 47, 77, 86, 106  
 difference 17, 86, 99  
 Dixon, J. K. 78  
 Doran, J. 51  
 dynamic ordering 78  
 dynamic weighting 52
- Ernst, G. 21, 86  
 evaluation function 47-48, 49, 51-54, 55-56, 60, 62, 64, 74, 92, 106  
 expansion of a node 34, 42
- Feigenbaum, E. A. 21  
 Feldman, J. 21  
 Fikes, Richard 31, 98  
 fixed ordering 70, 77  
 forward pruning 79  
 forward reasoning 16-18, 38, 42, 57  
 full-width search 78
- game tree 18, 32-33, 65  
 game-tree search 65-82

- Gelernter, Herbert 91  
General Problem Solver (GPS) 86-90, 99, 104  
generalized AND/OR graph 63  
generate-and-test 21-22  
geometry 91-93  
Gilligly, J. J. 74  
go 79  
goal 16, 26, 80, 86  
goal states 24  
goal-directed reasoning 17  
Graph Traverser 51  
Greenblatt, R. D. 73, 76, 79  
ground space 104
- Harris, L. R. 52, 53, 74, 78  
Hart, Peter 49  
heuristic information 45, 49, 57, 60, 72  
Heuristic Path Algorithm 51  
heuristic power 50  
heuristic search 1, 20, 21-22, 34, 45-64, 89, 91  
heuristic search method 45  
heuristics 20-22, 83  
hierarchical search 104  
hierarchical planning 104  
horizon effect 76
- information retrieval 16  
informedness of an algorithm 49  
initial states 24  
interdependent subproblems 43, 62-64  
intermediate OR node 28, 43  
iterative deepening 77
- Kadane, J. 52  
killer heuristic 78  
Knuth, D. E. 66, 68
- learning 74, 98
- length-first search 106  
Levi, G. 62  
live position 67  
Logic Theorist 17, 83-85, 86, 88, 91
- macro-operators 20  
MACROP 101  
Martelli, A. 50  
max cost 58, 59-60  
means-ends analysis 17, 46, 86, 89, 96, 99, 104  
method of analogies 79  
Michie, D. 51, 81-82  
minimax 65-68, 69, 70, 72, 75  
Minsky, M. 21  
Moore, R. W. 66, 68  
Moses, Joel 95  
mutilated chessboard problem 19
- negmax 66-67, 68  
Newell, Allen 21, 22, 80, 83, 86  
Nilsson, Nils 22, 31, 45, 47, 49, 60, 78, 98
- operator schemata 24  
operators 16, 23, 26, 57, 83, 86, 91, 94, 98, 104  
optimal solution 20, 48, 57  
optimality 50, 51, 62, 64  
ordered depth-first search 47, 77  
ordered search 46-48, 49, 55, 60-62, 63, 95
- partial development 46, 86  
partial expansion 46  
partial functions 24  
Pitrat, J. 81  
plan 81, 98, 100, 106  
planning 1, 20, 102, 104  
plausible-move generation 79  
ply 75

- Pohl, Ira 17, 39, 51, 52, 55  
 Polya, G. 21  
 potential solution 60-61, 62, 63  
 preconditions 98, 100, 104  
 predicate calculus 98  
 primitive problem 26, 28, 92  
 problem reduction 87, 91  
 problem representation 1-20, 23-33  
 problem solving 1, 45, 57, 83, 86, 91, 94, 98, 104  
 problem-reduction representation 18, 26-31, 41, 57, 86  
 propositional calculus 83, 88, 90  
 pruning 46, 67, 68, 77, 79, 92, 95, 99
- QA3 99  
 quiescence 75-77, 78
- random game tree 71  
 Raphael, Bertram 49  
 refutation move 78  
 robot problem solving 16, 98-107
- Sacerdoti, Earl 104  
 SAINT 94-95, 96-97  
 Samuel, A. L. 72, 73, 74  
 search 1, 18  
 search graph 19  
 search space 19-20, 45, 72  
 secondary search 76  
 Shannon, C. E. 72, 73-76, 78  
 Shaw, J. C. 21, 80, 83, 86  
 Simon, H. A. 21, 22, 52, 80, 83, 86  
 SIN 95-97  
 Sint, L. 56  
 Sirovich, F. 62  
 Siagle, James 62, 78, 94  
 Slate, D. J. 73, 77  
 SOLDIER 96  
 solution 24  
 solution graph 29, 41
- solution tree 29, 57-58, 60-61  
 solvable node 29  
 state 16, 23  
 state space 19, 23  
 state-space graph 18, 24-25, 32, 34, 47, 49, 57  
 state-space representation 17, 23-25, 26, 29-31, 34, 57, 86, 98  
 state-space search 21, 25, 34-40, 42, 45-56, 60, 62, 84  
 static evaluation function 67, 74, 76  
 STRIPS 1, 20, 31, 63, 98-103, 104, 106-107  
 successor node 18, 24, 34  
 sum cost 58, 59  
 symbolic integration 1, 16, 17, 90, 94-97  
 syntactic symmetry 92
- Table of Connections 87  
 tapered forward pruning 79  
 terminal node 28, 32  
 theorem proving 16, 19, 48, 57, 83, 88, 90, 91, 98  
 theorem-proving representation 18  
 tic-tac-toe 32, 72  
 tip node 62, 67  
 top-down reasoning 17  
 totally dependent game tree 71  
 Tower of Hanoi puzzle 26-27, 31  
 traveling-salesman problem 1, 25, 36, 48, 52, 53-54  
 triangle table 100-101
- uniform game tree 70-72  
 uniform-cost search 36-37, 39, 47, 49, 56  
 unsolvable node 29, 42
- Wilkins, D. 81  
 world model 16, 98, 104

# Representation of Knowledge

# The Representation of Knowledge

## Table of Contents

A. Introduction to Knowledge Representation . . . . .	121
B. Survey of Representation Techniques . . . . .	129
C. Representation Schemes . . . . .	137
1. Logic . . . . .	137
2. Overview of Semantic Networks . . . . .	151
3. Production Systems . . . . .	160
a. Overview of Production Systems . . . . .	160
b. Issues in the Design of Production Systems . . . . .	165
4. Procedural Representations of Knowledge . . . . .	173
5. Semantic Primitives . . . . .	183
6. Analogical Representations . . . . .	191
7. Frames and Scripts . . . . .	199
References . . . . .	208
Index . . . . .	218

## A. Introduction to Knowledge Representation

Artificial Intelligence research involves building computer systems capable of performing tasks like talking, planning, and playing chess. When we talk about people doing these things, we always talk about what they have to "know" in order to do them. In other words, we describe someone's ability to do something intelligent in terms of *knowledge*.

Similarly, we say that a computer program *knows* how to play cards, or understand spoken English, or manipulate a robot. In other words, knowledge is something that we *attribute* to programs in the same manner that we attribute it to each other, based on observing certain behavior. A *representation of knowledge* in AI is a combination of data structures and interpretive programs that, if used in the right way, will lead to "knowledgeable" behavior.

The nature of knowledge and intelligence has been pondered by psychologists, philosophers, linguists, educators and sociologists for hundreds of years. Since their research methodology involves the design of programs that exhibit intelligent behavior, AI researchers have often taken a rather pragmatic approach to this subject. Work on "Knowledge Representation" in AI has involved the design of several classes of data structures for storing information in computer programs, and the development of procedures that allow "intelligent" manipulation of these data structures to make inferences.

Keep in mind that a data structure is not "knowledge", any more than an encyclopedia is knowledge. We can say, metaphorically, that a book is a source of knowledge; but without a reader, the book is just ink on paper. Similarly we often talk of the facts or rules in an AI database as knowledge *per se*, when we really mean that a certain program uses the data structures to behave in a "knowledgeable" way.

Techniques and theories about knowledge representation have undergone rapid change and development in the last 5 years. The articles in this chapter try to give a general review of the different ideas that researchers have had, what they are good for, and what they are not so good for. Our understanding of these matters is still incomplete. This is one of the most active areas of AI research at the present time.

We hope that this article will help guide the reader's understanding of the various formalisms described in the Chapter. After a brief discussion of the kinds of "knowledge" we are talking about in particular, this article introduces some general issues that will serve as a vocabulary for talking about and comparing different representation methods. It should be stressed that these "dimensions" of representation schemes are neither complete nor orthogonal; in other words, there may be aspects of representation formalisms that are not accounted for by these dimensions, and the features themselves are interdependent.

The second article in the chapter is a brief survey of the most important representation formalisms, intended to give an overview of the kinds of systems we are talking about. The remaining articles describe, in more detail, the mechanics of the various representation schemes, their development, and some of the current research problems.

## Knowledge

What are the kinds of knowledge that we need to behave knowledgeably? What kinds of things is knowledge "about"? To get a handle on these questions, consider the following list of types of knowledge:

**Objects:** Typically, we think of knowledge as "facts" about objects in the world around us: *Birds have wings, Robins are birds, Snow is white*, etc. So, of course, there must be some representation of objects, classes or categories of objects, and descriptions of objects.

**Events:** We also know about acts and events in the world: *Bob kissed Mary behind the barn, The sky will fall tomorrow*. In addition to a representation for the events themselves, our formalism needs to somehow indicate the causes and effects of events, and their time course.

**Performance:** A behavior like riding a bicycle, a behavior involving "skill," must involve some sort of knowledge beyond objects and events. This is knowledge about *how* to do things: the performance of skills and procedures. Like bike-riding, most cognitive behaviors, eg. composing sentences and proving theorems, involve performance knowledge, and it is often hard to draw the line between performance- and object-knowledge. (Beware! Pushing too hard on this point leads right back to the fundamental philosophical issue of what knowledge is!)

**Meta-knowledge:** Another kind of knowledge that we seem to use is knowledge about the other knowledge we have, or meta-knowledge. Knowledge about knowledge includes facts about the extent, reliability, importance, and history of the facts we know about the world. Meta-knowledge seems to play a central role in human reasoning processes and is just becoming an active area in AI research in the representation of knowledge (Collins, 1979, Davis & Buchanan, 1977).

The questions of whether these kinds of knowledge are distinguishable or whether there are other varieties of knowledge, are interesting psychological issues. For now, however, we will leave the psychological aspects of the problem of knowledge behind (see Chapter Information Processing Psychology, on Information Processing Psychology, to see how the models described in this chapter have been applied to the psychology of memory.) In this article, we will discuss some of the features of AI representation schemes that make it more or less easy for computer programs to exhibit these various types of knowledge (objects, events, performance, meta-knowledge). The determining factor will be the way that knowledge is *used* in the system.

## Using Knowledge

The most important consideration in examining and comparing representation schemes is the eventual use of the knowledge. The goals of AI systems involve cognitive tasks like recognizing objects, answering questions, and manipulating robotic devices. But the actual use of the knowledge in these programs involves three stages: acquiring new knowledge, retrieving facts relevant to the problem at hand, and reasoning about these facts in search of a solution.



**Acquisition.** We usually describe *learning* in terms of the accumulation of knowledge, but it involves more than the addition of new facts to our brains. In fact, the acquisition of knowledge involves relating something new to what we already know in a complex and still mysterious way. AI systems often classify a new data structure before it is added to the database, so that it later can be retrieved when it is relevant. Also, in many kinds of systems, new structures can interact with old, interfering with tasks that had previously been performed properly. Finally, some representation schemes are concerned with acquiring knowledge in a form that is *natural* to humans, who serve as the source of new knowledge (see article Applications.B). If these processes do not occur during acquisition, the system can accumulate new facts or data structures without appearing to *know* anything new!

**Retrieval.** Determining what knowledge is relevant to a given problem becomes crucial when the system "knows" a lot of different things. Humans are incredibly proficient at this task, and most representation schemes that have been directly concerned with this issue have been based on ideas about human memory (see articles B2 and B7 and chapter Information Processing Psychology). The fundamental ideas that have been developed might be termed *linking* and *grouping*: if you know that one data structure is going to entail another in a predicted reasoning task, put in an explicit link between them; and if several data structures are typically going to be used together, group them into a larger structure.

**Reasoning.** When the system is required to do something that it has not been told explicitly how to do, it must reason, i.e. it must "figure out", from what it already "knows", what it needs to know. For instance, suppose a database retrieval program "knows" only that that *Robins are birds* and that *All birds have wings*. Remember that this means that it has data structures and programs which would allow it to answer the appropriate question properly:

<i>Are Robins birds?</i>	Yes
<i>Do all birds have wings?</i>	Yes

If we then ask it, *Do robins have wings?*, the program must "reason" to answer the query. In problems of any complexity, this ability becomes increasingly important: The system must be able to deduce and verify a multitude new facts beyond those it has been told explicitly.

For a given representation we must ask "What kind of reasoning is possible, easy, natural, etc., in this formalism?" There are many different kinds of reasoning one might imagine:

**Formal reasoning** involves the syntactic manipulation of data structures to deduce new ones following prespecified "rules of inference." Mathematical logic is the archetypal formal representation (see article B1).

**Procedural reasoning** uses simulation to answer questions and solve problems. When we ask a program *What is the sum of 3 and 4?*, it answers by using its procedural model of arithmetic (see article B4).

**Analogical reasoning** seems to be a very natural mode of thought for humans and, so far, difficult to accomplish in AI programs. The idea is that when you ask a question *Can robins fly?* the system might reason that "Robins are like sparrows, and I know sparrows can fly, so robins probably can fly." (See Article Problem Solving.C2 for a review of AI attempts at this kind of reasoning.)

Generalization and Abstraction) are also natural reasoning process for humans that are difficult to pin down well enough to implement in a program. If you know that *Robins have wings*, that *Sparrows have wings*, and that *Bluejays have wings*, eventually you will believe that *All birds have wings*. This capability may be at the core of most human learning, but it has not yet become a useful technique in AI.

Meta-knowledge can also be used in reasoning. To answer the question *What is Paul Newman's telephone number?* you might reason that "If I knew Paul Newman's number, I would know that I knew it, because it is a notable fact." This involves using "knowledge about what you know", in particular about the extent of your knowledge and about the importance of certain facts. Recent research in psychology and AI indicates that this kind of reasoning may be central to human cognitive processing (Gentner & Collins, 1979 and Flavell, 1979); furthermore some work on implementing this kind of knowledge in AI systems has also been done (Davis, 1976, Bobrow & Winograd, 1977, Brachman, 1978).

Two things need to be said about the uses of knowledge described here. First, they are very *interrelated*: When acquiring new knowledge the system must be concerned with how that knowledge will be retrieved and used later in reasoning. Second, when you get right down to it, *efficacy* is the primary consideration for knowledge-based AI systems: Although there is serious concern among AI researchers about the psychological validity of the various representation schemes, we are not yet in a position to prove that one scheme captures some aspect of human memory better than another. There is no *theory of knowledge representation*: We don't yet know why some schemes are good for certain tasks and not others.

We will now proceed to discuss some of the characteristics of representation schemes that have been used in the AI literature for describing and comparing the different formalisms.

### Scope and Grain

What portion of the external world can be represented in a system? In what detail are objects and events represented? And how much of this detail is actually needed by the reasoning mechanisms? Questions like these, concerning the *scope* and *grain* of a representation scheme, often are the first questions asked in determining the suitability of a given formalism. But they are not easy questions to answer.

For one thing, answers to these questions depend totally on the particular application of the representation formalism. The predicate calculus, for instance, might be an extremely fine grain representation in a mathematical reasoning program, but a crude approximation for a vision program. Exactly how much detail is desirable depends, of course, on the intended use (See McCarthy & Hayes, 1969). In general, uniformity of detail across the objects and events is desirable for a given reasoning task about those objects and events (Bobrow, 1975).

If one asks "Can everything I want be represented in the formalism?", the answer is almost always "Yes, but some things are more easily represented than others." Getting a

feeling for what it means "to be represented more easily"--which involves the representation, the domain, and the reasoning strategies--is part of the art of doing AI research these days. There is no formal metric for measuring the appropriateness of a formalism along these lines. Bobrow (1975) refers to the process of *mapping* the objects and events in the world in some internal encoding; then one can ask the mapping in a given situation is "easy," "natural," "psychologically valid," etc.

When one is concerned with representing things that are not objects in the world, like other people's beliefs, these issues become much more complicated.

### Modularity and Understandability

If one thinks of the data structures in a program as "pieces of knowledge", then adding new data structures is like adding knowledge to the system. One characteristic that is often used to compare representation schemes is *modularity*, which refers to the ability to add, modify, or delete individual data structures, with clear effects on what the system "knows", independent of the remainder of the database.

In general, humans find modular, or decomposable, systems easier to understand and work with (Simon, 1969). For example, the following situation will be familiar to readers who have helped to write large computer programs. A large system is composed of many procedures that "call" each other in a complex way, which becomes increasingly hard to follow as the system grows. Often modification of Procedure X, so that it will work properly when called by Procedure A, interferes with the proper functioning of X when it is called by Procedure B. In other words, in order to successfully modify a large system, the programmer must understand the interactions of all of its pieces, which can become an impossibly hard task.

Procedures are the data structures in *procedural representations* (see article B4), which are notoriously non-modular. In other representation schemes there are other data structures (production rules, logic formulae, etc.) which seem less inherently intertwined, but the control of the interaction of the various database entries is a very important characteristic of all representation schemes. No system is completely *modular*--i.e., in all systems there is some degree of interaction between the data structures that form the knowledge base. But some formalisms (predicate calculus, production systems) are more modular than others (procedural, semantic nets) (Winograd, 1975).

Here are some of the pros (and cons) of modularity:

**Context independence.** In a modular system, each fact is interpreted more or less globally--e.g., the new facts that can be deduced from a set of facts are always the same--whereas in less modular systems the current *context* may play an important role in the inference mechanism. This causes the "meaning" of a data structure to be *context-dependent* in a non-modular system, so that the "pieces of knowledge" metaphor is less appropriate.

**Modifiability.** This aspect refers to how adding, modifying, or deleting database entries affects the rest of the database. In the "ideal" modular system, facts can be added independent of the rest of the data structures. As described

above, in procedural systems, which are archetypically non-modular, a change to one procedure affects not only its own performance, but in directly affects all those procedures that "call" it.

**Codability.** On the other hand, procedural systems are often more directly codable--i.e., they correspond to our intuitive notions of how to do the task. The designer of a modular-type system for a given domain, like an expert system based on production-rules (see chapter Applications), typically spends a great deal of time finding ways of forcing the system to use the right rule at the right time. In general, our human understanding of a cognitive domain, like speaking a foreign language or playing tennis or doing mathematics, is complex, but we tend to communicate our understanding in terms of modular "rules" or sentences.

What aspects of a representation formalism are easy for humans to understand? There is a great difference between modular and not-so-modular systems: Winograd (1974) generalizes that in modular systems the facts are easy to recognize but the reasoning process may be quite opaque, and the opposite is often true in procedural representations. The degree to which the system is understandable by humans is important in several phases of its development and performance: Design and implementation, acquisition of knowledge from human experts, performance of the task, and in interaction and explanations for the eventual user. The modularity of a given representation scheme may be very helpful in increasing the understandability of the system.

#### **Explicit Knowledge and Flexibility**

Another issue to keep in mind when examining various representation schemes is what part of the system's knowledge is explicit. By this we mean: to what "facts" do the programmer (and the system) have direct, manipulatory access and what knowledge is "built-in". For example, an operating system has an explicit representation its the "priority queues", but its "knowledge" about scheduling jobs is typically hidden deep in the code. It is there, of course, but it is much more opaque than a set of scheduling "rules" would be, for example.

One particular advantage of explicit representation schemes is that, because the facts are in a form that allows a global interpretation, the same fact can be used for multiple purposes. In some large systems this feature has been a significant advantage. For example, in MYCIN (article Medical Applications.B) the production rules that form the system's knowledge about how to diagnose the possible causes of infectious diseases, are used not only by the diagnosis system itself, but also by the systems that explain the diagnosis system's reasoning to the consulting physician and that acquire new rules from expert physicians (Davis & Buchanan, 1977).

#### **Declarative vs. Procedural Knowledge**

On a closely related subject, the dispute over the relative merits of declarative vs. procedural knowledge representations is a historically important battle line over which much of the current state of representation theory was painfully developed around 1970 (Winograd, 1975). Many of the issues discussed in this article are aspects of the

procedural/declarative debate. The declarative systems were typified by resolution-based theorem provers (see Chapter Theorem Proving), and the procedural systems by Winograd's PLANNER-based SHRDLU (Articles AI Languages.C2 and Natural Language.F5). The Declarativists talked about the *flexibility* and *economy* of their representation, its *completeness* and the *certainty* of its deductions, and the ease of *modifiability* of the systems. The Proceduralists stressed directness and ease of *coding* reasoning procedures, as well as the *understandability* of the reasoning processes.

Although in retrospect these positions seem somewhat arbitrarily chosen over the space of possible features of representation schemes, the battle was an important one in AI. It dissolved, rather than resolved, and the result was a much greater respect for the importance of representation theory in current AI work.

### Meta-knowledge

One of the concerns of the more recent representation systems is how to use knowledge about knowledge: self-knowledge or meta-knowledge. A good example of human use of meta-knowledge is the tip-of-the-tongue phenomenon: When you run into someone you have met once before, it is sometimes the case that you can't remember his name. You may remember very well the first meeting, that he is the brother-in-law of your wife's boss, that he has a foreign-sounding name or that his name rhymes with spaghetti. And you might use all of this knowledge in trying to recall his name. This is the kind of introspective evidence that suggests a major role for meta-knowledge in human reasoning. And several AI representation formalisms have begun to incorporate schemes for explicitly stating meta-knowledge about data structures.

Implicit meta-knowledge was a common and necessary characteristic of previous formalisms. Wired into the systems were methods for determining many of the meta-level facts that are useful for reasoning. Here is a partial list of the kinds of things that might be useful to know, about what we know:

**Extent.** In particular, knowledge about what we don't know, as in the telephone number examples discussed above.

**Defaults.** We have expectations about objects and events that we encounter that can be viewed as default values for the parameters of new data structures.

**Criteriality.** How important some facts are relative to a given conclusion: A trunk is more indicative of an elephant than are four legs.

**Relevance.** Explicit knowledge about what kinds of knowledge may be relevant to certain tasks. This may come in the form of links between data structures, or groupings of structures into larger structures.

**History.** Knowledge about the source, age, and development of the data structure.

**Reliability.** How certain this knowledge is. If it is contradicted by other observations and facts, when it should be abandoned.

**Representation.** Knowledge of the representation formalism is always available implicitly, since it is necessary before the data structures can be modified. But explicit information about the representation is also possible, and it is especially useful for multiple uses of the same data structures.

**Meta-cognitive knowledge.** This is knowledge of the capabilities of the system. We humans learn this kind of information from experience (Flavell, 1979), but AI systems might also benefit from knowledge of their own performance. As an example, Bobrow (1975) discusses a robot who is planning a trip. His knowledge that he can read the street signs along the way to find out where he is is meta-cognitive knowledge.

### Final Remarks

This article has not been about representation formalisms per se, but a discussion of the pragmatics of epistemology, the study of the nature of knowledge. The intention has been to lay the groundwork for an appreciation of the problems inherent to incorporating knowledge in AI programs. This discussion may also help guide a critical comparison of the representation methods so far developed.

The following article is an quick overview of most of the representation schemes. The remaining articles in the chapter go into substantial detail about individual representation schemes, discussing their development, their technical features, their use in AI systems, and their shortcomings.

## B. Survey of Representation Techniques

As discussed in Article A1, AI research has had to deal in "experimental epistemology." In order to create programs that exhibit intelligent behavior, researchers in AI have had to develop schemes for incorporating and manipulating knowledge about the world in their programs; sometimes these representation schemes have been rather pragmatic or superficial, but sometimes they have touched on the very core of the nature of knowledge and cognition. This article reviews many of the representation schemes that have been used in AI programs that play chess, converse in English, and operate robots. Most of this research assumes that *what* needs to be represented is known a priori by the researcher, and his job is just figuring out *how* to encode the information in the system's data structures and procedures. The remaining articles in the chapter describe in some detail the development and current state of each of the representation techniques surveyed here.

### Special Purpose Representation Techniques

Some problems that AI researchers attack seem to supply their own natural representation for the knowledge required to do the problem. For example, a visual scene from a robot's camera is often encoded as an array representing a grid over the scene. The values of the elements of the array represent the average brightness over the corresponding area of the scene (see Chapters Vision and Robotics on Vision and Robotics). This representation is useful for some tasks, like finding the boundaries of the objects in the scene, but is clumsy for other tasks, like counting the number of objects. In the latter case a list, each element of which represents one object indicating its location, orientation and size, might be a more useful representation. (See the discussion in Bobrow, 1975).

This example illustrates a very important principle to keep in mind when comparing representation techniques: In some sense, these two (and all other) representation methods are interchangeable: If we know one representation in enough detail, we could for the most part construct the other one. It is the intended *use* of the knowledge about the scene that is the factor that recommends one representation scheme over another. In a big AI system, like the Speech Understanding programs, multiple representations of the same information may be used simultaneously for different purposes (see Chapter Speech).

Other special purpose representation schemes of particular interest are those used in the early Natural Language processing programs, like SAD-SAM and SIR (see Natural Language.F1), and the *discrimination net* used in the EPAM program (see Article Information Processing Psychology.B2).

### State-Space Search

Perhaps the earliest representation formalism used extensively in AI programs was the state-space representation, developed for problem-solving and game-playing programs. The basic idea is that from a given state in a problem or game, all possible next states can be determined using a small set of rules, called *transition operators* (or *legal move generators* in game playing programs.) For example, in a chess game, the original state is the board position at the beginning of the game. The move generators are just the rules for moving each piece.

So, all of the next states of the game (i.e., the possible board configurations after White's first move) can be generated by applying the move rules to the original positions of the pieces. Similarly, all of the possible states after Black's first response can be generated.

The space generated by this procedure can be viewed as a tree, called the *game tree*. In chess, the *root* of the tree is the starting board configuration. One branch is generated for each of White's possible moves. And from each of these offspring nodes, a branch is generated for each of Black's responses, etc. The leaf nodes of the tree correspond to checkmates or draws. Usually there are a great many moves possible at each node of the tree, some good and some bad, and the object of state-space search techniques is to find the "best" move, corresponding to a path through the tree that is most likely to bring success.

One rather straightforward way to find the winning move is to try all of the alternative moves, then try all of the opponents responses to these moves, then all of the possible responses to those, until all of the possible continuations of the game have been exhausted and it is clear which was optimal. The problem here is that for interesting problems, like chess, there are far too many possible combinations to try: the *combinatorial explosion*, as it has been called.

Basically, the solution adopted in AI research is to limit the number of alternatives examined at each stage of the look-ahead process and the *depth* examined for the consequences. The selection of nodes to examine is based on rules, called *heuristics*, that represent "educated guesses" about which are the most promising alternatives in certain classes of situations.

For a thorough discussion of search techniques see Chapter Search and Nilsson (1971). Newell & Simon (1972) have described human problem-solving behavior in terms of state-space search (see Article Information Processing Psychology.C). The knowledge represented in a search tree is the structure of a problem in terms of the alternatives available at each possible state of the problem. The issue is then how to make the best choice with a limited amount of computational power. In problem domains like game playing and puzzle solving, the state-space representation seems to capture the structure of the knowledge used. But in other situations, where for instance the *legal moves* are not clearly defined or too numerous to even list, the representation must incorporate the actual domain knowledge that is available.

### Logic

The classical approach to representing the knowledge contained in sentences like  
*All birds have wings*

is the predicate calculus, developed by philosophers and mathematicians as a formalization of the process of making inferences from facts. The example about birds' wings would be translated into the mathematical formula

$$\text{FOR-ALL } x. \text{ bird } (x) \Rightarrow \text{ HasWings } (x)$$

which reads: *For any object, x, in the world, if x is a bird, then x has wings.* The advantage of



using a formal representation is that there is a set of rules associated with the predicate calculus, called the *rules of inference*, by which facts that are known to be true can be used to derive other facts which *must* also be true. Furthermore, the truth of any statement can be checked, in a well specified manner, against the facts that are already known to be true. For example, suppose we add another fact to our database

$$\text{FOR-ALL } x. \text{ Robin } (x) \Rightarrow \text{Bird } (x)$$

which reads: *For any object, x, in the world, if x is a Robin, then x is a bird.* Then from these two facts, we can conclude that all robins have wings:

$$\text{FOR-ALL } x. \text{ Robin } (x) \Rightarrow \text{HasWings } (x)$$

Note that there is a specific rule of inference that allows this deduction, and that the system of formal logic assures us that the new fact is true so long as the original facts were true (see Article B1 on Logic).

One reason that this approach to the manipulation of knowledge has been so popular in AI systems is that this derivation of new facts from old can be mechanized. Using automated versions of proof techniques like the *resolution method*, programs have been written to grind through a database of facts represented as sentences in the predicate calculus, to determine the validity of a new statement of the same form. Mechanistic theorem provers of this sort have been used with some success in programs with relatively small databases. However, when the number of facts becomes large, we have to somehow search through them to decide which rules to apply to which facts at each step of the proof. This can be thought of as the same kind of state-space search we discussed above. At each step of the proof, all of the possible applications of any rule to any combination of facts must be considered. And we face the same combinatorial explosion: The number of alternative things to do at each step grows exponentially with the number of facts.

Heuristic search techniques have been applied to theorem provers to help alleviate the difficulties of finding the relevant facts. And AI languages, like PLANNER and QA4, automate certain pattern-directed proof procedures, allowing *procedural* specification of heuristics for guiding the proof in terms of routines that are called when a decision must be made between alternatives (see article B4 and Chapter AI Languages). Although success with real-world size problems (involving large databases) has remained limited, theorem proving is still an active area of AI research (see Chapter Theorem Proving).

The predicate calculus is a very general representation scheme. Many kinds of facts seem to be naturally represented as statements, and modern work in Logic has extended the formalism through attempts to encode both knowledge about beliefs and second-order knowledge about expressions in the formalism. The generality of the formalism comes from its close resemblance to the way we communicate knowledge in language. But, like language, the predicate calculus is not a completely general vehicle for expression. For example, imagine how a verbal description of how to ride a bicycle compares to the knowledge that is necessary to perform the act.

Unlike language, the predicate calculus and related formal systems are "guaranteed correct" to an extent that other representation schemes have not yet reached: The

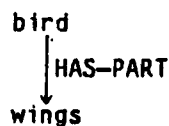
semantic *entailment* of a statement in the calculus (i.e., the inferences or conclusions that can be drawn from that statement) is completely specified by the *rules of inference*. Theoretically, the database can be kept logically consistent and if this is done, all conclusions drawn are guaranteed consistent. A definition and guarantee of logical consistency is a feature that other representation schemes are still striving for.

### Procedural Representation

Procedural representations of knowledge have found increasing popularity and utility since research on the PLANNER programming language, first used in Winograd's famous SHRDLU system (see Articles AI Languages.C2 and Natural Language.F5). In a procedural representation knowledge about the world is contained in the programs that know how to do things. For instance, in a parser for a natural language understanding system, the knowledge that a *noun phrase* may contain articles, adjectives, and nouns is represented in the program by calls (within the NP procedure) to routines that know about articles, nouns, and adjectives, respectively. The underlying knowledge, the grammar in this example, is never stated explicitly, and thus is not typically extractable in a form that humans can easily understand. The difficulty that humans have in verifying and changing procedural representations is the major flaw of these systems. Nevertheless, all AI systems use a procedural representation at some level of their operation, and there is a general consensus that there is a legitimate role for procedural representation in AI programs (Winograd, 1975; and see article B4).

### Semantic Nets

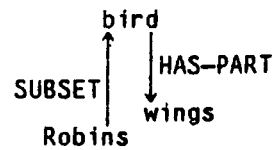
The semantic net, developed by Quillian (1968) and others, was invented as an explicitly psychological model of human associative memory. A net consists of nodes representing objects, concepts and events, and links between the nodes, representing their interrelations. Consider, for example, the simple net:



where *bird* and *wings* are nodes representing sets or concepts, and *HAS-PART* is the name of the link specifying their relationship. Among the many possible interpretations of this net fragment is the statement:

*All birds have wings*

As illustrated earlier, statements of this sort also have a natural representation in predicate calculus based representation systems. One key feature of the semantic net representation is that the important associations are often made explicitly and succinctly: relevant facts about an object or concept can be inferred from the nodes to which they are directly linked, without, for instance, a search through a large database of rules pertaining to the object. This is particularly true with respect to ISA and SUBSET links, which establish an *inheritance hierarchy* in the net. For example, the net segment:



might be interpreted to mean that since robins are birds, and birds have wings, then robins have wings.

The interpretation (semantics) of a net, however, depends solely on the program that manipulates it; there are no conventions about the meaning of a net structure. Therefore, inferences drawn by manipulation of the net are not assuredly *valid*, in the sense they were assured valid in a predicate calculus representation scheme.

Recent work on semantic network representations has followed two separate lines of research. One continues the psychological modeling aspects of the nets, as in the work of Norman & Rumelhart (1975) and Anderson (1976); the other involves establishing more formal semantic notions for network structures. The work of Woods (1975), Brachman (1976), and Hendrix (1975) is of particular interest in this regard.

One common feature of these more recent network schemes is the case-frame structure for concepts in the network. Nodes representing concepts and events inherit from their superiors (nodes higher in the inheritance hierarchy) the set of links they must have, and possibly some *default* links to other nodes. For example, a node that is intended to represent GIVING might be instantiated with GIVER, RECEIVER, and OBJECT-GIVEN links inherited from the GIVING concept node. This development arises from concern with case-like structures in English (see Article Natural Language.B4). It is useful in structuring knowledge both for comparing objects and events and for identifying missing pieces of information. This concern with inheritance of links is related to semantic primitives and frames, discussed below.

### Semantic Primitives

A characteristic feature of semantic nets is that there are generally many possible representations in the net for a single fact or event. For instance, the fact that robins have wings could be represented by the example net discussed above ("All robins are birds. All birds have wings."), or more simply by linking the *wings* node directly to the *robins*:



This inherent ambiguity is a general problem in network construction, and the programs that manipulate the net to make deductions are typically very flexible in terms of the exact

structures they look for. Although this feature might be used to great advantage, the ability to store things in the net redundantly, with an eye to future relevance, is usually outweighed by the confusion and expense caused by not knowing how a given fact might be stored when it comes time to look for it. In particular, the paraphrase task popular in natural language work research, where the program rephrases an input sentence or checks whether two sentences have the same "meaning", led researchers like Norman & Rumelhart (1975) and Schank & Abelson (1977) to use canonical internal representations based on *semantic primitives*. The technique used to establish a canonical representation reduces all structures to a set of primitives in such a way that two structures that mean the same thing reduce to the same network of primitive nodes. Current work by Wilks (1977b) on machine translation between English and French (see Article Natural Language.E2) also uses a set of primitive acts to establish an intermediate representation, or *interlingua*, for the underlying meaning of sentences in the two languages.

### Production Systems

One feature of a representation scheme that is generally valuable is *modularity*, the ability to distinguish and separately manipulate distinct pieces of knowledge within the database. Humans find modular databases easier to understand and work with, since adding, removing, and replacing various *facts* is a natural way for us to think about knowledge. The predicate calculus formalism, for example, is archetypically modular, since the statements and rules of inference can all be manipulated separately. A procedural representation, on the other hand, is notoriously unmodular, since a change to any one procedure will not only affect its performance, but will also change the results of all the procedures that "call" the modified one.

Production systems, developed by Newell & Simon (1972) for their models of Human Information Processing (see Chapter Information Processing Psychology), are a modular representation scheme that is finding increasing popularity in large AI programs. The basic idea of these systems is that the database consists of rules, called productions, in the form of condition/action pairs: "If this condition occurs, do this action."

For example, consider the following set of productions, which will PRINT the factorial of some positive integer, I:

1) N unbound	—> SET N TO I
2) N=1	—> PRINT I and STOP
3) ALWAYS	—> DECREMENT N and SET I TO I*N

Typically, the productions are executed according to some very simple sequencing rule like the following:

Cycle through the productions in order. The first time the condition (left-hand side) is found to be TRUE, do the corresponding right-hand side, and then restart the cycle at the first production.

Alternatively, the productions can be "fired in parallel", as in the following sequencing rule:

Find all of the productions whose left-sides are TRUE. Then choose one of the right-hand sides based on a set of rules about choosing appropriate productions.

Note that the sequencing strategy specified is critical. Current work on production systems has emphasized the control aspects of the formalism and the ability to develop self-modifying (learning) systems (see article B3).

Production systems have been found useful as a mechanism for controlling the interaction between pieces of declarative and procedural knowledge. Because they allow systems with large amounts of knowledge to be understood and modified in a modular fashion they have been used in current large applications systems like DENDRAL (Article Chemistry Applications.C), MYCIN (Article Medical Applications.B) and PROSPECTOR (Article Applications.D2).

#### Frames -- KRL

The latest direction in AI representation schemes is the *frame*. This formalism for knowledge representation is in its early development stage, with several projects exploring their own frame-like ideas. Basically a frame is a data structure that includes declarative and procedural information in pre-defined internal relations. Thus a frame for a dog might have "knowledge hooks", or *slots* for facts that are typically known about dogs, like the SPECIES, OWNER, NAME, and an "attached procedure" for finding out who the owner is if that is not known. In the frame-like language KRL, a dog-frame might look like this:

#### DOG

```
SELF:  an animal; a pet
SPECIES:
OWNER:  a person
        (TO FILL: find a person with pet=myself)
NAME:  a proper name; DEFAULT Rover
```

#### DOG-1

```
SELF:  a dog
SPECIES: mutt
Owner:  Jimmy
NAME:  Fido
```

The semantics of this example, as well as the ideas being developed in frame-like formalisms are discussed in Article B7.

An interesting feature of the frame is its capability to determine whether it is applicable in a given situation. If it is not, it could transfer control to a more appropriate frame (Minsky, 1975). There seem to be many issues about the possible implementations of frame-like systems that are unresolved, and there are probably other issues that haven't surfaced yet in this new area of AI research. But the basic idea of structuring units of knowledge, which is an extension of the case-frame structures discussed above, seems to be a useful idea.

### Conclusion

This brief summary indicates the variety of representation techniques being used in AI projects, and the remaining articles in this Chapter go into most of these schemes in greater detail. Article A1 deals with the *issues* discussed in representation theory, in an attempt to offer some ideas for comparing the various techniques. They each have strengths and weaknesses in different contexts. Many researchers feel that the representation of knowledge is the key issue at this point in the 20-year history of the field. Knowledge representation is also one area where AI and Cognitive Psychology share fundamental concerns, for the brain seems to use a variety of representation schemes whose natures are at present poorly understood, or completely unknown. The interested reader should peruse Chapter Information Processing Psychology on Information Processing models in Psychology.

## C. Representation Schemes

### C1. Logic

The study of formalized intelligence did not begin with the invention of the stored program computer. Rather, philosophers have been grappling with the problem of formalizing reasoning and knowledge of the world since the time of the ancient Greeks. This tradition, begun in the Academy and with the Sophists, formalized in the last half of the nineteenth century with the work of men like Boole, Frege and Russell, and expanded and amplified in the current century by philosophers such as Quine, Carnap and Tarski, is an important part of western intellectual history. This study of the possible modes of knowledge and reasoning has developed into the philosophical and mathematical study of logic. Artificial intelligence can build upon this accumulated work.

This article is about logic. The next two sections, dealing with the propositional and predicate logic, are an introduction to formal logic for the naive user. This particular introduction has been written with the artificial intelligence applications of logic in mind. The fourth section gives an example of an elementary expression of world knowledge in a formal logical formalism. Fifth is a survey of actual Artificial Intelligence systems which use logic. The final section considers logic in the context of artificial intelligence itself: how logic is used in A.I., and what the A.I. issues involving logic are.

#### The propositional calculus

Perhaps the oldest and most general representational structure applied to Artificial Intelligence is that of formal logic.

Logic, as used in A.I., has two important and interlocking branches. The first is consideration of "what you can say"; what relationships and implications one can formalize, the *axioms* of a system. The second is the deductive structure, the *rules of inference*. If one considers the "state-space search" paradigm for A.I. (see *Search Overview*), the axioms define the "states", and the rules of inference, the legal transitions between those states.

Logic is quite literally a *formal* endeavor. It is concerned with the *form* (syntax) of statements, and whether, by their *form*, they imply *TRUE* conclusions.

There are currently many different varieties of logic, such as *propositional calculus*, *predicate calculus* and *modal logic*. These are not mutually exclusive alternatives. Rather, logic systems result from building. One starts with a given notion, such as *propositional logic*, and, by inclusion of additional notions, for example, *predication and quantification*, extends to a stronger, more expressive logic. It therefore behooves our explanation of logic to "start at the bottom", and work upwards.

The most fundamental notion in logic is that of *truth*. A properly formed statement has one of two different truth values, *TRUE* and *FALSE*. (It is possible to extend to *multiple truth valued* logics; several A.I. systems have. However, discussion of multiple truth valued logics is beyond the scope of this article). So, if we have propositions *X*, *Y* and *Z*, we might say that *X* and *Y* are *TRUE* (have truth value *TRUE*) while *Z* is *FALSE*. Typical propositions

are "Bob's car is blue", "Seven plus six is twelve", and "Mary is John's uncle". Note that each of the quoted sentences is a proposition itself, not to be broken down into its constituent parts. Thus, we could assign truth value "TRUE" to "Mary is John's Uncle", with no regard for the *meaning* of "Mary is John's Uncle". Propositions are those things which we can call true or false. Terms such as "John's uncle" and "Seven plus four" would not be propositions, as we cannot assign a truth value to them.

Now, pure, disjoint propositions aren't very interesting. We obtain the simplest logic, *Propositional Calculus* by introducing *sentential connectives* to join propositions. There are five commonly employed connectives:

And	$\wedge$
Or	$\vee$
Not	$\neg$
Implies	$\supset$
Equivalent	$\equiv$

Depending upon the context, one often sees  $\wedge$  written as  $\&$ , and  $\supset$  as  $\rightarrow$ .

Intending that these connectives keep their *natural* interpretations, we shall define them as follows:

$X \wedge Y$  is TRUE if both  $X$  is TRUE and  $Y$  is TRUE and otherwise it is FALSE.

$X \vee Y$  is TRUE if either  $X$  is TRUE or  $Y$  is TRUE or both are TRUE and is FALSE if both  $X$  and  $Y$  are FALSE.

$\neg X$  is TRUE if  $X$  is FALSE, and FALSE if  $X$  is TRUE.

$X \supset Y$  is TRUE if  $Y$  is TRUE or  $X$  is FALSE or both.  $X \supset Y$  is meant to be the propositional calculus rendition of the notion, *If we assume that  $X$  is true, then  $Y$  must be so.* We use this concept in everyday speech with statements like *If Jenny is nine months old, then she can't do calculus.* Note that if Jenny can do calculus, or if Jenny is not nine months old, then this statement is true.

$X \equiv Y$  is TRUE if both  $X$  and  $Y$  are TRUE, or both  $X$  and  $Y$  are FALSE, FALSE if they are different.

This table, a compressed *truth table* summarizes these definitions: Here we use T for TRUE, and F for false.

X	Y	$X \wedge Y$	$X \vee Y$	$X \supset Y$	$\neg X$	$X \equiv Y$
T	T	T	T	T	F	T
T	F	F	T	F	F	F
F	T	F	T	T	T	F
F	F	F	F	T	T	T

From combinations of these variables and connectives, we can build sentences of propositional logic, just like the expressions of mathematics. Typical sentences are:



$$(X \supset (Y \wedge Z)) \equiv ((X \supset Y) \wedge (X \supset Z)) \quad (1)$$

$$\neg(X \vee Y) \equiv \neg(X) \wedge \neg(Y) \quad (2)$$

$$(X \wedge Y) \vee (\neg Y \wedge Z) \quad (3)$$

(Parentheses are used here just as in ordinary algebra.)

Of these three examples, (1) is a *tautology*. A tautology is a sentence that is always true. That is, no matter what values are assigned to the sentential constants (another name for variables that are either true or false), the sentence itself is always true.

The second example is a *fallacy* or contradiction. No matter what assignment of values is used, the value of the sentence is always false. The third example is neither a tautology nor a fallacy; it is true in some models (for example, when X, Y, and Z are all true) and false in others (such as when X, Y, and Z are all false). Logicians are generally less interested in this third kind of sentence than in either tautologies or fallacies, since they are not as useful in proofs.

In propositional calculus, we also encounter the first *inference rules*. An inference rule allows the deduction of a new true sentence from several previous true sentences. The most common (and famous) propositional calculus inference rule is that of *modus ponens*. It states that if we have sentences of the form  $X \supset Y$  and  $X$  then we can *deduce* the new sentence  $Y$ . Thus, if we know that *if John is an uncle then John is male* and *John is an uncle* we can conclude *John is male*.

In *natural deduction* systems for logical inference, there are typically one inference rule for the introduction, and another for the removal (elimination) of each of sentential connectives. Thus, the rule *modus ponens* is the  $\supset$  elimination rule, since it allows us to eliminate one instance of  $\supset$ .

### Predicate Calculus

For the practical purposes of A.I., propositional logic is not very useful. It seems at least necessary, if we are to describe our knowledge of some world, to be able to speak of individuals, to be able to postulate relationships between these individuals, and to be able to generalize these relationships over classes of individuals. We turn to the *predicate calculus* to accomplish these objectives.

The predicate calculus is an extension of the notions of the propositional calculus. The meanings of the connectives ( $\wedge$ ,  $\vee$ ,  $\supset$ ,  $\neg$ , and  $\equiv$ ) are retained, but the focus of the logic is changed. Instead of looking at sentences that are of interest merely for their structure, predicate calculus is used to make statements about specific objects. We call these objects *individuals*. Examples of individuals are *you*, *this paper*, *the number 1*, *the queen of hearts*, *Socrates*, and *that coke can*.

We will wish to make statements about the individuals, both by themselves, and in relation to other individuals. The statements are called *Predicates*. A predicate is applied to

a specific number of arguments, and has a value of either TRUE or FALSE when used with that many individuals. Predicate calculus, like propositional calculus, is a two valued logic; it has no "maybe".

An example of a predicate of one argument is the predicate *Is-Red*. Of the individuals mentioned above, *Is-Red* is TRUE of *the queen of hearts*, and *that coke can*, and FALSE of *this paper*, *Socrates*, and *I*. The solution of *Is-Red(you)* is left as an exercise for the reader. Other predicates of one argument are *Negative* (is X less than zero?) *Greek*, *Mortal*, and *Made-of-paper*.

Predicates can have more than one argument. An example of a two place predicate from mathematics is *Is-greater-than*. Real objects could be compared by the two place predicate, *Is-lighter-than*. A three place predicate from mathematics might be *Pythagorean*(X Y Z), which would be true whenever X and Y were the sides of a right triangle with Z as its hypotenuse. *Triplets*(Tom Dick Alphonse) would be true if the specific people referred to as Tom Dick and Alphonse were triplets. One very important and common two place predicate is = (equals). In predicate calculus with equality, individuals for which the = relationship is true may be used interchangeably.

Note that each one place predicate defines a *set* or *sort*. That is, for any one place predicate, P, all objects, x, can be divided into *satisfies P* (P(x) is True), or *doesn't satisfy P* (P(x) is False). We shall use the term *sort* and *one place predicate* interchangeably. Examples of sorts are *red objects*, *odd numbers*, *playing cards*, *men*, and *animals*. We see that some sorts include other sorts; all *men* are *animals*; all *knaves* are *playing cards*.

We shall often have occasion to refer to facts that we know to be true of all or some of the members of a sort. For this, we introduce two new notions, that of *Variable*, and of *Quantifier*. A variable is a place holder, one that is to be filled in by some constant.

There are two quantifiers,  $\forall$ , meaning *For all...*, and  $\exists$ , meaning *There exists...*: The English language sentence *All men are mortal*, is thus expressed in predicate calculus as  $\forall X.(MAN(X) \supset MORTAL(X))$  (loosely rendered *For all x, if x is a man (man of x is true) then x is mortal (mortal of x is true)*.) The English sentence *There is a playing card that is red and is a knave* becomes the predicate calculus statement  $\exists X.(PLAYING-CARD(X) \wedge KNAVE(X) \wedge IS-RED(X))$ . Here x is being used as a variable.

We can, of course, form more complicated sentences using these quantifiers, connectives, predicates, constants and variables. For example, if P is a binary (two place) predicate, Q a ternary (three place) predicate, A, B, and C are constants, and X, Y, and Z are variables, we can obtain the *well formed formulas* (or WFF's). (A WFF is essentially a syntactically correct sentence of one's logic.):

$$\forall X Y. \exists Z. ((P(A, X) \vee Q(Z, C, Y)) \supset (\neg \forall Z. P(A, B) \equiv Q(A, B, Z))) \quad (1)$$

$$P(A, B) \supset \exists Z. \forall X. \exists Y. Q(Y, Z, X) \quad (2)$$

$$\forall X Y. ((P(X, Y) \equiv P(Y, X)) \wedge \forall X. P(X, X)) \quad (3)$$

Sentence (3) states that the predicate P is commutative and reflexive.

For a typical natural deduction system, use of the quantifiers and predicates implies the introduction of four more inference rules, one for each of the introduction and elimination of each quantifier. Thus, the  $\forall$ -E (for all elimination, or universal specialization) rule states that, for any WFF  $\alpha$ , that mentions  $X$ , if we have

$$\forall X. \alpha(X)$$

we can conclude for any individual  $A$ , that:

$$\alpha(A)$$

Thus, if we know:

$$\forall X. (\text{MAN}(X) \supset \text{MORTAL}(X))$$

we can apply this to the individual Socrates, using the  $\forall$ -E rule, to get:

$$\text{MAN}(\text{SOCRATES}) \supset \text{MORTAL}(\text{SOCRATES})$$

The rules of the propositional calculus, extended by quantification and predicates, and the inference rules for quantifiers, results in the *Predicate Calculus*.

Predicate calculus, as we've described it, is very general, but still very clumsy. Two other additions to the logic will ease our expression, though still without extending what we can say. The first of these is the the notion of operators (or functions). Functions, like predicates, have a fixed number of arguments. Each of the arguments of a function can be either a variable, a constant, or a function (with its arguments). Thus, we could have functions like *absolute-value*, *father-of*, *plus*, and *left-arm-of*. Functions can, of course, be combined; we can speak of the *father-of (father-of (John))*, who would, of course, be John's paternal grandfather. Functional values, along with individual constants and predicates form the set of *terms*.

The second important addition is that of the predicate = (equals). Two individuals,  $X$  and  $Y$  are equal if and only if they are indistinguishable under all predicates and functions. More formally,  $X=Y$  if and only if, for all predicates  $P$ ,  $P(X) \supset P(Y)$ , and for all functions  $F$ ,  $F(X) = F(Y)$ . What we arrive at with these additions is now longer pure predicate calculus; rather, it is a variety of *first order logic*.

The reader may find this phrase, *first order logic*, familiar. A logic is first order if it permits quantification over individuals, but not over predicates and functions. A second order logic would permit quantification over functions and predicates. Thus one could state, in a second order logic, "for all functions,  $f \dots$ " or "there exists a predicate  $p$ , such that  $\dots$ ". Second (or higher) order logics might have a place in artificial intelligence. However, higher logics are *incomplete* and lack useful semantic models. Indiscriminate use of higher order logics can also lead to some paradoxes.

Note, however, that some aspects of higher-order logics -- i.e., universal quantification over predicates and functions (e.g., 'the following is true of all predicates') -- can be successfully and usefully merged into a first order logic system.

One useful instance of universal quantification over predicates is the use of axiom

schema. Mathematicians ought to be familiar with the axiom schema of mathematical induction; it states that if a property is both true of the number 0, and, if it is true of any number  $n$ , then it is true of the number  $n+1$ , then that property will be true of all numbers. If we let the symbol  $PP$  be a *predicate parameter*, for which we can substitute any monadic (one place) predicate, we could write this *axiom* as:

$$(PP(0) \wedge \forall X.(PP(X) \supset PP(X+1))) \supset \forall X.PP(X)$$

Thus, to prove that the sum of the integers from  $i=0$  to  $n$  ( $SIGMA(0,n,i)$ ) is  $n^*(n+1)/2$ , we need to first prove  $PP(0)$ ,

$$SIGMA(0,0,i) = 0^*(0+1)/2$$

and then that

$$\forall n.(SIGMA(0,n,i) = n^*(n+1)/2 \supset SIGMA(0,n+1,i) = (n+1)^*(n+2)/2)$$

Thus, we can conclude that

$$\forall n.(SIGMA(0,n,i) = n^*(n+1)/2)$$

Another comment on the value of first order logics is in order. First order logic is both *sound* (it is impossible to prove a false statement) and *complete* (any true statement has a proof). Higher order logics are not complete. Soundness is of importance in A.I. systems: one current debate is over whether one's system needs to be, or even ought to be sound. Present *real world* logic axiomatizations are so incomplete in their expression of the world, that logical completeness is not very important.

### Axiomatic Systems

What we have presented so far is not a solution to any artificial intelligence representation problem. Rather, we have laid out an alphabet, defined parts of speech and the grammar (predicate, term, WFF). We have neither declared the vocabulary, nor specified what sentences one should write. That is left to the user of logic, just as a programming manual might present the syntax of legal statements, but would not define which programs ought to be written.

However, a good programming manual ought to present example programs; we present a brief sample *axiomatization* of the *Tower of Hanoi* problem. The tower of Hanoi problem presents three pegs and a number of disks. Each disk is of a different size, and has a central hole. The disks can be stacked on the pegs. The problem initially has a number of disks, say 5, stacked on a single peg, largest on the bottom through smallest on the top. A free disk is any disk with no other disk on it. A move consists of taking any free disk and placing it either on an empty peg, or on a larger free disk. The problem asks, can one transfer the tower of disks from the first peg to the second, using only legal moves?

Problem expression and solution in logic has several parts. We must first specify the vocabulary of our domain; what are the variables, constants, predicates and functions.

Secondly, we define *axioms*, sentences (WFF's) in this logic, which we assert express the necessary relationships between the objects needed to model our domain.

Obvious objects (constants) for this axiomatization are the disks, D1, D2, D3, D4, and D5, and the pegs, P1, P2 and P3; obvious predicates are the sorts (sets) DISK and PEG. DISK(D1) is TRUE; PEG(D3) is FALSE.

We need also to be able to compare disk size; for that, we have a binary predicate, <. Using < as an infix operator, between its arguments, we define D1<D2 to mean that disk D1 is smaller than disk D2. If we have variables X, Y, and Z, we can express the transitivity of < with the axiom:

$$\forall X Y Z.((X<Y \wedge Y<Z) \Rightarrow (X<Z)) \quad (1)$$

The size relationships between the disks is stated by

$$D1 < D2 \wedge D2 < D3 \wedge D3 < D4 \wedge D4 < D5 \quad (2)$$

Note that by using (1) and (2), we can establish that, for example, D2 < D5.

One thing many systems fail to come to grips with is impossible situations. Our axiomatization also ought to know which disks are not smaller than others. Fortunately, we can establish this with one simple axiom:

$$\forall X. \neg X < X \quad (3)$$

Now, we are going to need to be able to talk about the status of our problem solving at some point, and to be able to compare the status after different series of moves. A common strategy used to deal with this difficulty is that of introducing a *situational* constant. A situation is a compression of where things are at any given point. Thus, in the tower of Hanoi problem, we might have disk D3 under disk D1 in situation SIT12. We therefore also have the sort SITUATION, a monadic predicate.

The vertical relationships of the disks and the pegs is, after all, the primary predicate of this problem. Hence, we need a ternary predicate ON(X,Y,S), which asserts that disk X is on disk (or peg) Y in situation S. A disk or peg could therefore be *FREE* in a situation S if nothing is on it. Axiomatically, this becomes:

$$\forall X S. (FREE(X,S) \equiv \neg \exists Y. (ON(Y,X,S))) \quad (4)$$

Notice how much more specific and concise the formal statement of these relationships can be.

We shall state that moving disk X onto disk Y will be *LEGAL* in situation S if both X and Y are free in S, X is a DISK, and if Y is bigger than X.

$$\forall X Y S. (LEGAL(X,Y,S) \equiv (FREE(X,S) \wedge FREE(Y,S) \wedge DISK(X) \wedge X < Y)) \quad (5)$$

Now all we lack is a way of generating new situations. The *function* MOVE, defined on two objects and a situation, produces a new situation where the first object is on top of the second. This will be a SITUATION if that move is LEGAL.

$$\forall X Y S. (\text{LEGAL}(X,Y,S) \equiv \text{SITUATION}(\text{MOVE}(X,Y,S))) \quad (6)$$

An what does this new situation look like? Well, X is on Y, and nothing else has changed. If S' is also a variable, we declare:

$$\begin{aligned} \forall S S' X Y. ((S' = \text{MOVE}(X,Y,S) \wedge \text{SITUATION}(S')) \supset \\ (\text{ON}(X,Y,S') \wedge \\ \forall Z Z1. ((\neg Z=X \wedge \neg Z1=Y) \\ \supset (\text{ON}(Z,Z1,S) \equiv \text{ON}(Z,Z1,S')))) \wedge \\ \forall Z. (\text{ON}(X,Z,S) \supset \text{FREE}(Z,S'))) \quad (7) \end{aligned}$$

What we have presented is enough to formalize the elementary notions of the Tower of Hanoi problem. However, what we're primarily interested in is whether a particular situation can be reached from another. For this we declare the predicate ACCESS, on two situations, with the intended meaning that the second situation can be reached from the first by a *series* of legal moves.

Now, axiomatizing this notion has its complexities. The idea of a "series of moves" (the transitive closure) is not really a first order notion. However, the following axioms should prove useful in our manipulations: First of all, if a situation can be used to reach a second situation, then every situation that can access the first can access the second.

$$\forall S S' S'' X Y. ((\text{ACCESS}(S,S') \wedge S'' = \text{MOVE}(X,Y,S') \\ \wedge \text{SITUATION}(S'')) \supset \text{ACCESS}(S,S'')) \quad (8)$$

We must also decide if ACCESS(S,S) is to be true. It doesn't make a lot of difference, so:

$$\forall S. \text{ACCESS}(S,S) \quad (9)$$

can be one of our axioms.

Several other axioms delimiting transitive closure might be useful. For example, we might state that if a situation S can access another situation S' (ACCESS(S,S')), then either they're equal, or there is some intermediate situation that can reach S' in one move, and can be accessed by S.

$$\begin{aligned} \forall S S'. ((\text{ACCESS}(S,S') \supset \\ (S=S') \vee \exists S'' X Y. (\text{SITUATION}(S'') \\ \wedge \text{MOVE}(X,Y,S'') = S' \wedge \text{ACCESS}(S,S''))) \quad (10) \end{aligned}$$

However, we don't achieve the full power of these situational variables unless we consider an *axiom schema* such as the *induction* schema on situation and access. It states that if a property PP is true of a situation S, and if, for all situations S' and S'', if PP is true of S', and S'' is a move away from S', then PP is true of S'', then PP is true of all situations accessible from S. More formally and specifically,

$$\begin{aligned} \forall S. ((\text{PP}(S) \wedge \\ \forall S' S'' X Y. ((\text{SITUATION}(S') \wedge \text{SITUATION}(S'') \\ \wedge S'' = \text{MOVE}(X,Y,S') \wedge \text{PP}(S')) \supset \text{PP}(S''))) \\ \supset \forall S'. (\text{ACCESS}(S,S') \supset \text{PP}(S'))) \quad (11) \end{aligned}$$

Now, all we have described so far is a language for expressing problems in the "Tower of Hanoi" world, and certain sentences in that language which, we assert are true.

The classical Tower of Hanoi problem can be expressed in this language as the sentence:

$$\begin{aligned} \exists S'. \forall S. ( & ( \text{ON}(D5, P1, S) \wedge \text{ON}(D4, D5, S) \wedge \text{ON}(D3, D4, S) \wedge \\ & \text{ON}(D2, D3, S) \wedge \text{ON}(D1, D2, S) \wedge \text{FREE}(D1, S) ) \supset \\ & ( \text{ON}(D5, P2, S') \wedge \text{ON}(D4, D5, S') \wedge \text{ON}(D3, D4, S') \wedge \\ & \text{ON}(D2, D3, S') \wedge \\ & \text{ON}(D1, D2, S') \wedge \text{FREE}(D1, S') \wedge \text{ACCESS}(S, S') ) ) \quad (12) \end{aligned}$$

### Applications of Logic to Artificial Intelligence

Logic, as we have described it, is in some sense the basic representation formalism for work in artificial intelligence. That this is so is indicated by the fact that other formalisms can be mapped into logic; their commitment to *representation* is the same. Of course, they differ in the processes they use, but logic makes no commitment in that area.

In this section we shall survey various artificial intelligence systems which use logic as their representation. We shall also discuss the processes they use to make deduction, since this is an equally important aspect. These processes vary, but the representation in each case is the same.

#### a) QA3 (Green, 1969)

QA3 was a general question-answering system, which solved simple problems from a number of domains.

The language used to represent facts was first-order logic. For example, the assertion *In state S there is nothing under the bananas* would be represented as

$$\forall X \neg \text{AT}(X, \text{under-bananas}, S)$$

The deductions were performed using the *resolution* rule of inference (see also Chapter Theorem Proving), with simple general heuristics for control. Space prohibits more discussion of the inference process here.

The system solved problems from domains such as chemistry, robot problem-solving, state-space transformations (e.g., monkey-and-bananas), and automatic programming. An example from the chemistry domain is:

Given the following facts (among others):

Ferrous sulfide is a dark-gray compound and it is brittle.  
Ferrous sulfide is a sulfide.

$$\text{sulfide}(\text{FES}) \wedge \text{compound}(\text{FES}) \wedge \text{darkgray}(\text{FES}) \wedge \text{brittle}(\text{FES})$$

What is the answer to the question:  
Is it true that no dark-gray thing is a sulfide?

$\neg \exists X (\text{darkgray}(x) \wedge \text{sulfide}(x))$

The answer is: No, FES is both dark-gray and a sulfide.

Despite its generality, and its success on simple problems, QA3 could not handle really hard problems. The fault lay in the method of deduction, resolution theorem proving. Unrestricted resolution is *complete*, in the sense that it will always return an answer if one is available, but it is too slow. Even the heuristics used by Green (referred to as *unit preference*, *set of support*, and *subsumption*) do not control the search sufficiently to make this approach feasible.

The use of first-order logic as a representation was felt to be successful. This success derived from the fact that logic is well-defined, unambiguous, and rather general. Green illustrated several different representations for the Towers of Hanoi problem, all using first-order logic; different versions made the solution of the problem easier or harder (Amarel, 1969 performed the same sort of analysis on the Missionaries and Cannibals problem). In general, logic seemed adequate for the representation part; the problems with QA3 lay in the deduction part.

b) STRIPS (Fikes and Nilsson, 1971; Fikes, Hart, & Nilsson, 1972)

STRIPS (short for Stanford Research Institute Problem Solver) was designed for the class of problems faced by a robot in rearranging objects and navigating in a cluttered environment. The interesting aspect of these problems is the fact that the representation of the world must include a large number of facts dealing with the position of the robot, objects, open spaces, etc; thus, simpler forms generally used for puzzles or games will not suffice. The world model (representation) chosen for STRIPS was the first order predicate calculus.

A simple problem was:

Given a robot at point a, and boxes at points b, c, and d,  
gather the boxes together.

The current situation is described as:

ATR(a)  
AT(BOX1,b)  
AT(BOX2,c)  
AT(BOX3,d)

and the goal as

$\exists X (\text{AT}(\text{BOX1},x) \wedge \text{AT}(\text{BOX2},x) \wedge \text{AT}(\text{BOX3},x))$

Problem-solving in a robot domain such as this involves two types of processes:

- (a) deduction in a particular world model, to find out whether a certain fact is true



- (b) searching thru a space of world models, to find one in which the given condition is satisfied (e.g., how can we get the three blocks together?)

The former process is usually called question-answering; the latter, planning. STRIPS used different methods to perform these two processes. The question-answering was done via resolution theorem proving, as in Green's system; the planning was performed via means-ends analysis, as in GPS. This dual approach allowed world models more complex and general than in GPS, and provides more powerful search heuristics than those found in theorem-proving programs.

STRIPS is too complicated to discuss further here; see article Search.05 for more details.

c) FOL (Weyhrauch, 1977; Filman & Weyhrauch, 1976)

FOL is, among other things, a proof checker for proofs stated in first order logic. The language used is first order logic, that is, first order predicate calculus, augmented with equality and functions. Note that neither of these extensions actually expands the kinds of things which can be said in the language; but, they do make certain things *easier* to express.

Deduction is done using the *natural deduction* of Prawitz, 1965, which includes the rules of inference listed in sections 2 and 3.

A trivial example of the use of FOL is the following.

Suppose we want to prove the syllogism mentioned in section (3).

Socrates is a man  
All men are mortal  
therefore  
Socrates is mortal

We could give FOL the following commands:

*We first declare the only constant (Socrates), the only variable (x), and a number of 1-place predicates (MAN and MORTAL). All input to FOL is prefaced with \*\*\*\*\*.*

```
*****DECLARE INDCONST Socrates;
*****DECLARE PREDCONST MORTAL MAN 1;
*****DECLARE INDVAR x;
```

*Tell FOL to assume that the two axioms are true.*

```
*****ASSUME MAN(SOCRATES) ^ YX.(MAN(X) => MORTAL(X));
```

1  $\text{MAN}(\text{SOCRATES}) \wedge \forall X. (\text{MAN}(X) \supset \text{MORTAL}(X))$  (1)

\*\*\*\*\*TAUT  $\forall X. (\text{MAN}(X) \supset \text{MORTAL}(X))$  1;

*This can be derived from (1) via the rule of  $\wedge\text{-}\exists$ .*

2  $\forall X. (\text{MAN}(X) \supset \text{MORTAL}(X))$  (1)

\*\*\*\*\* $\forall\exists$  2 Socrates;

*Use  $\forall\text{THX}\$$  to specialize this to Socrates.*

3  $\text{MAN}(\text{SOCRATES}) \supset \text{MORTAL}(\text{SOCRATES})$  (1)

\*\*\*\*\*TAUT  $\text{MORTAL}(\text{Socrates})$  1,3;

*Now, show that Socrates is mortal.*

4  $\text{MORTAL}(\text{Socrates})$  (1)

*FOL says that Socrates' mortality depends on the statements in (1). Try to incorporate them into the statement.*

\*\*\*\*\* $\supset\text{I}$  1 $\supset$ 4;

5  $(\text{MAN}(\text{SOCRATES}) \wedge \forall X. (\text{MAN}(X) \supset \text{MORTAL}(X))) \supset \text{MORTAL}(\text{SOCRATES})$

*And this is what we wished to show.*

FOL can handle more complicated problems, of course, and can more properly be viewed as a sophisticated environment for studying epistemological questions. Weyhrauch, 1978 has more details.

Thus we have examined some systems which use logic to represent the axioms and rules of inference of their problem domain. In the next section we will consider the more general case of the usefulness of logic to artificial intelligence.

### Logic and Representation

It is important to emphasize that this discussion of formal logic and representation theory is not centered about performance. Rather, it is a discussion of what things *can* be said, and how one can economically state them. We are dealing with representational structures, but not representational data structures. We are concerned with the *conceptual* structures for artificial intelligence. First order logic, as we have described it, demands a clean syntax, a clear semantics, and, above all, the notions of *truth* and *inference*. The conceptual knowledge inherent in a system of logic axioms can be encoded for use in a computer by a semantic net, a frame, a LISP list, or any of a large number of different data structures. Logical systems can be manipulated by a general purpose theorem prover or specific purpose inference systems. Systems cease to be logical when they become *ad hoc*, detached from the semantics of their primitives, or concerned not so much with the truth of their results, as their accessibility.

Discussion of *representational* formalisms often falls into the fallacy of discussing *heuristic* mechanisms. When we suggest using a logical formalism for the representation of knowledge, we are not implying any particular control structure. A system employing first order logic can use resolution, or pattern directed inference, or computation in a semantic model, or any other scheme the writer can devise for specifying which inference is to be selected next. Keeping the representational system separate from the deductive process permits the employment of all or any of these schemes, as need arises.

McCarthy & Hayes, 1969 were the first workers in artificial intelligence to make this distinction explicitly. The question of representation was termed the *epistemological* part of the problem, defined (in McCarthy, 1977, p. 1038) as "what kinds of facts about the world are available to an observer with given opportunities to observe, how these facts can be represented in the memory of a computer, and what rules permit legitimate conclusions to be drawn from these facts." The issue of processing, called the *heuristic* part, was defined as 'how to search spaces of possibilities and how to match patterns'.

We turn now to an evaluation of the efficacy of logic as a representation formalism -- i.e., what advantages and disadvantages are involved in the use of logic.

First, logic is *natural*; as McCarthy, 1977 and Filman, 1979 point out, the expression of a problem in logic often corresponds to our intuitive understanding of the domain. Green, 1969 also indicated that a logical representation was easier to reformulate; thus, experimentation is facilitated.

Second, it is *precise*; there exist standard methods (not discussed here) of determining the *meaning* of an expression in a logical formalism. Hayes, 1977 presents a complete discussion on this issue, and argues for the advantages of logic over other representation systems, on these grounds.

Third, logic is *flexible*; since logic makes no commitment to the kinds of processes which will actually make deductions, a particular fact can be represented in a single way, without having to consider its possible use.

The final benefit is *modularity*. Logical assertions can be entered in a data base independently of each other; thus, knowledge can grow incrementally, as new facts are

discovered and added. In other systems, the presence of a new fact might adversely affect the kinds of deductions which can be made.

The major disadvantage of logic is the same as its major advantage -- the separation of *representation* from *processing*. The difficulty with most current artificial intelligence systems lies in the heuristic, not epistemological, part (e.g., the failure of resolution theorem proving). Thus, separating the two aspects, and concentrating on the former, merely postpones addressing the problem. The work on such systems as PLANNER (Hewitt, 1972) and frames (Minsky, 1975) represents an effort to incorporate the heuristic aspect into the epistemological; GOLUX (Hayes, 1977b) and FOL (Weyhrauch, 1978) are attempts to formalize control of processing, while retaining the logical precision.

To conclude, representational issues must be considered if we are ever to create a generally "artificially intelligent" computer. There are many aspects of common knowledge and common sense that we do not even know how to formalize, much less how to convince a computer to do by itself.

## C2. Overview of Semantic Networks

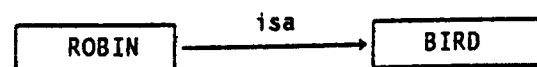
Many of the recent systems developed in Artificial Intelligence research use a class of knowledge representation formalisms called *semantic networks*. These representation formalisms are grouped together because they share a common notation and not because they share any simple set of common assumptions.

This common notation consists of *nodes* (drawn as dots, circles, or boxes in illustrations) and *arcs* or *links* (drawn as arrows) connecting the nodes. Both the nodes and the arcs can have labels. Nodes are usually used to represent *objects, concepts, or situations* in the domain, and the arcs are used to represent the *relationships* between them. The superficial similarity of this notation is all that some semantic network systems have in common. For example, researchers in psychology, such as Quillian (1968), Norman & Rumelhart (1975), and Anderson & Bower (1973), have developed semantic network systems primarily as psychological models of human memory. Researchers in computer science have been more concerned with developing an efficient knowledge representation. Because of differing goals, there is no simple set of unifying principles to apply across all semantic network systems. This article, however, will attempt to characterize some of the most common network schemes.

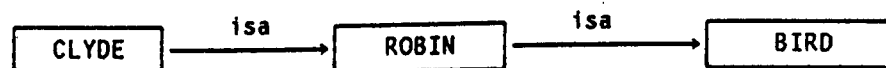
This article will present a description of how simple concepts are represented in semantic networks and review some AI systems that use semantic networks. Finally, some more difficult problems in representation, such as quantification, will be mentioned and some of the proposed solutions reviewed.

### A Basic Description of the Representation

Suppose we wish to represent a simple fact like *All robins are birds* in a semantic network. We might do this by creating two nodes to designate "robins" and "birds" with a link connecting them, as follows:

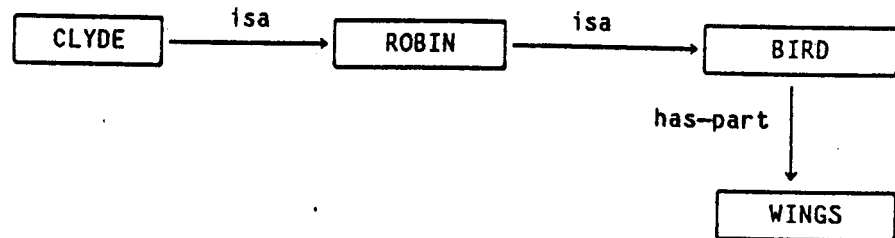


If Clyde is a particular individual whom we wished to assert is a robin, we could add a node for Clyde to the network as follows:



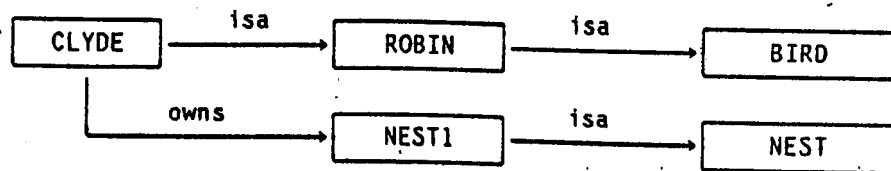
Notice that in this example we have not only represented the two facts we initially intended to represent, but we have also made it very easy to deduce a third fact, namely, that Clyde is a bird, simply by following the ISA links. The ease with which it is possible to make deductions about hierarchies such as the one above is one reason for the popularity of semantic networks as a knowledge representation. In a domain where much of the reasoning is based on a very complicated taxonomy, such as in the PROSPECTOR system (Article Applications.D2), a semantic network is a logical choice for a knowledge representation.

One usually desires to encode knowledge about facts other than taxonomies. For example, we might wish to express the fact "birds have wings" in our network. We could do this as follows:



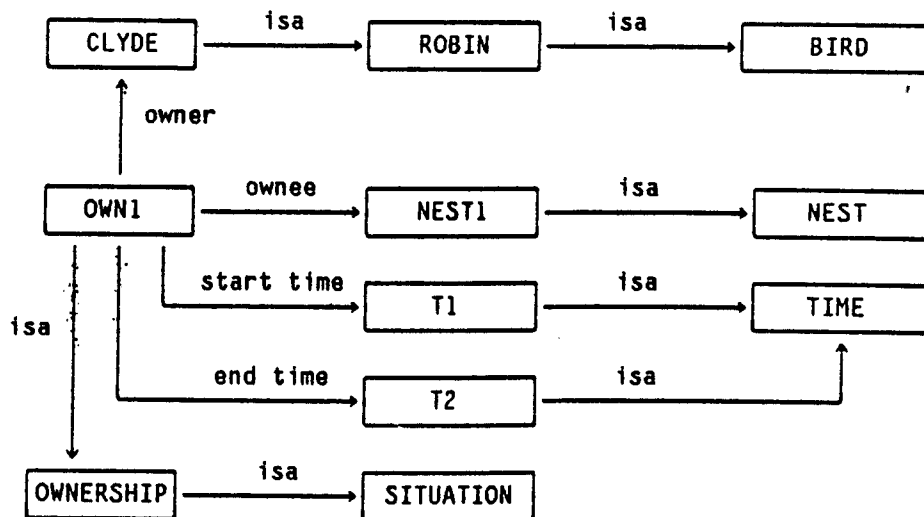
As in the previous example, our choice of representation has made it very easy to write a procedure to make the deductions that robins have wings and that Clyde has wings. All that is necessary is to trace up the ISA-hierarchy, assuming any facts asserted about higher nodes on the hierarchy can be considered assertions about the lower ones also, without having to represent these assertions explicitly in the net. This is called *property inheritance*, and the ISA link is often referred to as a *property inheritance link*. The ease with which it is possible to implement property inheritance in a hierarchy also contributes to the desirability of semantic net representations.

Suppose we wish to represent the fact *Clyde owns a nest*. Our first impulse may be to encode this fact using an ownership link to a node representing Clyde's nest:



In the above example, NEST1 is the nest that Clyde owns. It is an instance of NEST. The above representation may be adequate for some purposes, but it has shortcomings. Suppose one wanted to encode the additional information that Clyde owned NEST1 from time T1 to time T2. This is impossible to do in the current network because the ownership relation is encoded as a link and links, by their nature, can only encode binary relationships. What we might need is the equivalent of a four-place predicate for ownership that would include the start time and end time of the relationship.

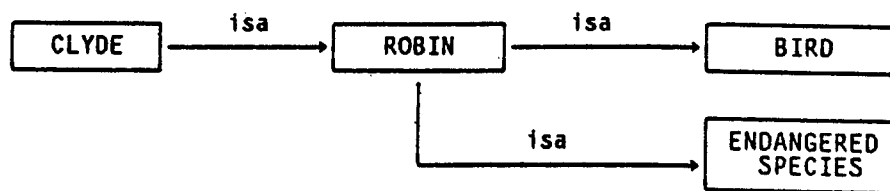
The solution to this problem that was proposed by Simmons (Simmons & Slocum, 1972), and later adopted in many semantic net systems, was to extend nodes to represent situations and actions, as well as objects and sets of objects. Each situation node can have a set of outgoing "case" arcs, which specify the various arguments to the situation predicate. Many semantic networks, including Simmons's, use a linguistically motivated case system; but this is not necessary to solve the problem at hand since arcs labeled "owner" and "ownee" can serve this purpose as well as those with more general labels such as "agent" and "object." Using situation nodes with case arcs to represent the fact "Clyde owned a nest from time T1 to time T2," our network now becomes



The node NEST1 is created to represent Clyde's nest, which, of course, ISA nest, as shown. The OWN1 node represents a particular instance of OWNERSHIP, namely, Clyde owning his nest. And like all instances of OWNERSHIP, it has case arcs to OWNER, OWNEE, START-TIME, and END-TIME. Although the addition of situation nodes and case arcs has complicated the network somewhat, its expressive power has been greatly increased.

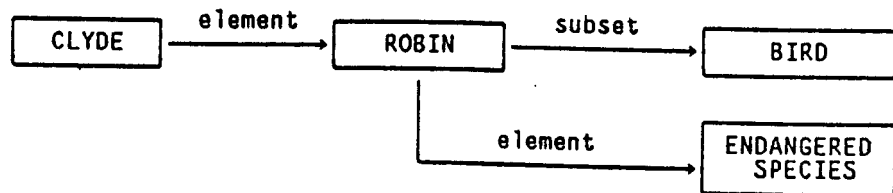
Notice that the representation described above lends itself to the expression of states and actions in terms of a small number of primitives. For example, "walk" might be considered a type of "motion" and could be represented by a node with an isa arc to "motion" and case arcs describing the specialized features of walking. The use of a small number of primitives has both advantages and disadvantages and is discussed more fully in B5.

There are still some serious problems with the representation as it has been developed so far. Suppose we wished to make the assertion "The robin is an endangered species" (which is not really true, but which we assume for the sake of argument). If we were not careful, we might create the following:



The problem with this representation is that we defined our reasoning procedures so that they would treat the ISA link as a property inheritance link. Therefore, the individual Clyde

inherits all the properties of ENDANGERED SPECIES, just as it does those of BIRD, and one would be led to make many incorrect deductions. For example, from "Naturalists study endangered species" one would conclude that "Naturalists study Clyde." The fault is that there is as yet no distinction in our network between an *individual* and a *class*, or set of individuals. To implement this distinction, the links *element* and *subset* have been differentiated. Clyde is an *element* of the set of robins, the set of robins is a *subset* of the set of birds, and the set of robins is an *element* of the set of endangered species. In terms of this set-theoretic taxonomy, the network can be drawn as follows:



And if we specify in the reasoning procedures that elements of a set inherit properties only over subset links and not over element links, then this distinction can be represented and we keep the naturalists off Clyde's case.

#### A Brief Survey of Semantic Network Systems

The node and link formalism of semantic networks has found use in many AI systems in different application domains. It is impossible to mention every one of these systems in this paper, but we can review some of the more well-known ones. This section does not intend to give a complete description of any of these systems but, rather, to give an introduction to what is available in the AI literature.

Quillian designed two early semantic-network-based systems (1968, 1969) that were intended primarily as psychological models of associative memory (see Section Information Processing Psychology.B3a). Quillian was the first to make use of the node and link formalism, but his networks were different from those described above. The network itself was very simple. It consisted of a number of chunks called *planes*. Each plane encoded knowledge about a particular concept through interconnecting nodes representing other concepts in the system. These interconnecting links represented a few simple types of concept jointure, *conjunction* and *disjunction* and the modification of one concept by another.

Quillian's procedures manipulated the network to make inferences by finding connections between a pair of concepts. The program would start from the nodes denoting each of the two concepts being compared and would "activate" all concepts connected to each of the initial two concepts. This spreading activation continued, forming an expanding sphere of activation around each concept. When some concept was activated simultaneously by both of the two original concepts, a connection had been found. The program would then print out a description of the connection in a stylized version of English and could then continue to find longer, probably more obscure, connections.

In an effort to solve some of the problems in the original system, which included a lack of direction in the search and the formation of a large number of spurious associations



between many concepts, Quillian Quillian (1969) designed a second semantic network system that was more complex than the one described above. The system was called the Teachable Language Comprehender (TLC).

Other semantic-network-based computer programs that were designed as psychological models were Anderson and Bower's HAM program (1973) and Lindsay, Norman and Rumelhart's Active Structural Networks (1975).

One of the earliest programs to use semantic network techniques was Bert Raphael's SIR (see Article Natural Language.F1 on early natural language systems). The program could perform a variety of simple reasoning tasks, such as "A finger is part of a hand, a hand is part of an arm, therefore a finger is part of an arm." Although he did not claim to use a node and link formalism, his use of binary predicates, such as PART-OF(FINGER, HAND), and reasoning procedures was much like the early semantic net systems and faced many of the same problems that these systems faced.

R. F. Simmons (1973) designed a semantic network, mentioned earlier, using a linguistic case frame approach (see Article Natural Language.B4) for use in a natural language understanding program. The system could recognize sentences using an ATN grammar, translate them into a network, reason using this network, and, finally, generate answers to questions from the semantic network, Simmons & Slocum, 1972.

Carbonell & Collins (1974) used a semantic network as the basis of the SCHOLAR program (see Article Education Applications.C1), which answered questions about geographical data and provided a mixed-initiative tutorial dialogue on the subject--i.e., SCHOLAR can answer questions posed by the student as well as pose appropriate questions and give timely hints on its own initiative.

Winston's concept-learning program was based on a semantic net encoding the relationships between objects in a Blocks World scene (see Article Learning.E).

Myopolous, Cohen, Borgida, and Sugar designed a system for grouping related parts of a semantic network into units called "scenarios" Myopolous, Cohen, Borgida, & Sugar, 1975. The network was the basis of the TORUS system, a program to provide natural language access to a relational database management system. Hayes, 1977b also designed a system that incorporates higher level structures similar to scenarios, which he calls "depictions."

Two of the speech understanding systems (see the Speech section) used semantic networks as a knowledge representation, namely, those of BBN Woods (1976) and SRI Walker et al., 1976.

In connection with the SRI speech understanding system, Hendrix, 1975 developed the idea of network partitioning, which provides a mechanism for dealing with a variety of difficult representation problems including quantification, representation of logical connectives, hypothetical worlds, and belief worlds.

Duda, 1978 have investigated the possibility of encoding judgmental knowledge in semantic networks in the PROSPECTOR system (see Article Applications.D2).

### Reasoning with Semantic Networks

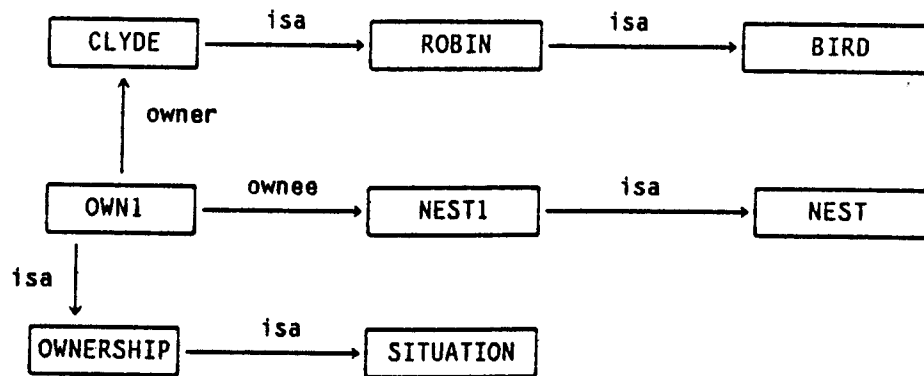
In semantic network representations, there is no *formal semantics*, as there is with predicate calculus, that assigns meaning to network structures. Meaning is assigned to a network structure by context, the procedures that manipulate the structure. It is for this reason that all semantic networks cannot be placed into a simple, well-defined category of representation systems. A wide variety of network-based systems have been designed that use totally different procedures for manipulating their networks.

It is therefore impossible to talk about a network representation without reference to the procedures that use it. Because of this close coupling of a network and its reasoning procedure, there is a wide range of choice as to what knowledge to encode explicitly in the network and what knowledge to encode procedurally. Using a semantic network does not result in a purely declarative representation. (See Article A1 for a discussion on declarative and procedural representations.)

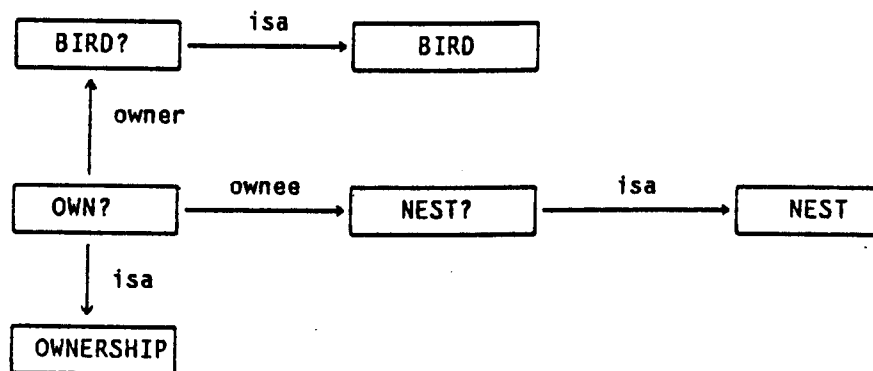
An example of a network reasoning procedure was Quillian's system (1968, 1969) described earlier based on a *spreading activation model*.

The reasoning mechanism used by many semantic network systems is based on **matching network structures**. A network fragment representing the query is constructed and matched against the network representing the system's knowledge. If the question was a yes-no question and if a match is found, then the question can be answered affirmatively. If the question is of the wh-type (e.g., *What is the capitol of New Mexico?*), then the particular object sought by the wh-query can be obtained from the bindings produced during the matching process. The reasoning procedure can continue finding matches after it has found the first one; each time producing different bindings, with different answers to the user's wh-question.

The matcher can use deductive rules during the matching process to create network structure that is not explicitly present in the network. For example, if A ISA B, and B ISA C, then the matcher can *deduce* the existence of an ISA link between A and C. For another example, suppose we have the following network as our knowledge base:



and suppose we wish to answer the question *Is there a bird who owns a nest?* We could translate that question into the following network structure:



In the above network fragment, BIRD?, NEST?, and OWN? represent the yet to be determined bird-nest-owning relationship. Notice that the query network does not match the knowledge network exactly. The deduction procedure would have to construct an ISA link from CLYDE to BIRD to make the match possible. The matcher would bind BIRD? to CLYDE, OWN? to OWN1, and NEST? to NEST1, and the answer to the question would be "yes." If, instead, the user had asked *Which bird owns a nest?*, the question-answering procedure would construct exactly the same query network except that instead of answering "yes," it would return the binding of BIRD? and answer "CLYDE."

A good example of a network deduction system constructed around this matching paradigm is the SNIFFER system (Fikes & Hendrix, 1977). SNIFFER has the full power of a natural deduction theorem prover (see Theorem Proving), if needed. It is also capable of taking advantage of procedural knowledge embedded in *selector functions*, which tell it which network elements should be matched next and which provide advice on how to match the selected element.

### Some Difficult Problems for Network Representations

As the scope of problems that researchers tackle increases, the expressive power of the semantic networks used must also be increased to deal adequately with the new domains. Semantic networks should be capable of representing any fact about the world. To represent some facts, it is necessary to have the full power of quantification offered by the first order predicate calculus. It is desirable to represent quantified expressions, but it is not immediately obvious how this should be done. The problem is difficult because the scope of each quantifier has to be made explicit in the net--for propositions with several quantifiers.

In simple semantic networks, universal quantification is implemented through the property inheritance mechanism. For example, the network structure shown previously, with a ROBIN node and a BIRD node connected by an ISA link, is essentially an encoding of the proposition

$$\text{FOR-ALL } x \text{ ROBIN}(x) \Rightarrow \text{BIRD}(x) .$$

However, this simple mechanism is not sufficient to handle cases where it is necessary to distinguish the scope of several universal and existential quantifiers, such as in the statement "Every positive integer is greater than some positive integer."

One possible approach, discussed by Woods (1975) and Shapiro (1971) encodes quantifiers as higher order propositions that take as arguments: a variable name, a specification of the range of the quantification, and the proposition to be quantified.

Schubert, 1975 and Kay, 1973 describe an approach to quantification that involves skolemization. The proposition to be represented in the network is converted to prenex normal form, with existentially quantified variables replaced by skolem functions. In the network, each node has an associated quantification property. The skolem functions are implemented by special links from existentially quantified nodes to their dependents, the universally quantified nodes. The disadvantage of the skolemization approach is that it is very difficult to store the denial of an existing proposition. It is not adequate to simply attach a negation operator to the structure, since all the universal quantifiers have to be converted to existential quantifiers, and vice versa. This could be a complex task when a complex proposition is involved.

Hendrix, 1975 has demonstrated that network partitioning can be used to represent quantifiers in a logically adequate manner while preserving a simple network structure.

There are also many other problems that have been discussed in the literature on semantic networks, but many of these problems are not problems unique to semantic networks but are difficulties for any representation system. These include the symbol-mapping problem (Fahlman, 1975), the representation of belief worlds, mass nouns, degrees of certainty, time and tense information, and processes that consist of sequenced actions taking place over time. The success of semantic networks in a wide variety of artificial intelligence problems is a good indication that it is a good formalism for the solution to these other problems as well.

**References**

See Anderson & Bower, 1973; Brachman, 1976; Fikes & Hendrix, 1977; Fillmore, 1968; Hayes, 1977; Hendrix, 1975; Kay, 1973; Myopolous, Cohen, Borgida, & Sugar, 1975; Norman & Rumelhart, 1975; Quillian, 1968; Quillian, 1969; Schubert, 1975; Shapiro, 1971; Simmons, 1973; Simmons & Slocum, 1972; Walker et al., 1976; and Woods, 1975.

### C3. Production Systems

#### C3a. Overview of Production Systems

Production systems were first proposed by Post (1943), but have since undergone such theoretical and application-oriented development that the current systems have little in common with Post's formulation, or in some case, with each other. Our purpose in this article is to illustrate the "basics" of a production system (PS) noting especially where certain design decisions could result in PSs with different characteristics. The following article will discuss the issues that arise as a result of these design alternatives.

A production system consists of a set of productions, an interpreter to control how the productions are used, and a database of symbols which may or may not "evoke" one or more productions. A *production* is a chunk of factual knowledge, cast as an IF-THEN statement. For example:

**FACT:** Always punt on 4th down with long yardage required.

**PRODUCTION:** *If it is fourth down and long yardage is required  
Then punt*

In general, the If part (condition-part, left-hand-side, or LHS) is a boolean expression and the Then part (action-part, right-hand-side, RHS) is an action of some kind. Typically, in AI systems, the action part does not involve a physical action like punting, but rather some symbolic computation which varies in complexity among different systems. In its simplest form the action part involves adding some data to the database.

Table 1 shows how a production system might be used to identify food items using a process similar to that used in the game of "20 questions." The condition-part of each of the productions corresponds to a question one might ask; the action part represents an addition to our knowledge. The *interpreter*, which specifies some control of computation in the system, is shown after the productions in Table 1. The *database* for the system is not shown explicitly in the Table, but is part of the *trace*. The trace shows the operation of the production system over time; in particular it shows the contents of the database (called the "attribute list" or A.L.) as each production is checked for applicability. The attribute list stores the PS's current knowledge about the food item to be identified. Initially, the system knows that the item is green and weighs over 15 pounds, i.e. attribute list: (*green, weighs 15 lbs*).

## PRODUCTIONS:

- P1. If the Attribute List (abbreviated A.L.) contains *green*  
Then Put *produce* on the A.L.
- P2. If the A.L. contains *packed in small container*  
Then Put *delicacy* on the A.L.
- P3. If the A.L. contains *stored in refrigerator*  
or *produce*  
Then Put *perishable* on the A.L.
- P4. If the A.L. contains *weighs 15 lbs*  
and does not contain *perishable*  
and is *inexpensive*  
Then Put *staple* on the A.L.
- P5. If the A.L. contains *perishable*  
and *weighs 15 lbs*  
Then Put *turkey* on the A.L.
- P6. If the A.L. contains *weighs 15 lbs*  
and *produce*  
Then Put *watermelon* on the A.L.

## INTERPRETER:

- R1. A production is applicable if its condition part is true.
- R2. If more than one production is applicable, then make any production which adds a duplicate attribute to the A.L. inapplicable.
- R3. If more than one production is applicable, then execute the action part of the first applicable production.
- R4. If only one production is applicable, execute its action part.

## TRACE:

1. Initial A.L.: (*green, weighs 15 lbs*)  
Applicable Production: P1  
Execute P1 because of R4
2. A.L.: (*produce, green, weighs 15 lbs*)  
Applicable Productions: P1, P3, P6  
Checking if any of P1, P3, OR P6 adds a duplicate attribute, (R2):  
P1: A.L.: (*produce, produce, green, weighs 15 lbs*)  
P3: A.L.: (*perishable, produce, green, weighs 15 lbs*)  
P6: A.L.: (*watermelon, produce, green, weighs 15 lbs*)  
Do not execute P1 because of R2  
Execute P6 because of R3

3. A.L.: (*perishable, produce, green, weighs 15 lbs*)  
 Applicable productions: P1, P3, P5, P6  
 Checking if any of P1, P3, P5, P6 adds a duplicate  
 attribute (R2):  
 P1: A.L.: (*produce, perishable, produce, green, weighs 15 lbs*)  
 P3: A.L.: (*perishable, perishable, produce, green, weighs 15 lbs*)  
 P5: A.L.: (*turkey, perishable, produce, green, weighs 15 lbs*)  
 P6: A.L.: (*watermelon, perishable, produce, green, weighs 15 lbs*)  
 Do not execute P1 or P3 because of R2  
 Do not execute P6 because of R3

Table 1. An Example Production System.

Some of the design issues for PSs will become clear by following the trace in Table 1. The PS is trying to identify a food item which it knows is *green* and weighs more than 15 pounds. Thus, the attribute list (A.L.) contains the initial knowledge of the system: (*green, weighs 15 lbs*). The interpreter matches each production to the contents of A.L. in order to find all applicable productions (see R1). The only production which is applicable is P1, so in accordance with R4 the interpreter executes the action part of P1, which involves adding the attribute *produce* to the attribute list. Thus, after the first step in the trace, the PS knows a new fact about the food item.

So far, this example introduces two important design issues. First, the knowledge of the PS is modular. This is obvious in the sense that each production is a small, self-contained chunk of knowledge. Less obvious is the fact that the productions are independent because they communicate indirectly via the database (attribute list), instead of "calling" each other as procedural knowledge representations do. (see article B4) Said differently, the effect of a production on any other production is manifest in the database. Removing or replacing a production, then, may detract from or alter the eventual contents of the database, but does not affect the contents of any other productions. This makes modification of the PS an easy task.

The second point is that the knowledge of this PS is accessible to the whole system. This feature has two consequences: first, the knowledge represented as productions is always available; any production can be activated at any time. Second, the knowledge in the database is always available, and can all be used to activate productions. There are advantages and disadvantages to this accessibility, but we postpone discussion of them.

We are currently at step 2 in the trace in Table 1. The attribute list contains (*produce, green, weighs 15 lbs*). At this point, the interpreter finds that P1, P3, and P6 are applicable, and so it invokes R2 and R3 to pick just one production to execute. R2 is necessary, because the order of the productions doesn't change, and so the first applicable production will always be the same one. (R2 is not the only possible resolution to this problem; the interested reader can certainly think of many more.) After the interpreter has applied R2, finds that P3 and P6 are both applicable. A choice must be made, and R3 constitutes one choice strategy. After executing P3, the attribute list contains an additional attribute, *perishable*.



Four further design issues are illustrated here. First, note that the connectives OR and AND appear in P3 and P6. The condition-part of a production can be arbitrarily complex, depending on the proposed application of the PS. Note also that although P1, P3, and P6 were applicable, P4 and P5 were nearly applicable. That is, both matched to the attribute *weighs 15 lbs*, and P4 also matched because A.L. did not contain *perishable*, but neither of them matched on all their condition subparts. This is called a *partial match*, and in AI applications it is handled in various ways by the interpreter. For our purposes, we will ignore it.

A third point is that more than one production was applicable, and the interpreter can only apply one at a time. The process of picking just one production is called *conflict resolution*, and AI systems employ numerous approaches to the problem, including applying each production in parallel. (See Issues article following this one.)

Before resuming the example in Table 1, it is worth noting that the "correct" identification for food item with properties (*green, weighs 15 lbs*) is watermelon, and that P6 would put watermelon on the attribute list. Therefore, if the productions were ordered so that P6 preceeded P3, or if some other conflict resolution strategy were used, the PS would correctly attribute the name watermelon to the food item. The point is that conflict resolution and the ordering of productions can both affect the performance of the PS, and in AI systems, both are used by programmers to make their systems run as efficiently as possible. It has been argued, however, that ordering the productions is in violation of the principle that all knowledge should be available at all times to the PS: One point of ordering productions is to insure that, in some circumstances, some productions are not executed (Davis and King 1975).

In the third step of the trace in Table 1 the interpreter finds that P1, P3, P5, and P6 are applicable, but rejects P1 and P3 because of R2, and rejects P6 because of R3. After executing P5 the attribute list contains

*(turkey, perishable, produce, green, weighs 15 lbs).*

Although this is in some sense the "wrong answer", since turkeys are not *green* and watermelons are, it is really not an "answer" at all because the interpreter has no rules for interpreting an attribute as the identity, or name, of a food item. The interpreter is so primitive that all it can do is find applicable productions and resolve (badly) conflicts between productions. It doesn't know it is looking for a name-attribute. A final related, shortcoming of the interpreter is that it has no rules for when to stop work. By default, this PS will stop executing productions in one more iteration because each of P1, P3, P5, and P6 will have been applied once and R2 prohibits applying a production more than once. But this method of stopping the PS is crude, and has no relation to the task that the PS was supposed to accomplish, namely, identifying a food item from a set of attributes.

In summary, this example has illustrated the following points about the design of production systems.

1. Knowledge in a production system is modular.
2. Knowledge (productions and database items) is accessible to the whole system.

3. Productions affect each other only indirectly by altering the database.
4. The condition-part of a production can be arbitrarily complex.
5. In some production systems, a partial match on the condition-part will suffice to execute the production.
6. Various methods are employed for conflict resolution, when more than one production is applicable.
7. Productions can be ordered or organized to achieve some control strategy, but this violates the principle that all knowledge should be available all the time.
8. The interpreter must have some rules to allow it to quit executing productions.

It was mentioned at the beginning of this article that the "basic" production system formalism has been substantially modified and elaborated upon in AI implementations. The points above characterize the "basic" AI productions system (far removed from Post's original formulation). Implementation issues that have been raised in AI research are discussed in the next article.

### C3b. Issues in the Design of Production Systems

In this article we will assume familiarity with production systems (see the previous article) and concentrate on issues that arise in the implementation of AI systems. First we will present a brief discussion of the advantages and disadvantages of production systems. Then, in the light of this discussion we will examine the characteristics of task domains which are especially amenable to implementation with production systems. Finally we will consider five design issues which have received substantial attention from the AI community.

#### Advantages and Disadvantages of Productions Systems

Production systems, as described in the last section, have most often been used in artificial intelligence programs to encode the body of "real-world knowledge" which gives a system its power. The intent of this section is to judge their usefulness for this task. Rychener, 1976 has pointed out that production systems can be used to model other popular knowledge representation formalisms, like semantic nets and frames, thus, they are useful in the sense of being quite general. The purpose of this section is not to discover how much can be represented with production systems, but rather, what sorts of knowledge and tasks can be represented *well*.

The most striking characteristic of PSs is their *modularity*--individual productions are independent of other productions. This has facilitated the building of the large systems discussed below. Knowledge can be expressed in understandable chunks, which can be integrated into a larger whole. The reason for this autonomy of individual rules is the fact that each rule must carry all its relevant context with it. Since one cannot know in advance when a rule will fire, no assumptions can be made about the state of the world at the time of firing. The modularity is enforced by the limited channel of interaction; productions cannot communicate directly with each other, but only through the global database. However, there are indications (Rychener, 1976) that the modularity is harder to maintain as one moves to larger systems. As we begin to implement very large systems (e.g., with thousands of productions), it is possible that some kind of structural subroutining in the rule base will have to be imposed; this would violate the basic assumption of a single uniform knowledge base. This question will not be resolved until such large systems are actually built. However, even if modularity can be preserved, the lack of interaction among rules will often lead to inefficiencies which could be avoided if more explicit interaction were permitted.

Another advantage is the *uniformity* imposed on knowledge by the PS approach. Since all information must be encoded within the relatively rigid structure of productions, it can often be understood, either by another person or by the system itself. For example, Waterman's learning system (Waterman, 1970) used the fact that its rules were expressed in a constrained manner to facilitate automatic modification of them. Similarly, Davis' TEIRESIAS (see article Applications.B) analyzed its rule base, and formed *rule models*, representing typical forms of rules used in certain situations. These models were then used by the system itself, as a guideline for learning new rules; if the human expert suggested a rule that did not fit into TEIRESIAS' model, the system asked for clarification. Unfortunately, uniformity also has a disadvantage. It will sometimes unduly restrict the kinds of information which can be handled. Note that the advantages of modularity and uniformity accrue most

directly to theoretically motivated systems. As the theoretical constraints are relaxed to achieve greater power, these advantages are weakened.

A further advantage of the production system formalism is the *naturalness* with which one can express knowledge to the system. Most knowledge can be expressed fairly easily in the production format, although there is some indication Lenat, 1976 that some heuristics have to be unnaturally "forced" into productions. Nevertheless, the range of knowledge that can be expressed is encouraging; the expressiveness of a formalism is a strong theoretical criterion. See A1, A1, and B1.

There are, however, significant disadvantages inherent in the PS formalism. One of these is inefficiency of program execution. The strong modularity and uniformity of PSs results in high overhead. Since PSs perform *every* action via the recognize/act cycle, they are incapable of taking advantage of certain knowledge which becomes available. For example, Barstow, 1977 found occurrences in which he wished to group productions together, to take "larger steps"; the restrictions of his rule-based system prevented him from doing this. MYCIN (see below) actually incorporated a feature like this, in the form of a set of *knowledge tables*, and associated *specialized functions*, to perform more advanced actions in a single step. A paper by Lenat (1977) represents an effort to amend such flaws, by sacrificing some of the advantages of production systems.

A second disadvantage of the production system formalism is that it is hard to follow the flow of control in a program. In general, algorithms are more comprehensible when they are written in a procedural form (as in ALGOL) than they are when they are represented as a set of productions. This problem arises from two characteristics of production systems. First, since productions do not "call" each other, as procedures do, the flow of control in a program is obscured. In order to follow this flow, one must discern what modification to the database was effected by a production such that another production was enabled to fire. A second factor that obscures the flow of control is that productions are not typically very complex; each represents a relatively primitive action. Consequently the level of detail at which one must follow the actions of a program is relatively low; simple tasks may involve a large number of productions.

#### Applicable Domains for PSs used as a Representation of Knowledge

Davis & King, 1977 have characterized the domains in which production systems are especially useful. They are

- a) domains in which the knowledge is diffuse, consisting of many facts (e.g., clinical medicine), as opposed to domains in which there is a concise, unified theory (e.g., physics);
- b) domains in which processes can be represented as a set of independent actions (e.g., a medical patient-monitoring system), as opposed to domains with dependent sub-processes (e.g., a payroll program);
- c) domains in which knowledge can be easily separated from the manner in which it is to be used (e.g., a classificatory taxonomy, like those used in biology), as opposed to cases in which representation and control are merged (e.g., a recipe).

Rychener rephrased this in artificial intelligence terms: if we can view the task at hand as a sequence of transitions from one state to another in a *problem space* (Newell & Simon, 1972), we can model this behavior with production systems, since each transition can be effectively represented by one (or several) production firings. (*Learning* in this context can be viewed as the addition of new productions, to represent newly-discovered associations between states and successors). The database in such a system will serve to represent the current state of the model, analogous to a *program state vector* in a conventional computer program. Within this framework, the PS approach is similar to that of GPS--i.e., an executive distributing problem-solving effort among loosely-controlled methods.

If the task cannot be viewed this way (for example, the transitions are of different sizes, or some of the state description information is only implicitly available), it is probably not suited for production systems, and might prove very hard to implement in PS style. For example, Lenat, 1976 found that there were times when heuristic knowledge had to be "coerced" into productions, to be stored in the system; the expression was not "natural". Or, it might prove easy to implement in PS style, but with extra expense. For example, if the domain does have a single unified theory, the additional power of production systems as a representation will be wasted, but will entail the usual extra overhead.

#### Examples of Production Systems in AI Research

Numerous AI projects use production systems. The four mentioned here were picked for their innovativeness.

Waterman, 1970, implemented an *adaptive production system*. His task-domain was the game of draw poker. The program started with a set of fairly simple heuristics (coded as productions) for playing the game, and through various training methods it extended and improved them. The significant aspect of this system was the degree to which production systems facilitated the learning process. Since knowledge in production systems is represented in a constrained, modular fashion, the program was able to analyze and manipulate its own representation. Other examples of production systems which model learning are those of Hedrick (1976) and Vere (1977).

Lenat, 1976 modeled the process of discovery in mathematics using a production system, by viewing it as a heuristic search. His system, called AM (see article Applications.C2), started with a minimal knowledge of mathematical concepts represented as *frames* (see article B7). Heuristics, represented as productions, were used to expand these concepts and derive new ones. In the course of its operation, AM discovered a large number of already-known mathematical concepts, such as prime numbers, the arithmetic functions, etc, and also two interesting concepts which had never been discovered before. Lenat's system is especially important because of its sophisticated data-and control structures, and its large number (250) of heuristics.

The MYCIN system (Shortliffe, 1974; Davis & Buchanan, 1977; Medical Applications.B) acts as a medical consultant, aiding in the selection of therapy for a patient with a bacteremia infection. It carries out an interactive dialogue with a physician, and is capable of explaining its reasoning. It also includes a rule acquisition subsystem, which enables expert

physicians to expand or modify its rule base. MYCIN has about 200 productions which are used to identify a bacteremia infection from signs and symptoms. It is distinguished by merit of a *backward chaining* control structure, (see below), and *inexact reasoning*, in which confidence factors are attached to the conclusion part of each production, and are used to discriminate between hypotheses about the infection.

Rychener (1976) did not design a novel AI system, but rather implemented a number of extant systems using the production system formalism. In particular, he implemented Bobrow's STUDENT, Newell and Simon's GPS, Feigenbaum's EPAM, Winograd's SHRDLU, and Berliner's CAPS. His intent was not to produce high performance systems, but rather, to show that the production system formalism was a natural one for programming. His primary problem was that the uniformity and modularity of productions make it hard to build very complex control structures.

### Production System Issues

#### Complexity of Left- and Right-hand Sides of Production Rules

Production rules (usually just called "productions") consist of two parts: a left-hand side ("condition part" or "test") and a right-hand side ("consequent" or "action"). The system operates by repeatedly matching the L.H.S., and executing the symbols of the R.H.S.

In Post's original formalism, both parts were simple strings of symbols; the L.H.S. was matched against the database, and if the match was successful, the R.H.S. was inserted in its place.

This structure has been progressively extended as the size and complexity of systems has increased. The form of the antecedent was augmented; first to include predicate calculus forms with free variables, and then to allow arbitrary LISP function calls. Thus, the "test" part of a cycle is performed by EVALuating the L.H.S. of the rules; the functions called examine the database, and return the answer, binding the appropriate variables and constants if successful. Even in the most complex systems, however, the testing of the LHS is not permitted to have any side effects.

Similarly, the form of the consequent has been extended, first to include variables whose values are bound during the "test" phase, and then to allow arbitrary programs which are executed, rather than asserted. These programs usually specify actions in terms of a set of domain-specific conceptual primitives. In some systems (Riesbeck, 1975; Marcus, 1976; Rychener, 1976) these actions could include activation or deactivation of sets of other productions. Again, this represents a (radical) extension of the original production system formalism.

The complexity of the LHS and RHS is effectively an indication of the "grain size" of the production system - that is, how much knowledge is encoded into a single production. The production is the largest unit of knowledge recognized by the system; for larger operations, one must rely on sequences or collections of productions. Domain-dependent systems tend to require much larger "chunks" than their theoretical (or psychologically-motivated) counterparts. For example, Rychener's (1975; 1976) demonstration systems

tended to have low level rules, whereas MYCIN (Shortliffe, 1974) operated in larger steps, and HearsayII, in larger steps yet.

### Structure of the Rule Base and Database

As PSs have become bigger and more complex, questions of efficiency have imposed structure and order upon both the rule base (the set of productions) and the database (the set of facts about the task). These were originally unordered, to be accessed in some unspecified manner. The structure which has been imposed is not intended to affect the *power* of the system, in terms of the kinds of things that can be accomplished; rather, it is an effort to find computationally efficient ways to perform the given tasks. As such, it is more of an implementation detail than a theoretical question.

The most common way to organize the *rule base* is to *index* the productions according to the aspect of the system they affect. For example MYCIN Shortliffe; 1974 had all rules indexed according to the part of the database they accessed or changed. This made elimination of irrelevant rules in given situations much easier. As Lenat, 1977 point out, this can be carried further, to allow *partitioned* production memory (i.e., multiple production memories, which are active at different times), analogous to the use of *subroutines* in the algorithmic languages. Of course, this defeats one of the fundamental tenets of PS work, that all knowledge be potentially available at all times, but it does allow the designer to take advantage of inherent features of the domain (e.g., that certain types of knowledge are needed only at certain times).

To make the information in the *database* more accessible, it might be *ordered*. That is, new additions would be inserted at one end, and older elements would eventually be shifted out the other. This provides recency information -- the newest elements are those nearest the input end. This information might be used for conflict resolution (see (c) below).

The data base might also benefit from *partitioning*. For example, one might wish to have one database which contains the current state of the system, and another which is a record of the goals that the system is currently trying to accomplish.

A good example of the work in organizing the rule base and database is that of McDermott, Newell, & Moore, 1976. They used a series of *filters*, to reduce the cost of the "test" cycle; the filters, implemented via a discrimination net, performed a "pre-screening" of the set of productions, so that only a small subset of this set actually had to be fully tested. The filters depended upon information derived from a cross-indexing of the elements in the antecedents of productions, and the elements currently in the database. McDermott showed that, for his particular system, the time taken for the test cycle in the bare system (without the filters) was bilinear in the product of the sizes of the database and the rule base, and that the use of sufficiently-powerful filters could eliminate both these dependencies.

### Conflict Resolution

The discussion thus far has assumed that exactly one rule will have its LHS satisfied on each cycle, and that that rule will be fired. In practice, there are often many rules which

can be fired; the system is required to choose one from among this set (called the *conflict set*). The process of choosing is referred to as *conflict resolution*.

The conflict resolution phase is used to implement some basic system features, such as sequencing of production firing, attention focussing, interruptability, control of instability, etc.

Several approaches to conflict resolution have been used, including the following:

- a) choose the first rule to be satisfied, where "first" is defined in terms of some explicit linear order of the rule base
- b) choose the highest priority rule, where priority is defined by the programmer according to the demands and characteristics of the task.
- c) choose the most specific rule -- i.e., the one with the most specific expression in the L.H.S. In this context, specific means the opposite of general: A rule which has only one clause in its condition part will be easier to satisfy and more general than one with numerous clauses. The latter would fire only in very specific circumstances.
- d) choose the rule which references the element most recently added to the database
- e) choose a rule instantiation (i.e., rule/binding pair) which has not occurred previously
- f) choose a rule according to some specially-defined function
- g) choose a rule arbitrarily
- h) explore *all* the applicable rules in parallel

Various systems use different methods. For example, DENDRAL (Buchanan, Sutherland & Feigenbaum, 1969) uses (b), OPS (Forgy, 1977) uses a complicated sequence involving five of the six methods, and MYCIN simply performs exhaustive search, allowing *all* relevant rules to fire eventually. ZBIE (Siklossy, 1972) used a "best-match" criterion (similar to (c)), followed by production ordering (a). LISP70 (Tesler, 1973) offered the programmer a choice of (a), (c), or (f). AM (Lenat, 1976) and Hearsay II ( ) use complicated scheduling algorithms, which try to select the most "useful" rule to fire.

Good discussions of this issue can be found in (Davis & King, 1977) and (McDermott & Forgy, 1976). McDermott identifies two characteristics a production system should have: *sensitivity* (the ability to respond quickly to changes in the environment), and *stability* (the ability to carry out relatively long sequences of actions), and discusses the way in which different conflict resolution strategies serve these goals. He concludes that no single conflict resolution strategy can be completely satisfactory, with respect to the two characteristics mentioned; a combination of approaches must be used.



The *test* phase of the test/execute cycle can now be seen to split naturally into two phases: (i) *match* the set of productions against the database, to determine which ones are potentially applicable, and (ii) *resolve* the conflict by choosing the appropriate element(s) of this set. The preceding discussions of the complexity of rules, and the structure of the rule base and database, dealt with efficiency on the match phase; this section has discussed the resolution phase. Of the three phases (match, resolve, execute), the match has proven to be the most expensive in terms of computational resources.

### Direction of Inference

Classical PSs operate by matching the LHSs of productions, and executing the appropriate RHS. This approach is called *forward chaining*, or data-directed inference.

Some recent systems operate in the opposite direction -- that is, finding a RHS which provides the required information, and trying to instantiate the corresponding LHS. This is called *backward chaining* or goal-directed inference. This approach has been restricted to the performance-oriented systems such as MYCIN.

For example, if MYCIN had hypothesized that the current organism was bacteroides, it might use the rule given in the earlier discussion to verify this. It would do this by looking for information concerning the three premises of the rule. Some, like information concerning the site of the culture, would be available directly from patient data. Others, like the identity of the infection, might have to be deduced. In this case, MYCIN would search for rules which might provide such information, and try to establish them--that is, try to verify their premises. This would continue until the chain was traced back to given data.

The use of backward chaining creates PSs with different characteristics, and affects the type of domains to which they are applicable. The back chaining systems have tended to be larger than their forward chaining counterparts, with more domain knowledge encoded in the rule base; they are also less flexible, allowing fewer choices with respect to the accompanying control structures.

Other systems using backward chaining include PROSPECTOR (Article Applications.D2) and RITA (Applications.D3).

Note that this is not merely a feature of production systems, but rather a fundamental question of reasoning. Work on deductive inference (via, e.g., predicate calculus) has always recognized the difference between these two approaches; Hewitt's (1972) PLANNER provides a good explication of this dichotomy.

### Representation Vocabulary

As mentioned, the form of the antecedents and consequents of rules has been extended, from mere symbols to be matched, to arbitrary programs. While providing more power, this extension has unfortunately led to increased complexity. In particular, the range of expressions allowed by the rules is much harder to comprehend, especially if the language is not formally specified.

In practice, the language usually turns out to have a syntax like the predicate calculus (i.e., a predicate or function name, followed by a list of arguments). Theoretically, much more complicated structures are possible.

A significant aspect of the representation language concerns the choice of *conceptual primitives* -- i.e., the functions or predicates, in terms of which the rules and data base elements are coded. This set of primitives might be called the *vocabulary* of the system. Different systems will define their vocabulary at higher or lower levels, depending upon the task to be accomplished.

For example, MYCIN allowed conditions which examined the values of certain parameters of the current patient (using predicates like SAME, NOTSAME, KNOWN, and UNKNOWN), and actions which asserted values of other parameters (using a function named CONCLUDE). Rychener's (1976) PS implementation of EPAM (Feigenbaum, 1963) used a lower-level vocabulary with tests like COMPATNEG (meaning that the result of this compatibility test was negative) and actions like RESTIM (which signals the current stimulus-response pair to be fed through the net again). OPS Forgy, 1977 uses a set of match functions, including ANY and NOTANY (which perform element matching at the top level) and CONTAINS and NOTCONTAINS (which test subelements at arbitrary levels), and actions which perform basic symbol manipulation, including modification of other productions.

Earlier, the *match* phase -- the manner in which conditions are checked against the database -- was discussed. It is apparent now that this is a question of the language chosen; different languages will affect the ease of matching.

The significant trend in all these features, at least for the applications-oriented systems, is toward more powerful and domain-specific approaches, with fewer claims to generality. This often means giving the rule-writer access to the other parts of the system as well--i.e., not only providing him with high level primitives which are convenient for his particular task, but allowing him to change parts of the system which are usually prespecified. For example, the user might be allowed to dynamically adjust the conflict resolution strategy, so that his program is always using a strategy which is appropriate to the current situation. Or, the interpreter itself might be accessible for modification, so that, e.g., the match algorithm could be tuned to reflect efficiency considerations. The result of this is usually a gain in power, at the expense of modularity.

This article has reviewed some of the advantages, disadvantages, and issues of design for production systems. Many of the issues are still evolving, since production systems are a relatively recent tool in AI. For a more detailed discussion of these issues, the reader is directed to Davis & King, 1977.

#### C4. Procedural Representations of Knowledge

Representations such as predicate calculus are described as *declarative*; a fact is stored without regard for the ways in which it may be *used*. This article will deal with a contrasting class of representations, referred to as *procedural*. These are motivated by the thesis of *Procedural Embedding of Knowledge*: The useful knowledge of a domain is intrinsically bound up with the specialized procedures for its use (Hewitt, 1975, P. 196).

The extreme case of a procedural representation is a *program* for accomplishing some task. Compared to the declarative representation schemes, a program is *explicit*; all desired actions are expressed. For example, a system for operating in a mathematical domain might have a specific algorithm for performing differentiation, etc. In systems like this, all the knowledge is expressed in the form of code. All the interactions and procedure calls must be indicated explicitly; one lacks the ability to simply *search* for relevant knowledge.

The approach of using straightforward programs is suitable for many domains, but not, typically, for artificial intelligence. Most A.I. problems are sufficiently ill-structured that we cannot specify in advance exactly which knowledge will be useful, and exactly when and where each routine should be used.

Thus, the first 'procedural' systems in A.I. were somewhat more powerful than this, and less extreme in their procedural outlook. Although the *reasoning* was done via purely procedural knowledge, the *facts* were stored in a data base similar to those used by the theorem-proving programs.

Such an approach was used by several of the early artificial intelligence systems, one of which was Raphael's (1968) *SIR* program (article Natural Language.F1). *SIR* could answer questions about simple logical relationships, such as 'Is a finger part of a person?'. Its knowledge was stored in two forms: simple property attachment for facts, and LISP procedures for performing inference. Thus, to answer the question given above, it would use two facts (that a finger is part of a hand, and a hand is part of a person), and one *procedure* (a complicated routine called 'PARTRUGUQ', which basically traced 'part-of' links).

Woods (1968) implemented the most sophisticated of the early procedural systems. His program handled questions dealing with a subset of the Official Airlines Guide. The questions were 'pre-processed', from English into syntax trees. These trees were translated into function calls stated in a semantic *query language* for referencing the actual data base. For example, the question 'What American Airline flights go from Boston to Chicago?' would be expressed in the query language as:

```
(FOR EVERY X1/FLIGHT; EQUAL(OWNER(X1), AMERICAN-AIRLINES)
AND CONNECT(X1, BOSTON, CHICAGO); LIST(X1))
```

The *procedural* aspect resulted from the fact that these were actual function calls, to be executed in a LISP environment in which the appropriate functions had been defined. Thus, like *SIR*, the system operated with a data base of facts (the airlines schedule), and a set of procedures (which implemented the primitive functions of the query language). For example, the above query would be answered by a series of commands which would retrieve a list of all flights in the data base, find out which of those were owned by American Airlines, find out which of those went from Boston to Chicago, and print the final list.

This has some advantages, including the fact that the actual implementation of the procedures can be changed quite easily, if needed (e.g., if the organization of the data base is changed in some way).

In order to understand the motivation behind procedural representations, it is necessary to see the position that they are arguing *against*. The extreme form of declarative system is a predicate-logic-based theorem-prover, using a uniform proof procedure such as resolution (article Theorem Proving.B). Such a system performs deductions in a simple, straightforward way, essentially checking all knowledge in the system for applicability at each point.

This 'blind search' approach tends to result in serious efficiency problems; as the amount of potentially relevant knowledge increases, *combinatorial explosion* of the number of possible deductions occurs. Only limited kinds of control information are permitted, in the form of syntactic *strategies* such as unit-preference or set-of-support. There is no provision for encoding domain-specific heuristics or special-case information that we might happen to know. The nature of such systems is *declarative* (i.e., what things *can* be done), rather than *imperative* (i.e., what things *should* be done). A more extensive treatment of this position will be found in article B1 on logic, which should be read as a prerequisite for this one.

A representation such as predicate logic, expresses its knowledge in the form of assertions, without regard to their use. The statement 'All birds can fly' might be represented as:

$$\forall x \text{ Bird}(x) \supset \text{Flies}(x)$$

Note that there is no information here concerning how to *use* this information; the only restriction is that we must observe standard rules of inference. We might know that Clyde is a bird, and thus conclude that he can fly; or, we might know that Clyde cannot fly, and infer from this that he is not a bird. The crucial point here is that there is no way, within predicate calculus, to state which meaning is intended, or indicate which situations will probably be appropriate for the use of this knowledge.

We can consider declarative and procedural representations to be opposite ends of a spectrum. On the one hand, the declarative representations are general, flexible, and modular; on the other, procedural representations are more efficient. This efficiency is the most important aspect of procedural representations. The airlines question-answering system, for example, could be implemented using a deductive formalism, but we would be limited, the any question, to merely stating the *logical formula* of the answer, without giving any 'suggestions' for finding it; this might result in the combinatorial explosion mentioned earlier.

The advantages and disadvantages of these approaches will be discussed further in the last section. The next section deals with an expanded class of representations which is also referred to as 'procedural'.

### More Sophisticated Procedural Representations

#### a) the goals and methods of procedural representations

Even the powerful procedural representations used by Raphael and Woods were soon found to be inadequate for general artificial intelligence work; the restrictions on flexibility and modularity were too great.

For example, SIR was successful within its limited domain, but not easily extensible. The kinds of information that it could handle were limited; the high degree of interaction between the various parts of the system made modification difficult. If a new property (e.g., SIZE) was added to the system, it was necessary to formulate a complete set of routines for performing inference about that property. Similarly, Woods' program ran into the same difficulty of having to specify, for every property in the data base, a set of routines for working with it.

The ideal formalism would require the generality and power of the theorem-proving systems which were common in the late 1960's (such as the declarative systems which were discussed near the end of the previous section). In such a system, one can add or modify knowledge without difficulty, since the parts of the data base are independent of one another. Similarly, interactions happen 'automatically', in that the theorem prover will find all theorems which happen to be relevant at a given point in the deduction process. The best example of this work was the QA3 system, developed by Green, 1969. Thus, there was a trend, in the late 1960's, to search for a merger of the two forms -- combining the flexibility of logic-based systems, with the efficiency of the earlier procedural versions. This work was also referred to as 'procedural', but it is important to recognize that the class of systems included here is wider than the versions discussed in the previous section. The remainder of this article will use the term 'procedural' to refer to this larger class. The essence of this approach is to represent knowledge (of the type typically encoded in predicate calculus) *with instructions for its use*. Winograd, 1974 discusses the question of the use of a representation, in terms of four sub-issues:

*control* - what should be done next ?

*retrieval* - what knowledge is potentially relevant ?

*matching* - how does that knowledge apply to the problem at hand ?

*application* - what are the results of using it ?

The declarative approach to knowledge deals primarily with the latter two; the *procedural* approach is an attempt to handle all four. In the remainder of this article, we will use the term 'control' to cover the issues involved in both control and retrieval.

In a sense, the difference between declarative and procedural representations is a result of the kinds of domain referred to; declarative representations typically describe an external world, whereas procedural representations also encode information about the kinds of processing which are permitted. This point is made explicitly by Hayes, 1977.

The thrust of the work in procedural representations has been to try to find better ways of expressing this control information. If we look at the process of deduction as consisting of searching a tree, the procedural representations are seen as an effort to provide the ability to indicate which branches should be explored.

The information about how to use a piece of knowledge might concern various aspects of processing. One form of control is to indicate the 'direction' in which an implication can be used. To continue with the example given earlier, a typical procedural piece of knowledge (with 'procedural' used in the general sense indicated above) might be stated as:

(IF-NEEDED FLIES (X)  
TRY BIRD (X))

Thus, if we are trying to prove that Clyde can fly, this tells us to try to prove that he is a bird. This knowledge will not be used in the other direction; thus, if we learn that Fred is a bird, this will not enable us to assert anything about Fred being able to fly.

Another use of 'procedural' knowledge occurs when we try to specify which knowledge will be relevant at a particular point. For example, if we want to prove that something is a fish, and we suspect that the 'theorems' (i.e., pieces of knowledge) THEOREM1 and THEOREM2 will be useful, we might write:

(GOAL FISH (X) (USE THEOREM1 THEOREM2))

which would express our intention to the system.

In essence, what we are doing here is straightforward deduction, of the kind provided by *resolution* or *natural deduction* for the predicate calculus. However, the deduction here is *controlled* somewhat; we have told the system how and when it can use the knowledge that it has been given.

There have been three major *methods* of specifying control information:

- 1) to specify control by the way in which one states the facts; this is the approach used in Planner (Hewitt, 1972), and in the examples above.
- 2) to encode the representation language at a lower level, so that the user has access to the set of mechanisms for specifying the reasoning process; this is the approach of CONNIVER (Sussman & McDermott, 1972).
- 3) to define an additional language, used to express control information, which works together with the representation language; this idea was the foundation for the GOLUX project (Hayes, 1973), and for Kowalski's (1975) predicate calculus programming.

Of the three approaches, the work on Planner was seminal for the current generation of procedural representations, as well as being the most widely used. Accordingly, this article will devote most attention to this one effort. Since approach (2) is often combined with (1), it will be discussed at the same time. The last approach, which is quite different from the other two, will be considered (briefly) later.

## b) PLANNER and successors

Planner was designed as a representation and control language, to expedite the process of deduction. The specific intent of the Planner system was to subordinate the deductive system to the control structure. That is, the concern was not with the class of inferences which were *possible* (as would be the focus in theorem-proving), but with the inferences which would actually be *made*. This can create problems; there are some quite straightforward deductions which Planner is unable to make. This will be discussed later.

Full PLANNER was never implemented. A subset, referred to as micro-Planner, was implemented by Sussman, Winograd, & Charniak, 1970, and used in a few systems, most notably Winograd's (1972) thesis. The examples given here will use the syntax of micro-Planner, since that is the best-known version.

Some of the features of Planner include the ability to specify whether theorems should be used in a forward or backward direction, and the ability to recommend the use of specific theorems in given situations. In fact, the ability to recommend pieces of knowledge was somewhat more general than indicated previously. Besides recommending theorems by name, it was possible to suggest general *classes* of theorems, via the use of *filters*. For example, if we were trying to prove that something was a fish, and we wished to indicate that the most promising way was to use theorems about zoology, we could write

(GOAL FISH(X)(FILTER ABOUT-ZOOLOGY)).

Two other features were significant. First was a method of procedure call referred to as *pattern-directed invocation*. When trying to solve a problem, it was not necessary to tell the system which theorems (i.e., implications) to try. Rather, the system itself could search for theorems whose structure indicated that they might be relevant to the goal at hand. For example, if we are trying to prove that block A is on block B (expressed as, e.g., (ON A B)), we would only try theorems whose results would assert something of this nature. Although the idea of pattern-directed invocation is common in theorem-proving, its extension to programming languages was novel.

The other interesting aspect of Planner was its capability to perform *search*, implemented as *backtracking*. If Planner was ever required to make a decision (e.g., selecting a particular block to be the base of a tower that it was building), it recorded this as a *choicepoint*. If, later on, a failure occurred, the system would back up to this point, make a different decision (i.e., select a different block), and try again. Again, this capability for automatic backtracking was common in theorem-proving work, but new to programming languages (although the GPS system Search.02 had contained a similar feature).

Planner thus serves as the language in which both the *problem* (i.e., the logical aspect) and the methods of solution (i.e., the *procedural* aspect) are stated. The intent is that the user be able to state as much or as little subject-dependent knowledge as required.

More information on PLANNER can be found in chapter AI Languages on AI Programming Languages.

The most extensive use of PLANNER (actually, micro-Planner) was as a part of Winograd's (1972) SHRDLU system (Natural Language.F5). In this case, PLANNER was used

to represent information about a restricted domain of toy blocks, and move those blocks about.

For example, a theorem for putting one block on top of another was defined:

```
(DEFTHEOREM TC-PUT
  (THCONSE (X Y Z) ( PUT $?X $?Y)
    (CLEAR $?Y (SIZE $?X) $?X)
    (SUPPORT $?Y (SIZE $?X) $?X)
    (THGOAL ( GRASP $?X) (THUSE TC-GRASP))
    (THSETQ $?Z (TCENT $?Y (SIZE $?X)))
    (THGOAL ( MOVEHAND $?Z) (THUSE TC-MOVEHAND))
    (THGOAL ( UNGRASP) (THUSE TC-UNGRASP))))
```

which essentially says 'to put a block (X) on another block (Y), make sure there is space on Y, and that Y can support X; then pick up X, move it to the appropriate space on Y, and let go of it.'

A number of artificial intelligence languages followed Planner, including CONNIVER Sussman & McDermott, 1972, QA4 Rulifson, Derkson, & Waldinger, 1972, POPLER Davies, 1972, and QLISP Reboh et al., 1976. These involved a mixture of the approaches (1) and (2) above. As Hayes, 1977 points out, there was confusion in many cases about whether the ultimate goal was a *representation language* or a *programming language*. Thus, comparison was difficult.

For further discussion of many of these languages see the Handbook chapter on AI programming languages.

#### (i) Planner languages and extended logic

In some cases, procedural representations merely enable us to add control information to a normal deductive system. However, there are some inferences permitted by procedural representations which are beyond those found in classical logical systems. These include the various forms of 'default' reasoning Reiter, 1978. One form of default, as implemented in Planner, is the THNOT primitive. An expression

```
(THNOT A B)
```

says 'unless we can prove that A is true, assume that B is true'. An example of this might be a form to be executed when we are discussing a bird

```
(THNOT OSTRICH (X)
  ASSUME FLIES (X))
```

which says 'assume that X can fly, unless we can deduce that he is an ostrich.'



THNOT can only function correctly if certain aspects of our knowledge are 'complete'. In the above example, we assume that, if X were an ostrich, we would know that fact (or, we would have some way to deduce it). If we do not have this 'complete' knowledge, the system might make incorrect inferences. This is not necessarily a serious problem; there might be times that we want the system to 'jump to conclusions'. This will be discussed later.

THNOT and similar functions take us beyond the realm of ordinary logic, since they violate the property of *monotonicity*. Monotonicity states that, if a conclusion is derivable from a certain collection of facts, the same conclusion remains derivable if more facts are added. Obviously, Planner, with the THNOT primitive, does not observe this property. In essence, procedural and declarative systems implement *different* logics; this is discussed in Reiter, 1978. As Hewitt, 1972 points out, the logic of Planner is a combination of classical logic, intuitionistic logic, and recursive function theory. Winograd, 1979 outlines a taxonomy of 'extended inference modes' which are outside the provision of ordinary logic.

### (c) the 'separate language' approach

The work on the (3) approach, by Hayes and Kowalski, has had a radically different emphasis. The intent here is to use a standard *representation language*, like the predicate calculus, with an associated inference structure, but to build 'on top of it' a separate *control language*, for restricting the inferences which will actually be made. The control language is subservient to the representation language; that is, it should not affect the final results which are achieved by the system, only the efficiency with which it performs. Unlike the Planner family, for instance, the languages in this class preserve monotonicity.

This approach has not had the impact on artificial intelligence of the Planner work. However, it has provided a clarification of the issues involved.

Since this approach is so different from the other two, it will not be discussed further; the term 'procedural' is usually used in artificial intelligence to refer to representations of the Planner class.

The next section will consider the effectiveness of procedural representations in A.I.

We have discussed three phases in the development of procedural representations: simple programs, programs with data bases, and more general A.I. systems with pattern-matching capabilities. Within the last group, we have identified three approaches, typified, respectively, by Planner, Conniver, and Golux, of which the first is dominant.

In this section we will evaluate the procedural representations, with respect to their usefulness in artificial intelligence and their differences from declarative systems. The discussion here will consider only currently developed systems; it is possible that future procedural representations might not have the same set of advantages and disadvantages discussed below.

The work on procedural representations occurred mostly in the period from 1969 to 1973, during which a number of new representation languages were developed to test out such ideas.

This work has now changed direction somewhat, for reasons which will be discussed later. It is now possible, given the perspective of time, to evaluate the contributions and failings of such efforts. A similar discussion of these issues appears in Winograd, 1975.

First, the advantages of procedural representations. Chief among these is the ability to represent *heuristic* knowledge -- i.e., domain-dependent information which might enable more effective deduction processes. This includes information about whether a theorem should be used in a backward or forward direction, which knowledge should be used in a given situation, and which subgoals should be tried first.

The most important result of this ability to encode heuristic knowledge is the *efficiency* realized by such systems. We are not dependent on general search to solve our problem, but can put in as much knowledge of the particular as we happen to have. Efficiency of this sort was the motivation behind most of the work on procedural representations.

A related advantage is the ability to perform *non-logical* inferences, as discussed in the previous section. These are methods, such as default reasoning, which cannot easily be expressed in a more uniform deduction system (but, for an example of an attempt to include this in predicate calculus, see McCarthy, 1977 ). Winograd, 1979 discusses this issue further, arguing that these types of non-logical reasoning are necessary in a system which attempts to model human reasoning.

Another advantage accruing to procedural representations is the ability to do *modeling*. That is, certain actions or events are difficult to represent in a declarative way; a naive declarative formulation will often encounter the *frame* problem, concerning what effects actions have on the state of the world.

To take an example from the 'blocks world', if A is on B, and we move B, we should update the location of A; but, if we move C (a completely different block), the location of A does not change. The propagation of those facts which have not changed (which causes combinatorial problems in a declarative system) is much easier in a procedural system: we simply 'perform' the desired actions, and update the world accordingly; the data base represents the current state of the world. This method does have one drawback, discussed below. Interestingly, the frame problem is avoided here for precisely the same reasons as in *analogical* representations, discussed in article B6.

There are, however, some serious difficulties associated with the procedural approach (at least, as it has been developed thus far).

One of these concerns the fact that many procedural systems are not *complete*; there are cases in which one possesses all the facts required to reach a certain conclusion, but the language is not powerful enough to make the required deductions. One example of this is Planner's handling of disjunction. As Moore, 1975 points out, if we are given the facts (expressed in a simple propositional calculus):

$$\begin{array}{l} A \supset C \\ B \supset C \\ A \vee B \end{array}$$

we should be able to infer C; however, Planner lacks the machinery for doing this. Note that a resolution-based theorem prover could do this easily; thus, this is one area in which Planner is inferior to declarative formalisms.

Note, however, that completeness is not necessarily desirable. There are many cases in which we want the system to work quickly, and are not so concerned about accuracy. If the system is complete, it might take a long time to find a particular answer (or, to conclude that it cannot find the answer), when we would prefer that it return quickly, confessing its ignorance.

A different aspect of deductive systems is *consistency*; a system is consistent if all its deductions are correct -- that is, if the conclusion necessarily follows from the premises. Again, most theorem-proving systems have this property, but procedural systems often do not. For example, the use of default reasoning can introduce inconsistency, in the presence of incomplete knowledge. Thus, if we use the fact that Fred is a bird to conclude that he can fly, and later discover that he is an ostrich, we will have inconsistency. Inconsistency can be harmful because, in certain kinds of deductive systems (including most of those based on first-order logic), it can lead to completely wrong inferences. Hewitt, 1975 refers to this as the 'Garbage In -- Garbage Out' principle.

Like completeness, consistency is not necessarily desirable. McDermott & Doyle, 1978 argue that much of our reasoning is done by revising our beliefs in the presence of new information. Similarly, Hewitt indicates that most of our knowledge is not *absolute*; we regularly accept caveats and exceptions. Also, if we control the reasoning sufficiently tightly in the presence of inconsistency, the 'garbage in -- garbage out' effect will not operate.

Another drawback of procedural representations (in their current form) is that the control information sometimes 'gets in the way'. For example, if we want to prove  $(A \wedge B)$ , Planner enables us to express this as  $(\text{THAND } A \ B)$ , which essentially says 'prove A, then B'. There is no way to omit this type of information, even if we want to. Similarly, we are required to state for each theorem, whether it is to be used in a forward or backward direction, even if we have no idea, *a priori*, which to choose.

Procedural systems also suffer from a lack of generality. With uniform theorem provers it is necessary only to state the appropriate knowledge; the reasoning system will perform the correct deductions. With procedural systems, we must encode not only the knowledge, but also instructions for its use. Thus, more work is required to adapt a system for a new domain. In this respect, procedural systems have not attained the generality which was one of the major goals.

The *modeling* which procedural languages facilitate, discussed earlier, has an inherent limitation: we are unable to *reason* about different states of the world. That is, we cannot represent statements such as 'this tower is taller now than it was five minutes ago', since the modeling approach does not provide a way of talking about states of the world existing 'now' and 'five minutes ago'.

In general, procedural systems experience difficulty in reasoning with some kinds of *incomplete* information (this is not related to the question of deductive completeness, discussed above). As mentioned earlier, the 'default' reasoning can produce inconsistent

results in the absence of complete information about the domain. Similarly, the *modeling* discussed in the previous paragraph requires complete information to function correctly. Thus, domains of incomplete information are often ill-suited for procedural representations. Note that this characteristic is shared with *analogical* representations. (B6)

Another feature which is sacrificed in the procedural approach is *modularity*. In a procedural representation, the interaction between various pieces of knowledge is high. Thus, a change (or addition) to the knowledge base might have more far-reaching effects than a similar change in a base of predicate calculus assertions. In essence, this is the price we pay for the greater degree of control permitted in procedures. Note that this criticism is most applicable to 'pure' procedural systems. Languages like Planner, with pattern-directed invocation, retain a somewhat modular flavor; in fact, one of the virtues of Planner is that it provides a uniform way of expressing procedural knowledge.

A disadvantage which specifically appears in Planner is the use of automatic backtracking. Planner will try as many ways as it can to achieve a particular goal, sometimes retrying different versions of a goal which a more 'intelligent' system would have rejected after the first failure. This leads to obvious problems concerning efficiency. Note that this is, in some sense, another version of the problem of combinatorial explosion which occurs with uniform theorem provers. This particular problem has been removed in the later procedural languages by forcing the user to indicate explicitly where, when, and how he wants backtracking to occur.

The consensus among artificial intelligence researchers (see, e.g., Moore, 1975, ch. 5) is that there should be ways to embed control in a deductive system, but that the methods tried thus far have many flaws in them. For example, two specific criticisms have been directed at Planner's method of specifying control. First, it is too *local*; Planner is unable to consider the overall shape of the goal tree, which would enable it to search the tree more effectively. Second, Planner is unable to 'reason' about its control information; ideally, it should be able to make decisions on the basis of facts about control, as it can now make decisions on the basis of facts about the world.

The interest in procedural representations has been replaced by other formalisms such as frames (B7). These embody a slightly different approach to the problem of organizing and using knowledge.

## C5. Semantic Primitives

Many AI programs are problem solvers operating in a specific, limited problem domain. Others--like GPS, REF-ARF, and general-purpose theorem provers--aim at applicability to problems from a wide variety of domains. In either case, the system must include a language for stating problems and a representation, in some combination of data structures and procedures, of the knowledge that can be brought to bear in problem solving.

If the program is a specialized one, its representations of problems and of knowledge can be tailored very specifically to the problem domain. If the program is general, it is likely to work by applying a single, general algorithm to problems that, however varied, have been reduced to something like a least common denominator. REF-ARF, for example, required all problems to be cast in the terms of a particular programming language; GPS, that they be stated in a language of objects and operators together with a Table of Connections; theorem provers, that they be axiomatized in the predicate calculus.

There is a class of problems to which neither of these two approaches (the specialized problem solver and the generalized one) applies. By definition such problems require that input to the system be in some subset of natural language; so a system requiring previous translation to some other format would miss the mark. The problem might be to paraphrase the input, to translate it to another natural language, to answer questions about it, or simply to store it in some internal representation on the basis of which the other tasks might be carried out. One aspect of these tasks involves algorithms for parsing and, to some extent, text generation; these are considered in chapter , Natural Language. The aspect of concern in this article is one on which less progress has been made. The question is, what would be a good internal representation of natural language input?

This brings us to a second characteristic of the class of problems considered here: The natural language to be represented is relatively unspecialized in its content. What is sought is a representation for problems and knowledge, not in a world of toy blocks (Winograd, 1972), or lunar rocks (Woods, 1968), or of any other subject for a specialized data base [LIFER], but in a world of everyday human action and commonsense understanding. Typical problems have as their subject matter events like eating in a restaurant or going to a birthday party. Expert special-purpose problem solving systems are not what one wants for such topics. The requirement is for a representation that can accommodate general human knowledge.

Several of the other articles in this chapter discuss possible formalisms for the internal representation of natural language input and of the commonsense knowledge needed to operate on that input in appropriate ways. Such formalisms include the predicate calculus, semantic nets, and frames. Whatever the choice of formalism, however, there is another major question concerning the vocabulary to be used within that formalism. In a predicate calculus representation, for example, what predicates are to be used? In a semantic net, what node and link types should be provided. The use of *semantic primitives* gives one possible kind of answer to such questions.

The term *semantic primitive* is very far from having a clearcut definition. As a starting point, one may think of a primitive as any symbol that is used but not defined within the formalism. The term is so used by Wilks, for example, who accordingly concludes that "primitives are to be found in all natural language understanding systems--even those . . .

that argue vigorously against them." (Wilks, 1977c, p. 19) A second and narrower usage takes semantic primitives to be elements of meaning into which the meanings of words and sentences can be decomposed; examples of such work come from linguistics (e.g., Jackendoff, 1975; Jackendoff, 1976) and psychology (e.g., Miller, 1975; Miller, 1976) as well as from AI.

Additional issues exist about what primitives "really are," how they may be used in reasoning, and what alternatives there are to using primitives. Winograd, 1978 provides a general survey and analysis of such questions. The remainder of this article illustrates some of the possible positions through a review of the two major AI systems for language understanding that are characterized by their authors as using semantic primitives.

### Wilks's System

Yorick Wilks, now of the University of Essex, has been developing a natural language understanding system since about 1968 (Wilks, 1968; see article VE2, Wilks's Machine Translation Work). The system accepts paragraphs of English text, producing from them an internal representation that is a structure made up of semantic primitives. From this structure, a French translation of the input is generated. The translation serves as a test of whether the English has been "understood," and the test is objective in a way that inspection of the internal representation would not be. The translation task also has the advantage, Wilks suggests, that correct translation of the input may often require only a shallower understanding than would the ability to answer arbitrary questions about it. Consistently with these reasons for the choice of translation as a task, most of the effort in Wilks's system is spent in converting the English input to the internal representation.

The first major problem that Wilks addressed was the resolution of word-sense ambiguity; for this was the problem on which earlier machine translation efforts had foundered (see article VE1, Mechanical Translation Overview). For example, in "the policeman interrogated the crook," the program must be able to determine that "crook" means a criminal, not a shepherd's crook. In "the old salt was damp," it is necessary to look beyond the sentence to the surrounding context to decide whether "salt" means a chemical compound or a sailor. Some other problems of ambiguity that Wilks has also treated include the following:

- (1) Resolving prepositional (case) ambiguity, as in "he put the number in the table" ("in" as "part of," "table" as a list) versus "he put the fork in the table" ("in" as "into," "table" as a physical object).
- (2) Resolving pronoun references on the basis only of conceptual knowledge of word meanings, as in "I bought the wine, sat on a rock, and drank it."
- (3) Resolving pronoun references where additional world knowledge is required, as in "the soldiers fired at the women, and I saw several of them fall."

Finally, Wilks emphasizes the importance of allowing for new or extended word senses, as in "my car drinks gas."

The general idea of Wilks's approach, which he calls *preference semantics*, is to use knowledge of possible word meanings to disambiguate other words. Part of the meaning of "drink," for example, is that it prefers a fluid object; and part of the meanings of "wine" and "gas" is that they are fluids. If the best fit among possible word senses does not satisfy all preferences (such as the preference of "drink" for an animate subject), then an extended word sense can be accepted. The formalism within which preferences are expressed, Wilks suggests, is closer to a *frame representation* than to a *semantic net*.

As the description above should make clear, a central requirement in Wilks's system is a dictionary distinguishing among the various senses of words that can appear in the input text. Definitions in the dictionary use a vocabulary of semantic primitives, grouped into five classes. Examples from each class are given below.

#### Substantives

MAN (a human)  
 STUFF (a substance)  
 PART (a part of an entity)

#### Actions

CAUSE (causing something to happen)  
 BE (being as equivalence or predication)  
 FLOW (moving as liquids do)

#### Cases

TO (direction toward something)  
 IN (containment)

#### Qualifiers

GOOD (morally correct or approved)  
 MUCH (much, applied to a substance)

#### Type indicators

HOW (being a type of action—for adverbial constructions)  
 KIND (being a quality—for adjectival constructions)

In addition to the primitive elements, of which there are currently over eighty, Wilks uses several elements, distinguished by names beginning with an asterisk, that are defined as equivalent to a class of primitives. For example, \*ANI (animate) encompasses MAN, FOLK (a human group), BEAST (a nonhuman animal), and certain others. A typical definition using the primitives is that for one sense of the word "break":

```
((*HUM SUBJ)
 (*PHYSOB OBJE)
 (((NOTWHOLE KIND) BE) CAUSE) GOAL)
 (THING INST)
 STRIK)
```

In English, this says roughly that "break" means a STRIKing, done preferably by a HUMAN SUBJECT and preferably with an INSTRUMENT that is a THING, with the GOAL of CAUSING a PHYSICAL OBJECT to BE NOT WHOLE. Words other than verbs are also defined by such structured formulas. For example, one of the senses of "crook" is

((((NOTGOOD ACT) OBJE) DO)  
(SUBJ MAN))

--i.e., a man who does bad acts. A detailed description of the syntax of such word-sense definitions, or *semantic formulas*, is given in Wilks, 1977c .

The completed representation of a text is a structure made up of such word-sense formulas. At a level corresponding to the clause or simple sentence, formulas are arranged into triples, or *templates*, standing for an agent, an action, and an object; and any of the three may itself be qualified by other formulas. For example, "Small men sometimes father big sons" would be structured as follows:

$$\begin{array}{ccccc}
 [\text{man}] & \longleftrightarrow & [\text{father}] & \longleftrightarrow & [\text{sons}] \\
 \uparrow & & \uparrow & & \uparrow \\
 [\text{small}] & & [\text{sometimes}] & & [\text{big}]
 \end{array}$$

Here the bracketed English words should be imagined as replaced by the formulas representing their proper senses. Relationships among templates are indicated at a still higher level of structure. For more detail, see article VE2.

What is the status of the primitive vocabulary in Wilks's system First, he argues, primitives are not essentially different from natural language words. A semantic description in terms of primitives is just a description in "a reduced micro-language, with all the normal weaknesses and vagueness of a natural language" Wilks, 1977c. The justification for using a language of primitives, then, is just that it provides "a useful organizing hypothesis . . . for an AI natural language system" (Wilks, 1977c).

Second, individual primitives have their meaning in the same way that English words do: neither by direct reference to things, nor by correspondence to nonlinguistic psychological entities, but only by their function within the overall language.

Third, in light of the nature of primitives, there is no one correct vocabulary for a primitive language, any more than there is a correct vocabulary for English. The test of the adequacy of a particular set of primitives is an operational one: the success or failure of the linguistic computations that use it. As suggestive evidence that Wilks's own set of primitives will indeed turn out to be adequate, he observes that it is very similar to the eighty words that are most frequently used in definitions in Webster's Third New International Dictionary.

Finally, there are nevertheless some general considerations to be taken into account in choosing a set of primitives. Wilks identifies the following properties as desirable for the set to have (Wilks, 1977c, p. 3):

1. **Finitude:** The number of primitives should be finite and should be smaller than the number of words whose meanings it is to encode.
2. **Comprehensiveness:** The set should be adequate to express and distinguish among the senses of the word set whose meanings it is to encode.



3. Independence: No primitive should be definable in terms of other primitives.
4. Noncircularity: No two primitives should be definable in terms of each other.
5. Primitiveness; No subset of the primitives should be replaceable by a smaller set.

A qualification should be noted concerning the property of comprehensiveness: the definition in primitives of a word-sense is not required to be exhaustive of meaning. Wilks cites "hammer," "mallet," and "axe" as terms among which a representation in primitives cannot be expected to distinguish (Wilks, 1977c, p. 11). In addition, the definition of a term is not expected to say everything; Wilks distinguishes between word meanings, which definitions express, and facts about things. The definition of "water," for example, might say that water is a liquid substance, but not that water freezes into ice. Facts of the latter sort are expressed in commonsense inference rules, which are separate from the dictionary and are used only as a last resort in disambiguation.

### Schank's Conceptual Dependency

Another system based on semantic primitives is the Conceptual Dependency theory of Roger Schank (now of Yale University). The theory has evolved since about 1969 (Schank & Tesler, 1969; Schank, 1972); its most distinctive feature, however, which is the attempt to provide a representation of all actions using a small number of primitives, was first introduced in 1972 (Schank, Goldman, Rieger, & Riesbeck, 1972).

There are significant differences between Schank and Wilks, both in the general outline of their systems and in their views of primitives. Wilks's system, for example, is oriented toward the task of machine translation, whereas Conceptual Dependency theory makes broader claims. First, Schank emphasizes task independence; and in fact the theory has been used as the basis of programs that, among other things, can paraphrase an input, translate it to another language, draw inferences from it, or answer questions about it. Second, the theory is offered not only as a basis for language-understanding computer programs but also as an intuitive theory of human language processing.

Consistently with these claims, Schank holds that it is the business of an adequate representation of natural language utterances to capture their underlying conceptual structure. A first requirement is that the representation be unambiguous, even though the input may contain syntactic ambiguity, as in "I saw the Grand Canyon flying to New York," or semantic ambiguity, as in "The old man's glasses were filled with sherry." The speaker of an ambiguous sentence usually intends an unambiguous meaning; so the representation is expected to reflect only the most likely version of what was intended.

A second requirement is that the representation be unique--that is, that distinct sentences with the same conceptual content should have the same representation. Some examples of groups of sentences that are all represented the same way are

- I want a book.
- I want to get a book.
- I want to have a book.

and

Don't let John out of the room.

Prevent John from getting out of the room.

The principle of uniqueness of representation has been characterized as the basic axiom of the system. It also has been justified as accounting for human abilities to paraphrase and translate text, and to do so in a computationally efficient manner. The problem of paraphrase--"how sentences which were constructed differently lexically could be identical in meaning" (Schank, 1975c) --is a major theme throughout Schank's work.

To obtain unique, unambiguous representations of meaning, Schank's system relies principally on a set of eleven primitive ACTs (Schank, 1975a pp. 40-44; Schank & Abelson, 1977 pp. 12-15). These are as follows:

Physical acts

PROPEL	apply a force to a physical object
MOVE	move a body part
INGEST	take something to the inside of an animate object
EXPEL	force something out from inside an animate object
GRASP	grasp an object physically

Acts characterized by resulting state changes

PTRANS	change the location of a physical object
ATRANS	change an abstract relationship, such as possession or ownership, with respect to an object

Acts used mainly as instruments for other acts

SPEAK	produce a sound
ATTEND	direct a sense organ toward a stimulus

Mental acts

MTRANS	transfer information
MBUILD	construct new information from old information

There are several other categories, or concept types, besides the primitive ACTs in the representational system. They are:

Picture Producers (PPs), which are physical objects. Some special cases included among the PPs are natural forces like wind and three postulated divisions of human memory: the Conceptual Processor, where conscious thought takes place; the Intermediate Memory; and the Long Term Memory.

Picture Aiders (PAs), which are attributes of objects.

Times.

Locations.

Action Aiders (AAs), which are attributes of ACTs.

Only a little work has been done on reducing these latter categories to a primitive set; see Russell, 1972 and Lehnert, 1978 on the analysis of PPs, and Schank, 1975a on the treatment of PAs as attribute-value pairs.

Detailed rules are provided for the ways that elements of these categories can be combined into meaning representations. There are two basic kinds of combinations, or *conceptualizations*: One involves an actor (a PP) doing a primitive ACT; the other involves an object (a PP) and a description of its state (a PA). Conceptualizations can be tied together by relationships of instrumentality or causation, among others.

The primitive elements that occur in conceptualizations are not words, according to Schank, but concepts; they reflect a level of thought underlying language rather than language itself. Consequently, representations of text in Conceptual Dependency are said to be language-free. The task of translation, then, becomes only one task among many; it is accomplished by parsing from one language into Conceptual Dependency, and then generating text in the second language from the Conceptual Dependency representation.

The notion of language-free primitive concepts requires explication. For Schank, as for Wilks, the justification for using primitives is functional. Schank differs from Wilks, however, in his choice of a general sort of function to be optimized, as well as in his view of primitives as language-free and psychologically plausible. Schank particularly emphasizes the computational advantages, to both programs and people, of storing propositions in a canonical form. (Schank, 1975b p. 38) This requires, in Schank's view, that information implicit in a sentence be made explicit. (Schank, 1975a pp. 9, 16; Schank & Abelson, 1977 p. 11) Obtaining the implicit information in turn requires inferencing. (Schank, 1975a p. 17) And it is as an aid to inferencing that the use of primitives receives its most important justification. Schank says, for example:

Rather than stating that if you see something, then you know it and if you hear something then you know it and if you read something then you know it and so on, we simply state that whenever an MTRANS exists, a likely inference is that the MTRANSed information is in the mental location LTM [Long Term Memory] (our representation for "know"). This is a tremendous savings of time and space. (Schank, 1975b p. 40)

Each primitive ACT, then, entails its own set of inferences. As a fuller example, the following are the main inferences from the fact that X PTRANSed Y from W to Z:

- 1) Y is now located at Z.
- 2) Y is no longer at location W.
- 3) If Z = X, or Z is human and requested the PTRANS, then Z will probably do whatever one ordinarily does with Y. Moreover, Z probably will become pleased by doing this. (Schank, 1975a p. 71)

Such inferences provide both the criterion for choosing a set of primitives and the definition of what primitives are. The primitive ACTs, Schank states, (Schank & Abelson, 1977 p. 75; see also Schank, 1975a p. 68, Schank, 1975b p. 40) are no more than the sets of inferences to which they give rise. Moreover:

The theoretical decision for what constitutes a primitive ACT is based on whether the proposed new ACT carries with it a set of inferences not already accounted for by an ACT that is already in use. Similarly, a primitive ACT is dropped when we find that its inferences are already handled by another ACT. (Schank, 1975b p. 40)

In his earlier work (Schank, 1973a, p. 14; see also Schank, 1975c, p. 238), Schank claimed that the primitive ACTs of Conceptual Dependency, together with some set of possible states of objects, were sufficient to represent the meaning of any English verb. It soon became clear, however, that additional mechanisms would be needed for a general-purpose language-understanding system. For example, there are problems of quantification and of metaphor, which have not yet been addressed (Schank & Abelson, 1977, p. 167); there are problems raised by the fact that natural-language communications often presuppose a great deal of background knowledge, some of which has to do with the typical course of events in commonplace situations like eating in a restaurant or taking a bus (see article on B7); and, of particular importance with respect to the use of primitives, there are problems arising from the fact that Conceptual Dependency generally expresses the meaning of an action verb only in terms of its physical realization. One example is the reduction of "kiss" to "MOVE lips to lips." (see Schank & Abelson, 1977, p. 130) The inadequacy of this representation becomes especially apparent in light of the claim that no information is lost by the use of primitive ACTs to represent actions. (Schank, 1975c, pp. 238, 239)

Recently Schank has added several new devices to his representational system to reflect the purposive aspects of actions as well as their physical descriptions. These include *goals*, which can be realized by appropriate sequences of acts; *scripts*, which provide such sequences in simple stereotyped situations; *plans*, which provide a more flexible way of specifying the appropriate action sequences, including the treatment of a whole set of alternative subsequences as a single subgoal; and *themes*, which include people's occupations (e.g., lawyer), their relationships with others (e.g., love), and their general aims (e.g., getting rich), and which are offered as the source of their goals. The representation of a piece of text is thus extended to try to take into account not only what caused what but also what was intended to cause what and why the actor might have had such an intention in the first place. In addition, Schank has recently supplemented the primitive ACTs with several social ACTs--AUTHORIZE, ORDER, DISPUTE, and PETITION--in order to represent yet another dimension of human actions more readily. None of these devices, however, is characterized as introducing a new set of primitives.

## C6. Analogical Representations

There is a class of representation schemes with properties different from the schemes previously discussed. These are the *analogical* or *direct* representations, which include maps, models, diagrams, sheet music, etc.

Analogical representations have been defined as systems in which "properties of and relations between parts of the representing configuration represent properties and relations of parts in a complex represented configuration, so that the structure of the representation gives information about the structure of what is represented." (Sloman, 1971). The significant point here is the requirement of correspondence between relations in the representation, and relations in the represented situation. Hayes, 1974 indicates that the connection between the representation and the situation is thus one of *homomorphism* (structural similarity) rather than *denotation*. For example, the proximity of two points on a map indicates the geographical proximity of the two points in the world. Analogical representations may be contrasted with the more prevalent *propositional* or *fregean* forms (so called after Frege, who invented the predicate calculus), which include most common types of representation. Fregean forms do not require this correspondence between relations in the representation and relations in the situation; for example, proximity of assertions in a predicate calculus data base does not indicate proximity between the subjects of these assertions in the real domain.

The degree to which a representation can be described as analog is affected by the routines which operate on that representation. That is, unless the interpretive processes make use of the full structure of the representation, power may be sacrificed. Continuing with the map example, if a routine for examining a map retrieved all distances from an internal table, rather than by looking at the map, it would be pointless to say that the map was analogical with respect to distance. (Pylyshyn, 1975, Pylyshyn, 1978) refers to the interpretive processes collectively as the *Semantic Interpretation Function* (S.I.F.), and suggests that the terms "analogical" and "Fregean" can only be applied to a *system* (representation paired with an S.I.F.), rather than to the representation alone.

When we use the term "analogical" we mean *analogical with respect to certain properties*. For example, a map (with a reasonable S.I.F.) is analogical with respect to location (and hence distance), but not (usually) with respect to elevation.

The analogical/propositional distinction has been the subject of discussion in psychology, where researchers have presented conflicting arguments about the nature of human memory. Pylyshyn, 1973 presents a thorough treatment of this issue.

Analogical representations are less general than propositional ones, and therefore suited for a smaller class of task domains. However, for those

specific domains, the analogical representation has some significant advantages. In particular, the problem of updating the representation becomes much simpler. For example, if we add a new city to a map, we need not explicitly state its distance from all the old cities, but can trust that the distance on the map accurately represents the distance in the world. Thus, for those properties which are analogically represented, the frame problem (McCarthy & Hayes, 1969), of updating the state of all objects in a scene after one has been altered, is minimized.

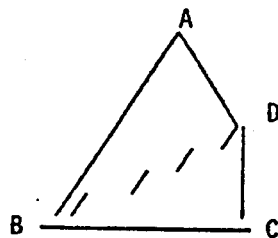
The next section of this article presents some systems which use analogical representations; the final section returns to a discussion of the advantages and disadvantages of these representations.

### Systems Using Analogical Representations

The Geometry-Theorem Prover (Gelernter, 1959 and article Search.D3) was one of the earliest theorem provers of any sort, and was distinguished by its reliance on a diagram to control search. The system proved theorems in Euclidean geometry, of the simple sort often done by high school students. The system operated via *problem-reduction* Search.B2, working backward from the goal, and trying to prove the subgoals generated.

Together with a statement of the problem, the system was given a simple diagram, which it used in two ways. The most important of these was the pruning heuristic, "Reject as false any statement (goal) which is not true in the diagram". Only those subgoals which were not contradicted in the diagram were pursued via the formal proof methods. Note that the system might miss a valid proof (if, e.g., some fact which was true did not happen to be true in the diagram), but would never accept an invalid one, since the final proofs were performed formally; any fact which was "accidentally" true in the diagram would be contradicted later in the development of the proof.

A sample problem is:

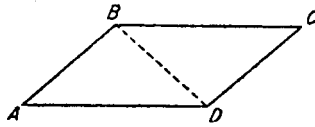


Given: Angle ABD equals angle DBC.  
 Segment AD perpendicular segment AB.  
 Segment DC perpendicular segment BC.

Prove: Segment AD equals segment CD.

The other use of the diagram was to establish "obvious" facts, concerning, for example, the ordering property of points on a line, and the intersection properties of lines in a plane. Many of these are self-evident from the diagram, but would be tiresome to prove from fundamental axioms. In certain such circumstances, the program would assume the fact true if it were true in the diagram, while explicitly noting that it had made such an assumption.

The program did not construct the diagram itself; rather, the diagram was input along with the problem statement. However, the program was able to add additional lines to the diagram, when necessary. For example, given the following problem:



Given: Quad-lateral ABCD.  
 Segment BC parallel to AD.  
 Segment BD equals AD.

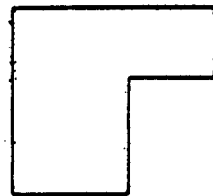
Prove: Segment AB equals segment CD.

Solution is difficult without hypothesizing the line BD. Eventually, the program will construct that line, then use congruence of triangles to establish the proof.

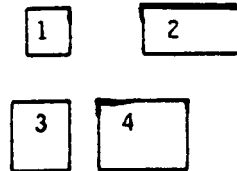
The use of the diagram resulted in the pruning of about 995 out of every 1000 subgoals, at each level of search. The final improvement, therefore, was several orders of magnitude.

Work on the *General Space Planner* (Eastman, 1970, Eastman, 1973) addressed the general class of *space planning* problems. The goal in these tasks is to arrange a set of *domain units* (e.g., objects) in a *space* (e.g., a room), subject to given *constraints* (e.g., adjacency, line-of-sight). A simple problem is:

Given the space



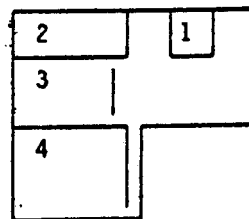
and the objects



and the constraints:

{3} must be adjacent to {4}  
 {2} must be adjacent to {3}  
 {1} must be visible from {3}  
 {1} must not be adjacent to any other objects,

one solution is:



A concrete version of this problem might be a machine room, with the objects being pieces of equipment to be placed.

The system used a representation, called a *variable domain array* Eastman, 1970 which

was a specialization of the sort of 2-d diagram used by Gelernter. It consisted of partitioning the diagram into a set of rectangles, to save space in memory. This did not affect the fundamental properties of the representation; in particular, the structure of the representation reflected the structure of the space, with respect to the properties of size, shape, and position (so the system could be described as "analogical", with respect to those properties).

Also, the system had two properties identified by Eastman as useful for the space planning task:

- (1) both filled and empty space are represented (useful for solution of problems like the one here)
- (2) overlaps between objects can be easily detected (necessary for more sophisticated problems)

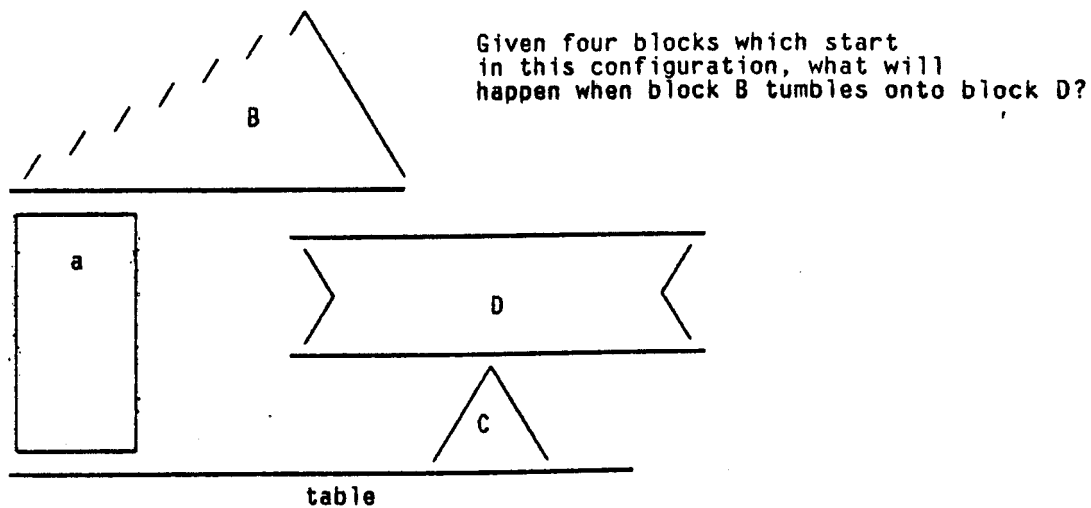
The system solved the problems via a depth-first search algorithm (*Search.Overview*), locating successive objects and backing up when necessary. The search was facilitated by a *constraint graph*, which represented, via restrictions on the amount of area left, the effects of constraints between pairs of objects. Thus, the search could attack the most restrictive constraint first, and produce an efficient search. This method has been called *constraint structured planning*.

Note that Eastman's work is in one sense the reverse of Gelernter's. Gelernter's system performed search in a propositional space (sets of formal statements) using an analogical representation (the diagram) for guidance. Eastman's performed search in an analogical space (the diagrammatic array) using a propositional form (the constraint graph) for heuristic guidance.

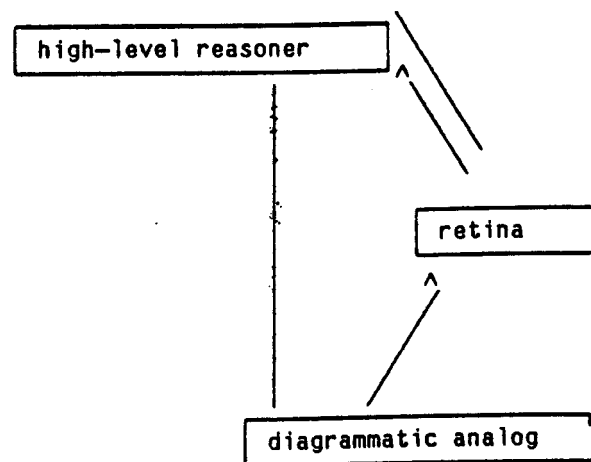
The WHISPER system Funt, 1976 was a system designed to function completely via the analogical representation, unlike Gelernter's theorem prover which used the diagram merely to augment its reasoning.

The system operated in a simplified blocks-world environment. An example problem is:





The system consisted of the following components:



The "diagram" was an array, which represented the 2-dimensional scene in the obvious way, as shown above. The retina, used to view the diagram, consisted of a set of parallel receptors, arranged in concentric circles; each receptor viewed a small (local) part of the diagram. The high-level reasoner, containing qualitative physical knowledge, was the domain-dependent part of the system; it employed information regarding the behavior of rigid bodies when acted upon by gravity.

The significance of the diagram to WHISPER lay in the fact that there were two types of analogs present:

- (a) between the static states of the diagram and the static states of the world
- (b) between dynamic behavior of objects in the diagram, and similar behavior of objects in the world

The correspondences between the diagram and the world were simple and well-defined; no complicated processes were required to map from one to the other. A number of properties, such as position, orientation, and size of blocks, were represented analogically. For these properties, it was not necessary to perform complicated deductions, since they "fell out" of the diagrams. For example, to test whether or not a particular area of the world was "empty" (i.e., not occupied by any block), the system had only to "look at" the corresponding area of the diagram (the Space Planner system also had this property). With most propositional representations (e.g., Fahlman, 1974, ), it would be necessary to enumerate each block individually, testing whether or not that block overlapped the space in question.

Note that certain properties (e.g., color, weight) were not represented in the analog. To reason about these, normal deduction would be necessary.

The retina, used for viewing the diagram, also provided a number of *perceptual primitives*, including center of area, contact finding, similarity testing, etc. The high-level reasoner never looked at the diagram directly, but only thru the retina.

WHISPER thus constituted an example of a system for solving problems using an exclusively analogical representation.

Baker, 1973 had earlier suggested a similar representational formalism. Like Funt, he envisioned a 2-dimensional array to represent the diagram; however, he also discussed the possibility of retaining spatial "smoothing" information *within* each cell of the array, to remove some of the error induced by the coarseness of the array.

Both Funt and Baker suggested that the *parallelism* of their systems coupled well with the analogical representations. The elements of their processors (in Funt's case, the retina) could operate autonomously, with connections only to their (spatially) neighboring cells. Via this sort of network, arbitrary transformations, via combinations of translation and rotation, could be represented.

#### Issues Concerning Analogical Representations

The work done to date on direct representations raises a number of issues. These will be discussed here, but not resolved.

First, we will justify our characterization of an analogical representation. Sloman, 1975 presents some popular misconceptions about such representations, and clarifies them. Analogical representations need not be continuous; a sorted list (of, e.g., numbers) is

analogical with respect to size. Analogical representations need not be 2-dimensional; again, the sorted list is an example of a 1-dimensional representation. Analogical representations, like Fregean ones, may have a *grammar* which defines them. Thus, the difference reduces to the statement given in the overview; there must be a correspondence between aspects of the structure of the representation, and the structure of the represented situation.

One of the advantages of analogical representations over their propositional counterparts relates to the difference between *observation* and *deduction*: the former is relatively cheap (in some situations) while the latter can require a great deal of computation. To take an example from Funt, if we wish to discover the behavior of an experiment with a pile of blocks on the moon, we can perform the same experiment on earth; if the analogy between our representation (the experiment on earth) and the situation (the experiment on the moon) is strong enough, the results will be valid. Thus, we have avoided the necessity for deduction, by ensuring that the relevant properties were "carried over" into our (analogical) representation. In general, "observation" occurs in the form of computation; we transform the representation in ways which correspond to events in the real world.

Funt relates a more abstract justification for the use of analogical representations. A Fregean representation (consisting of, e.g., a set of predicate calculus axioms) will often admit to several models -- i.e., situations of the world which satisfy the axioms. Thus, any proofs from the axioms will be valid in all such models. Analogical representation, on the other hand, are usually more exhaustive and specific, admitting fewer models. Thus, results proved about these representations are less general. Since generality is 'expensive' (i.e., general results are usually harder to prove than specific ones), we often prefer to work with the most specific form -- in this case, the analogical.

Additionally, as illustrated by Geiernter's work, the use of analogical representations can facilitate search. Constraints in the problem situation are represented by constraints on the types of transformations applied to the representation, so that impossible strategies are rejected immediately.

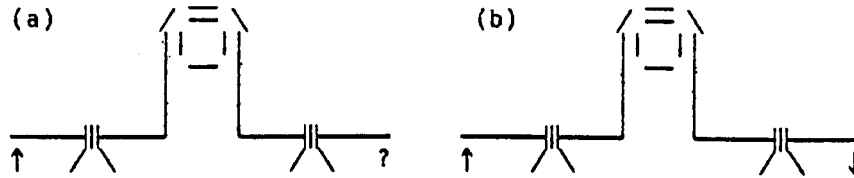
There are, however, some disadvantages to the use of these direct representations. First is the general/specific question mentioned earlier; analogical representations tend to restrict us to more specific problems. But, as Sloman, 1975 points out, there are times when generality is needed. For example, consider the problem "If I start in room A and then move back and forth between room A and room B, which room will I be in after exactly 377 moves?". For this case, the best approach is not to simulate the action, but to generalize the effects of odd and even numbers of moves.

Second, as Funt points out, some features of the analog may not hold in the actual situation, and we might not know which ones these are. This is related to the general question of knowing the limits of the representation.

Third, analogical representations become unwieldy for certain types of incomplete information. That is, if a new city is added to a map, its distance from other cities is obtained easily. But, suppose that its location is known only indirectly -- e.g., that it is equidistant from cities Y and Z. then, the distance to other cities must be represented as equations, and the power of the analog has been lost.

Sloman (1971) suggested generalizing the concept of *valid inference*, to include work

with direct representations <extends to the notion of proof in the analogical medium>. For example, given the situation (a) below, we want to infer (b).



We have two levers, connected by a rope passing over a pulley (as shown). If one lever is pushed up (as indicated by the arrow), what will be the effect upon the opposite end of the other lever?

The answer, as shown in (b), is that it will move down.

Unfortunately, these rules of inference will be hard to specify. As yet, the problem is ill-understood.

To conclude, analog representations are analog *with respect to some properties* of the situation being represented. Some properties (especially physical ones) may be relatively easily represented analogically, and the savings in computation may be very worthwhile.

### C7. Frames and Scripts

One major observation by psychologists (Bartlett, 1932) about our daily cognitive activity is that we use a large, well-coordinated body of knowledge based on previous experiences to actively interpret new situations. For example, when we visit a restaurant that we have never been to before, we have an extensive array of expectations based on experience in other restaurants about what we will find: Maitre d's, menus, tables, etc. In addition to these expectations about the *objects* in a typical restaurant, we have strong expectations about the *sequence of events* that are likely to take place. In AI this knowledge about typical objects and events for a given context have been called *frames* and *scripts*, respectively. Frames were originally proposed by Minsky (1975) as a basis for understanding visual perception, natural language dialogues, and other complex behaviors. Scripts have been developed by Schank & Abelson (1977). They are discussed in detail in the articles Natural Language.F6 and Natural Language.F3, but they are mentioned here because they serve the same purpose for Schank and Abelson as frames, namely, they organize knowledge in a way that directs attention and facilitates inference and other cognitive processes.

The structure of a frame is based on the observation that many of our normal cognitive activities are stereotyped and general, but are "fine-tuned" to whatever a specific task involves. These tasks have a strongly interpretive flavor to them. Frames provide a way of interpreting a collection of data as an instance of some concept. As a simple example, consider the concept of "chair". Deciding whether an object is a chair is an interpretive task in which general and stereotyped characteristics are "filled in" (or instantiated) by a particular instance of a chair:

Frame: Chair	Instance of: Chair
Number of legs:	4 legs
Style of back:	Cushioned back
Style of seat:	Embroidered seat cover
Number of arms:	Armless

The characteristics which are filled in whenever an object is recognized as a chair are called *slots*. A frame is a collection of slots, although a slot may be associated with more than one frame. In particular, two or more frames may share a slot, for example, the concept of "seat" is common to chairs, cars, and bicycles. Also, the slot may itself be a frame, for example, the concept of "seat" has many characteristics (slots): size, firmness, durability, etc. Thus, frames can be "nested" within each other, which is consistent with the psychological idea of defining concepts in terms of other concepts.

One can imagine slots as "gaps" in the frame that must be filled as the situation is interpreted. There are two types of concepts, (1) object descriptors such as color, location, size, etc. and (2) action or event sequences such as going to a restaurant or riding a bicycle. Groupings of object descriptors have been referred to as frames (Minsky, 1975), prototypes (Aikins, 1979), hypotheses (Szolovitz, Hawkinson, & Martin, 1977), schemas (Davis, 1976), and units (Winograd, 1975); groupings of action or causal sequences, referred to as scripts (Schank & Abelson, 1977).

The slots are initially empty when a frame is chosen but are rapidly filled with specific

values as the frame is matched to the situation. Since the collection of slots composing the frame are assumed to be the most relevant concepts to use for the particular situation, they serve as a focus-of-attention device that guides the direction of cognitive processing. The slots resemble questions that the frame is asking of the situation.

Frames are organized into larger *frame systems*, collections of specific task-related frames. Any changes of emphasis or attention in performing the task are reflected in the transformations occurring between frames within a frame system. When frames within a system share the same slots, it is possible to share slot-values between the frames, instead of recomputing the common slot-values for each. Thus, frame systems provide a means of value *inheritance* that reduces the amount of cognitive labor a system has to perform.

In the following sections we will describe some of the major ideas from the frames literature and briefly describe some of the systems that have explored them.

#### Organizing Knowledge in Frames and Scripts

To illustrate some of the current ideas about slots and frames and how they might be used by a frame-driven system, consider the following example: A frame that specifies the characteristics (slots) of a restaurant. An incomplete example of such a frame is given below. The terminology used for this example is not in use by any one system; it is intended only to give the flavor of frame systems. (See Winograd, 1975, Schank & Abelson, 1977, and Szolovitz, Hawkinson, & Martin, 1977 for more system-specific details). Annotations are in *italics*.

```

Restaurant
AKindOf: BusinessEstablishment          this is a Restaurant frame
EstablishmentName:                       restaurants are places of business
    default: Joe's Grill                 a standard name
    if-needed:                          and how to find the real name
                                          Sign is another frame
                                          (Find a Sign with Words (Restaurant or Grill or Place))
Location:
    default: (an Address in SanFrancisco) ... as are Address and
                                                SanFrancisco, etc.
    range: (an Address in World)
FoodStyle:
    default: Burgers
    range:                                delimits possible values
                                          for this slot
    if-added:                            (Burgers, Szechewan, Mandarin, FreshSeafood, HauteCuisine)
                                          if we find this out,
                                          update the Alternatives
                                          (Update Alternatives of Restaurant)
TimesofOperation: (a Timeofday that is IntheEvening)
PaymentForm:
    default: CreditCard
    range: (Cash, CreditCard, Check, WashingDishesScript)
PriceRange:
    default: Moderate
    range: (FreeLunch, Average, Moderate, Expensive)
    if-needed: (AveragePrices of RestaurantMenu)
EventSequence:
    default: EatatRestaurant             a script, see below
    if-needed:                          (if ChefPerson then (Try CookatRestaurant)
                                          if WaitPerson then (Try ServeatRestaurant))
FoodQuality:
    default: Mediocre
    range: (Ugh, Mediocre, Palatable, Ok, Excellent, Magnifique)
    if-needed: ((Ask a Connoisseur) or (TasteTest))
Alternatives:
    if-needed: (Find all Restaurants with the same FoodStyle)
Examples: (Sam's Grill, Tadich, Scott's, Pacific Cafe)
:
:
:

```

A visitor to a specific restaurant can be guided in his expectations or actions, using this generic or *prototypical* Restaurant frame. It provides a way of organizing and using the information about restaurants and thus establishes the concept or class called Restaurant. A specific example, or instance of a restaurant say Sam's Grill, might be represented by the following assignment of values to the slots specified by the prototypic Restaurant frame:

**Sam's Grill frame**

ASpecializationOf: Restaurant  
 NameOfEstablishment: Sam's Grill  
 Location: (300 Bush, SanFrancisco)  
 TimesofOperation: (M-F, 10am-8:30pm)  
 FoodStyle: FreshSeafood  
 PriceRange: Expensive  
 ServiceQuality: Excellent  
 FoodQuality: Excellent  
 PaymentForm: (CreditCards, Cash, Check)  
 Alternatives: (Tadich, Pacific Cafe)

Note that not all of the slots specified in the general Restaurant frame need to be filled in for the instance of the restaurant. Default values are supplied for those unspecified slots in the instance frame. These will be used unless new information is found to update the value for that slot. For example, the default EventSequence that was probably followed at Sam's is described by the EatatRestaurant *script* described below:

**EatatRestaurant Script**

Props: (Restaurant, Money, Food, Menu, Tables, Chairs)  
 Roles: (HungryPersons, WaitPersons, ChefPersons)  
 PointofView: HungryPersons  
 TimeofOccurence: (TimesofOperation of Restaurant)  
 PlaceofOccurence: (Location of Restaurant)  
 EventSequence: *a default sequence*  
   first: EnterRestaurantScript  
   then: if (WaitToBeSeatedSign or Reservations)  
         then GetMaitred'sAttentionScript  
   then: PleaseBeSeatedScript  
   then: OrderFoodScript  
   then: EatFoodScript unless (LongWait) when  
         ExitRestaurantAngryScript  
   then: if (FoodQuality was better than Palatable)  
         then ComplementsToTheChefScript  
   then: PayForItScript  
   finally: LeaveRestaurantScript

This is a very crude rendition of the Restaurant script given by Schank & Abelson, 1977Natural Language.FB. In particular, this script is presented in English, while Schank's scripts are represented in conceptual dependency format (B5).

We see that this script specifies a normal or default sequence of events as well as exceptions and possible error situations. The script also requires the use of a few static descriptions such as Props and Roles that are filled by other frames. Scripts are being researched extensively by Schank and his colleagues (Schank & Abelson, 1977).

These frames and scripts form a frame system for dealing with most of the scenarios involving restaurants. They might be part of other frame systems such as a system to recognize BusinessEstablishments like Banks and ServiceStations.



We will now consider how frames can be used to direct reasoning and provide a focus-of-attention for a system that is frame-driven.

### Procedural Knowledge in Frames and Scripts

The preceding discussion emphasized the *declarative* aspects of frames and scripts, that is, how facts about objects and situations might be represented and stored. However, frames not only accommodate facts and pieces of information as they are gathered, but they also determine the direction in which they are sought. In particular, we will see how procedures attached to slots can guide the reasoning of the system as it attempts to apply a frame to a situation.

**FILLING OUT SLOTS.** Filling out the slots of a frame or script is the primary process that occurs after the frame or script is selected. Script slots provide *expectations* of events to occur, just as frame slots provide expectations of objects and properties to be found in a scene. Filling the slots provides confirmation that the frame or script is appropriate for the scene or event, and amounts to *understanding* it. For example, Schank's script-based story understander (Natural Language.F6) can be said to have understood a story when each slot in the appropriate script has been filled by an explicit event, or by an implied one. As discussed above, one way of filling a slot is to use default values specified in a generic frame until other information becomes available. These values are typical assumptions that can be made without great repercussions and are used in lieu of other knowledge. For example, the default value for the TimesOperation slot of the generic Restaurant frame might be "Open Evenings Except Mondays."

Information to update a slot value can also be found in a number of other ways. The slot could inherit a value from another frame in the same frame system that has generated it. The Restaurant script, for example, inherits the times of operation and the location from the Restaurant frame. Default and inherited values are relatively inexpensive methods of finding new information for slots. These methods account for a large part of the power of frames; any new frames interpreting the situation can make use of values determined by prior experience, without having to recompute them.

However, for some slots a certain amount of reasoning or a specific procedure might be required to obtain these values. In these cases, the method of *procedural attachment* provides a way of specifying slot-specific processes. These procedures attempt to determine the value of a slot on demand and are sometimes referred to as *servants*. The use of attached procedures allows us to specify methods that can take advantage of the current context by supplying slot-specific heuristics for the determination of the slot values. These heuristics might propose some simple inquiries to the database, some theorems to be proved, a set of production rules to be fired, or an arbitrary procedure. For example, when we want to know what the food tastes like (to fill in the FoodQuality slot), we are advised to try some (TasteTest) or ask someone we trust who might have tried it before (Ask a Connoisseur).

Another frequently used form of procedural attachments are those that are activated by changing the value of a slot. These procedures are invoked when a value is found for a slot. Since these procedures are event-driven, that is, triggered by the addition of a value, they resemble demons. For example, a procedure might update the Alternatives slot for a new restaurant when values for FoodStyle had been determined.

**TRIGGERING FRAMES.** Attached procedures have another important ability: They can be used to suggest or *trigger* other frames. Thus, these procedures could recommend other, perhaps more applicable frames within the same frame system, or they could recommend an entirely new frame system to deal with the situation, suggesting that a new point of view was more applicable. For example, if we walked into a restaurant and there was no maitre d', but there was a food line, we might want to switch to the Cafeteria frame.

This triggering procedure has been used in various frame-driven systems that attempt to diagnose a patient, (Szolovitz, Hawkinson, & Martin, 1977) . Since a number of related diseases might share a core set of signs and symptoms, the ability to make a differential diagnosis depends heavily on the ability to detect those factors that rule out or confirm a particular diagnosis. Typically, in medicine, when a disease is ruled out, another more applicable disease is indicated, which is probably very similar.

The ability to trigger other frames through procedural attachment provides a method to select frames by which to pursue the slot-filling task. It also provides a method of *confirmation* that the current frame and frame system is the most appropriate for the situation. The frame can determine whether the values confirm or disconfirm its applicability to the situation. It could then suggest that other frames be used to perform the interpretation. The triggering ability and the ability to use default or inherited values allow a frame-driven system to be reasonably efficient at determining the interpretation or point of view most applicable to the current situation.

#### CURRENT SYSTEMS USING FRAMES AND SCRIPTS

A number of experimental prototype systems have been implemented to explore the idea of frame-based processing. The following descriptions are intended to give an indication of the domains and problems the researchers in this area addressed; the references should be consulted for more details.

Bobrow, Kaplan, Norman, Thompson, and Winograd Bobrow et al. (1977a) have experimented with frame-based natural language understanding in their GUS system. Designed to help a person make airline reservations, the system attempted to demonstrate how various aspects of dialogue understanding--such as handling mixed-initiative dialogues, indirect answers and anaphoric references--could be performed using a frame's ability to provide expectations and defaults, and therefore structure the conversation. This system was also used to explore procedural attachment issues. A sample interaction with the GUS system is given below:

Hello. My name is GUS. I can help you plan a simple trip by air.  
Where do you want to go?

*The reservations frame indicates that all reservations for traveling by air need a destination.*

\*\* I WANT TO GO TO SAN DIEGO ON MAY 28

*The client specifies more than is requested, the date of departure.*

What do you want to leave?

**\*\* I MUST BE IN SAN DIEGO BEFORE 10 AM**

*GUS must infer from this indirect answer that the flights can leave anytime as long as they arrive before 10 am.*

Would you like PSA flight 102 that arrives at 9:15 am?

**\*\* WHAT IS THE NEXT FLIGHT?**

*...which GUS interprets to mean the next flight in order of departure.*

Air California flight 310 that leaves at 8:30 am and arrives at 10 am

**\*\* I'LL TAKE THE FIRST ONE**

*Anaphoric reference resolved to mean the PSA flight.*

In what name should I make the reservations, please?

**\*\* DAN**

I have confirmed the following flight..

To guide this dialogue, the following frames dealing with aspects of air travel were used to understand requests made by the passenger as well as to drive GUS to ask for needed information.

<b>TripSpecification</b>	<i>the major topic of discussion</i>
<b>HomePort:</b>	
range: (a City)	
default: PaloAlto	
<b>ForeignPort:</b>	<i>where passenger is traveling on</i>
range: (a City)	<i>the OutwardLeg</i>
<b>OutwardLeg:</b>	
range: (a TripLeg)	
<b>InwardLeg:</b>	<i>for return flights, not detailed</i>
range: (a TripLeg)	<i>in this example dialogue</i>
<b>TripLeg</b>	
<b>FromPlace:</b> (HomePort of TripSpecification)	

ToPlace:  
  range: (a City)  
  if-needed: (AskClient)

TravelDate:  
  range: (a Date)  
  if-needed: (AskClient)

DepartureSpec:  
  range: (a TimeRange)  
  if-needed: (AskClient)

ArrivalSpec:  
  range: (a TimeRange)  
  if-needed: (AskClient)

ProposedFlights:  
  range: (a SetofFlights)

FlightChosen:  
  range: (a Flight)  
  if-needed: (AskClient)

Traveller:  
  range: (a Person)  
  if-needed: (AskClient)

Notice that the dialogue follows directly the order of the slots in both the TripSpecification frame and the TripLeg frame.

Concurrently with the development of GUS, KRL (Knowledge Representation Language, Bobrow & Winograd, 1977) was developed to extend the ideas and intuitions found during the implementation of GUS and to provide a new basis for representation. A report of a number of project sketches implemented in KRL-0 can be found in Bobrow et al., 1977a. The report details a number of difficulties and shortcomings experienced in their attempt to use the abilities outlined above.

Other work with frame-based systems includes the NUDGE system, developed by Goldstein and Roberts (Goldstein & Roberts, 1977), which was used to understand incomplete and possibly inconsistent management-scheduling requests and to provide a complete specification for a conventional scheduling algorithm. Implemented in FRL-0, the system also used a frame-based semantics to resolve anaphoric requests Bullwinkle, 1977.

A program that solves physics problems stated in English was developed by Novak (1977). It used a set of canonical object frames--such as Point, Mass, or Pivot--to interpret the actual objects and their relations in a number of static problems. These canonical object frames were used to construct a "view" of an actual object as an abstract object, thereby simplifying the problem representation.

A final example of a frame-based system is that of Schank et al. (Schank & Abelson, 1977) He has been using scripts to investigate the notions of causality and understand sequences of events. In particular, the SAM program Natural Language.F6 attempts to understand short stories using a script to guide the interpretation or inference of occurrences in the story. Consider the following sample story that makes use of our RestaurantScript:

John went to a restaurant. He asked the waitress for coq au vin.  
He paid the check and left.

This simple story corresponds very well to our default EventSequence with the assignment of "the waitress" to WaitPerson, "coq au vin" to Food, and "check" to PaymentForm. After establishing the appropriate frame and filling some of its slots with known information, SAM can infer (from default information) answers to questions like *Did John read the menu?* The mechanisms from performing these inferences are the subject of much research. (See Aihandbook article on SAM for a detailed description.)

#### SUMMARY

Frames and scripts are recent attempts by AI researchers to provide a method for organizing the large amounts of knowledge needed to perform cognitive tasks. The ideas behind the development of frames are based on a number of psychological observations that have been made about how humans perform in different cognitive situations. The development of large-scale organizations of knowledge and the concomitant ability of these organizations to provide a basis for active cognitive processing is very recent, and a number of major questions must be answered before the conjectured benefits of frames can be fully realized.

#### REFERENCES

See Bartlett, 1932; Bobrow et al., 1977a; Bobrow & Winograd, 1977; Bullwinkle, 1977; Charniak, 1978; Goldstein & Roberts, 1977; Kuipers, 1975; Minsky, 1975; Novak, 1977; Rosenberg, 1977; Schank & Abelson, 1977; Szolovitz, Hawkinson, & Martin, 1977; Winograd, 1975; Winston, 1977a; and Winston, 1977b.

## References

- Aikins, J. S. Prototypes and production rules: An approach to knowledge representation for hypothesis formation. *IJCAI* 6, 1979.
- Amarel, S. On Representations of Problems of Reasoning About Actions. In D. Michie & B. Meltzer (Eds.), *Machine Intelligence 3*. Edinburgh: Edinburgh University Press, 1969. Pp. 131-171.
- Anderson, J. *Language, Memory, and Thought*. Hillsdale, NJ: Lawrence Erlbaum, 1976.
- Anderson, J., & Bower, G. *Human Associative Memory*. Washington, D.C.: Winston & Co., 1973.
- Anderson, R. H. The Use of Production Systems in RITA to Construct Personal Computer Agents. *SIGART News*, No. 63, June 1977.
- Baker, R. A Spatially-Oriented Information Processor which Simulates the Motions of Rigid Objects. *Artificial Intelligence*, 1973, 4, 29-40.
- Barstow, D. R. Automatic Construction of Algorithms & Data Structures Using a Knowledge Base of Programming Rules, TR 641, CSD, Stanford, 1977
- Bartlett, F. C. *Remembering*. Cambridge: Cambridge University Press, 1977.
- Berliner, H. J. Some necessary conditions for a master chess program, *Proc IJCAI3*, 1973
- Bernstein, M. I. Knowledge-Based Systems: A Tutorial, TM-(L)-5903/000/00A, SDC, June 1977
- Bobrow D. G., Kaplan R. M., Kay M., Norman D. A., Thompson H., Winograd T. GUS, A Frame-Driven Dialog System. *Artificial Intelligence* 8, 1977, 155-173. (a)
- Bobrow, D. Dimensions of Representation. In D. Bobrow & A. Collins (Eds.), *Representation and Understanding*. New York: Academic Press, 1975.
- Bobrow, D. G. A question-answering system for high-school algebra word problems, *Proc. AFIPS Fall Joint Computer Conference*, 1964, pp 591-614
- Bobrow, D. G., & Winograd, T. An overview of KRL, a knowledge representation language. *Cognitive Science*, 1977, 1, 3-46.
- Brachman, R. J. What's in a concept: Structural foundations for semantic networks (BBN Rep. No. 34333). Cambridge: Bolt, Beranek, & Newman, 1976.
- Brachman, R. J. A structural paradigm for representing knowledge. BBN Report No. 3605, May, 1978.

- Brooks, R. Production Systems as control Structures for Programming Languages, Proc Workshop
- Buchanan, B. G., G. L. Sutherland, and E.A. Feigenbaum Heuristic DENDRAL: A Program for Generating Explanatory Hypotheses in Organic Chemistry, in Machine Intelligence, 4, B. Meltzer and D. Michie (eds), 1969
- Buchanan, B.G., and E.A. Feigenbaum DENDRAL and Meta-DENDRAL: Their Applications Dimension, Memo HPP-78-1, Stanford Heuristic Programming Project, 1978
- Bullwinkle, C. Levels of complexity in discourse for anaphoric disambiguation and speech act interpretation. IJCAI 5, 1977, 43-49.
- Carbonell, J. R., & Collins, A. M. Natural semantics in AI. Amer. Journal of Comput. Ling., 1974, Microfiche 3.
- Charniak, E. With spoon in hand this must be the Eating frame. TINLAP-2, 1978, 187-193.
- Collins, A. Human Plausible Reasoning. BBN Report No. 3810, 1979.
- Davies, D. J. M. Popler: A Pop-2 Planner, MIP-89, School of AI, University of Edinburgh, Edinburgh, Scotland, 1972.
- Davis, R. The Application of Meta-Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases, AIM-286, Stanford, 1976
- Davis, R., Buchanan, B. G., and Shortliffe, E. H. Production Rules as a Representation for a Knowledge-Based consultation Program, AIM 266, Stanford, 1975 (need a ref to the AIJ version)
- Davis, R., and Buchanan, B. G. Meta-level knowledge: Overview and applications, IJCAI 5, 1977.
- Davis, R., and King, J. J. An Overview of Production Systems, in Machine Intelligence 8, E. Elcock and D. Michie (eds), Chichester, Ellis Horwood, 1977
- Dijkstra, E. W. Guarded Commands, Nondeterminacy & Formal Derivation of Programs, CACM, 18, 1975, 453-457
- Duda, R. O., Hart, P. E., Nilsson, N. J., and Sutherland, G. L. Semantic Network Representations in Rule-Based Inference Systems, in Pattern-Directed Inference Systems, D.A. Waterman and F. Hayes-Roth (eds), New York, Academic, 1978
- Eastman, C. M. Automated Space Planning. Artificial Intelligence, 1973, 4, 41-64.
- Eastman, C. M. Representations for Space Planning. CACM, 1970, 4, 242-250.
- Erman. ???
- Fahlman, S. E. Symbol-mapping and frames. SIGART Newsletter, No. 53, August 1975, pp. 7-8.

- Fahlman, S. E. A Planning System for Robot Construction Tasks. *Artificial Intelligence*, 1974, 5, 1-49.
- Feigenbaum, E. A. The simulation of verbal learning behavior, in *Computers and Thought*, E.A. Feigenbaum and J. Feldman (eds), New York, McGraw-Hill, 1963
- Feigenbaum, E. A. The Art of Artificial Intelligence: Themes and Case Studies of Knowledge Engineering, Proc. 5th Int'l Joint Conf A.I., 1977
- Fikes, R., & Nilsson, N. J. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 1971, <2>, 189-208.
- Fikes, R., & Hendrix, G. A Network-based knowledge representation and its natural deduction system. *IJCAI 5*, 1977, 235-246.
- Fikes, R., Hart, P. E., & Nilsson, N. J. Learning and Executing Generalized Robot Plans. *Artificial Intelligence*, 1972, 3, 223-250.
- Fillmore, C. J. The Case for case. In E. Bach & R. Harms (Eds.), *Universals in Linguistic Theory*. Chicago: Holt, 1968.
- Filman, R. E. The Interaction of Observation and Inference, Stanford AI Memo 327, Computer Science Dept., Stanford University, 1979.
- Filman, R. E., & Weyhrauch, R. W. An FOL Primer. Stanford AI Memo 288, Stanford University, Dept. of Computer Science, 1976.
- Flavell, J. Metacognition and cognitive monitoring: A new area for cognitive-developmental inquiry. Dept. of Psychology, Stanford Univ., 1979.
- Forgy, C. and J. McDermott OPS, A Domain-Independent Production System Language, *IJCAI5*, 1977.
- Forgy, C. A Production System Monitor for Parallel Computers, tech rpt, CMU CSD, 1977.
- Funt, B. V. WHISPER: A Computer Implementation Using analogs in Reasoning. Technical Report 76-09, Computer Science Department, University of British Columbia, 1976.
- Funt, B. V. WHISPER: A Problem-Solving System Utilizing Diagrams and a Parallel Processing Retina, *IJCAI 5*, 1977, 459-464.
- Gelernter, H. Realization of a Geometry Theorem-Proving Machine. Proc. of Int. Conf. on Information Processing. Paris: UNESCO House, 1959. Pp. 273-282.
- Gentner, Dedre, and Collins, A. M. Knowing about knowing: Effects of meta-knowledge on inference. Submitted to *Cognitive Psychology*, 1979.
- Goldstein, I. P., & Roberts, R. B. NUDGE, a knowledge-based scheduling program. *IJCAI 5*, 1977, 257-263.



- Green, C. The Application of Theorem-Proving to Question-Answering Systems. *IJCAI* 1, 1969, pp. 219-237.
- Hayes, P. On Semantic nets, frames, and associations. *IJCAI* 5, 1977, 99-107. (b)
- Hayes, P. J. Computation and Deduction. *Symposium on Mathematical Foundations of Computer Science*, Czech. Academy of Science, 1973.
- Hayes, P. J. In Defence of Logic. *IJCAI* 5, 1977, pp. 559-565.
- Hayes, P. J. Some Problems and Non-Problems in Representation Theory. *British Computer Society, AI & Simulation of Behavior Group Summer Conference*, Univ. of Sussex, 1974. Pp. 63-79.
- Hedrick, C. Learning Production Systems from Examples, *AI Journal*, 1976.
- Hendrix, G. Expanding the utility of semantic networks through partitioning. *IJCAI* 4, 1975, 115-121.
- Hendrix, G. G., Thompson, C. W., & Slocum, J. Language Processing via Canonical Verbs and Semantic Models. *IJCAI* 3, 1973, 262-269.
- Hewitt, C. Description and Theoretical Analysis (Using Schemata) of Planner: A Language for Proving Theorems and Manipulating Models in a Robot. *MIT AI-TR 258*, 1972.
- Hewitt, C. How to Use What You Know. *IJCAI* 4, 1975, pp. 189-198.
- Hopcroft, J. E., & Ullman, J. D. *Formal Languages and Their Relation to Automata*. New York: Addison-Wesley, 1969.
- Jackendoff, R. A System of Semantic Primitives. *TINLAP*, 1975, 28-33.
- Jackendoff, R. Toward an Explanatory Semantic Representation. *Linguistic Inquiry*, 1976, 7, 89-150.
- Kay, M. The MIND system. In R. Rustin (Ed.), *Natural Language Processing*. New York: Algorithmics Press, 1973.
- Kowalski, R. Predicate Logic as a Programming Language. *Information Processing 74*. Amsterdam: North-Holland, 1974. Pp. 569-574.
- Kuipers, B. A frame for frames. In D. Bobrow & A. Collins (Eds.), *Representation and Understanding*. New York: Academic Press, 1975.
- Lehnert, W. C. *The Process of Question Answering*. Hillsdale, N.J.: Erlbaum, 1978.
- Lenat, D. B. AM: An artificial intelligence approach to discovery in mathematics as heuristic search. *AIM 286*, Stanford University, 1976.
- Lenat, D. B., and McDermott, J. Less than General Production System Architectures, *IJCAI* 5, 1977.

- Manna, Z. *Introduction to the Mathematical Theory of Computation*. New York: McGraw-Hill, 1973.
- Marcus, M. A Design for a Parser for English, Proc. A.C.M. Nat'l conference, Houston, 1976, 62-67.
- McCarthy, J. Epistemological Problems of Artificial Intelligence. *IJCAI 5*, 1977, pp. 1038-1044.
- McCarthy, J. Towards a mathematical science of computation, in *Information Processing 1962*, C.M. Popplewell (ed), Amsterdam, North-Holland, 1963.
- Mccartyh, J., and Hayes P. J. Some Philosophical Problems from the Standpoint of Artificial Intelligence, in D. Michie and B. Meltzer (Eds.), *Machine Intelligence 4*. Edinburgh University Press, 1969.
- McDermott, D. Artificial intelligence meets natural stupidity. *SIGART Newsletter*, 1976, 57, 4-9.
- McDermott, D. V., & Doyle, J. *Non-Monotonic Logic I*, MIT AI Memo 486, MIT, 1978.
- McDermott, J., and Forgy, C. *Production System Conflict Resolution Strategies*, CSD, CMU, 1976b.
- McDermott, J., Newell, A., & Moore ;, J. *The Efficiency of Certain Production system Implementations*, CMU, 1976a.
- Miller, G. A. *Comments on Lexical Analysis*. *TINLAP*, 1975, 34-37.
- Miller, G. A., & Johnson-Laird, P. N. *Language and Perception*. Cambridge: Harvard Univ. Press, 1976).
- Minsky M. *Computation Finite and Infinite Machines*. New Jersey: Prentice-Hall, 1972.
- Minsky, M. A. A framework for representing knowledge. In P. Winston (Ed.), *The Psychology of Computer Vision*. New York: McGraw-Hill, 1975.
- Minsky, M., & Papert, S. *Progress Report*, AI Memo 252, Artificial Intelligence Laboratory, MIT, 1972.
- Moore, R. C. *Reasoning from Incomplete Knowledge in a Procedural Deductive System*, TR 347, MIT AI Lab, MIT, 1975.
- Myopolous, J., Cohen, P., Borgida, A., & Sugar, L. Semantic networks and the generation of context. *IJCAI 4*, 1975, 134-142.
- Nash-Webber, B. L. *Semantics and Speech Understanding* (BBN No. 2896). Cambridge: Bolt, Beranek, & Newman, 1974.
- Newell, A. *Production systems: Models of Control Structure*, in *Visual Information Processing*, W. Chase (ed), New York, Academic Press, 1973.

- Newell, A., and Simon, H. A. GPS, a program that simulates human thought, in *Computers and Thought*, E.A. Feigenbaum and J. Feldman (eds), NY, McGraw-Hill, 1963.
- Newell, A., & Simon, H. *Human Problem Solving*. New Jersey: Prentice-Hall, 1972.
- Nilsson, N. J. *Problem-Solving in Artificial Intelligence*. New York: McGraw-Hill, 1971.
- Norman, D. A., Rumelhart, D. E., & the LNR Research Group *Explorations in Cognition*. San Francisco: Freeman, 1975.
- Norman, D. A. *Memory and Attention*, 2nd edition, New York: Wiley, 1976.
- Novak, G. S. Representation of knowledge in a program for solving physics problems. *IJCAI* 5, 1977, 286-291.
- Post, E. Formal reductions of the general combinatorial problem, *American Jour. Math*, 65, 1943, 197-268.
- Prawitz, D. *Natural Deduction--a proof-theoretical study*. Stockholm: Almqvist & Wiksell, 1965.
- Pylyshyn, Z. Do we need images and analogs? *TINLAP-1*, 1975.
- Pylyshyn, Z. Imagery and artificial intelligence. In C. W. Savage (Ed.), *Perception and Cognition: Issues in the Foundation of Psychology* (Vol IX of Minnesota Studies in the Philosophy of Science), U. of Minn. Press, 1978.
- Pylyshyn, Z. What the mind's eye tells the mind's brain: a critique of mental imagery. *Psychological Bulletin*, 1973, 13, 1-24.
- Quillian, M. R. Semantic memory. In M. Minsky (Ed.), *Semantic Information Processing*. Cambridge, Mass.: MIT Press, 1968.
- Quillian, M. R. The teachable language comprehender: A simulation program and the theory of language. *CACM*, 1969, 12(8), 459-476.
- Raphael, B. SIR: Semantic Information Retrieval. In M. Minsky (Ed.), *Semantic Information Processing*. Cambridge: MIT Press, 1968. Pp. 33-145.
- Reboh, R., Sacerdoti, E., Fikes, E. R., Sagalowicz, D., Waldinger, R. J., & Wilber, M. *QLISP: A Language for the Interactive Development of Complex Systems* TN 120, AAI Center, SRI International, Inc., 1976.
- Reiter, R. On Reasoning by Default. *TINLAP-2*, 1978, 210-218.
- Riesbeck, C. K. Conceptual Analysis, in *Conceptual Information Processing*, R.C. Schank (ed), Amsterdam, North-Holland, 1975.
- Rosenberg, S. Frames-based text processing, MIT AI Memo 431, 1977.

- Rulisfon, J., Derkson, J. A., & Waldinger, R. J. **QA4: A Procedural Calculus for Intuitive Reasoning**, TN 83, AI Center, SRI International, Inc., 1972.
- Russell, S. W. **Semantic Categories of Nominals for Conceptual Dependency Analysis of Natural Language**. Stanford AI Project, Memo AIM-172, July 1972.
- Rychener, M. D. **Production Systems as a Programming Language for Artificial Intelligence Applications**, CSD, CMU, December, 1976.
- Rychener, M. D. **The STUDNT Production System: A Study of Encoding Knowledge in Production Systems**, 1977 .
- Rychener, M. D. **Control Requirements for the Design of Production System Architectures**, Proc. ACM Symposium on Artificial Intelligence and Programming Languages, 1977, 37-44.
- Sandewall, E. **Conversion of Predicate-Calculus Axioms, Viewed as Non-Deterministic Programs, to Corresponding Deterministic Programs**. IJCAI 3, 1973, pp. 230-234.
- Schank, R. C. **Identification of Conceptualizations Underlying Natural Language**. In R. C. Schank & K. M. Colby (Eds.), **Computer Models of Thought and Language**. San Francisco: W. H. Freeman, 1973. Pp. 187-247. (b)
- Schank, R. C. **The Structure of Episodes in Memory**. In D. G. Bobrow & A. Collins (Eds.), **Representation and Understanding**. New York: Academic Press, 1975. Pp. 237-272. (c)
- Schank, R. C. **Conceptual Information Processing**. Amsterdam: North-Holland, 1975. (a)
- Schank, R. C. **The Fourteen Primitive Actions and Their Inferences**, Stanford AI Laboratory, Memo AIM-183, March 1973. (a)
- Schank, R. C. **Conceptual Dependency: A Theory of Natural Language Understanding**. **Cognitive Psychology**, 1972, 3, 552-631.
- Schank, R. C. **What Makes Something 'Ad Hoc'**. TINLAP-2, 1978, pp. 8-13.
- Schank, R. C., & Tesler, L. G. **A Conceptual Parser for Natural Language**. IJCAI 1, 1969, 569-578.
- Schank, R. C., Goldman, N., Rieger, C. J., & Riesbeck, C. K. **Primitive Concepts Underlying Verbs of Thought**, Stanford AI Project, Memo AIM-162, February 1972.
- Schank, R. C., Goldman, N., Rieger, C. J., & Riesbeck, C. K. **MARGIE: Memory, Analysis, Response Generation, and Inference on English**. IJCAI 3, 1973, 255-261.
- Schank, R. C., Tesler, L. G., & Weber, S. **Spinoza II: Conceptual Case-based Natural Language Analysis**. Stanford AI Project, Memo AIM-109, January 1970.

- Schank, R. C., & Abelson, R. P. **Scripts, Plans, Goals, and Understanding**. Hillsdale, N.J.: Lawrence Erlbaum, 1977.
- Schank, R.C. The Primitive ACTs of Conceptual Dependency. TINLAP, 1975, pp. 38-41. (b)
- Schubert, L. K. Expanding the expressive power of semantic networks. *IJCAI* 4, 1975, 158-164.
- Shapiro, S. C. A net structure for semantic information storage and deduction, and retrieval. *IJCAI* 2, 1971, 512-523.
- Shortliffe, E. H. MYCIN: A rule-based computer program for advising physicians regarding antimicrobial therapy selection, AIM 251, Stanford Univ., 1974. (Also published as: MYCIN: Computer-based Medical Consultations, American Elsevier, 1976.)
- Siklossy, L. Natural Language Learning by Computer, in Representation and Meaning, H.A.Simon and L.Siklossy (eds), Prentice-Hall, N.J., 1972
- Simmons, R. F. Semantic networks: Their computation and use for understanding English sentences. In R. Schank & K. Colby (Eds.), **Computer Models of Thought and Language**. San Francisco: Freeman, 1973.
- Simmons, R. F., & Slocum, J. Generating English discourse from semantic nets. *Communications of the ACM*, 1972, 15:10, 891-905.
- Simon, H. A. **The Sciences of the Artificial**. MIT Press, 1969.
- Sloman, A. Afterthoughts on analogical representations. TINLAP-1, 1975.
- Sloman, A. Interactions between philosophy and AI--the role of intuition and non-logical reasoning in intelligence. *Artificial Intelligence*, 1971, 2, 209-225.
- Sussman, G., & McDermott, D. V. **Conniver Reference Manual**, MIT AI Memo 259, MIT, 1972.
- Sussman, G., Winograd, T., & Charniak, E. **Micro-Planner Reference Manual**, MIT AI Memo 203, MIT, 1970.
- Szolovitz, P., Hawkinson, L. B., & Martin, W. A. An overview of OWL, a language for knowledge representation, MIT LCS-TM-86, 1977.
- Tesler, L.G., H.J. Enea, and D.C.Smith The LISP70 Pattern Matching System, Proc IJCAI3, 1973
- Vere, S. A. Induction of Concepts in the Predicate Calculus, Proc IJCAI4, 1975
- Vere, S. A. Relational Production Systems, *Artificial Intelligence*, 8, 1977, 47-68
- Walker, D. E., et. al. **Speech Understanding Research**, SRI Final Tech. Report. SRI International, Inc., Menlo Park, CA, 1976.

- Waterman, D. A. Generalization learning techniques for automating the learning of heuristics. *Artificial Intelligence*, 1970, 1, 121-170.
- Weyhrauch, R. W. A Users Manual for FOL, Stanford AI Memo 235.1, Stanford University, Computer Science Dept., 1977.
- Weyhrauch, R. W. *Prolegomena to a theory of formal reasoning*, AI Memo 315, Stanford Computer Science Dept., Stanford University, 1978.
- Wilks, Y. A. *Grammar, Meaning and the Machine Analysis of Language*. London: Routledge & Kegan Paul, 1972.
- Wilks, Y. A. Good and Bad Arguments about Semantic Primitives. DAI Research Report No. 42, Dept. of Artificial Intelligence, University of Edinburgh, May 1977. (a)
- Wilks, Y. A. An Artificial Intelligence Approach to Machine Translation. In R. C. Schank & K. M. Colby (Eds.), *Computer Models of Thought and Language*. San Francisco: W. H. Freeman, 1973. Pp. 114-151.
- Wilks, Y. A. An Intelligent Analyzer and Understander of English. *CACM*, 1975, 18, 264-274. (a)
- Wilks, Y. A. Knowledge Structures and Language Boundaries. *IJCAI* 5, 1977, 151-157. (b)
- Wilks, Y. A. *Computable Semantic Derivations*, Systems Development Corp., Santa Monica, SP-3017, 1968.
- Wilks, Y. A. A Preferential, Pattern-Seeking Semantics for Natural Language Inference. *Artificial Intelligence*, 1975, 6, 53-74. (b)
- Wilks, Y. A. Methodological Questions about Artificial Intelligence: Approaches to Understanding Natural Language. *Journal of Pragmatics*, 1977, 1, 69-84. (c)
- Wilks, Y. A. Primitives and Words. *TINLAP*, 1975, pp. 42-45. (c)
- Winograd, T. *Language as a Cognitive Process*. In press, 1978.
- Winograd, T. *Five Lectures on Artificial Intelligence*, Stanford AI-Memo 246, AI Lab, Stanford University, 1974.
- Winograd, T. *Understanding Natural Language*. New York: Academic Press, 1972.
- Winograd, T. Extended Inference Modes in Computer Reasoning Systems. In *Artificial Intelligence*, 1979, in press.
- Winograd, T. Frame representations and the declarative/procedural controversy. In D. Bobrow & A. Collins (Eds.), *Representation and Understanding*. New York: Academic Press, 1975. Pp. 185-210.
- Winograd, T. On Primitives, Prototypes, and Other Semantic Anomalies. *TINLAP-2*, 1978, pp. 25-32.

Winston, P. H. Learning structural descriptions from examples. In P. H. Winston (Ed.), *The Psychology of Computer Vision*. New York: McGraw-Hill, 1975.

Winston, P. H. *Artificial Intelligence*. Cambridge, Mass.: Addison-Wesley, 1977. (a)

Winston, P. H. Learning by hypothesizing and justifying transfer frames, MIT AIM-414, 1977. (b)

Woods, W. et al. Speech understanding systems: Final Report. BBN Report No. 3438, 1976.

Woods, W. A. What's in a Link: foundations for semantic networks. In D. Bobrow & A. Collins (Eds.), *Representation and Understanding*. New York: Academic Press, 1975.

Woods, W. A. Procedural Semantics for a Question-Answering Machine. *Proc. Fall Joint Computer Conference, AFIPS*, 1968, 33(1), 457-471.

## Index

Abelson, Robert 199  
 analogical 191

backtracking 177

combinatorial explosion 130, 131, 174  
 constraint structured planning 194

depth 130  
 direct 191  
 discrimination net 129

Eastman 193  
 EPAM 129

frame 135  
 frame systems 200  
 frame problem 180  
 frames 199  
 frames, declarative knowledge 203  
 frames, procedural knowledge 203  
 fregean 191  
 FRL-0 206  
 Funt 194

Gelernter 192  
 General Space Planner 193  
 Geometry Theorem Prover 192  
 grain 124  
 Green, Cordell 173  
 GUS 204

Heuristic search 131  
 heuristics 130  
 human problem-solving 130

inheritance 200  
 inheritance hierarchy 132  
 interlingua 134

KRL 206

legal move generators 129

Micro-PLANNER 177  
 models 180, 181  
 modularity 134, 182

Natural Language processing 129  
 NUDGE 206

parallelism 196  
 pattern-directed invocation 177  
 perceptual primitives 196  
 PLANNER 131, 132, 177, 181  
 power/generalizability tradeoff 197  
 predicate calculus 130  
 problem-reduction 192  
 procedural attachment 203  
 Procedural Embedding of Knowledge 173  
 procedural representation 132  
 production systems 134  
 productions 134

QA3 173  
 QA4 131

Raphael, Bertram 173  
 resolution method 131  
 rules of inference 131, 132

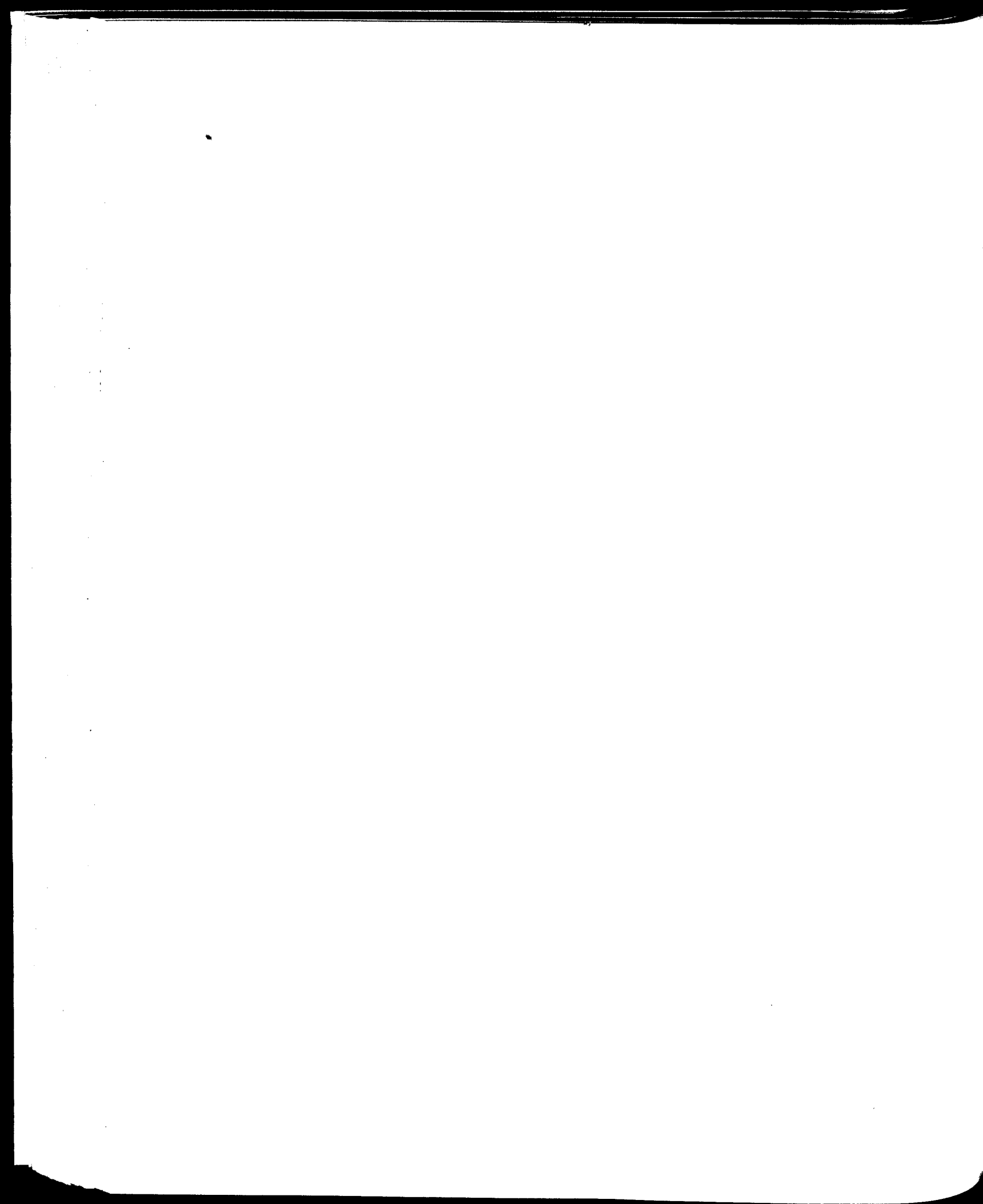
SAD-SAM 129



Schank, Roger 199  
scope 124  
scripts 199  
Semantic Interpretation Function 191  
semantic net 132  
semantic primitives 134  
SHRDLU 132  
SIR 129, 173  
slots 199  
Space planning 193  
spreading activation 154  
state space search 129  
state-space search 131

theorem prover 131  
transition operators 129

WHISPER 194  
Woods, William 173



# Natural Language Understanding

# Natural Language

## Table of Contents

A. Natural Language Processing Overview . . . . .	223
B. Mechanical Translation . . . . .	228
C. Grammars . . . . .	233
1. Formal Grammars . . . . .	233
2. Transformational Grammars . . . . .	238
3. Systemic Grammar . . . . .	242
4. Case Grammars . . . . .	244
D. Parsing . . . . .	247
1. Overview of Parsing Techniques . . . . .	247
2. Augmented Transition Nets . . . . .	252
3. The General Syntactic Processor . . . . .	256
E. Text Generation . . . . .	261
F. Natural Language Processing Systems . . . . .	267
1. Early Natural Language Systems . . . . .	267
2. Wilks's Mechanical Translation System . . . . .	273
3. LUNAR . . . . .	276
4. SHRDLU . . . . .	279
5. MARGIE . . . . .	283
6. SAM and PAM . . . . .	287
7. LIFER . . . . .	296
References . . . . .	302
Index . . . . .	312

## A. Natural Language Processing Overview

The most common way that human beings communicate is by speaking or writing in one of the "natural" languages, like English, French, or Chinese. Computer programming languages, on the other hand, seem awkward to humans. These "artificial" languages are designed to have a rigid format, or *syntax*, so that a computer program reading and compiling code written in an artificial language can understand what the programmer means. In addition to being structurally simpler than natural languages, the artificial languages can express easily only those concepts that are important in programming: "Do this then do that," "See if such and such is true," etc. The things that can be expressed in a language are referred to as the *semantics* of the language.

The research on understanding natural language described in this section of the Handbook is concerned with programs that deal with the full range of meaning of languages like English. Computers that can understand what people mean when typing (or speaking) English sentences will be easier to use and will fit more naturally into people's lives. In addition, artificial intelligence (AI) research in natural language processing hopes to extend our knowledge of the nature of language as a human activity. Programs have been written that are quite successful at understanding somewhat constrained input: the user is limited in either the structural variation of his sentences (syntax constrained by an artificial *grammar*) or in the number of things he can "mean" (in domains with constrained semantics). Some of these programs are adequate for many useful computer-interface tasks and are available commercially. But the fluent use of language as humans use it is still elusive, and natural language (NL) processing is an active area of research in AI.

This article presents a brief sketch of the history of natural language processing research in AI, and it attempts to give some idea of the current state of the art in NL and related research in representing knowledge about the world within the language understanding programs. The next article is a historical sketch of the very earliest ideas about processing language with computers, to achieve *mechanical translation* of one language into another. It is followed by two sections containing technical articles on some of the grammars and *parsing* techniques that AI researchers have used in their programs. Then, after an article on *text generation*, which involves the creation of sentences by the program to express what it wants to say, there are a half dozen articles describing some of the most important NL systems.

Two other sections of the Handbook are especially relevant to NL research. Speech Understanding research attempts to build computer interfaces that actually understand spoken language. Speech and natural language understanding research have been closely linked. Increasingly inseparable from NL research is the study of Knowledge Representation, because AI researchers have come to believe that a very large amount of knowledge about the world is used in even simple dialogue. Research in the representation of knowledge explores ways of making this *world knowledge* accessible to the computer program by "representing" it in internal data structures.

### History

Research in *computational linguistics*, the use of computers in the study of language, started in the 1940s, soon after computers became available commercially. The machine's

ability to manipulate symbols was first used to compile lists of word occurrences (word lists) and concordances (their contexts in written texts). Such surface-level machine processing of text was of some value in linguistic research, but it soon became apparent that the computer could perform much more powerful linguistic functions than merely counting and rearranging data.

In 1949, Warren Weaver proposed that computers might be useful for "the solution of the world-wide translation problem" (Weaver, 1955, p. 15). The resulting research effort, called *mechanical translation*, attempted to simulate with a computer the presumed functions of a human translator: looking up each word in a bilingual dictionary; choosing an equivalent word in the output language; and, after processing each sentence, arranging the resulting string of words to fit the output language's word order. Despite the attractive simplicity of the idea, many unforeseen problems arose, both in selecting appropriate word equivalences and in arranging them to produce a sentence in the output language. Article B discusses the history, problems, and current state of research on mechanical translation.

In the 1960s a new group of computer programs was developed that attempted to deal with some of the more complex issues of language that had led to the difficulties in the mechanical translation efforts. These early natural language programs mark the beginning of artificial intelligence work in understanding language. They no longer assume that human communication is a process of word manipulation. Instead, they view human language as a complex cognitive ability involving many different kinds of knowledge: the structure of sentences, the meaning of words, a model of the listener, the rules of conversation, and an extensive shared body of general information about the world. Several of these programs are described briefly in Article F1.

The focus of modern work in natural language processing in AI is "understanding" language. Several different tasks have been used as the criterion for defining what constitutes a demonstration that the program understands a piece of text; these tasks include *paraphrasing*, *question answering*, *mechanical translation*, and *information retrieval*. Many design issues depend on which type of task the program is to perform, but the general approach has been to model human language as a knowledge-based system for processing communications and to create a computer program that serves as a working model of this system.

AI researchers in natural language processing expect their work to lead both to the development of practical, useful language understanding systems and to a better understanding of language and the nature of intelligence. The computer, like the human mind, has the ability to manipulate symbols in complex processes, including processes that involve decision making based on stored knowledge. It is an assumption of the field that the human use of language is a cognitive process of this sort. By developing and testing computer-based models of language processing that approximate human performance, researchers hope to understand better how human language works.

#### Approaches to NL Processing

Natural language research projects have had diverse goals and used diverse methods, making their categorization somewhat difficult. One coherent scheme, borrowed from

Winograd (1972), groups natural language programs according to how they represent and use knowledge of their subject matter. On this basis, natural language programs can be divided into four historical categories.

The earliest natural language programs sought to achieve only limited results in specific, constrained domains. These programs used ad hoc data structures to represent "knowledge." Programs like BASEBALL, SAD-SAM, STUDENT, and ELIZA (see Article F1) searched their input sentences, which were restricted to simple declarative and interrogative forms, for key words or patterns representing known objects and relationships. Domain-specific rules, called *heuristics*, were used to derive the required information from the key words in the sentence and the knowledge in the database. Though they performed relatively small tasks and avoided or ignored many of the complexities in language, their results and methods were the impetus to dealing with more difficult problems.

The second category can be called *text-based systems*. These programs, such as PROTOSYNTHESIS I (Simmons, Burger, & Long, 1966) and the Teachable Language Comprehender, TLC (Quillian, 1969), attempted to expand beyond the limits of a specific domain. The programs dealt with full English text as a base, rather than with key words or phrases. Input text was interpreted as a request to access a structured information store, and a variety of clever methods were used to identify the proper response. Though more general than their predecessors, these programs still failed to deal with the underlying *meaning* of the English language input. They were able to give only responses that had been pre-stored as data--they had no deductive power.

To try to deal with the problem of how to characterize and use the meaning of sentences, a group of programs was developed called *limited logic systems*. In systems like SIR (Raphael, 1968), DEACON (Thompson, 1968), and CONVERSE (Kellogg, 1968), the information in the database is stored in a formal, albeit ad hoc, notation, and mechanisms are provided for translating input sentences into the same form. The function of the formal notation is to attempt to liberate the informational content of the input from the structure of English. The overall goal of these systems was to accept complex input information (e.g., information containing quantifiers and relationships), use it to perform inferences on the database, and thus realize answers to complex questions. Problems, however, arose from the fact that the complexity of the stored information was not really part of the database but was built into the system's routines for manipulating the database. PROTOSYNTHESIS II (Simmons, 1966), for example, contained statements of the form "A is X" and "X is B" and tried to answer "Is A B?", based on transitivity. The deductive mechanism required for these inferences was embedded in special-purpose subroutines, rather than in the database as a "theorem," and thus was not available to be used to perform more involved inferences, which require a longer chain of reasoning.

### Representing Knowledge in NL Programs

The fourth approach to building language understanding programs might be called *knowledge-based systems* and is closely intertwined with current research on the representation of knowledge (see the Knowledge Representation section of the Handbook). Among the most important knowledge representation schemes explored in NL research have been: procedural semantics, semantic networks, case systems, and frame systems.

In the early 1970s, two systems were built that attempted to deal with both syntactic and semantic problems in a comprehensive way. William Woods's LUNAR system (Article F4) answered questions about the samples of rock brought back from the moon, using a large database provided by the National Aeronautics and Space Agency. It was one of the first programs to attack the problems of English grammar using an *augmented transition network* parser (Article D2). It used a notion of *procedural semantics* in which queries were first converted in a systematic way into a "program" to be executed by the retrieval component. Terry Winograd's SHRDLU system (Article F5) carried on a dialogue with a user in which the system simulated a robot manipulating a set of simple objects on a table top. The naturalness of the dialogue, as well as SHRDLU's apparent reasoning ability, made it particularly influential in the development of AI ideas. These two systems integrate syntactic and semantic analysis with a body of world knowledge about a limited domain, enabling them to deal with more sophisticated aspects of language and discourse than had previously been possible.

Central to these two systems is the representation of knowledge about the world as procedures within the system. The meanings of words and sentences were expressed as programs in a computer language, and the execution of these programs corresponded to reasoning from the meanings. Direct procedural representations are often the most straightforward way to implement the specific reasoning steps needed for a natural language system. Most of the actual working systems that have been developed have made heavy use of specialized procedural representations, to fill in those places where the more *declarative* representation schemes--those where the "knowledge" is encoded in passive data structures that are interpreted by other procedures--are insufficient. (The *procedural/declarative controversy* has been an important focus in the history of AI. See Article Representation.B.)

Perhaps the most influential declarative representation scheme is the *semantic network*. Semantic networks were first proposed by Quillian (1968) as a model for human associative memory. They used the concepts of graph theory, representing words and meanings as a set of linked nodes. By using a systematic set of link types, it was possible to program simple operations (such as following chains of links) that corresponded to drawing inferences. Another important declarative scheme is the use of standard *logic* formulas (Article Representation.C1), which are subject to mathematical rules of deduction for drawing inferences. The advantage of semantic networks over standard logic is that some selected set of the possible inferences can readily be done in this specialized and efficient way. If these correspond to the inferences that people make easily, then the system will be able to do a more natural sort of reasoning than can be easily achieved using formal logical deduction.

Semantic networks have been the basis for a number of systems, including most of the *speech understanding* systems (see Speech Understanding). Recently there has been a good deal of work on formalizing the network notions so that there is a clear correspondence between the graph operations and the formal semantics of the statements represented (see Article Representation.C2).

Case representations extend the basic notions of semantic nets with the idea of a *case frame*, a cluster of the properties of an object or event into a single concept (see Article C4). There have been a large number of variations on this notion, some of which remain close



to the linguistic forms. Others such as *conceptual dependency* are based on the notion of *semantic primitives*, the construction of all semantic notions from a small set of "primitive" concepts. The MARGIE system (Article F3), built in the early 1970s by Roger Schank and his students, uses the conceptual dependency representation.

As with semantic networks, the advantage of case representations lies in their focus on clustering relevant sets of relationships into single data structures. The idea of clustering structures in a coherent and efficient way has been carried much further in representation schemes based on the notion of a *frame* (Minsky, 1975; see also Article Representation.C6). Where case representations deal primarily with single sentences or acts, frames are applied to whole situations or complex objects or series of events. In analyzing a sentence, narrative, or dialogue, a language understanding system based on frame representations tries to match the input to prototypes for the objects and events in its domain that are stored in its database.

For example, Roger Schank's SAM system (Article F6) makes use of simple, linear *scripts*, which represent stereotyped sequences of events, to understand simple stories. It assumes that the events being described will fit (roughly) into one of the scripts in its knowledge base, which it then uses to fill in missing pieces in the story. The GUS system (Bobrow, et al., 1977) is a prototype travel consultant, carrying on a dialogue to help a person schedule an air trip. It uses frames representing standard trip plans. GUS uses the experimental frame language KRL (Bobrow & Winograd, 1977; see also Article Representation.C6).

The important common element in all of these systems is that the existence of prototype frames makes it possible to use *expectations* in analysis. When a sentence or phrase is input that is ambiguous or underspecified, it can be compared to a description of what would be expected based on the prototype. Assumptions can be made about what was meant, if there is a plausible fit to the expectation. This *expectation-driven* processing seems to be an important aspect of the human use of language, where incomplete or ungrammatical sentences can be understood in appropriate contexts. Research on script- and frame-based systems is the most active area of AI research in natural language understanding at the present time.

The current state-of-the-art in working (non-experimental) NL systems is exemplified by ROBOT (Harris, 1977), LIFER (Hendrix, 1977a), and PHLQA1 (Landsbergen, 1976).

## References

General discussions of natural language processing research in AI can be found in Boden (1977), Wilks (1974), Winograd (1974), Charniak & Wilks (1976), Schank & Abelson (1977), and Winograd (forthcoming). Waltz (1977) contains more than fifty brief summaries of current projects and systems. In addition, many historically important NL systems are described in Feigenbaum & Feldman (1963), Minsky (1968), Rustin (1973), Schank & Colby (1973), and Winograd (1972). COLING (1976), TINLAP-1 (1975), Bobrow & Collins (1975), and TINLAP-2 (1978) are proceedings of recent conferences describing current work in the field.

## B. Mechanical Translation

The concept of translation from one language to another by machine is older than the computer itself. According to Yehoshua Bar-Hillel (1960), one of the early investigators in the field, the idea was perhaps first conceived as early as the early 1930s by P. P. Smirnov-Troyansky of the Soviet Union and G. B. Artsouni of France (see Bar-Hillel, 1960, p. 7). Their work apparently never received much attention, lying dormant until a decade later when the climate was much more favorable, due to the recent invention of the digital computer. In certain quarters of the scientific world people imagined--with some justification--that computers would lead to many entirely new and far-reaching ideas about man and--perhaps less justifiably--that computers would help bring about a new world order. In short, there was tremendous excitement over the potential of these new *thinking machines*, as they were quickly dubbed. This was also the time when Claude Shannon was formulating his ideas on information theory, when Norbert Wiener was devising the concept of *cybernetics*, and when Pitts and McCullough were developing their ideas on neural nets and brain function. Moreover, computing had just passed its initial tests, during the war, with flying colors--in such strategic tasks as breaking codes and calculating complicated nuclear cross sections.

It would be well to bear in mind that, when machine translation work began, programming was done by wiring boards and machine language was the only computer language available. Such concepts as arrays and subroutines were still to appear, not to mention pushdown stacks, compiler languages, recursive procedures, and the like. Furthermore, no one had heard of *context-free* and *context-sensitive grammars*, or of *transformational grammars*, or *augmented transition networks*. At the forefront of computational linguistics, the application of the computer to the study of language, were statistical experiments with language, such as compiling matrices of letter frequencies and of transition frequencies between successive letters. Such matrices could be used to produce interesting samples of *pseudo-language*, by producing words from randomly generated letters with the same characteristics as English words. (Also, see the discussion of Yngve's *random text generation* system in Article E).

### First Attempts

The real genesis of machine translation dates from a series of discussions between Warren Weaver and A. Donald Booth in 1946. Both men were familiar with the work on code breaking by computers, based on letter-frequency and word-frequency tables. It seemed to them that some of the same methods would be applicable to translation and that the principal obstacle would be incorporating a full dictionary of the two languages. Of course they recognized that simply having a dictionary would not solve all problems. Some of the remaining problems would be the following: (a) Many words have several translations, depending upon context; (b) word orders differ from language to language; and (c) idiomatic expressions cannot be translated word for word but must be translated *in toto*. Nevertheless, it appeared plausible, at the time, that the major problem in translating between two languages was simply that of vocabulary--and so at least a large part of translation seemed mechanizable.

In 1947, Booth and D. H. V. Britten worked out a program for dictionary lookup. This was a full-form dictionary, in that each variant of any basic word (e.g., love, loves, loving, etc.) had to be carried as a separate entry in the dictionary. In 1948, R. H. Richens suggested the addition of rules concerning the inflections of words, so that the redundancy

of the multiple dictionary entries could be eliminated. In 1949, Warren Weaver distributed a memorandum entitled *Translation* to about two hundred of his acquaintances, and a considerable wave of interest ensued. In addition to the idea that all languages have many features in common, three other items from that memorandum are worth repeating. The first is the notion of a *window* through which one can view exactly  $2N + 1$  words of text; Weaver suggests that when  $N$  is sufficiently large, one will be able to determine the unique, correct translation for the word that sits in the middle of the window. He then points out that  $N$  may be a function of the word, rather than a constant, and discusses the idea of choosing a value of  $N$  such that, say, 95% of all words would be correctly translated 98% of the time. The second is this intriguing statement: "When I look at an article in Russian, I say, *This is really written in English, but it has been coded in some strange symbols. I will now proceed to decode.*" This certainly carries to an extreme the concept that source text and translated text "say the same thing." In fact, it leads naturally to the third provocative idea of the memorandum that translating between languages A and B means going from A to an intermediate "universal language," or *interlingua*, that, supposedly, all humans share, and thence to B. This idea, of an intermediate representation of the *semantics* or meaning of an utterance, appears often in modern natural language processing work in AI under the heading *representation of knowledge* (see discussion in the Overview and in the Handbook Section on Knowledge Representation).

After Weaver's memorandum, work sprang up in several centers in the United States. Erwin Reifler conceived the idea of two auxiliary functions to be performed by human beings, those of *pre-editor* and *post-editor*. The pre-editor would prepare the input text to be as free as possible of ambiguities and other sources of difficulty; the post-editor would take the machine-translated text and turn it into grammatical, comprehensible prose.

A 1952 conference produced recommendations to implement a dictionary-lookup program and to work towards the invention, or discovery, of the hypothetical universal language, called *Machinese*, which Weaver had proposed as an intermediate language in mechanical translation.

A. G. Oettinger was one of the first to design a program that carried out word-for-word translation of Russian text into English. A very high percentage of the Russian words had more than one possible translation; so all of them were listed in the output English, enclosed in parentheses. Thus, a sample of English output text read as follows:

(In, At, Into, To, For, On) (last, latter, new, latest, lowest, worst) (time, tense) for analysis and synthesis relay-contact electrical (circuit, diagram, scheme) parallel-(series, successive, consecutive, consistent) (connection, junction, combination) (with, from) (success, luck) (to be utilize, to be take advantage of) apparatus Boolean algebra. (Oettinger, 1955, p. 55)

A cleaned-up version of this sentence reads: "In recent times Boolean algebra has been successfully employed in the analysis of relay networks of the series-parallel type" (p. 58). Readers of the translated text were expected to discern from the jumble of synonyms what the cleaned-up text really should be. Clearly, there was still a long, long way to go toward mechanical translation.

In the next year or two, most of the effort was directed toward devising ways to handle different endings of inflected words and estimating the size of vocabulary needed for

translations of varying degrees of quality. In 1954 a journal of mechanical translation was founded, called *MT*. Machine translation received considerable public attention when a group from IBM and Georgetown University made grand claims for a program that translated from Russian to English, although this program was not particularly advanced over any others. In any case, machine translation became an "in" thing and groups sprang up in many countries.

Early attempts focusing on syntactic information were able to produce only low-quality translation and led eventually to extreme pessimism about the possibility of the endeavor. It has since become clear that high-quality translation systems must in some sense *understand* the input text before they can reconstruct it in a second language. For the first time, it was becoming apparent that much "world knowledge" is used implicitly when human beings translate from one language to another. Bar-Hillel gave as an example the pair of sentences, "The pen is in the box," and "The box is in the pen." To quote Bar-Hillel,

I now claim that no existing or imaginable program will enable an electronic computer to determine that the word pen in the [second] sentence . . . has the [meaning *an enclosure where small children can play*] . . . (Bar-Hillel, 1960, p. 159)

He goes on to remark that, to his amazement, no one had ever pointed out that in language understanding there is a world-modeling process going on in the mind of the listener and that people are constantly making use of this subconscious process to guide their understanding of what is being said. Bar-Hillel continues:

A translation machine should not only be supplied with a dictionary but also with a universal encyclopedia. This is surely utterly chimerical and hardly deserves any further discussion. . . . We know . . . facts by inferences which we are able to perform . . . instantaneously, and it is clear that they are not, in any serious sense, stored in our memory. Though one could envisage that a machine would be capable of performing the same inferences, there exists so far no serious proposal for a scheme that would make a machine perform such inferences in the same or similar circumstances under which an intelligent human being would perform them. (pp. 160-161)

Bar-Hillel despaired of ever achieving satisfactory machine translation. His sentiments were perhaps not universally shared, but certainly the momentum of the machine-translation effort slowed considerably in the next decade.

### Current Status

A signal event toward the revival of MT research was the publication in 1957 of Noam Chomsky's *Syntactic Structures*, in which *transformational grammars* were introduced. This book spurred many new developments in the analysis of syntax. Concurrently, new computer languages and new types of data structures were being explored by computer scientists, leading to the creation (in 1960) of both ALGOL and LISP, with their features of lists, recursion, etc. These languages were the first in a series of languages geared more toward symbol manipulation than "number crunching," as discussed in the AI Programming Languages Section of the Handbook.

The 1960s saw considerable progress toward natural language understanding, such as the development of systemic grammars and of particular programs that carried on a dialogue of sorts with the user: BASEBALL, SAD-SAM, STUDENT, SIR, etc., which are described in Article F1. There were still efforts going on around the world in "pure" machine translation, but in general it seemed that people had agreed that "understanding" language was prerequisite to translation.

The early 1970s have seen some revival of interest in machine translation, partly because some progress has been made in the internal *representation of knowledge*. The programs of Wilks (Article F2) and Schank (Articles F3 and F6), for example, translate input sentences into internal data structures based on *semantic primitives*, which are intended to be "language independent"--elements of meaning that are common to all natural languages. The internal representation can be manipulated relatively easily by procedures that carry out *inferences* forming in effect an internal language or *interlingua* for modeling the world. The data structure(s) derived from a particular sentence could be considered to be a translation of that sentence into Weaver's *Machinese*. If the reverse problem can be solved (i.e., Machinese to, say, Russian), then one would have the realization, on some level, of Weaver's idea (see Article E for research on the *generation* of text.)

In fact, more-or-less practical machine translation has been in use for some years to provide crude, but readable, translation of technical papers. Large machine translation systems have also been built for varied applications. Two examples are the system developed for the 1975 Apollo-Soyuz joint mission and the system currently under development at the European Economic Community Computer Center. The first system is the more sophisticated in that it allows the use of free syntax, while the second has a large vocabulary of approximately 100,000 words (English-French).

It is difficult to evaluate the practicality of machine translation, because there is a complicated quality/usefulness trade-off. In some applications it is worthwhile to have even a very bad translation, if it can be done by a computer in a much shorter time (or much more cheaply) than by humans. In others (such as the preparation of instruction manuals) it is possible to deal with input texts that use a specially restricted form of the language, thereby making translation easier. There is also the possibility of machine-human interactive translating, in which the output of the computer is used not by the ultimate reader but by someone engaged in producing the final translation. The computer can be used to do sub-tasks like dictionary lookup, or can produce more-or-less complete translations that are then checked and polished by a person (who perhaps does not know the original language).

At the current time, computers are being used in these ways in a number of translation systems. There is also a renewed interest in fully automatic translation, based on some of the techniques for dealing with meaning described below. However it is not clear whether we are yet ready to reattack the goal of "fully automatic high quality translation." Much current work on language is based on a belief that deep understanding of what is being said is vital to every language use. Applied to translation, this means that we must first have a program that understands a subject before we can translate material about that subject. Since our ability to model large areas of knowledge is still primitive, this places a strong limit on the scope of material we might handle.

**References**

A brief, popular review of current work in mechanical translation can be found in Wilks (1977a). For the earliest history, see the introduction to Locke & Booth (1955).

See also Bar-Hillel (1960), Bar-Hillel (1964), Bar-Hillel (1970), Booth (1967), Oettinger (1955), Rustin (1973), Schank (1975), Weaver (1955), and Wilks (1973).

## C. Grammars

A *grammar* of a language is a scheme for specifying the sentences allowed in the language, indicating the rules for combining words into phrases and clauses. In natural language processing programs, the grammar is used in *parsing* to "pick apart" the sentences in the input to the program to help determine their meaning and thus an appropriate response. Several very different types of grammars have been used in NL programs and are described in the articles which follow.

### C1. Formal Grammars

One of the more important contributions to the study of language was the theory of *formal languages* introduced by Noam Chomsky in the 1950s. The theory has developed as a mathematical area, not a linguistic one, and has strongly influenced computer science in the design of computer programming languages (artificial languages). Nevertheless, it is useful in connection with *natural language* understanding systems, both as a theoretical and a practical tool.

#### Definitions

A *formal language* is defined as a (possibly infinite) set of strings of finite length formed from a finite vocabulary of symbols. (For example, the strings might be sentences composed from a vocabulary of words.) The *grammar* of a formal language is specified in terms of the following concepts:

1. The *syntactic categories*, such as <SENTENCE> and <NOUN PHRASE>. These syntactic categories are referred to as *nonterminal symbols* or *variables*. Notationally, the nonterminals of a grammar are often indicated by enclosing the category names in angle brackets, as above.

2. The *terminal symbols* of the language, for example the words in English. The terminal symbols are to be concatenated into strings called *sentences* (if the terminals are words). A language is then just a subset of the set of all the strings that can be formed by combining the terminal symbols in all possible ways. Exactly which subset is permitted in the language is specified by the rewrite rules:

3. The *rewrite rules* or *productions* specify the relationships that exist between certain strings of terminals and nonterminal symbols. Some examples of productions are:

```

<SENTENCE> -> <NOUN PHRASE> <VERB PHRASE>
<NOUN PHRASE> -> the <NOUN>
<NOUN> -> dog
<NOUN> -> cat
<VERB PHRASE> -> runs

```

The first production says that the (non-terminal) symbol <SENTENCE> may be "rewritten" as the symbol <NOUN PHRASE> followed by the symbol <VERB PHRASE>. The second permits <NOUN PHRASE> to be replaced by a string composed of the word *the*, which is a terminal symbol, followed by the nonterminal <NOUN>. The next two allow <NOUN> to be replaced by

either dog or cat. Since there are sequences of productions permitting <NOUN PHRASE> to be replaced by the dog or the cat, the symbol <NOUN PHRASE> is said to *generate* these two terminal strings. Finally, <VERB PHRASE> can be replaced by the terminal runs.

4. The **start symbol**. One nonterminal is distinguished and called the "sentence" or "start" symbol, typically denoted <SENTENCE> or S. The set of strings of terminals that can be derived from this distinguished symbol, by applying sequences of productions, is called the *language generated by the grammar*. In our simple example grammar, exactly two sentences are generated:

The cat runs.  
The dog runs.

The important aspect of defining languages formally, from the point of view of computational linguistics and natural language processing, is that if the structure of the sentences is well understood, then a parsing algorithm for analyzing the input sentences will be relatively easy to write (see Section □1 on parsing).

#### The Four Types of Formal Grammars

Within the framework outlined above, Chomsky delineated four types of grammars, numbered 0 through 3. The most general class of grammar is type 0, which has no restrictions on the form that rewrite rules can take. For successive grammar types, the form of the rewriting rules allowed is increasingly restricted, and the languages that are generated are correspondingly simpler. The simplest formal languages (types 2 and 3) are, as it turns out, inadequate for describing the complexities of human languages. On the other hand, the most general formal languages are difficult to handle computationally. (See Article C2 for a fuller discussion of these difficulties. There is an intimate and interesting connection between the theory of formal languages and the theory of computational complexity; see Hopcroft & Ullman, 1969.) The following discussion gives a formal account of the different restrictions applied in each of the four grammar types.

Formally, a grammar  $G$  is defined by a quadruple  $(VN, VT, P, S)$  representing the nonterminals, terminals, productions, and the start symbol, respectively. The symbol  $\underline{V}$ , for "vocabulary," is used to represent the union of the sets  $\underline{VN}$  and  $\underline{VT}$ , which are assumed to have no elements in common. Each production in  $\underline{P}$  is of the form

$$X \rightarrow Y$$

where  $\underline{X}$  and  $\underline{Y}$  are strings of elements in  $\underline{V}$ , and  $\underline{X}$  is not the empty string.

**Type 0.** A type-0 grammar is defined as above: a set of productions over a given vocabulary of symbols with no restrictions on the form of the productions. It has been shown that a language can be generated by a type-0 grammar if and only if it can be recognized by a Turing machine; that is, we can build a Turing machine which will halt in an ACCEPT state for exactly those input sentences that can be generated by the language.

**Type 1.** A type-0 grammar is also of type 1 if the form of the rewrite rules is



restricted so that, for each production  $X \rightarrow Y$  of the grammar, the right-hand side  $Y$  contains at least as many symbols as the left-hand side  $X$ . Type-1 grammars are also called *context sensitive* grammars. An example of a context-sensitive grammar with start symbol  $S$  and terminals  $a$ ,  $b$ , and  $c$  is the following:

$$\begin{aligned} S &\rightarrow aSBC \\ S &\rightarrow aBC \\ CB &\rightarrow BC \\ aB &\rightarrow ab \\ bB &\rightarrow bb \\ bC &\rightarrow bc \\ cC &\rightarrow cc \end{aligned}$$

The language generated by this grammar is the set of strings  $abc$ ,  $aabbcc$ ,  $aaabbbccc$  . . . . This language, where each symbol must occur the same number of times and must appear in the right position in the string, cannot be generated by any grammar of a more restricted type (i.e., type 2 or type 3).

An alternate (equivalent) definition for context-sensitive grammars is that the productions must be of the form

$$uXv \rightarrow uYv$$

where  $X$  is a single nonterminal symbol;  $u$  and  $v$  are arbitrary strings, possibly empty, of elements of  $V$ ; and  $Y$  is a nonempty string over  $V$ . It can be shown that this restriction generates the same languages as the first restriction, but this latter definition clarifies the term *context-sensitive*:  $X$  may be rewritten as  $Y$  only in the context of  $u$  and  $v$ .

**Type 2. Context-free grammars** or type-2 grammars are grammars in which each production must have only a single non-terminal symbol on its left-hand side. For example, a context-free grammar generating the sentences  $ab$ ,  $aabb$ ,  $aaabbb$  . . . is:

$$\begin{aligned} S &\rightarrow aSb \\ S &\rightarrow ab \end{aligned}$$

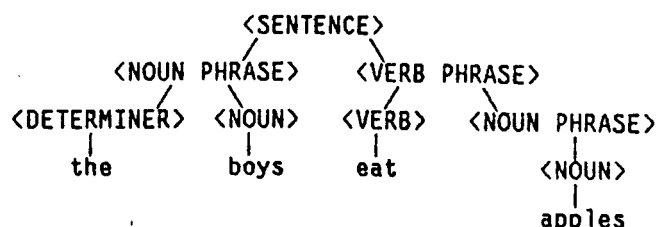
Again, it is not possible to write a context-free grammar for the language composed of the sentences  $abc$ ,  $aabbcc$ ,  $aaabbbccc$  . . . --having the same number of  $c$ 's at the end makes the language more complex. The simpler language here, in turn, cannot be generated by a more restricted (type-3) grammar.

An example of a context-free grammar that might be used to generate some sentences in natural language is the following:

$$\begin{aligned} \langle \text{SENTENCE} \rangle &\rightarrow \langle \text{NOUN PHRASE} \rangle \langle \text{VERB PHRASE} \rangle \\ \langle \text{NOUN PHRASE} \rangle &\rightarrow \langle \text{DETERMINER} \rangle \langle \text{NOUN} \rangle \\ \langle \text{NOUN PHRASE} \rangle &\rightarrow \langle \text{NOUN} \rangle \\ \langle \text{VERB PHRASE} \rangle &\rightarrow \langle \text{VERB} \rangle \langle \text{NOUN PHRASE} \rangle \\ \langle \text{DETERMINER} \rangle &\rightarrow \text{the} \\ \langle \text{NOUN} \rangle &\rightarrow \text{boys} \\ \langle \text{NOUN} \rangle &\rightarrow \text{apples} \\ \langle \text{VERB} \rangle &\rightarrow \text{eat} \end{aligned}$$

In this example, the, boys, apples, and eat are the terminals in the language and <SENTENCE> is the start symbol.

An important property of context-free grammars in their use in NL programs is that every derivation can conveniently be represented as a tree, which can be thought of as displaying the structure of the derived sentence. Using the grammar above, the sentence "the boys eat apples" has the following *derivation tree*:



Of course, "the apples eat boys" is also a legal sentence in this language. Derivation trees can also be used with context-sensitive (type-1) grammars, provided the productions have the alternate form  $uXv \rightarrow uYv$ , described above. For this reason, context-free and context-sensitive grammars are often called *phrase-structure grammars* (see Chomsky, 1959, pp. 143-144, and Lyons, 1968, p. 236).

**Type 3.** Finally, if every production is either of the form

$$X \rightarrow aY \quad \text{or} \quad X \rightarrow a$$

where  $X$  and  $Y$  are single variables and  $a$  is a single terminal, the grammar is a type-3 or *regular grammar*. For example, a regular grammar can be given to generate the set of strings of one or more  $a$ s followed by one or more  $b$ s (but with no guarantee of an equal number of  $a$ s and  $b$ s):

$$\begin{aligned} S &\rightarrow aS \\ S &\rightarrow aT \\ T &\rightarrow b \\ T &\rightarrow bT \end{aligned}$$

#### Discussion: Language and Computational Algorithms

Because of the increasingly restricted forms of productions in grammars of type 0, 1, 2, and 3, each type is a proper subset of the type above it in the hierarchy. A corresponding hierarchy exists for formal languages. A language is said to be of type  $i$  if it can be generated by a type- $i$  grammar. It can be shown that languages exist that are context-free (type 2) but not regular (type 3); context-sensitive (type 1) but not context-free; and type 0 but not context-sensitive. Examples of the first two have been given above.

For regular and context-free grammars, there are practical parsing algorithms to determine whether or not a given string is an element of the language and, if so, to assign to it a syntactic structure in the form of a derivation tree. Context-free grammars have considerable application to programming languages. Natural languages, however, are not

generally context-free (Postal, 1962; Chomsky, 1963), and they also contain features that can be handled more conveniently, if not exclusively, by a more powerful grammar. An example is the requirement that the subject and verb of a sentence be both singular or both plural. Some of the types of grammars and parsing algorithms that have been explored as more suitable for natural language are discussed in the articles that follow.

### References

For a general discussion of the theory of formal grammars and their relation to automata theory, see Hopcroft & Ullman (1969). Their use in NL research is discussed in Winograd (forthcoming).

Also of interest, of course, are the works of Chomsky (especially 1956, 1957, and 1959), as well as Lyons (1968), Lyons (1970), and Postal (1964).

## C2. Transformational Grammars

The term *transformational grammar* refers to a theory of language introduced by Noam Chomsky in *Syntactic Structures* (1957). In the theory an utterance is characterized as the surface manifestation of a "deeper" structure representing the "meaning" of the sentence. The deep structure can undergo a variety of "transformations" of form (word order, endings, etc.) on its way up, while retaining its essential meaning. The theory assumes that an adequate grammar of a language like English must be a *generative grammar*, that is, that it must be a statement of finite length capable of (a) accounting for the infinite number of possible sentences in the language and (b) assigning to each a structural description that captures the underlying knowledge of the language possessed by an idealized native user. A *formal system of rules* is such a statement; it "can be viewed as a device of some sort for producing the sentences of the language under analysis" (Chomsky, 1957, p. 11). The operation of the device is not intended to reflect the processes by which people actually speak or understand sentences, just as a formal proof in mathematics does not purport to reflect the processes by which the proof was discovered. As a model of abstract knowledge and not of human behavior, generative grammar is said to be concerned with competence, as opposed to performance.

### The Inadequacy of Phrase-structure Grammars

Given that a grammar is a generative rule-system, it becomes a central task of linguistic theory to discover what the rules should look like. In *Syntactic Structures* (1957) and elsewhere (see Chomsky, 1963, Postal, 1964), it was shown that English is neither a *regular* nor a *context-free* language. The reason is that those restricted types of grammars (defined in Article C1) cannot generate certain common constructions in everyday English, such as the one using "respectively":

Arthur, Barry, Charles, and David are the husbands of Jane, Joan, Jill,  
and Jennifer, respectively.

It was not determined whether a more powerful (i.e., context-sensitive) grammar could be written to generate precisely the sentences of English; rather, such a grammar was rejected for the following reasons.

1. It made the description of English unnecessarily clumsy and complex--for example, in the treatment required for conjunction, auxiliary verbs, and passive sentences.
2. It assigned identical structures (derivation trees) to sentences that are understood differently, as in the pair:

The picture was painted by a new technique.  
The picture was painted by a new artist.

3. It provided no basis for identifying as similar sentences having different surface structures but much of their "meaning" in common:

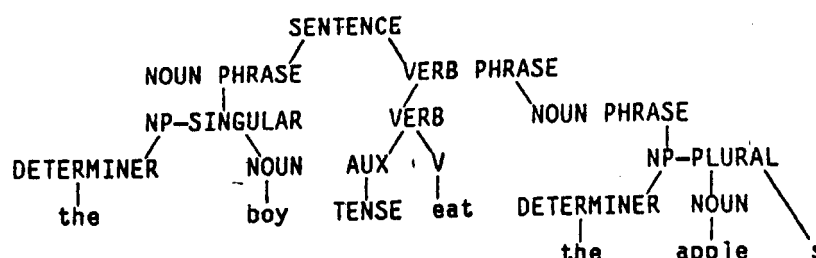
John ate an apple.  
 Did John eat an apple?  
 What did John eat?  
 Who ate an apple?

The failure of phrase-structure grammar to explain such similarities and differences was taken to indicate the need for analysis on a higher level, which transformational grammar provides.

### Transformational Rules

In *Syntactic Structures*, Chomsky proposed that grammars should have a tripartite organization. The first part was to be a phrase-structure grammar generating strings of morphemes representing simple, declarative, active sentences, each with an associated phrase marker or derivation tree. Second, there would be a sequence of *transformational rules* rearranging the strings and adding or deleting morphemes to form representations of the full variety of sentences. Finally, a sequence of morphophonemic rules would map each sentence representation to a string of phonemes. Although later work has changed this model of the grammar, as well as the content of the transformational rules, it provides a basis for a simple illustration.

Suppose the phrase-structure grammar is used to produce the following derivation tree:



To generate "the boy ate the apples," one would apply transformations mapping "TENSE + eat" to "eat + PAST"; a morphophonemic rule would then map "eat + PAST" to ate. To derive "the boy eats the apples," the transformational rule used would select present tense and, because the verb follows a singular noun phrase, would map "TENSE + eat" to "eat + s." It is noteworthy that the transformational rule must look at nonterminal nodes in the derivation tree to determine that "the boy" is in fact a singular noun phrase. This example illustrates one respect in which transformational rules are broader than the rules of a phrase-structure grammar.

The transformations mentioned so far are examples of *obligatory transformations*, insuring agreement in number of the subject and the verb. To obtain "the apples were eaten by the

boy," it would be necessary first to apply the optional *passive* transformation, changing a string analyzed as

NOUN-PHRASE-1 + AUX + V + NOUN-PHRASE-2  
to  
NOUN-PHRASE-2 + (AUX + be) + (en + V) + by + NOUN-PHRASE-1

In other words, this *optional transformation* changes "the boy TENSE eat the apples" to "the apples TENSE be (en eat) by the boy," and then forces agreement of the auxiliary verb with the new plural subject. Further obligatory transformations would yield "the apples be PAST eaten by the boy" (where "be + PAST," as opposed to "be + s + PAST," is ultimately mapped to were). The *ordering* of transformational rules is thus an essential feature of the grammar.

### Revisions to the Model

In *Aspects of the Theory of Syntax* (1965), Chomsky made several revisions to the model presented in *Syntactic Structures*. The version outlined in the more recent book has been called the "standard theory" of generative grammar and has served as a common starting-point for further discussion. In the standard theory (as summarized in Chomsky, 1971), sentence generation begins from a context-free grammar generating a sentence structure and is followed by a selection of words for the structure from a *lexicon*. The context-free grammar and lexicon are said to form the *base* of the grammar; their output is called a *deep structure*. A system of transformational rules maps deep structures to *surface structures*; together, the base and transformational parts of the grammar form its *syntactic component*. The sound of a sentence is determined by its surface structure, which is interpreted by the *phonological component* of the grammar; deep structure, interpreted by the *semantic component*, determines sentence meaning. It follows that the application of transformational rules to deep structures must preserve meaning: This was the Katz-Postal hypothesis, which required enlarging the generative capacity of the base and revising many of the transformational rules suggested earlier (Katz & Postal, 1964).

The place of the semantic component in the standard theory has been the major source of current issues. For example, the following pairs of sentences have different meanings, but their deep structures, in the standard theory, are the same.

Not many arrows hit the target.  
Many arrows didn't hit the target.

Each of Mary's sons loves his brothers.  
His brothers are loved by each of Mary's sons.

Chomsky's response was to revise the standard theory so that both the deep structure of a sentence and its subsequent transformations are input to the semantic component (Chomsky, 1971). He exemplifies the position of *interpretive semantics*, which keeps the syntactic component an autonomous system. The opposing view, called *generative semantics*, is that syntax and semantics cannot be sharply separated and, consequently, that a distinct level of syntactic deep structure does not exist. (This issue is discussed in Charniak & Wilks, 1976.)

There have been a number of developments within the theory of transformational grammar since the work reviewed here, and current debates have called into question many of the basic assumptions about the role of transformations in a grammar. For current discussions of these issues, see Akmajian, Culicover and Wasow (1977) and Bresnan (1978).

### References

The classic references here are, of course, Chomsky (1957) and Chomsky (1965). Chomsky (1971) is a shorter and more recent discussion. Akmajian, Culicover, & Wasow (1977) and Bresnan (1978) are the latest word on transformation theory.

Also see Charniak & Wilks (1976), Chomsky (1956), Chomsky (1959), Chomsky (1963), Harman (1974), Katz & Postal (1964), Lyons (1968), Lyons (1970), Postal (1964), and Steinberg & Jakobovits (1971).

### C3. Systemic Grammar

Systemic grammar, developed by Michael Halliday and others at the University of London, is a theory within which linguistic structure as related to the *function* or use of language, often termed *pragmatics*, is studied. According to Halliday (1961, p. 141), an account of linguistic structure that pays no attention to the functional demands we make on language is lacking in perspicacity, since it offers no principles for explaining why the structure is organized one way rather than another. This viewpoint is in contrast to that of *transformational grammar*, which has been concerned with the syntactic structure of an utterance apart from its intended use.

#### The Functions of Language

Halliday distinguishes three general functions of language, all of which are ordinarily served by every act of speech.

The *ideational function* serves for the expression of content. It says something about the speaker's experience of the world. Analyzing a clause in terms of its ideational function involves asking questions like: What kind of process does the clause describe--an action, a mental process, or a relation? Who is the actor (the logical subject)? Are there other participants in the process, such as goal (direct object) or beneficiary (indirect object)? Are there adverbial phrases expressing circumstances like time and place? The organization of this set of questions is described by what Halliday calls the *transitivity system* of the grammar. (This is related to the ideas of *case grammars* discussed in Article C4.)

The *interpersonal function* relates to the purpose of the utterance. The speaker may be asking a question, answering one, making a request, giving information, or expressing an opinion. The *mood system* of English grammar expresses these possibilities in terms of categories such as statement, question, command, and exclamation.

The *textual function* reflects the need for coherence in language use (e.g., how a given sentence is related to preceding ones). Concepts used for analysis in textual terms include: (1) theme, the element that the speaker chooses to put at the beginning of a clause; and (2) the distinction between what is new in a message and what is given--the latter being the point of contact with what the hearer already knows.

#### Categories of Systemic Grammar

The model of a grammar proposed by Halliday has four primitive categories:

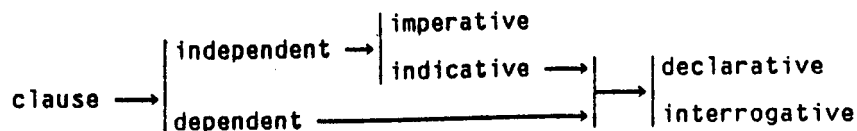
1. The units of language, which form a hierarchy. In English, these are the sentence, clause, group, word, and morpheme. The "rank" of a unit refers to its position in the hierarchy.

2. The structure of units. Each unit is composed of one or more units at the rank below, and each of these components fills a particular role. The English clause, for example, is made up of four groups, which serve as subject, predicator, complement, and adjunct.



3. The classification of units, as determined by the roles to be filled at the level above. The classes of English groups, for instance, are the verbal, which serves as predicator; the nominal, which may be subject or complement; and the adverbial, which fills the adjunct function.

4. The system. A system is a list of choices representing the options available to the speaker. Since some sets of choices are available only if other choices have already been made, the relationship between systems is shown by combining them into networks, as in the simple example below:



The interpretation is that each clause is independent or dependent; if independent, it is either imperative or indicative; and if either indicative or dependent, then it is either declarative or interrogative. In general, system networks can be defined for units of any rank, and entry to a system of choices may be made to depend on any Boolean combination of previous choices.

### Conclusion

Systemic grammar views the act of speech as a simultaneous selection from among a large number of interrelated options, which represent the "meaning potential" of the language. If system networks representing these options are suitably combined and carried to enough detail, they provide a way of writing a generative grammar quite distinct from that proposed by *transformational grammar* (see Hudson, 1971, 1977; McCord, 1975; and Self, 1975). Furthermore, this formalism has been found more readily adaptable for use in natural language understanding programs in AI (see especially Winograd's SHRDLU system, Article F5).

### References

Halliday (1961) and Halliday (1970b) are the most general original references. Winograd (1972) discusses the application of systemic grammar in an NL program.

Also see Halliday (1967-68), Halliday (1970a), Hudson (1971), Hudson (1976), McCord (1975), McIntosh & Halliday (1966), and Self (1975).

#### C4. Case Grammars

Case systems, as used both in modern linguistics and in artificial intelligence, are descendants of the concept of *case* that occurs in traditional grammar. Traditionally, the case of a noun was denoted by an inflectional ending indicating the noun's role in the sentence. Latin, for example, has at least six cases: the nominative, accusative, genitive, dative, ablative, and vocative. The rules for case endings make the meaning of a Latin sentence almost independent of word order: The function of a noun depends on its inflection rather than its position in the sentence. Some present-day languages, including Russian and German, have similar inflection systems, but English limits case forms mainly to the personal pronoun, as in I, my, me, and to the possessive ending 's. Case functions for nouns are indicated in English by using word order or by the choice of preposition to precede a noun phrase--as in "of the people, by the people, and for the people."

The examples above describe what have been called "surface" cases; they are aspects of the *surface structure* of the sentence. Case systems that have attracted more recent attention are "deep" cases, proposed by Fillmore (1968) in his paper *The Case for Case*, as a revision to the framework of *transformational grammar*. The central idea is that the proposition embodied in a simple sentence has a deep structure consisting of a verb (the central component) and one or more noun phrases. Each noun phrase is associated with the verb in a particular relationship. These relationships, which Fillmore characterized as "semantically relevant syntactic relationships," are called *cases*. For example, in the sentence

John opened the door with the key ,

John would be the AGENT of the verb opened, the door would be the OBJECT, and the key would be the INSTRUMENT. For the sentence

The door was opened by John with the key ,

the case assignments would be the same, even though the surface structure has changed.

It was important to Fillmore's theory that the number of possible case relationships be small and fixed. Fillmore (1971b) proposed the following cases:

Agent	-- the instigator of the event.
Counter-Agent	-- the force or resistance against which the action is carried out.
Object	-- the entity that moves or changes or whose position or existence is in consideration.
Result	-- the entity that comes into existence as a result of the action.
Instrument	-- the stimulus or immediate physical cause of an event.
Source	-- the place from which something moves.
Goal	-- the place to which something moves.
Experiencer	-- the entity which receives or accepts or experiences or undergoes the effect of an action.

A more recent proposal (Fillmore, 1971a) recognizes 9 cases: Agent, Experiencer, Instrument, Object, Source, Goal, Location, Time, and Path.

Verbs were classified according to the cases that could occur with them. The cases for any particular verb formed an ordered set called a *case frame*. For example, the verb "open" was proposed to have the case frame

[ OBJECT (INSTRUMENT) (AGENT) ]

indicating that the object is obligatory in the deep structure of the sentence, whereas it is permissible to omit the instrument ("John opened the door") or the agent ("The key opened the door"), or both ("The door opened"). Thus, verbs provide *templates* within which the remainder of the sentence can be understood.

### The Case for Case

The following are some of the kinds of questions for which case analysis was intended to provide answers:

1. In a sentence that is to contain several noun phrases, what determines which noun phrase should be the subject in the surface structure? Cases are ordered, and the highest ranking case that is present becomes the subject.
2. Since one may say "Mother is baking" or "The pie is baking," what is wrong with "Mother and the pie are baking"? Different cases may not be conjoined.
3. What is the precise relationship between pairs of words like "buy" and "sell" or "teach" and "learn"? They have the same basic meaning but different case frames.

One way of looking at deep cases is to view the verb as a predicate taking an appropriate array of arguments. Fillmore has extended the class of predicates to include other parts of speech, such as nouns and adjectives, as well as verbs. Viewing warm as a predicate, for example, enabled case distinctions to account for the differences among the following sentences:

I am warm.	[experiencer]
This jacket is warm.	[instrument]
Summer is warm.	[time]
The room is warm.	[location]

### The Representation of Case Frames

In artificial intelligence programs, such predicates and their arguments can readily be

equated to nodes in *semantic networks*; and the case relations, to the kinds of links between them. Systems making such identifications include those of Simmons (1973), Schank (1975), Schank & Abelson (1977), and Norman & Rumelhart (1975). Semantic nets and related work on *semantic primitives* and *frames* are discussed in the section on Knowledge Representation and in Articles F3 and F6 which describe the MARGIE and SAM systems.

Many other systems using case representations exist. As pointed out in an extensive survey by Bruce (1975), considerable variation exists in both the sets of cases adopted and the ways in which case representation is used. The number of cases used varies from four or five (Schank) to over thirty (Martin). Bruce's proposal on criteria for choosing cases, which departs significantly from Fillmore's original goal of finding a small, fixed set of relationships, is that:

A case is a relation which is "important" for an event in the context in which it is described. (Bruce, 1975)

Case notation has been used to record various levels of sentence structure. As Fillmore introduced it, within the transformational grammar framework, deep cases were "deep" in the sense that "John opened the door" and "the door was opened by John" were given the same representation. They can also be viewed as relatively superficial, however, in that "John bought a car from Bill" and "Bill sold a car to John" could have distinct representations since they have different verbs. At this level, cases have been used in parsing (Wilks, 1976b; Taylor & Rosenberg, 1975); in the representation of English sentences, as opposed to their underlying meanings, as discussed above (Simmons, 1973); and in *text generation* (see Article E).

Systems using case at the deepest level, on the other hand, may represent the meaning of sentences in a way that collapses *buy* and *sell*, for example, into a single predicate (Schank, 1975; Norman & Rumelhart, 1975). A typical problem attacked by these systems is *paraphrasing*, where identifying sentences with the same deep structure is the goal. Schank (1975) also requires that all cases be filled, even if the information required was not explicitly given in the sentences represented. Charniak (1975) suggests that the appropriate use of case at this level of representation is in *inferencing*: The "meaning" of a case would then be the set of inferences one could draw about an entity knowing only its case. In the view of some writers, however, the function of case in natural language understanding systems is usually only as a convenient notation (see Charniak, 1975; Welin, 1975).

### References

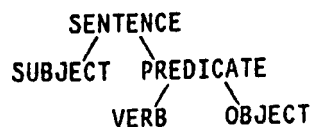
Fillmore (1968) is the classic reference on case grammars. Bruce (1975) is a thorough review of different approaches to case grammar.

Also see Charniak (1975), Fillmore (1971a), Fillmore (1971b), Norman & Rumelhart (1975), Samlowksi (1976), Schank (1973), Schank (1975), Schank & Abelson (1977), Simmons (1973), Taylor & Rosenberg (1975), Welin (1975), Wilks (1976b), and Wilks (1976a).

## D. Parsing

### D1. Overview of Parsing Techniques

Parsing is the "delinearization" of linguistic input, that is, the use of syntax and other sources of knowledge to determine the functions of the words in the input sentence in order to create a data structure, like a *derivation tree*, that can be used to get at the "meaning" of the sentence. A parser can be viewed as a *recursive pattern matcher* seeking to map a string of words onto a set of meaningful syntactic patterns. For example, the sentence "John kissed Mary" could be matched to the pattern:



The set of syntactic patterns used is determined by the *grammar* of the input language. (Several types of grammars are described in the articles in Section C.) In theory, by applying a comprehensive grammar, a parser can decide what is and what is not a grammatical sentence and can build up a data structure corresponding to the syntactic structure of any grammatical sentence it finds. All natural language processing computer systems contain a parsing component of some sort, but the practical application of grammars to natural language has proven difficult.

The design of a parser is a complex problem, both in theory and implementation. The first part of the design concerns the specification of the grammar to be used. The rest of the parsing system is concerned with the method of *use* of the grammar, that is, the manner in which strings of words are matched against patterns of the grammar. These considerations run into many of the general questions of computer science and artificial intelligence concerning process control and manipulation of knowledge.

### General Issues of Parser Design

The design considerations discussed below overlap; that is, a decision in one dimension affects other design decisions. Taken together they present a picture of the variety of issues involved in natural language parsing.

**Uniformity.** Parsers may *represent* their knowledge about word meanings, grammar, etc., with a single scheme or with specialized structures for specific tasks. The representation scheme affects the complexity of the system and the application of knowledge during parsing. If rules and processes are based on specialized knowledge of what the input to the parser will contain, it is possible to do things more quickly and efficiently. On the other hand, if one has a simple uniform set of rules and a consistent algorithm for applying them, the job of writing and modifying the language understanding system is greatly simplified, since all the knowledge in the system is uniformly explicated. In general, there is a trade-off between efficiency and uniformity; an algorithm specially designed for only one language can perform more efficiently than one that could uniformly handle any language.

**Multiple Sources of Knowledge.** Parsing, as originally developed (and still used in programming language compilers), was based purely on syntactic knowledge--knowledge about the *form* of sentences allowed in the language. However, it is possible to design systems in which syntax-based parsing is intermixed with other levels of processing, such as word recognition and use of word meanings. Such methods can alleviate many of the problems of language complexity by bringing more information to bear. Present systems tend toward such intermixed structures, both for effective performance and more psychologically valid modeling of human language understanding (see, for example, Article F5 on SHRDLU and the extensive discussion of multiple sources of knowledge in Article Applications.C3 on the SOPHIE system and the *blackboard* model in the Speech Understanding section).

**Precision.** Another major trade-off involved in parser design is *precision vs. flexibility*. Humans are capable of understanding sentences that are not quite grammatical; even if a person knows that a sentence is "wrong" syntactically, he can often understand it and assign it a structure (and more importantly, a meaning). Some natural language processing systems (e.g., PARRY and ELIZA) have been designed to incorporate this kind of flexibility. By looking for key words and using loose grammatical criteria, these systems can accept far more sentences than would a precise parser. However, these "knowledge-poor" *flexible* parsers lose many benefits of the more complete analysis possible with a precise system, since they rely on vaguer notions of sentence meaning than a precise system. While they reject less often, flexible systems tend to misinterpret more often. Many systems attempt to use additional *knowledge sources*, especially domain-specific knowledge, to increase flexibility while retaining precision.

**Type of structure returned.** As mentioned, parsing is the process of assigning structures to sentences. The form of the structure can vary, from a representation that closely resembles the surface structure of the sentence to a deeper representation in which the surface structure has been extensively modified. Which form is chosen depends upon the use to which the parse structure will be put. Currently, most work in natural language favors the deep structure approach.

These four issues--uniformity, types of knowledge used, precision, and level of representation--are very general questions, and are dealt with in different ways by different systems. In implementing a parser, after settling such general design questions, natural language programmers run up against another set of problems involving specific parsing *strategies*.

### Parsing Strategies

**Backtracking versus parallel processing.** Unfortunately for computational linguists, the elements of natural languages do not always possess unique meanings. For example, in going through a sentence the parser might find a word that could either be a noun or a verb, like "can," or pick up a prepositional phrase that might be modifying any of a number of the other parts of the sentence. These and many other ambiguities in natural languages force the parser to make choices between multiple alternatives as it proceeds through a sentence. Alternatives may be dealt with all at the same time, via *parallel processing*, or one at a time using a form of *backtracking*--backing up to a previous choice-point in the computation and trying again (see Article AI Languages.B3 on *control mechanisms* in AI programming languages). Both of these methods require a significant amount of bookkeeping to keep track of the

multiple possibilities: all the ones being tried, in the case of parallel processing; or all the ones not yet tried, in the case of backtracking. Neither strategy can be said to be innately superior, though the number of alternatives that are actually tried can be significantly reduced when backtracking is guided by "knowledge" about which of the choices are more likely to be correct--called *heuristic* knowledge (see Search.Overview).

**Top-down versus bottom-up.** In deriving a syntactic structure, a parser can operate from the goals, that is, the set of possible sentence structures (*top-down processing*), or from the words actually in the sentence (*bottom-up processing*). A strictly top-down parser begins by looking at the rules for the desired top-level structure (sentence, clause, etc.); it then looks up rules for the constituents of the top-level structure, and progresses until a complete sentence structure is built up. If this sentence matches the input data, the parse is successfully completed, otherwise, it starts back at the top again, generating another sentence structure. A bottom-up parser looks first for rules in the grammar to combine the words of the input sentence into constituents of larger structures (phrases and clauses), and continues to try to recombine these to show how all the words in the input form a legal sentence in the grammar. Theoretically, both of these strategies arrive at the same final analysis, but the type of work required and the working structures used are quite different. The interaction of top-down and bottom-up process control is a common problem in AI and is not restricted natural language programs (see, for example, the discussion in the Speech.A).

**Choosing how to expand or combine.** With either a top-down or bottom-up technique, it is necessary to decide how words and constituents will be combined (bottom-up) or expanded (top-down). The two basic methods are to proceed systematically in one direction (normally left to right) or to start anywhere and systematically look at neighboring chunks of increasing size (this method is sometimes called *island driving*). Both these methods will eventually look at all possibilities, but the choice of how to proceed at this level can have a significant effect on the efficiency of the parser. This particular feature is especially relevant to language processing in the presence of input uncertainty, as occurs, for example, in the *speech understanding* systems.

**Multiple knowledge sources.** As mentioned above, another important design decision that was especially apparent in the speech understanding systems was the effective use of multiple sources of knowledge. Given that there are a number of possibly relevant sets of facts to be used by the parser (phonemic, lexical, syntactic, semantic, etc.), which do you use when?

The issues discussed here under parsing strategies are all questions of *efficiency*. They will not in general affect the final result if computational resources are unlimited, but they will affect the amount of resources expended to reach it.

### Actual Parsing Systems

Every natural language processing program deals with these seven issues in its own fashion. Several types of parsers have developed as experience with natural language systems increases.

**Template matching.** Most of the early NL programs (e.g., SIR, STUDENT, ELIZA) performed "parsing" by matching their input against a series of predefined *templates*--binding

the variables of the template to corresponding pieces of the input string (see Article F1). This approach was successful, up to a point--given a very limited topic of discussion, the form of many of the input sentences could be anticipated by the system's designer who incorporated appropriate templates. However, these systems were *ad hoc* and somewhat inextensible, and the template matching was soon abandoned in favor of more sophisticated methods.

**Simple phrase-structure grammar parsers.** These parsers make use of a type of *context-free grammar* with various combinations of the parsing techniques mentioned above. The advantage of a phrase-structure grammar is that the structures derived correspond directly to the grammar rules; thus, the subsequent semantic processing is simplified. By using large grammars and skirting linguistic issues that are outside their limitations (such as some types of agreement), a phrase-structure grammar parser can deal with a moderately large subset of English. Phrase-structure grammars are used primarily to produce systems (like SAD-SAM) with useful performance on a limited domain, rather than to explore more difficult language-processing issues.

**Transformational grammar parsers.** These parsers attempt to extend the notions of *transformational grammar* into a parsing system. Transformational grammar is a much more comprehensive system than phrase-structure grammar, but it loses phrase-structure's direct, rule-to-structure correspondence. Moreover, methods that have been tried, such as analysis by synthesis (building up all possible sentences until one matches the input) and inverse transformations (looking for transformation rules that might have produced the input), have often failed because of *combinatorial explosion*--the proliferation of alternatives the system must examine--and other difficulties with reversing transformations. One of the major attempts to implement a transformational parser was that by Petrick (1973).

**Extended grammar parsers.** One of the most successful AI approaches to parsing yet developed has been to extend the concept of phrase-structure rules and derivations by adding mechanisms for more complex representations and manipulations of sentences. Methods such as *augmented transition net grammars* (ATNs) and *charts* provide additional resources for the parser to draw on beyond the simple, phrase-structure approach (see Articles D2 and D3). Some of these mechanisms have validity with respect to some linguistic theory, while others are merely computationally expedient. The very successful systems of Woods & Kaplan (1971), Winograd (1972), and Kaplan (1973), as described in the articles in Section F, use extended grammar parsers.

**Semantic grammar parsers.** Another very successful modification to the traditional phrase structure grammar approach is to change the conception of grammatical classes from the traditional <NOUN>, <VERB>, etc., to classes that are motivated by concepts in the domain being discussed. For instance, such a *semantic grammar* for a system which talks about airline reservations might have grammatical classes like <DESTINATION>, <FLIGHT>, <FLIGHT-TIME>, and so on. The *rewrite rules* used by the parser would describe phrases and clauses in terms of these semantic categories (see Article Applications.C3 for a more complete discussion). The LIFER and SOHPHIE systems (Articles F7 and Applications.C3) use semantic grammar parsers (Hendrix, 1977a, and Burton, 1976).

**Grammarless parsers.** Some NL system designers have abandoned totally the traditional use of grammars for linguistic analysis. Such systems are sometimes referred to as "ad hoc," although they are typically based on some loose theory that happens to fall



outside the scope of standard linguistics. These "grammarless" parsers opt for flexibility in the above-mentioned precision/flexibility trade-off. They are based on special procedures (perhaps centered on individual words rather than syntactic elements) that use semantics-based techniques to build up structures relevant to meaning, and these structures bear little resemblance to the normal structures that result from syntactic parsing. A good example of this approach can be found in the work of Riesbeck (1975).

### Conclusion

Recent research in parsing has been directed primarily towards two kinds of simplification: providing simplified systems for dealing with less than full English, and providing simplified underlying mechanisms that bring the computer parsing techniques closer to being a theory of syntax. Systems such as LIFER (Article F7) have been developed which use the basic mechanisms of augmented grammars in a clean and easily programmable way. Systems like these cannot deal with the more difficult problems of syntax, but they can be used quickly and easily to assemble specialized parsers and are likely to be the basis for natural language "front ends" for simple applications.

At the same time, there has been a reevaluation of the fundamental notions of parsing and syntactic structure, viewed from the perspective of programs that understand natural language. Systems such as PARSIFAL (Marcus, 1978) attempt to capture in their design the same kinds of generalizations that linguists and psycholinguists posit as theories of language structure and language use. Emphasis is being directed toward the interaction between the structural facts about syntax and the control structures for implementing the parsing process. The current trend is away from simple methods of applying grammars (as with phrase-structure grammars), toward more "integrated" approaches. In particular, the grammar/strategy dualism mentioned earlier in this article has been progressively weakened by the work of Winograd (1972) and Riesbeck (1975). It appears that any successful attempt to parse natural language must be based upon some more powerful approach than traditional syntactic analysis. Also, parsers are being called upon to handle more "natural" text, including discourse, conversation, and sentence fragments. These involve aspects of language that cannot be easily described in the conventional, grammar-based models.

### References

Again, much of this discussion is borrowed from Winograd (forthcoming). For a review of recent work, the proceedings of the TINLAP conferences (TINLAP-1, 1975 and TINLAP-2, 1978) are recommended.

## D2. Augmented Transition Nets

Augmented transition networks (ATNs) were first developed by William Woods (1970) as a versatile representation of grammars for natural languages. The concept of an ATN evolved from that of a *finite-state transition diagram*, with the addition of tests and "side-effect" actions to each arc, as described below. These additions resulted in the power needed for handling features of English like *embedding* and *agreement* that could not be conveniently captured by regular (or even context-free) grammars. An ATN can thus be viewed as either a grammar formalism or a machine.

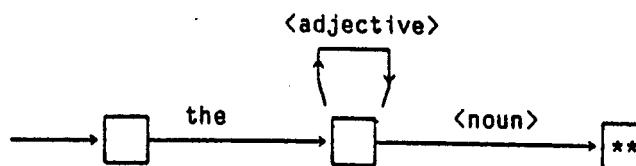
Many current language processors use an ATN-like grammar; in some ways, it may be considered state-of-the-art, at least for actual working systems.

### Preliminary Theoretical Concepts

A finite-state transition diagram (FSTD) is a simple theoretical device consisting of a set of states (nodes) with arcs leading from one state to another. One state is designated the START state. The arcs of the FSTD are labeled with the terminals of the grammar (i.e., words of the language), indicating which words must be found in the input to allow the specified transition. A subset of the states is identified as FINAL; the device is said to *accept* a sequence of words if, starting from the START state at the beginning of the sentence, it can reach a FINAL state at the end of the input.

FSTDs can "recognize" only regular or type-3 languages (see the discussion of *formal languages* in Article C1). To recognize a language, a machine must be able to tell whether an arbitrary sentence is part of the language or is not. Regular grammars (those whose rewrite rules are restricted to the form  $Y \rightarrow aX$  or  $Y \rightarrow a$ ) are the simplest, and FSTDs are only powerful enough to recognize these languages. In other words, it is impossible to build an FSTD that can dependably distinguish the sentences in even a context-free language.

For example, the following FSTD, in which the start state is the left-most node and the final state is labeled **\*\***, will accept any sentence that begins with the, ends with a noun, and has an arbitrary number of adjectives in between.

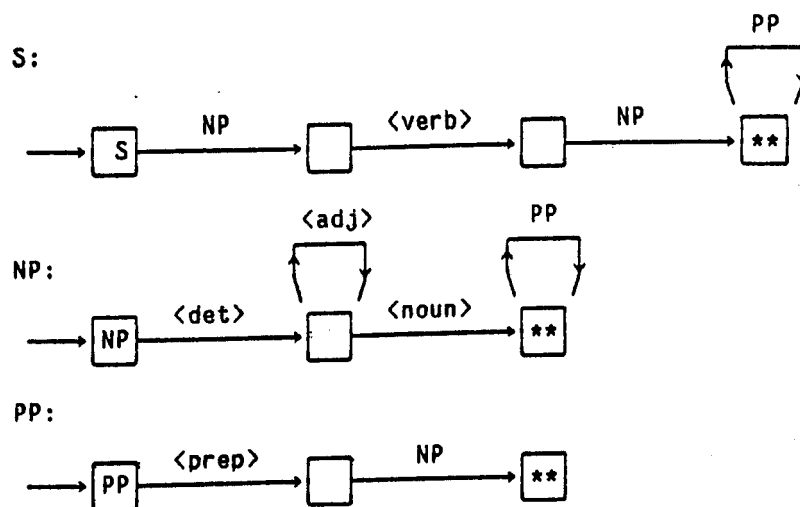


Let's follow through the net with the input sentence "the pretty picture." We start in the START state and proceed along the arc labeled the, because that is the left-most word in the input string. This leaves us in the middle box, with "pretty picture" left as our string to be parsed. After one loop around the adjective arc, we are again at middle node, but this time with the string "picture" remaining. Since this word is a noun, we proceed to the FINAL node, **\*\***, and arrive there with no words remaining to be processed. Thus the parse is successful; in other words, our example FSTD accepts this string.

However, regular grammars are inadequate for dealing with the complexity of natural language, as discussed in Article C2. A natural extension to FSTDs, then, is to provide a recursion mechanism that increases their recognition power to handle the more inclusive set of context-free languages. These extended FSTDs are called *recursive transition networks* (RTNs). An RTN is a finite-state transition diagram in which labels of an arc may include not only terminal symbols but also nonterminal symbols that denote the name of another subnetwork to be given temporary control of the parsing process.

An RTN operates similarly to an FSTD. If the label on an arc is a terminal (word or word class), the arc may be taken (as in FSTDs) if the word being scanned matches the label. For example, the word ball would match an arc labeled <noun> but not one labeled <adjective>. Otherwise, if the arc is labeled with a nonterminal symbol, representing a syntactic construct (e.g., PREPOSITIONAL PHRASE) that corresponds to the name of another network, the current state of the parse is put on a stack and control is transferred to the corresponding named subnetwork, which continues to process the sentence, returning control when it finishes or fails.

Whenever an accepting state is reached, control is transferred to the node obtained by "popping the stack" (i.e., returning to the point from which the subnetwork was entered). If an attempt is made to pop an empty stack, and if the last input word was the cause of this attempt, the input string is accepted by the RTN; otherwise, it is rejected. The effect of arcs labeled with names of syntactic constructs is that an arc is followed only if a construction of the corresponding type follows as a phrase in the input string. Consider the following example of an RTN:



Here NP denotes a noun phrase; PP, a prepositional phrase; det, a determiner; prep, a preposition; and adj, an adjective. Accepting nodes are labeled \*\*. If the input string is "The little boy in the swimsuit kicked the red ball," the above network would parse it into the following phrases:

NP: The little boy in the swimsuit  
 PP: in the swimsuit  
 NP: the swimsuit  
 Verb: kicked  
 NP: the red ball

Notice that any subnetwork of an RTN may call any other subnetwork, including itself; in the above example, for instance, the prepositional phrase contains a noun phrase. Also notice that an RTN may be nondeterministic in nature; that is, there may be more than one possible arc to be followed at a given point in a parse. Parsing algorithms handle nondeterminism by *parallel processing* of the various alternatives or by trying one and then *backtracking* if it fails. These general parsing issues are discussed in Article D1.

Context-free grammars, however, are still insufficient to handle natural language. The RTNs, then, must be extended, to provide even more parsing power.

### ATNs

An augmented transition network (ATN) is an RTN that has been extended in three ways:

1. A set of *registers* has been added; these can be used to store information, such as partially formed *derivation trees*, between jumps to different networks.
2. Arcs, aside from being labeled by word classes or syntactic constructs, can have arbitrary *tests* associated with them that must be satisfied before the arc is taken.
3. Certain *actions* may be "attached" to an arc, to be executed whenever it is taken (usually to modify the data structure returned).

This addition of registers, tests, and actions to the RTNs extends their power to that of Turing machines, thus making ATNs theoretically powerful enough to recognize any language that might be recognized by a computer. ATNs offer a degree of expressiveness and naturalness not found in the Turing machine formalism, and are a useful tool to apply to the analysis of natural language.

The operation of the ATN is similar to that of the RTN except that if an arc has a test then the test is performed first, and the arc is taken only if the test is successful. Also, if an arc has *actions* associated with it, then these operations are performed after following the arc. In this way, by permitting the parsing to be guided by the parse history (via tests on the registers) and by allowing for a rearrangement of the structure of the sentence during the parse (via the actions on the registers), ATNs are capable of building deep structure descriptions of a sentence in an efficient manner. For a well-developed and clear example, the reader is referred to Woods (1970).

### Evaluation of ATNs and Results

ATNs serve as a computationally implementable and efficient solution to some of the problems of recognizing and generating natural language. Their computational power provides the capability to embed different kinds of grammars, making them an effective testbed for new ideas. Two of the features of ATNs, the test and the actions on the arcs, make them especially well suited to handling *transformational grammars*. The ability to place arbitrary conditions on the arcs provides context sensitivity, equivalent to the preconditions for applying transformational rules. The capability to rearrange the parse structure, by copying, adding, and deleting components, provides the full power of transformations (see Article C2).

The ATN paradigm has been successfully applied to question answering in limited (closed) domains, such as the LUNAR program, which is described in Article F4. Also, ATNs have been used effectively in a number of *text generation* systems. In addition, the BBN *speech understanding* system, SPEECHLIS, uses an ATN control structure (see Article Speech.B3).

There are limitations to the ATN approach; in particular, the heavy dependence on syntax restricts the ability to handle ungrammatical (although meaningful) utterances. More recent systems (see especially Riesbeck's work, Article F3) are oriented toward meaning rather than structure and can thus accept mildly deviant input.

### References

The principal references here are, of course, Woods (1970), Woods & Kaplan (1971), and Woods (1973a). Also see Bobrow & Fraser (1969), Conway (1963), Matuzceck (1972), and Winograd (1975).

### D3. The General Syntactic Processor

Ronald Kaplan's (1973) General Syntactic Processor (GSP) is a versatile system for the parsing and generation of strings in natural language. Its data structures are intuitive and the control structures are conceptually straightforward and relatively easy to implement. Yet, by adjusting certain control parameters, GSP can directly emulate several other syntactic processors, including Woods's ATN grammar (Article D2), Kay's MIND parser (Kay, 1973), and Friedman's *text generation* system (Article E).

GSP represents an effort both to synthesize the formal characteristics of different parsing methods and to construct a unifying framework within which to compare them. In this respect, GSP is a "meta-system"--it is not in itself an approach to language processing, but rather it is a system in which various approaches can be described.

#### Data Structure: Charts

GSP gains much of its power through the use of a single, basic data structure, the *chart*, to represent both the grammar and the input sentence. A chart can be described as a modified *tree*, which is usually defined as a set of nodes that can be partitioned into a root and a set of disjoint subtrees. A tree encodes two sorts of relations between nodes: DOMINANCE, the relation between a parent and daughter node; and PRECEDENCE, the relation between a node and its right-hand sister node. Figure 1 shows a tree representing a particular noun phrase.

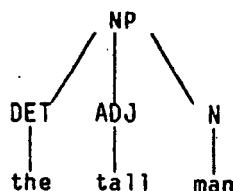


Figure 1. A tree for a noun phrase.

A chart is basically a tree that has been modified in two ways:

1. The arcs of the tree have been rearranged to produce a binary tree, that is, a tree in which each node has at most two dangling nodes (this rearrangement is described by Knuth [1973, p. 333] as the "natural correspondence" between trees and binary trees).
2. The nodes and arcs have been interchanged; what were previously nodes are now arcs, and vice versa.

For example, Figure 2 is the chart representation for the tree of Figure 1:

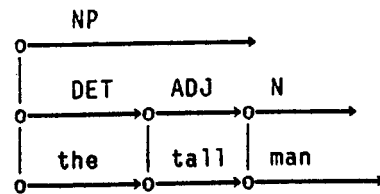


Figure 2. A chart for a noun phrase.

The chart representation has a number of advantages, including ease of access for certain purposes. For example, in Figure 1 there is no direct connection from DET to ADJ. In Figure 2 this connection has been made; that is, the PRECEDENCE relationships have been made explicit, and the DOMINANCE ones have been removed. This explicit encoding of precedence can be helpful in language processing, where the concept of one element following another is a basic relation.

Also, the chart can be used to represent a "string of trees" or "forest"--that is, a set of disjoint trees. For example, Figure 3a shows a string of two disjoint trees, headed by NP and V. Note that these trees cannot be connected, except with a dummy parent node (labeled ?). In Figure 3b, the equivalent chart representation is shown.

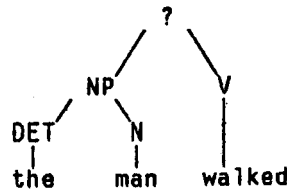


Figure 3a. Two disjoint trees.

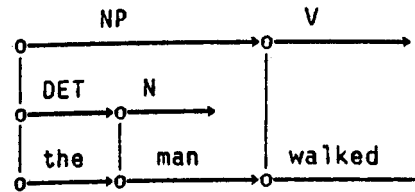


Figure 3b. The equivalent chart.

Finally, the chart provides a representation for multiple interpretations of a given word or phrase, through the use of multiple *edges*. The arcs in a chart are called edges and are labeled with the names of words or grammatical constructs. For example, Figure 4 represents the set of trees for "I saw the log," including the two interpretations for the word saw.

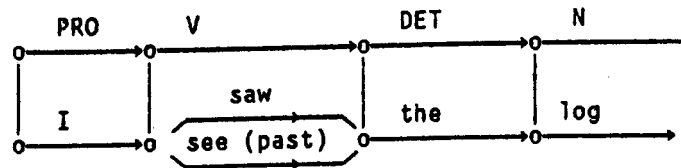


Figure 4. A chart showing multiple interpretations.

The chart allows explicit representation of ambiguous phrases and clauses, as well as of words.

Note that ambiguity could also be represented by distinct trees, one for every possible interpretation of the sentence. However, this approach is inefficient, as it ignores the possibility that certain subparts may have the same meaning in all cases. With the chart representation, these common subparts can be merged.

As defined earlier, the arcs in a chart are called edges and are labeled with the names of words or grammatical constructs. The nodes are called *vertexes*. The chart can be accessed through various functions, which enable one to retrieve specific edges, sets of edges, or vertexes.

At any given moment, the attention of the system is directed to a particular point in the chart called the CHART FOCUS. The focus is described by a set of global variables: EDGE (the current edge), VERTEX (the name of the node from which EDGE leaves), and CHART (the current subchart being considered by the processing strategy). GSP's attention is redirected by changing the values of these variables.

When the chart is initialized, each word in the sentence is represented by an edge in the chart for each category of speech the word can take. Figure 4 is an example of an initial chart configuration, preparatory to parsing. Each analysis procedure that shares the chart is restricted to adding edges, which gives later analyses the ability to modify or ignore earlier possibilities without constraining future interpretations. In this way, the individual syntactic programs remain relatively independent while building on each other's work in a generally *bottom-up* way.

It should be emphasized that the chart is just a data structure and is not directly related to the grammar. It merely serves as the global *blackboard* upon which the various pieces of the grammar operate. We still must specify the sorts of operations that use the chart--that is, the form of the grammar itself.

#### Data Structure: Grammatical Rules

Grammars for syntactic processing of language can be understood in terms of a network model like Woods's ATN grammar. That is, a grammar is viewed as a series of *states*, with transitions between the states accomplished by following *arcs* (see Article D2).

The grammars encoded by GSP fit this description. What gives GSP its power, however, is the fact that a grammar can be represented in the same way as a chart. That is, we can use the chart manipulation mechanisms, already developed, to operate upon the grammar itself. There is a difference, of course. The chart is merely a passive data store; the grammar contains instructions for: (a) acting on the chart--adding pieces and shifting attention; and (b) acting on the grammar--shifting attention (i.e., moving from one grammar state to another).

#### Control Structure

To handle the full complexity of grammars, GSP has some extra features. These include:



1. **REGISTERS.** As in ATNs, these are used as pointers to structures.
2. **LEVELSTACK.** This is a stack used to implement recursion. The chart focus, grammar focus (state), and register list are saved before a recursive call.
3. **NDLIST** (nondeterminism list). This is a list of choice points in the grammar. Whenever a choice is made, the user can optionally save the current configuration on NDLIST, to allow for backtracking.
4. **PROCSTACK.** This is a list of suspended processes. GSP allows a *co-routining* facility, under which processes can be suspended and resumed (ATNs have no equivalent to this).

In an ATN system, such features as recursion, *backtracking*, and movement of the pointer through the input sentence are implicit. With GSP, the user must handle all of these. This approach can be beneficial, particularly with features such as backtracking: automatic backtracking can be a less-than-desirable feature in a grammar (see the discussion in the AI Programming Languages Section).

### Using GSP

Note one facet of the approach outlined: All operations on the grammar and chart must be *explicitly* stated. Thus, GSP has placed much power in the hands of the grammar designer, with a corresponding cost in complexity.

GSP appears to be similar to an ATN, with three extensions:

1. The data structure used is a chart, instead of simply a string of words.
2. The grammar is encoded in the same manner as the chart; thus it is accessible to the system.
3. Processes can be suspended and resumed.

ATNs do not fully demonstrate the power of GSP. Kaplan also used GSP to implement Kay's MIND parser (a context-free, bottom-up system) and Friedman's transformational grammar text-generation system. The latter two made more extensive use of GSP's capabilities, in particular: (a) the possibilities of multiple levels in the chart; (b) the ability to suspend and restart processes; and (c) the ability to rearrange the chart, changing it as necessary. The Kay algorithm, in particular, made extensive use of the ability to modify the chart "on the fly," adding sections as required.

### Conclusions and Observations

GSP provides a simple framework within which many language processing systems can be described. It is not intended to be a high-level system that will do many things for the user; rather, it provides a "machine language" for the user to specify whatever operations

he wants. GSP's small set of primitive operations seems to be sufficient for representing most desirable features of syntax-based parsing. The clean, uniform structure enables GSP to be used as a tool for comparison (and possibly evaluation) of different systems.

The chart seems to be an effective data structure for representing the syntax of natural language sentences. It provides convenient merging of common subparts (i.e., to prevent re-scanning known components), while permitting representation of various forms of ambiguity. As Kay explained, the function of the chart is to "record hypotheses about the phraseological status of parts of the sentence so that they will be available for use in constructing hypotheses about larger parts at some later time" (Kay, 1973, p. 16').

The similarity of the chart and grammar network structures allows GSP to "bootstrap" compilers, thus simplifying compiler development and encouraging experimentation.

The backtracking mechanism is very general and thus can be inefficient if used too enthusiastically. Kaplan points out that heuristic ordering of alternatives is possible by altering the function that retrieves configurations from the NDLIST, though compilers should in any case attempt to minimize backtracking.

One flaw of GSP is its dependence on syntax; this approach, once popular in computational linguistics, is now recognized to be somewhat restrictive. GSP could conceivably be extended (or rather, used in different ways) to incorporate more emphasis on semantics and *pragmatics*.

### References

Kaplan (1973) is the principal reference. See also Friedman (1971), Kay (1973), Knuth (1973), and Woods (1970).

## E. Text Generation

Text generation is, in a sense, the opposite of natural language understanding by machine--it is the process of constructing text (i.e., phrases, sentences, paragraphs) in a natural language. Although this field has been pursued for fifteen years, few coherent principles have emerged, and the approaches have varied widely. Attempts at generating text have been made with two general research goals: (a) generating random sentences to test a grammar or grammatical theory and (b) converting information from an internal representation into a natural language.

### Random Generation

This approach, the random generation of text constrained by the rules of a test grammar, is of limited interest to workers in Artificial Intelligence, being oriented more toward theoretical linguistics than functional natural language processing systems. The objective of implementing a generation system of this sort is to test the descriptive adequacy of the test grammar, as illustrated by the following two systems.

Victor Yngve (1962) was one of the first researchers to attempt English text generation. He used a generative *context-free grammar* and a random-number generator to produce "grammatical" sentences: The system selected one production randomly from among those that were applicable at each point in the generation process, starting from those productions that "produced" <SENTENCE>, and finally randomly selecting words to fill in the <NOUN>, <VERB>, etc., positions. This is an example of the text produced by the system:

The water under the wheels in oiled whistles and its  
polished shiny big and big trains is black.

Joyce Friedman's (1969, 1971) system was designed to test out the effectiveness of *transformational grammars* (Article C2). It operated by generating *phrase-structure markers* (derivation trees that could contain syntactic information as well as words at the leaves) and by performing transformations on them until a *surface structure* (a derivation tree whose leaf nodes are all words) was generated. This use of transformations made her system more effective than Yngve's, since transformational grammars are more powerful than context-free grammars. The generation was random, but the user could specify an input phrase-structure marker and semantic restrictions between various terminals in order to test specific rules for "grammatical" validity.

These two systems, while relevant to work in linguistics, are only peripherally related to Artificial Intelligence. The fundamental emphasis in AI text-generation work has been on the meaning, as opposed to the syntactic form, of language.

### Surface Realization of Meaning

The goal of text generation systems in the AI paradigm is to take some internal representation of the "meaning" of a sentence and convert it to surface structure form, i.e. into the appropriate string of words. (Knowledge representation schemes like *predicate*

*calculus*, *semantic nets*, *conceptual dependency*, and PLANNER assertions are discussed in the Overview, and in more detail in the Knowledge Representation section of the Handbook.) The text-generation work of this sort has been heavily influenced by the type of internal knowledge representation used. That is, certain representation formalisms may conveniently express certain types of information, and a text-generation routine using the formalism will tend to make use of this fact. Text-generation modules are an important component of interactive AI systems of all types, since they are part of the interface between the internal structures and the user. In natural language systems that do question answering, paraphrasing, mechanical translation, information retrieval, etc., they play an especially crucial role.

Ross Quillian (1968) did pioneering work in the representation of knowledge and was also one of the first to deal with the problems of text generation. His system used a *semantic net* to represent the relations between words, which can be interpreted as their meaning. The task the system was then to perform was to compare two words, that is, find some semantic relation between them. For example:

Compare: Plant, Live

Answer: PLANT IS A LIVE STRUCTURE.

This relationship between the two words was discovered as a path in the net between the nodes that represented the words. Although this was a primitive semantic net scheme, many fundamental issues were first raised by Quillian's system, including several involving text generation.

Perhaps the most important point brought up by Quillian was that paths in the semantic net do not necessarily correspond to sentences in any simple way. There are many things that the system can "say" about a given path in its memory. And many of these, having not been input directly to the system, require *inferences* to be made. For example, another relationship the system found between plant and live was:

PLANT IS STRUCTURE WHICH GET-FOOD FROM AIR. FOOD IS THING WHICH BEING HAS-TO TAKE INTO ITSELF TO KEEP LIVE.

In order to have found this connection, the system has to discover connections between PLANT, AIR, FOOD, and LIVE that were not directly input.

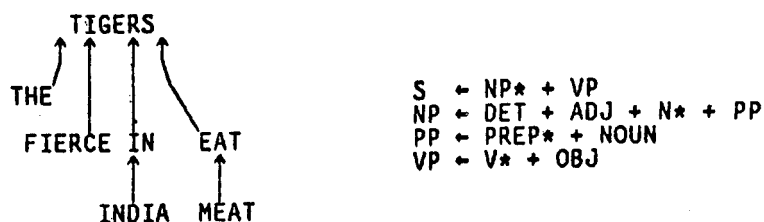
Although Quillian's semantic net system was limited, it strongly influenced much of the later work in NL and the representation of knowledge in AI (see Article Representation.C2). This influence reflected Quillian's stress on the importance of the semantic versus the surface components of language:

As a theory, the program implies that a person first has something to say, expressed somehow in his own conceptual terms (which is what a path is to the program), and that all his decisions about the syntactic form that a generated sentence is to take are then made *in the service* of this intention. (Quillian, 1968, p. 255)

This is a strong statement about language, and this view, of a cognitive process manipulating an internal representation, is perhaps the essence of the AI perspective.

Sheldon Klein (1965) was the first person to attempt to generate a *paraphrase* of a paragraph of text via an internal representation of that text. He used a knowledge representation scheme called *dependency grammar*, a context-free grammar with word-dependency information attached to each production. That is, the right-hand side of each rule in the grammar has a "distinguished symbol"; the "head" or governor of the phrase associated with that production is the head of the phrase that is associated with the distinguished symbol. All other words that are part of the phrase associated with the production are said to "depend" on this head.

For instance, given a simple dependency grammar and the sentence "the fierce tigers in India eat meat," the following dependency tree would be generated:



The symbols followed by \* are the distinguished symbols in the productions. Generation would then proceed from the set of dependency trees produced during the input analysis. The dependency trees from the individual sentences of the input paragraph were bound together with "two-way dependency" links between similar nouns.

The grammar used in generation was similar to the one used for analysis. Rule selection was random (as in Yngve's method) but with the added constraint that all dependencies generated must either be explicit in the initial dependency trees or derivable from them using a restricted transitivity. For example, in the above tree, EAT depends on TIGERS, and MEAT depends on EAT; using transitivity we can say that MEAT depends on TIGERS.

The actual paraphrase effect came from three features of the generation routine: (a) the random ordering of rule selection during generation; (b) the ability to combine input from different sentences into one output sentence; and (c) a simple synonym capability. Although this method attempted to recreate the sense of the original text, the device of dependency trees can encode only the crudest of semantic relationships present in the paragraph. In fact, the dependency relationship between words only indicates that some direct semantic relation exists between them without really specifying the nature of the relationship.

Terry Winograd's *blocks world* program, SHRDLU (1972), was able to answer questions concerning the state of the table-top domain and certain internal states of the program. The system, which is described in Article F5, had limited response-generating capabilities in order to make its output more fluent.

The basic text-generation techniques used were "fill-in-the-blank" and stored response patterns. For example, if an unfamiliar word was used, SHRDLU responded "I don't know the word ...." Similarly, a "why"-question was answered with "because <event>" or "in

order to <event>," where <event> is the relevant step of the program's manipulation of its world. The appropriate event was retrieved from the program's history list, and its description was generated as follows: To generate noun phrases, every known object had an English word associated with it; adjectives and relative clauses were added until a unique object (within the domain of discourse) was described. Events, such as "(PUTON OBJ1 OBJ2)," had generation patterns attached to them; in this case, the pattern would be: "(<correct form of to put>, <noun phrase for OBJ1>, ON, <noun phrase for OBJ2>)."

The stilted text generated by this scheme was moderated by the (heuristic) use of pronouns for noun phrases: Whenever the referent of a noun phrase had been mentioned in the same sentence or in the previous one, an appropriate pronoun was selected for it. SHRDLU's limited domain of discourse allowed it to exhibit surprisingly natural dialogue with such simple techniques.

Simmons and Slocum (1972) developed a natural language system that generated sentences from a *semantic network* representation of knowledge based on a semantic case structure. Input sentences were parsed by an *augmented transition network* using a *case grammar* (see Articles D2 and C4); the program then produced surface structure from the network, with possible transformations performed for *paraphrase* and *inference*, using a second ATN, not derivable from the first.

The semantic network included language-specific syntactic information such as number, determiner, agent, object, and instrument. For example, the sentence "John broke the window with a hammer" would be represented as:

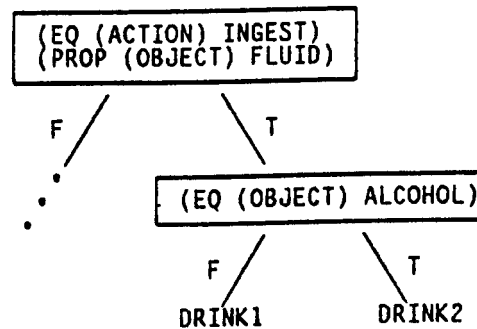
C1	TOKEN	BREAK	C3	TOKEN	WINDOW
	AGENT	C2		NUMBR	SING
	OBJ	C3		DET	DEF
	INST	C4			
	TIME	PAST			
C2	TOKEN	JOHN	C4	TOKEN	HAMMER
	NUMBR	SING		NUMBR	SING
	DET	DEF		DET	INDEF

C1, C2, C3, and C4 are nodes in the network (as are PAST, DEF, WINDOW, BREAK, JOHN, SING, INDEF, and HAMMER) and TOKEN, AGENT, OBJ, INST, etc., are types of arcs. In such nets, the verb node can be looked at as the "governor" since each verb has associated with it a set of case arguments that dictate which arcs may be present in the net. Each case argument is a case relation, such as AGENT or GOAL, and a value.

The actual generation was accomplished by "running the ATN backwards." The ATN proved to be an effective vehicle for generation. Uniformities in surface language were easily expressed, and the grammar was quite natural. Wong (1975) has extended this approach, incorporating features to handle extended discourse.

Neil Goldman's (1975) program generates surface structure from a database of *conceptual dependency* networks, as the text-generation part of Roger Schank's MARGIE system, described in Article F3. The conceptual dependency (CD) knowledge representation scheme, discussed further in Article F6 on Schank's SAM system, is based on *semantic primitives* (Article Representation.C5) and is therefore language independent, so the actual

word selection for output must be performed by Goldman's text-generation subsystem, called BABEL. This is accomplished by means of a *discrimination net* (a kind of binary decision tree, see Article Information Processing Psychology.C) that operates on a CD network that is to be verbalized. This discrimination net is used to select an appropriate *verb sense* to represent the event specified by the CD. (A verb sense is a meaning of the verb: DRINK, for example has two senses, to drink a fluid and to drink alcohol.) Essentially, there are only a small number of possible verbs that can represent the event, and a set of predicates determines which one to use. For instance, DRINK can be used to describe an INGEST event if the <object> has the property FLUID. The section of the discrimination net that handles DRINK might look like this:



Once a verb sense has been selected, an associated framework is used to generate a case-oriented syntax net, which is a structure similar to the semantic net of Simmons and Slocum. These frameworks include information concerning the form of the net and where in the conceptualization the necessary information is located. After the framework has been filled out, other language-specific functions operate on the syntax net to complete it syntactically with respect to such things as tense, form, mood, and voice. Finally, an ATN is used to generate the surface structure, as in the Simmons and Slocum program.

Yorick Wilks (1973) has developed a program that generates French from a semantic base of *templates* and *paraplates*. This is part of a complete *machine translation* system described in Article F2.

## Discussion

As the richness and completeness of the underlying semantic representation of the information has increased, the quality of the resulting paraphrase has improved. Like other areas of AI, the basic problem is to determine exactly what the salient points are and to obtain a good representation of them; progress in generation seems to be closely tied to progress in knowledge representation. Future work in generation will also have to address areas such as *extended discourse*, *stylistics*, etc. In this direction, Clippinger (1975) has looked at psychological mechanisms underlying discourse production, and Perrault, Allen, & Cohen (1978) have studied the planning of speech acts for communication in context.

**References**

See Clippinger (1975), Friedman (1969), Friedman (1971), Goldman (1975), Klein (1965), Perrault, Allen, & Cohen (1978), Quillian (1968), Quillian (1969), Simmons & Slocum (1972), Wilks (1973), Winograd (1972), Wong (1975), and Yngve (1962).



## F. Natural Language Processing Systems

### F1. Early Natural Language Systems

Early work on machine processing of natural language assumed that the syntactic information in the sentence, along with the meaning of a finite set of words, was sufficient to perform certain language tasks--in particular, answering questions posed in English. Several of these early natural language programs are reviewed here: their techniques, their successes, and their shortcomings. These programs were restricted to dialogues about limited-knowledge domains in simple English and ignored most of the hard grammatical problems in the complex constructions found in unrestricted English. Through work with programs of this genre, it became apparent that people constantly use extensive world-knowledge in processing language and that a computer could not hope to be competent without "understanding" language. These programs bridge the gap between the early *mechanical translation* attempts of the 1950s and current, semantics-based natural language systems (see the Overview Article, Article B, and the Articles on recent NL systems in this section).

#### SAD-SAM

SAD-SAM (Syntactic Appraiser & Diagrammer - Semantic Analyzing Machine) was programmed by Robert Lindsay (1963) at Carnegie Institute of Technology in the IPL-V list-processing language (see Article AI Languages.A). The program accepts English sentences about kinship relationships, builds a database, and answers questions about the facts it has stored.

It accepts a vocabulary of Basic English (about 1,700 words) and follows a simple *context-free grammar*. The SAD module parses the input from left to right, builds a syntactic tree structure, and passes this structure on to SAM, which extracts the semantically relevant (kinship-related) information to build the family trees and find answers to questions.

Though the subset of English processed by SAD is quite impressive in volume and complexity of structure, only kinship relations are considered by SAM; all other semantic information is ignored. SAM does not depend on the order of the input for building the family trees; if a first input assigns offspring B and C to X, and offspring D and E to Y, two "family units" will be constructed, but they will be collapsed into one if we learn later that E and C are siblings. (Multiple marriages are illegal.) However, SAM cannot handle certain ambiguities; the sentence "Joe plays in his Aunt Jane's yard" indicates that Jane is either the sister or sister-in-law of Joe's father, but SAM assigns one and only one connection at a time and therefore cannot use the ambiguous information: The structure of the model permits storing definite links but not possible *inferences*.

#### BASEBALL

Also in the early 1960s, Bert Green and his colleagues at Lincoln Labs wrote a program

called BASEBALL (Green, Wolf, Chomsky, & Laughery, 1963), again using the IPL-V programming language. BASEBALL is essentially an *information retrieval* program, since its database of facts about all of the American League games during one year is not modified by the program. Acceptable input questions from the user must have only one clause, no logical connectives (and, or, not), no comparatives (highest, most), and no facts about sequences of events; and most words must be recognized by the (extensive) dictionary.

The parsing system uses 14 categories of parts of speech and *right-to-left* scanning to structure the input question into functional phrases. Using this structure and the *key-words* found in the question, the input is transformed into a specification list that is the *canonical expression* for the meaning of the question. For example, the question "How many games did the Yankees play in July?" becomes:

```
TEAM = YANKEES
MONTH = JULY
GAMES (number of) = ?
```

The answer is found by searching the database for data items matching the specification list, storing them on a "found" list, and eventually processing and outputting them.

The size of the dictionary and the *heuristics* used for resolving syntactic ambiguities (score can be a verb or a noun; Boston, a place or the name of a team) make BASEBALL able to answer most reasonable English questions about the games stored in the database. However, BASEBALL does not return to the user for clarification of semantic ambiguities. It simply fails to parse the sentence and asks for a rephrasing of the question. Though the database is not modified by the program, BASEBALL exhibits deductive abilities in parsing questions and processing the "found-list."

## SIR

SIR (Semantic Information Retrieval) was written in LISP by Bertram Raphael (1964) as part of his thesis research at MIT. It is a prototype "understanding" machine since it can accumulate facts and then make deductions about them in order to answer questions.

SIR accepts a restricted subset of English. It matches sentences against 24 matching patterns of the following sort:

```
* is *
* is part of *
is * * ?
How many * does * have ?
What is the * of * ?
```

where the elements matched against \* are nouns, possibly modified by a "quantifier" such as: a, the, every, each, or a number. The match with a rule triggers the appropriate action in the program. A sample conversation might look like this:

```
Every boy is a person
I UNDERSTAND
```

A finger is part of a hand  
 THE ABOVE SENTENCE IS AMBIGUOUS PLEASE REPHRASE IT

There are two hands on each person  
 I UNDERSTAND

How many fingers does John have ?  
 THE ABOVE SENTENCE IS AMBIGUOUS BUT I ASSUME (has) MEANS  
 (has as parts)  
 I DON'T KNOW WHETHER FINGER IS PART OF JOHN

John is a boy  
 I UNDERSTAND

How many fingers does John have ?  
 THE ABOVE SENTENCE IS AMBIGUOUS BUT I ASSUME (has) MEANS  
 (has as parts)  
 HOW MANY FINGERS PER HAND ?

Every hand has five fingers  
 THE ABOVE SENTENCE IS AMBIGUOUS BUT I ASSUME (has) MEANS  
 (has as parts)  
 THE ANSWER IS 10

Who is President of the US ?  
 STATEMENT FORM NOT RECOGNIZED

SIR's world is concerned with only a small group of relations between objects, as specified by the 24 matching patterns. Declarative input statements are used to build the model, and the patterns have been chosen to express concepts that commonly occur in human conversation: set inclusion, spatial relationship, etc. The program has certain built-in knowledge (like transitivity of set inclusion) that enables it to answer questions about some facts not explicitly stated during input. SIR can also interact with the user to gather more information or to resolve ambiguities.

## STUDENT

STUDENT is another pattern-matching natural language program, written by Daniel Bobrow (1968) as his doctoral research project at MIT. STUDENT is able to read and solve high-school-level algebra story problems like the following:

If the number of customers Tom gets is twice the square of 20 percent of the number of advertisements he runs, and the number of advertisements he runs is 45, what is the number of customers Tom gets?

The entire subset of English recognized by STUDENT is derived from the following set of basic patterns:

(WHAT ARE * AND *)	(FIND * AND *)
(WHAT IS *)	(* IS MULTIPLIED BY *)
(HOW MANY *1 IS *)	(* IS DIVIDED BY *)
(HOW MANY * DO * HAVE)	(* IS *)
(HOW MANY * DOES * HAVE)	(* (*1/VERB) *1 *)
(FIND *)	
(* (*1/VERB) * AS MANY * AS * (*1/VERB) *)	

A \* sign indicates a string of words of any length, \*1 indicates one word, and (\*1/VERB) means the matching element must be recognized as a verb by the dictionary.

To construct the algebraic equations that will lead to the solution, the problem statement is scanned, first for linguistic forms associated with the equality relation (such as [\* IS \*]), then for algebraic operators. STUDENT then builds a list of the answers required, the units involved in the problem, and a list of all the variables in the equations. Then STUDENT invokes the SOLVE module with the set of equations and the desired unknowns.

If SOLVE fails, STUDENT applies *heuristics* such as: expanding idioms, identifying two previously "slightly different" variables, or invoking the REMEMBER module that contains special facts like:

(FEET IS THE PLURAL OF FOOT)  
 (ONE HALF ALWAYS MEANS .5)  
 (THE PERIMETER OF A RECTANGLE SOMETIMES MEANS TWICE  
 THE SUM OF THE LENGTH AND WIDTH OF THE RECTANGLE)  
 (DISTANCE EQUALS SPEED TIMES TIME)  
 (ONE FOOT EQUALS 12 INCHES)

If all else fails, STUDENT can request more information from the user.

Each time a heuristic succeeds in finding a new equation, the augmented set is sent to SOLVE. STUDENT finally prints the solution (using the pattern "variable IS value") or reports its inability to solve. In this example, STUDENT would print

(The number of customers Tom gets is 162) .

STUDENT's simple pattern-matching scheme, combined with its small set of well-chosen heuristics, does an impressive job of parsing typical high-school algebra word problems. The program can be tricked easily ("Bill has 3 apples, 2 pears, and one dog" would be parsed into two clauses); but as implemented, it solved the problems submitted as fast as humans could (in one case, several times faster than an MIT graduate student who was not using the special "age problem" heuristics). A slightly modified version was found to be a good model for *human problem solving*, in a study conducted by Paige & Simon (1966).

## ELIZA

Perhaps the most famous of these "pattern-matching" natural language programs was ELIZA, written by Joseph Weizenbaum (1966) at MIT. This program assumes the role of a Rogerian or "non-directive" therapist in its dialogue with the user. Although ELIZA's dialogue

with the user appears surprisingly realistic, the program uses no more intelligent mechanisms than those described above. Rather, the nature of therapist-patient dialogue lends itself to simulation in this manner. A sample dialogue might look like this (with ELIZA in caps):

Men are all alike.  
IN WHAT WAYS

They're always bugging us about something or other.  
CAN YOU THINK OF A SPECIFIC EXAMPLE

Well, my boyfriend made me come here.  
YOUR BOYFRIEND MADE YOU COME HERE

He says I'm depressed much of the time.  
I AM SORRY TO HEAR THAT YOU ARE DEPRESSED

It's true. I'm unhappy.  
DO YOU THINK COMING HERE WILL HELP YOU NOT TO BE UNHAPPY

I need some help, that much seems certain.  
WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP

Perhaps I could learn to get along with my mother.  
TELL ME MORE ABOUT YOUR FAMILY

...  
WHAT ELSE COMES TO MIND WHEN YOU THINK OF YOUR FATHER

Bullies.  
DOES THAT HAVE ANYTHING TO DO WITH THE FACT THAT YOUR  
BOYFRIEND MADE YOU COME HERE

...

ELIZA is programmed in the SLIP language (Symmetric List Processor), developed by Weizenbaum (1963). An offline *script* composed of *keylists* specifies the nature of the input language by specifying for each keyword a set of patterns to look for and the possible transformations associated with each pattern. For example, assuming the keyword is me, a pattern used for matching might be

(O YOU O ME)

and one of the transformation rules associated with this pattern might be

(WHAT MAKES YOU THINK I 3 YOU)

where O in the pattern matches any string of words, and 3 in the transformation rule designates the third element matched (in this case, everything between YOU and ME). The input is scanned from left to right for a keyword; a systematic *hashing* procedure very quickly eliminates words that cannot be keywords. ELIZA deletes the part of text preceding a punctuation delimiter if no keyword has been found; otherwise, the part following it (insuring thus that the transformation will be applied to one single phrase or sentence). If

several keywords are found, they are stored in turn in a "keystack" according to the rank of precedence associated with each of them--then the input is matched against each decomposition rule in turn. These patterns can be ordered in the keylist so that the more complex ones are tried first; for the keyword "I" the pattern

(O I O DEPRESSED O)

is hard to match, but if a match is achieved, the answer can be more spectacular than the transformations for the "general match" pattern

(O I O).

When a match is found, ELIZA generates a response, using the reassembly rules for this decomposition rule in a cyclic manner. If no decomposition rule matches for a given keyword, the keystack is popped and the pattern-matching procedure is repeated for the new keyword. If the keystack is empty, a response like "Please go on," "I see," or "Very interesting" will always do.

Several other tricks--like substituting for keywords in its response, associating keywords with a class or situation (Mother implies family), and remembering these keyword affiliates over the course of the conversation--help enhance the illusion of intelligent dialogue.

### Conclusions

None of these early natural language systems dealt with the syntax of language in any sophisticated way. In these early programs, the semantic knowledge needed to respond to the user was implicit in the patterns and the ad hoc rules used for parsing. Modern natural language programs maintain large databases of explicit world-knowledge that they use to assist in parsing the sentence as well as in interpreting it.

### References

For general reference, see Boden (1977), for lucid discussions of several of these systems; also, Simmons (1965), Simmons (1970), Wilks (1974), and Winograd (1974). The collections in Feigenbaum & Feldman (1963) and Minsky (1968) contain much of the original material.

## F2. Wilks's Mechanical Translation System

Current work in machine translation of languages is exemplified by Wilks's system (1973), which can produce good French from small English paragraphs. The system is entirely semantics based; that is, no use is made of conventional linguistic syntax in either the analysis or the generation stages. The input English text is first converted to a semantic representation and then converted to the final translated text. (The use of an intermediate representation bears some similarity to the Weaver's idea of *interlingua*, discussed in Article B.) Wilks stresses that his semantic representation is designed for mechanical translation and may not be appropriate for other NL tasks like question answering. The rationale for this is that an explicit representation of the logical implications of a sentence, which is necessary for some tasks, may not be necessary for translation: If the two languages are similar, an appropriate target sentence with the *same* implications can often be found in a more straightforward way.

Wilks's system first fragments the input text into substrings of words; it then matches the fragments against a set of standard *templates*, that is, deep semantic forms that try to pick out the meaning conveyed by the input-text fragments. The output of this stage is a first approximation to a semantic representation of each of these fragments. The system then tries to tie together these representations to produce a more densely connected representation for the complete text. When this process has been completed, the generation of the output text is accomplished by unwinding the interlingual representation using functions that interpret them in the target language.

The interlingual representation is based on *semantic primitives* (see Article Representation.C5) that Wilks calls *elements*. Elements express the entities, states, qualities, and actions about which humans communicate. In the system as reported in Wilks (1973), there were 60 of these elements, which fall into 5 classes, as shown in the following examples.

1. Entities:           MAN (human being),  
                          PART (parts of things),  
                          STUFF (substances).
2. Cases:               TO (direction),  
                          IN (containment).
3. Sorts:               CONT (being a container),  
                          THRU (being an aperture).
4. Type indicators:    KIND (being a quality),  
                          HOW (being a type of action).
5. Actions:            CAUSE (causes to happen),  
                          BE (exists),  
                          FLOW (moving as liquids do).

The elements are used to build up "formulas," which each represent one sense of a word. The verb drink, for example, is represented by the following formula:

((\*ANI SUBJ)  
 (((FLOW STUFF) OBJE)  
 ((\*ANI IN) (((THIS (\*ANI (THRU PART))) TO) (BE CAUSE)))))) .

Drink is thus an action [(BE CAUSE)], done by animate subjects [( \*ANI SUBJ)], to liquids [((FLOW STUFF) OBJE)]. It causes the liquid to be in the animate object [( \*ANI IN)] via a particular aperture of the animate object [((THIS (\*ANI (THRU PART))) TO)].

Formulas can also express preferences. The formula for the word big, for example, is

((\*PHYSOB POSS) (MUCH KIND)),

that is, a property preferably possessed by physical objects (*ds* opposed to substances). These preferences are used by the system to determine the correct word-senses in the input text.

The system's dictionary contains formulas for all the word-senses paired with *stereotypes* in the target language which produce the translated words. The following is an example of the stereotype for the word advise (into French):

(ADVISE (CONSEILLER A (FN1 FOLK MAN))  
 (CONSEILLER (FN2 ACT STATE STUFF)))

The two functions, FN1 and FN2, are used to distinguish the two possible constructions in French involving conseiller, *conseiller a . . .* and simply *conseiller . . .*. Functions like these in stereotypes are evaluated by the generation routines. Each function either evaluates to NIL, in which case it fails, or to a word that will appear in the output text. Hence the stereotypes also serve the purpose of a *text generation* grammar, providing complex context-sensitive rules where required, without search of a large store of such rules. This is an example of *procedural representation of knowledge* (see Article Representation.C4).

The first stage in analysis, then, involves replacement of the original words with their formula-stereotype pairs. This representation for the text is then separated into fragments, where the fragment boundaries are determined by punctuation marks, conjunctions, prepositions, and so on. The next stage involves matching these fragments against a built-in list of *templates*, networks of formulas based on a basic actor-action-object triple. Examples of bare templates are "MAN CAUSE THING," and "MAN DO THING." It is assumed that it is possible to build up a finite inventory of such bare templates that would be adequate for the analysis of ordinary language. The inventory for this system has been determined empirically and is easily modified.

At the next stage of the analysis, an attempt is made to expand the bare templates. This is done by noting semantic dependencies between the formulas in a fragment and selecting the structuring of formulas with the most dependencies. Notice is taken of the possible preferences expressed by the formulas at this stage. Thus the formulas specify not only word-senses but also procedures to be used in construction of complete templates. The preference of the overall system for *semantic density* is one of the key ideas in Wilks's work and produces a good solution to the problem of ambiguity.



In the final stage of analysis, the templates for individual fragments are tied together with higher level dependencies, expressed in terms of *paraplates*. Once again, the chief preference of the system is for semantic density. After this has been done, the system uses some commonsense inference rules to deal with situations in which more explicit world-knowledge is required. At the completion of this stage in the analysis, the input text has been replaced by an interlingual representation with suitable markers, and other information is used by the generation routines in a relatively straightforward manner to produce the final output text.

### Summary

Wilks's machine translation system is founded on three principles: (a) the use of templates for analysis, (b) the use of stereotypes for generation, and (c) the notion that the word-senses and much of the grammar will be properly interpreted by construction of a highly connected semantic representation for the input text. The fact that no explicit use is made of syntactic information reflects the current emphasis on understanding as a prerequisite for translation.

### References

This description of Wilks's work is based primarily on Wilks (1973). More recent descriptions include Wilks (1975), Wilks (1977a), Wilks (1977b), and Wilks (1978).

Also of interest: Charniak & Wilks (1976) and Schank (1975).

### F3. LUNAR

LUNAR is an experimental, natural language *information retrieval* system designed by William Woods at BBN (Woods, 1973b; Woods, Kaplan, & Nash-Webber, 1972) to allow geologists to access, compare, and evaluate chemical-analysis data on moon rock and soil composition obtained from the Apollo 11 mission (see Article Applications.F4 for a discussion of AI information retrieval systems). The primary goal of the designers was research on the problems involved in building a man-machine interface that would allow communication in ordinary English. A "real-world" application was chosen for two reasons: First, it tends to focus effort on the problems really in need of solution (sometimes this is implicitly avoided in "toy" problems); second, the possibility of producing a system capable of performing a worthwhile task lends some additional impetus to the work.

LUNAR operates by translating a question entered in English into an expression in a formal *query language* (Codd, 1974). The translation is done using an *augmented transition network* parser coupled with a rule-driven semantic interpretation procedure, which is used to guide the analysis of the question. The "query" that results from this analysis is then applied to the database to produce the answer to the request. The query language is a generalization of the *predicate calculus* (Article Representation.C1). Its central feature is a quantifier function that is able to express, in a simple manner, the restrictions placed on a database-retrieval request by the user. This function is used in concert with special enumeration functions for classes of database objects, freeing the quantifier function from explicit dependence on the structure of the database. LUNAR also served as a foundation for the early work done on *speech understanding* at BBN (see Article Speech.B3).

#### Detailed Description

The following list of requests is indicative of the types of English constructions that can be handled by LUNAR (shown as they would actually be presented to the system):

1. (WHAT IS THE AVERAGE CONCENTRATION OF ALUMINUM IN HIGH-ALKALI ROCKS?)
2. (WHAT SAMPLES CONTAIN P205?)
3. (GIVE ME THE MODAL ANALYSES OF P205 IN THOSE SAMPLES)
4. (GIVE ME EU DETERMINATIONS IN SAMPLES WHICH CONTAIN ILM)

LUNAR processes these requests in the following manner:

**Syntactic analysis** using an augmented transition network parser and *heuristic* information (including semantics) to produce the most likely *derivation tree* for the request;

**Semantic interpretation** to produce a representation of the meaning of the request in a formal query language; and

**Execution** of the query language expression on the database to produce the answer to the request.

LUNAR's language processor contains a grammar for a large subset of English, the

semantic rules for interpreting database requests, and a dictionary of approximately 3,500 words. As an indication of the capabilities of the processor, it is able to deal with tense and modality, some anaphoric references and comparatives, restrictive relative clauses, certain adjective modifiers (some of which alter the range of quantification or interpretation of a noun phrase), and embedded complement constructions. Some problems do arise in parsing conjunctive constructions and in resolving ambiguity in the scope of quantifiers. Emphasis has been placed on the types of English constructions actually used by geologists so that the system knows how they habitually refer to the objects in its database.

### The Query Language

The formal query language contains three types of objects: "designators," which name classes of objects in the database (including functionally defined objects); "propositions," which are formed from predicates with designators as arguments; and "commands," which initiate actions. Thus, if S10046 is a designator for a particular sample, OLIV is a designator for the mineral olivine, CONTAIN is a predicate, and TEST is a truth-value testing command, then "(TEST (CONTAIN S10046 OLIV))" is a sample expression in the query language. The primary function in the language is the quantifier function FOR, which is used in expressions of the following type:

( FOR <quant> X|CLASS : PX ; QX )

where <quant> is a quantifier like each or every, or a numerical or comparative quantifier; X is a variable of quantification; CLASS determines the class of objects over which the quantification is to range; PX specifies a restriction on the range; and QX is the proposition or command being quantified. FOR is used with enumeration functions that can access the database. Thus, FOR itself is independent of the database structure. As an example (taken from Woods, 1973b), if SEQ is an enumeration function used to enumerate a precomputed list, and if PRINTOUT is a command that prints a representation for the designator given as its argument, then

( FOR EVERY X1|(SEQ TYPECS) : T ; (PRINTOUT X1) )

prints the sample numbers for all type-C samples. In this case there is no restriction on the range of quantification in that  $PX = T$ , the universally true proposition.

A more complete example of the operation of LUNAR (simplified slightly from the same source) is shown below.

Request:

(DO ANY SAMPLES HAVE GREATER THAN 13 PERCENT ALUMINUM?)

Query Language Translation (after parsing):

((((TEST (FOR SOME X1/(SEQ SAMPLES) : T ; (CONTAIN X2  
(NPR\* X3/'AL203) (GREATER THAN 13 PCT))))))

Response:

YES

LUNAR is perhaps the best operational example of a finely tuned ATN parsing system applied to a real-world problem. Since the system has limited performance goals (i.e., facilitating database inquiry as opposed to holding an interesting conversation), many of the complications inherent in language understanding are avoided.

#### References

See Codd (1974), Woods (1973b), Woods & Kaplan (1971), and Woods, Kaplan, & Nash-Webber (1972).

## F4. SHRDLU

SHRDLU was written by Terry Winograd (1972) as his doctoral research at MIT. It was written in LISP and MICRO-PLANNER, a LISP-based programming language (see Article AI Languages.C2). The design of the system is based on the belief that to understand language, a program must deal in an integrated way with syntax, semantics, and reasoning. The basic viewpoint guiding its implementation is that meanings (of words, phrases, and sentences) can be embodied in procedural structures and that language is a way of activating appropriate procedures within the hearer. Thus, instead of representing knowledge about syntax and meaning as rules in a grammar or as patterns to be matched against the input, Winograd embodied the knowledge in SHRDLU in pieces of executable computer code. For example, the context-free rule saying that a sentence is composed of a noun phrase and a verb phrase,

$$S \rightarrow NP VP$$

is embodied in the MICRO-PLANNER procedure:

```
(PDEFINE SENTENCE
  (((PARSE NP) NIL FAIL)
   ((PARSE VP) FAIL FAIL RETURN)))) .
```

When called, this program, called SENTENCE, uses independent procedures for parsing a noun phrase followed by a verb phrase. These, in turn, can call other procedures. The process FAILs if the required constituents are not found. With these special *procedural representations* for syntactic, semantic, and reasoning knowledge, SHRDLU was able to achieve unprecedented performance levels in dialogues simulating a *blocks world* robot.

SHRDLU operates within a small "toy" domain so that it can have an extensive model of the structures and processes allowed in the domain. The program simulates the operation of a robot arm that manipulates toy blocks on a table. The system maintains an interactive dialogue with the user: It can accept statements and commands as well as answer questions about the state of its world and the reasons for its actions. The implemented system consists of four basic elements: a parser, a recognition grammar for English, programs for semantic analysis (to change a sentence into a sequence of commands to the robot or into a query of the database), and a general problem solver (which knows about how to accomplish tasks in Blocks World).

Each procedure can make any checks on the sentence being parsed, perform any actions, or call on other procedures that may be required to accomplish its goal. For example, the VERB PHRASE procedure called above contains calls to functions that establish verb-subject agreement by searching through the entire derivation tree for other constituents while still being in the middle of parsing the VP. SHRDLU's knowledge base includes a detailed model of the blocks world it manipulates, as well as a simple model of its own reasoning processes, so that it can explain its actions.

## Reasoning in SHRDLU

SHRDLU's model of the world and reasoning about it are done in the MICRO-PLANNER

programming language, which facilitates the representation of problem-solving procedures, allowing the user to specify his own heuristics and strategies for a particular domain. Knowledge about the state of the world is translated into MICRO-PLANNER assertions, and manipulative and reasoning knowledge is embodied in MICRO-PLANNER programs. For example, the input sentence "The pyramid is on the table" might be translated into an assertion of the form:

(ON PYRAMID TABLE)

SHRDLU's problem solver consists of a group of "theorems" about the robot's environment and actions, represented as MICRO-PLANNER procedures. The system's reasoning process tries to simulate that of a human understander. It is not based on a set of well-defined primitive symbols but tries to categorize its reality as humans do--with many redundant interconnections between fuzzily defined concepts. In operation, the theorem prover manipulates the state of the domain by running MICRO-PLANNER programs that perform the actions requested by the user.

The philosophy and implementation of PLANNER are described in the AI Programming Languages section of the Handbook, but a brief discussion here will illustrate its use in SHRDLU. The main idea of PLANNER is to solve problems using specific procedures built into the problem statements themselves, as well as using general problem-solving rules. The advantage of using these problem-specific rules or *heuristics* is that they can radically increase the efficiency of the process. Furthermore, the problem statements are programs and thus can carry out actions in the problem-solving process. Thus, to put one block on another, there might be a MICRO-PLANNER program of the form:

```
(THGOAL (ON ?X ?Y)
        (OR (ON-TOP ?X ?Y)
            (AND (CLEAR-TOP ?X)
                 (CLEAR-TOP ?Y)
                 (PUT-ON ?X ?Y))))))
```

which means if X is not already on Y, clear off everything that is stacked on top of X (so that the robot can move X), clear off Y (so that X can be placed on top of Y) and then put X on Y. This procedure resembles a *predicate calculus* theorem, but there are important differences. The PLANNER procedure is a program and its operators carry out actions. The THGOAL procedure finds an assertion in the database or proves it using other procedures. AND and OR are logical connectives. The crucial element is that though PLANNER may end up doing a proof, it does so only after checking some conditions that may make the proof trivial, or impossible, and it only performs the proof on relevant arguments, rather than checking all entities in the database as a blind theorem prover might. In addition to the article on PLANNER AI Languages.C2, the reader is referred to the Knowledge Representation section for a general discussion of these issues.

### Grammar, Syntax, and Semantics

SHRDLU's grammar is based on the notion of *systemic grammar*, a system of choice networks that specifies the features of a syntactic unit, how the unit functions, and how it influences other units, discussed in Article C3. Thus, a systemic grammar contains not only the constituent elements of a syntactic group but also higher level features such as mood, tense, and voice.

In order to facilitate the analysis, the parsing process looks for syntactic units that play a major role in meaning, and the semantic programs are organized into groups of procedures that are applicable to a certain type of syntactic unit. In addition, the database definitions contain *semantic markers* that can be used by the syntactic programs to rule out grammatical but semantically incorrect sentences such as "The table picks up blocks." These markers are calls to semantic procedures that check for such prohibited structures, such as that only animate objects pick up things. These semantic programs can also examine the context of discourse to clarify meanings, establish pronoun referents, and initiate other semantically guided parsing functions.

### Parsing

To write SHRDLU's parser, Winograd first wrote a programming language, embedded in LISP, which he called PROGRAMMAR. PROGRAMMAR supplies primitive functions for building systemically described syntactic structures. The theory behind PROGRAMMAR is that basic programming methods, such as procedures, iteration, and recursion, are also basic to the cognitive process. Thus, a grammar can be implemented in PROGRAMMAR without additional programming paraphernalia; special syntactic items (such as conjunctions) are dealt with through calls to special procedures. PROGRAMMAR operates basically in a top-down, left-to-right fashion but uses neither a *parallel processing* nor *backtracking* strategy in dealing with multiple alternatives (see Article □1). PROGRAMMAR finds one parsing rather directly, since decisions at choice-points are guided by the semantic procedures. By functionally integrating its knowledge of syntax and semantics, SHRDLU can avoid trying all choices in an ambiguous situation. If the choice made does fail, PROGRAMMAR has primitives for returning to the choice-point with the reasons for the failure and informing the parser of the next best choice based on these reasons. This "directed backup" is far different from PLANNER's automatic backtracking in that the design philosophy of the parser is oriented toward making an original correct choice rather than establishing exhaustive backtracking.

The key to the system's successful operation is the interaction of PLANNER reasoning procedures, semantic analysis, and PROGRAMMAR. All three of these elements examine the input and help direct the parsing process. By making use of this *multiple-source* knowledge and programmed-in "hints" (heuristics), SHRDLU successfully dealt with language issues such as pronouns and referents. The reader is referred to Winograd's *Understanding Natural Language* (1972), pages 8-15, for an illustrative sample dialogue with SHRDLU.

### Discussion

SHRDLU was a significant step forward in natural language processing research because of its attempts to combine models of human linguistic and reasoning methods in the language understanding process. Before SHRDLU, most AI language programs were linguistically simple; they used keyword and pattern-oriented grammars. Furthermore, even the more powerful grammar models used by linguists made little use of inference methods and semantic knowledge in the analysis of sentence structure. A union of these two techniques gives SHRDLU impressive results and makes it a more viable theoretical model of human language processing.

SHRDLU does have its problems, however. Like most existing natural language

systems, SHRDLU lacks the ability to handle many of the more complex features of English. Some of the problem areas are agreement, dealing with hypotheses, and handling words such as the and and.

Wilks (1974) has argued that SHRDLU's power does not come from linguistic analysis but from the use of problem-solving methods in a simple, logical, and closed domain (blocks world), thus eliminating the need to face some of the more difficult language issues. It seems doubtful that if SHRDLU were extended to a larger domain, it would be able to deal with these problems. Further, the level at which SHRDLU seeks to simulate the intermixing of knowledge sources typical of human reasoning is embedded in its processes rather than made explicit in its control structure, where it would be most powerful. Lastly, its problem solving is still highly oriented to predicate calculus and limited in its use of inferential and heuristic data (Winograd, 1974, pp. 46-48).

#### References

Winograd (1972) is, of course, the principal reference on SHRDLU. Boden (1977) presents a clear and concise discussion of the system. Also of interest are Sussman, Winograd, & Charniak (1970), the MICRO-PLANNER manual; and Wilks (1974), Wilks (1976a), Winograd (1971), Winograd (1974) and Winograd (forthcoming).

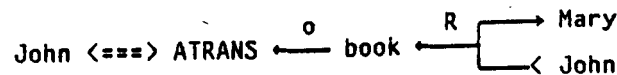


## F5. MARGIE

MARGIE (Meaning Analysis, Response Generation, and Inference on English) was a program developed by Roger Schank and his students at the Stanford AI Lab (Schank, 1975). Its intent was to provide an intuitive model of the process of natural language understanding. More recent work by Schank and his colleagues at Yale on *story understanding* and *conceptual dependency* theory are described in Article F6 on their SAM system.

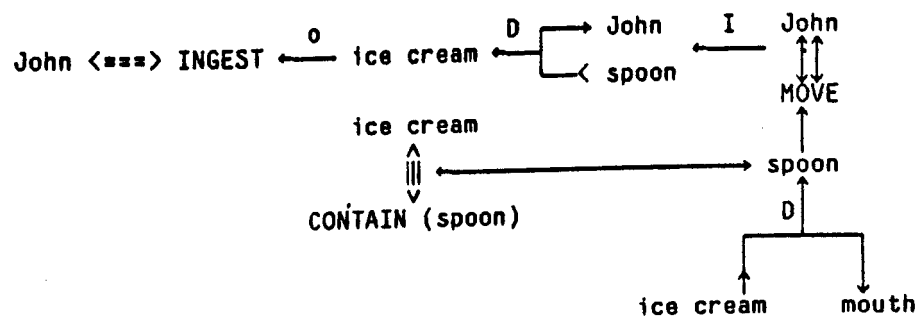
## Conceptual Dependency Theory

The central feature of the MARGIE system was the use of a knowledge representation scheme called Conceptual Dependency. Conceptual dependency is intended to represent meaning in a sufficiently deep manner so that all ambiguity is eliminated. Every sentence maps into a canonical form, and any two sentences with the same "meaning" will have the same representation. This goal was approached by designing a graph-structure formalism based on a set of *primitive* concepts. There are 6 basic types of concepts: things, actions, attributes of things, attributes of actions, times, and locations (the first four correspond roughly to nouns, verbs, adjectives, and adverbs). Relations among concepts are called *dependencies*, and there are 15 types of these. Among them are *case* relationships such as those between an act and its object, its direction, or its recipient and donor (see Article C4 on case grammars). Graphically, each type of dependency is denoted with a special arrow symbol (link), and each concept is denoted by a word representing it. For example, "John gives Mary a book" would be expressed as:



where John, book, and Mary are concept nodes. Also, the concept node ATRANS (abstract transfer--i.e., transfer of possession) is one of a small set of primitive verbs (about twelve) from which all actions must be built up. Other primitives include PTRANS (physical transfer--i.e., movement) and PROPEL (apply a force). The complicated, three-pointed arrow labeled R indicates a recipient-donor dependency between Mary and John, since Mary got the book from John. The arrow labeled o indicates an "objective" dependency; that is, the book is the object of the ATRANS, since it is the thing being given. Dependency links may link concepts or other conceptual dependency networks.

Another example, "John eats the ice cream with a spoon" would be represented as:



where the D and I arrows indicate DIRECTION and INSTRUMENT, respectively. Notice that in this example, "mouth" has entered the diagram as part of the conceptualization, even though it was not in the original sentence. This is part of the fundamental difference between conceptual dependency networks and the syntactic tree that a grammar may produce in parsing a sentence. John's mouth as the recipient of the ice cream is inherent in the "meaning" of the sentence, whether it is expressed or not. In fact, the diagram can never be finished, because we could add such details as "John *INGEST*ed the ice cream by *TRANS*ing the ice cream on a spoon to his mouth, by *TRANS*ing the spoon to the ice cream, by *GRASP*ing the spoon, by *MOV*ing his hand to the spoon, by *MOV*ing his hand muscles," and so on. Such an analysis is known to both the speaker and the hearer of the sentence and normally would not need to be expanded. (However, if we were actually designing a robot to perform such an action, we would want access to a more detailed network that would represent the robot's procedural knowledge about eating.)

For some tasks, like *paraphrasing* and *question answering*, this style of representation has a number of advantages over more surface-oriented systems. In particular, the fact that sentences like

Shakespeare wrote Hamlet  
and  
The author of Hamlet was Shakespeare ,

which in some sense have the same meaning, map into the same deep structure. They can thus be seen to be paraphrases of each other. Another important aspect of conceptual dependency theory is its independence from syntax; in contrast with earlier work in the *transformational grammar* paradigm, a "parse" of a sentence in conceptual dependency bears little relation to the syntactic structure. Schank (1975) also claims that conceptual dependency has a certain amount of psychological validity, in that it reflects intuitive notions of human cognition.

#### MARGIE

The MARGIE system, programmed in LISP 1.6, was divided into three components. The first, written by Chris Riesbeck, was a *conceptual analyzer*, which took English sentences and converted them into an internal conceptual dependency representation. This was done through a system of "requests," which were similar to *demons* or *production systems*. A request is essentially a piece of code that looks for some surface linguistic construct and takes a specific action if it is found. It consists of a "test condition," to be searched for in the input, and an "action," to be executed if the test is successful. The test might be as specific as a particular word or as general as an entire conceptualization. The action might contain information about: (a) what to look for next in the input, (b) what to do with the input just found, and (c) how to organize the representation. The flexibility of this formalism allows the system to function without depending heavily on syntax, although it is otherwise quite similar to the tests and actions that make *ATNs* such a powerful parsing mechanism.

The middle phase of the system, written by Chuck Rieger, was an inferencer designed to accept a proposition (stated in conceptual dependency) and deduce a large number of facts from the proposition in the current context of the system's memory. The motivation for this component was the assumption that humans "understand" far more from a sentence than

is actually stated. Sixteen types of inferences were identified, including "cause," "effect," "specification," and "function." The inference knowledge was represented in memory in a modified *semantic net*. Inferences were organized into "molecules," for the purpose of applying them. An example of this process might be:

John hit Mary.

from which the system might infer (among many other things):

John was angry with Mary.  
Mary might hit John back.  
Mary might get hurt.

The module does relatively unrestricted forward inferencing, which tended to produce large numbers of inferences for any given input.

The last part of the system was a *text generation* module written by Neil Goldman. This took an internal conceptual dependency representation and converted it into English-like output, in a two-part process:

1. A *discrimination net* was used to distinguish between different word-senses. This permitted the system to use English-specific contextual criteria for selecting words (especially verbs) to "name" conceptual patterns.
2. An ATN was used to linearize the conceptual dependency representation into a surface-like structure.

The text generation module is also discussed in Article E.

MARGIE ran in two modes: *inference mode* and *paraphrase mode*. In inference mode, it would accept a sentence and attempt to make inferences from that sentence, as described above. In paraphrase mode, it would attempt to restate the sentence in as many equivalent ways as possible. For example, given the input:

John killed Mary by choking her.

it might produce the paraphrases:

John strangled Mary.  
John choked Mary and she died because she was unable to breathe.

### Discussion

MARGIE is not, and was not intended to be, a "finished," production-level system. Rather, the goal was to provide a foundation for further work in computational linguistics. Of particular interest in MARGIE was the use of conceptual dependency as an *interlingua*, a language-independent representation scheme for encoding the meaning of sentences. Once the sentence was processed, the surface structure was dropped and all further work was done with the conceptual dependency notation. This method has certain beneficial effects

on the control structure: All interprocess communication can be done through conceptual dependency, without the need to resort to the surface level, although the more subtle information in the surface structure may be lost. Since the intermediate representation is "language-free," it should facilitate translation of the original sentence into another language, as Weaver indicated in his original discussion of *Machineese* (see Article B). As mentioned above, the existence of a unique representation for any fact should also facilitate tasks like paraphrasing and question answering.

### References

Conceptual dependency theory and all three parts of the MARGIE system are described well in Schank (1975). Since the early work described here, the theory has evolved considerably and several new systems have been built using the CD formalisms, all described very well in Schank & Abelson (1977). Other important references for MARGIE are Schank (1973) and Schank, Goldman, Rieger, & Riesbeck (1973).

## F6. SAM and PAM

### Story Understanding

SAM (Script Applier Mechanism) and PAM (Plan Applier Mechanism) are computer programs developed by Roger Schank, Robert Abelson and their students at Yale to demonstrate the use of *scripts* and *plans* in understanding simple stories (Schank et al., 1975, and Schank & Abelson, 1977). Most work in natural language understanding prior to 1973 involved parsing individual sentences in isolation; it was thought that text composed of paragraphs could be understood simply as collections of sentences. But just as words are not formed from the unconstrained juxtaposition of morphemes, and sentences are not unconstrained collections of words, so paragraphs and stories are not without structure. The structures of stories have been analyzed (Propp, 1968; Rumelhart, 1973; Thorndyke, 1977), and it is clear that the context provided by these structures facilitates sentence comprehension, just as the context provided by sentence structure facilitates word comprehension (see the Overview; also, the Speech.A article discusses *top-down processing* in speech understanding research). For example, if we have been told in a story that John is very poor, we can expect later sentences to deal with the consequences of John's poverty, or steps he takes to alleviate it.

Different researchers have very different ideas about what constitutes the structure of a story. Some *story grammars* are rather "syntactic," that is, they describe a story as a collection of parts, like setting, characters, goal introduction, plan, etc., determined by their sequential position in the story, rather than by their meaning. The work of Schank and Abelson reported here has a more semantic orientation. They propose an underlying representation of each phrase in a story which is based on a set of *semantic primitives*. This representation, called *conceptual dependency*, is the theoretical basis for more complex story structures such as *themes*, *goals*, *plans*, and *scripts*. The SAM and PAM programs understand stories using these higher level structures. (Article F3 describes the early work on conceptual dependency theory, and Articles Representation.C5 and Representation.C8 discuss related representation schemes.)

### Parsing: A Brief Introduction to Conceptual Dependency

Prior to his work with stories, Schank (1973) developed conceptual dependency (CD) for representing the meaning of phrases or sentences. The "basic axiom" of conceptual dependency theory is:

For any two sentences that are identical in meaning, regardless of language, there should be only one representation of that meaning in CD.

Schank thus allies himself with the early machine translation concept of *interlingua*, or intermediate language (see Articles B and Overview), and has in fact done some *mechanical translation* research in conjunction with the story understanding project. A second important idea is:

Any information in a sentence that is implicit must be made explicit in the representation of the meaning of the sentence.

This idea is the basis for much of the sophisticated inferential abilities of SAM and PAM: We shall see a sense in which the fact that "John ate food" is implicit in the sentence "John went to a restaurant," and how the former sentence can be inferred at the time that the program reads in the latter.

A third important idea is that conceptual dependency representations are made up of a very small number of semantic primitives, which include primitive acts and primitive states (with associated attribute values). Examples of *primitive acts* are:

ACTS:

- PTRANS The transfer of the physical location of an object. For one to "go" is to PTRANS oneself. "Putting" an object somewhere is to PTRANS it to that place.
- PROPEL The application of physical force to an object.
- ATRANS The transfer of an abstract relationship. To "give" is to ATRANS the relationship of "ownership."
- MTRANS The transfer of mental information between people or within a person. "Telling" is an MTRANS between people; "seeing" is an MTRANS within a person.
- MBUILD The construction of new information from old. "Imagining," "inferring," and "deciding" are MBUILDs.

In the most recent version of CD theory, Schank & Abelson (1977) included 11 of these primitive acts.

Examples of *primitive states* include:

STATES:

- |                          |                     |
|--------------------------|---------------------|
| Mary HEALTH(-10)         | Mary is Dead.       |
| John MENTAL STATE(+10)   | John is Ecstatic.   |
| Vase PHYSICAL STATE(-10) | The Vase is Broken. |

The number of primitive states in conceptual dependency theory is much larger than the number of primitive actions. States and actions can be combined; for example, the sentence:

John told Mary that Bill was happy

can be represented as

John MTRANS (Bill BE MENTAL-STATE(5)) to Mary.

An important class of sentences involves *causal chains*, and Schank and Abelson have worked out some rules about causality that apply to conceptual dependency theory. Five important rules are:

1. Actions can result in state changes.
2. States can enable actions.
3. States can disable actions.
4. States (or acts) can initiate mental events.
5. Mental events can be reasons for actions.

These are fundamental pieces of knowledge about the world, and conceptual dependency theory includes a shorthand representation of each (and combinations of some) called *causal links*.

Conceptual dependency representation is, in fact, the interlingua that is produced when SAM or PAM parse sentences. The parser which is used by these programs is an extension of the one developed by Chris Riesbeck (1975) for the MARGIE system (Article F3). As this program encounters words, it translates them into conceptual dependency representation; but, in addition, it makes predictions about what words and linguistic structures (verbs, prepositions, etc.) can be expected to occur and what conceptual dependency structures should be built in that eventuality.

Conceptual dependency is the underlying representation of the meaning of sentences upon which SAM and PAM operate. We turn now to higher level knowledge structures: scripts, plans, goals, and themes. Schank and Abelson make a distinction between scripts and plans that must be clear before the differences between SAM and PAM become apparent.

### Scripts

A script is a standardized sequence of events that describes some stereotypical human activity, such as going to a restaurant. Schank and Abelson's assumption is that people know many such scripts and use them to establish the context of events. A script is functionally similar to a *frame* (Minsky, 1975) or a *schema* (Bartlett, 1932, and Rumelhart, 1973), in the sense that it can be used to anticipate the events it represents. For example, the RESTAURANT script (see Figure 1) involves going to the restaurant, being seated, consulting the menu, and so on. People who are presented with an abbreviated description of this activity, e.g., the sentence "John went out to dinner," infer from their own knowledge about restaurants that John ordered, ate, and paid for food. Moreover, they anticipate from a sentence which fills part of the script ("John was given a menu") what sort of sentences are likely to follow, e.g., "John ordered the lamb." Scripts attempt to capture the kind of knowledge that people use to make these inferences. (Article Representation.C6 discusses scripts, frames and related representation schemes.)

-----  
 Players: customer, server, cashier

Props: restaurant, table, menu, food, check, payment, tip

Events:

1. customer goes to restaurant
2. customer goes to table
3. server brings menu
4. customer orders food
5. server brings food
6. customer eats food
7. server brings check
8. customer leaves tip for server
9. customer gives payment to cashier
10. customer leaves restaurant

Header: event 1

Main concept: event 6  
 -----

Figure 1. Restaurant Script

Two components of scripts are of special importance. We will discuss later how the *script header* is used by SAM to match scripts to parsed sentences. The second important component is the *main concept* or *goal* of the script. In the restaurant script the goal is to eat food.

The scripts used in SAM grew out of Abelson's (1973) notion of scripts as networks of causal connections. However, they do not depend on explicit causal connections between their events. In hearing or observing events which fit a standard script, it is not necessary to analyze the sequence of events in terms of causes, since they can be expected just from knowing that the script applies. The identification of events as filling their slot in the script gives us the intuition of "understanding what happened."

Scripts describe everyday events, but frequently these events (or our relating of them) do not run to completion. For example:

I went to the restaurant. I had a hamburger.  
 Then I bought some groceries.

This story presents several problems for a system like SAM that matches scripts to input sentences. One problem is that the restaurant script is "left dangling" by the introduction of the last sentence. It is not clear to the system whether the restaurant script

- 1) has terminated, and a new (grocery shopping) script has started;
- 2) has been distracted by a "fleeting" (one-sentence) grocery script; or



3) is interacting with a new grocery script (e.g., buying groceries in the restaurant).

Another thing that can happen to everyday scripts is that they can be thwarted, as in:

I went to the gas station to fill up my car.  
But the owner said he was out of gas.

This is called an "Obstacle".

Scripts describe rather specific events, and although it is assumed that adults know thousands of them, story comprehension cannot simply be a matter of finding a script to match a story. There are just too many possible stories. Moreover, there are clear cases where people comprehend a story even though it does not give enough information to cause a program to invoke a script, as in

John needed money. He got a gun and went to a liquor store.

Schank and Abelson point out that even if the program had a script for Robbery, this story offers no basis for invoking it. Nonetheless, people understand John's goals and his intended actions.

There must be relevant knowledge available to tie together sentences that otherwise have no obvious connection. . . . The problem is that there are a great many stories where the connection cannot be made by the techniques of causal chaining nor by reference to a script. Yet they are obviously connectable. Their connectability comes from these stories' implicit reference to plans. (Schank & Abelson, 1977, p. 75)

### Plans

Schank and Abelson introduce *plans* as the means by which goals are accomplished, and they say that understanding *plan-based* stories involves discerning the goals of the actor and the methods by which the actor chooses to fulfill those goals. The distinction between script-based and plan-based stories is very simple: In a script-based story, parts or all of the story correspond to one or more scripts available to the story understander; in a plan-based story, the understander must discern the goals of the main actor and the actions that accomplish those goals. An understander might process the same story by matching it with a script or scripts, or by figuring out the plans that are represented in the story. The difference is that the first method is very specialized, because a script refers to a specific sequence of actions, while plans can be very general because the goals they accomplish are general. For example, in

John wanted to go to a movie. He walked to the bus-stop.

We understand that John's *immediate* goal (called a *delta-goal* because it brings about a change necessary for accomplishment of the ultimate goal) is to get to the movie theater. *Going somewhere* is a very general goal, and does not apply just to going to the movies. In Schank and Abelson's theory, this goal has associated with it a set of planboxes, which are

standard ways of accomplishing the goal. Planboxes for *going somewhere* include riding an animal, taking public transportation, driving a car, etc.

Obviously, a story understander might have a "go to the movies" script in its repertoire, so that analysis of John's goals would be unnecessary--the system would just "recognize" the situation and retrieve the script. This script would be the standardized intersection of a number of more or less general goals and their associated planboxes. It would be a "routinized plan" made up of a set of general subplans: Go to somewhere (the theater), Purchase something (a ticket), Purchase something (some popcorn), etc.

A routinized plan can become a script, at least from the planner's personal point of view.

Thus, plans are where scripts come from. They compete for the same role in the understanding process, namely as explanations of sequences of actions that are intended to achieve a goal. (Schank & Abelson, 1977, p. 72)

The process of understanding plan-based stories involves determining the actor's goal, establishing the subgoals (delta- or D-goals) that will lead to the main goal, and matching the actor's actions with planboxes associated with the D-goals. For example, in

John was very thirsty. He hunted for a glass.

we recognize the D-goal of PTRANSing liquid, and the lower level goal (specified in the planbox for PTRANSing liquid) of finding a container to do it with.

### Goals and Themes

In story comprehension, goals and subgoals may arise from a number of sources. For example, they may be stated explicitly, as in

John wanted to eat ,

they may be nested in a planbox, or they may arise from *themes*. For example, if a LOVE theme holds between John and Mary, it is reasonable to expect the implicit, mutual goal of protecting each other from harm: "Themes, in other words, contain the background information upon which we base our predictions that an individual will have a certain goal" (Schank & Abelson, 1977, p. 132).

Themes are rather like production systems in their situation-action nature. A theme specifies a set of actors, the situations they may be in, and the actions that will resolve the situation in a way consistent with the theme. The goals of a theme are to accomplish these actions. Schank and Abelson have proposed seven types of goals; we have already considered D-goals. Other examples are:

- A- or Achievement-goals. To desire wealth is to have an A-Money goal.
- P- or Preservation-goal. To protect someone may be a P-Health or P-Mental State goal.
- C- or Crisis-goal. A special case of P-goals, when action is immediately necessary.

The LOVE theme can be stated in terms of some of these goals:

X is the lover; Y is the loved one; Z is another person.

SITUATION	ACTION
Z cause Y harm	A-Health(Y) and possibly cause Z harm
	or C-Health(Y)
not-Love(Y,X)	A-Love(Y,X)
General goals:	A-Respect(Y) A-Marry(Y) A-Approval(Y)

To summarize the knowledge-structures we have discussed, we note their interrelationships:

Themes give rise to goals.

A plan is understood when its goals are identified and its actions are consistent with the accomplishment of those goals.

Scripts are standardized models of events.

Scripts are specific; plans are general.

Plans originate from scripts.

Plans are ways of representing a person's goals. These goals are implicit in scripts, which represent only the actions.

A script has a header, which is pattern-matched to an input sentence. Plans do not have headers, but each plan is subsumed under a goal.

## SAM

Both SAM and PAM accept stories as input; both use an English-to-CD parser to produce an internal representation of the story (in conceptual dependency). Both are able to paraphrase the story and to make intelligent inferences from it. They differ with respect to the processing that goes on after the CD representation has been built.

SAM understands stories by fitting them into one or more scripts. After this match is completed, it makes summaries of the stories. The process of fitting a story into a script has three parts, a PARSER, a memory module (MEMTOK), and the script applier (APPLY). These modules cooperate: The parser generates a CD representation of each sentence, but APPLY gives it a set of Verb-senses to use once a script has been identified. For example, once the restaurant script has been established, APPLY tells the parser that the appropriate sense of the verb "to serve" is "to serve food" rather than, for example, "to serve in the army."

The parser does not make many inferences; thus it does not realize that "it" refers to the hot dog in "The hot dog was burned. It tasted awful." This task is left to MEMTOK. This module takes references to people, places, things, etc., and fills in information about them. It recognizes that the "it" in the sentence above refers to the hot dog, and "instantiates" the "it" node in the CD representation of the second sentence with the "hot dog" node from the first sentence. Similarly, in a story about John, MEMTOK would replace "he" with "John" where appropriate, and would continually update the "John" node as more information became available about him.

The APPLY module has three functions. First, it takes a sentence from the parser and checks whether it matches the current script, a concurrent (interacting) script, or *any* script in the database. If this matching is successful, it makes a set of predictions about likely inputs to follow. Its third task is to instantiate any steps in the current script that were "skipped over" in the story. For example, if the first sentence of a story is "John went to a restaurant," APPLY finds a match with the *script header* of the restaurant script in its database (see Figure 1). APPLY then sets up predictions for seeing the other *events* in the restaurant script in the input. If the next sentence is "John had a hamburger," then APPLY successfully matches this sentence into the restaurant script (event 6). It then assumes events 2-5 happened, and instantiates structures in its CD representation of the story to this effect. Events 7-10 remain as predictions.

When the whole story has been mapped into a CD representation in this manner, the SAM program can produce a summary of the story, or answer questions about it. (See Schank & Abelson, 1977, pp. 190-204, for an annotated sample protocol with the program.) Consistent with the idea of Interlingua, SAM can produce summaries in English, Chinese, Russian, Dutch, and Spanish. An example of a SAM paraphrase follows; note the powerful inferences made by instantiating intermediate script steps:

ORIGINAL:           John went to a restaurant. He sat down. He got mad.  
                          He left.

PARAPHRASE:        JOHN WAS HUNGRY. HE DECIDED TO GO TO A RESTAURANT.  
                          HE WENT TO ONE. HE SAT DOWN IN A CHAIR. A WAITER  
                          DID NOT GO TO THE TABLE. JOHN BECAME UPSET. HE  
                          DECIDED HE WAS GOING TO LEAVE THE RESTAURANT. HE  
                          LEFT IT.

SAM inferred that John left the restaurant because he did not get any service. The basis for this inference is that in the restaurant script, event 3 represents the waiter coming over to the table after the main actor has been seated. SAM knows that people can get mad if their expectations are not fulfilled, and infers that John's anger results from the nonoccurrence of event 3.

## PAM

Wilensky's (1978) PAM system understands stories by determining the goals that are to be achieved in the story and attempting to match the actions of the story with the methods that it knows will achieve the goals. More formally:

The process of understanding plan-based stories is as follows:

- a) Determine the goal,
- b) Determine the D-goals that will satisfy that goal,
- c) Analyze input conceptualizations for their potential realization of one of the planboxes that are called by one of the determined D-goals.

(Schank & Abelson, 1977, p. 75)

PAM utilizes two kinds of knowledge structure in understanding goals: *named plans* and *themes*. A named plan is a set of actions and subgoals for accomplishing a main goal. It is not very different from a script, although the emphasis in named plans is on goals and the means to accomplish them. For example, a script for rescuing a person from a dragon would involve riding to the dragon's lair and slaying it--a sequence of actions--but a named plan would be a list of subgoals (find some way of getting to the lair, find some way of killing the dragon, etc.) and their associated planboxes. When PAM encounters a goal in a story for which it has a named plan, it can make predictions about the D-goals and the actions that will follow. It will look for these D-goals and actions in subsequent inputs. Finding them is equivalent to understanding the story.

Themes provide another source of goals for PAM. Consider the sentences:

- a) John wanted to rescue Mary from the dragon.
- b) John loves Mary. Mary was stolen away by a dragon.

In both of these cases, PAM will expect John to take actions that are consistent with the goal of rescuing Mary from the dragon, even though this goal was not explicitly mentioned in (b). The source of this goal in (b) is the LOVE theme mentioned above, because in this theme, if another actor tries to cause harm to a loved one, the main actor sets up the goal of Achieving-Health of the loved one, and possibly harming the evil party. (It is assumed that the dragon stole Mary in order to hurt her.)

PAM determines the goals of an actor by

1. their explicit mention in the text of the story,
2. establishing them as D-goals for some known goal, or
3. inferring them from a theme mentioned in the story.

To understand a story is to "keep track of the goals of each of the characters in a story and

to interpret their actions as means of achieving those goals" (Schank & Abelson, 1977, p. 217). The program begins with written English text, converts it into CD representation, and then interprets each sentence in terms of goals (predicting D-goals and actions to accomplish them) or actions themselves (marking the D-goals as "accomplished"). When this process is completed, PAM can summarize the story and answer questions about the goals and actions of the characters.

### Summary

Scripts, plans, goals, and themes are knowledge structures built upon conceptual dependency theory. SAM is a program for understanding script-based stories. It matches the input sentences of a story to events in one or more of the scripts in its database. As such, it is relatively *top-down* in its operation--it processes input based on *expectations* it has built up from the scripts. PAM understands plan-based stories by determining the goals of the characters of the story and by interpreting subsequent actions in terms of those goals or subgoals that will achieve them. A great deal of inference can be required of PAM simply to establish the goals and subgoals of the story from the input text.

Schank and Abelson argue that human story understanding is a mixture of applying known scripts and inferring goals (where no script is available or of obvious applicability). They are experimenting with interactions of SAM and PAM, in particular, with using SAM to handle script-based sub-stories under the control of PAM.

### References

The recent book by Schank & Abelson (1977) is the most complete and readable source on both of these systems and on the current state of Conceptual Dependency theory. For the whole truth about PAM, see the Ph.D. dissertation by Wilensky (1978a).

Also of interest: Abelson (1973), Bartlett (1932), Minsky (1975), Propp (1968), Riesbeck (1975), Rumelhart (1973), Schank (1973), Schank et al. (1975), Thorndyke (1977)..skipc

### F7. LIFER

The natural language systems described in the preceding articles fall into two categories: those built to study natural language processing issues in general and those built with a particular task domain in mind. In contrast, LIFER, built by Gary Hendrix (1977a) as part of the internal research and development program of SRI International, Inc., is designed to be an "off-the-shelf" natural language utility available to systems builders who want to incorporate an NL front-end interface to improve the usability of their various applications systems. The bare LIFER system is a system for generating natural language interfaces; the interface builder can augment LIFER to fit his particular application, and even the eventual users can tailor the LIFER-supported front-end to meet their individual styles and needs.

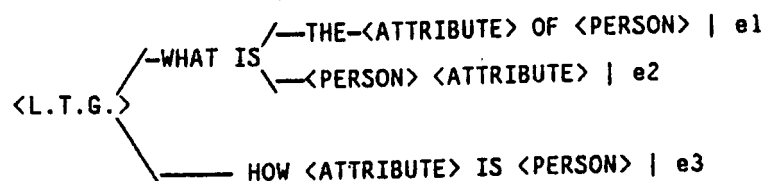
### Language Specification and Parsing

The LIFER system has two major components: a set of interactive language specification functions and a parser. Initially it contains neither a grammar, nor the semantics of any language domain. An interface builder uses the language specification functions to define an *application language*, a subset of English that is appropriate for interacting with his system. The LIFER system then uses this language specification to *interpret* natural language inputs as commands for the application system.

The interface builder specifies the language primarily in terms of grammatical *rewrite rules* (see Article C1). LIFER automatically translates these into transition trees, a simplified form of augmented transition networks (Article D2). Using the transition tree, the parser interprets inputs in the application language. The result is an interpretation in terms of the appropriate routines from the applications system, as specified by the interface builder. The parser attempts to parse an input string top-down and left to right (see Article D1) by nondeterministically tracing down the transition tree whose root node is the start symbol (known as <L.T.G.> for "LIFER top grammar"). For example suppose the interface builder has specified the following three production rules as part of his application language:

```
<L.T.G.> -> WHAT IS THE <ATTRIBUTE> OF <PERSON> | e1
<L.T.G.> -> WHAT IS <PERSON'S> <ATTRIBUTE> | e2
<L.T.G.> -> HOW <ATTRIBUTE> IS <PERSON> | e3
```

If an input matches one of these patterns, the corresponding expression (e1, e2, or e3) is evaluated--these are the appropriate *interpretations* that the system is to make for the corresponding input. The transition tree built by the language specification functions would look like this:



Sentences such as:

```

What is the age of Mary's sister?
How old is Mary's sister?
What is John's height?
How tall is John?

```

might be parsed using this simple transition tree, depending on how the *nonterminal* symbols, or meta-symbols, <ATTRIBUTE> and <PERSON> are defined.

During parsing, LIFER starts at the symbol <L.T.G.> and attempts to move towards the expressions to be evaluated at the right. The parser follows a branch only if some portion at the left of the remaining input string can be matched to the first symbol on the branch. Actual words (such as what or of in the above example) can be matched only by themselves. Meta-symbols (such as <ATTRIBUTE> or <PERSON>) can be matched in a number of ways, depending on how the interface builder has defined them:

- (a) as a simple set (e.g., <PERSON> = the set {Mary, John, Bill});
- (b) as a predicate that which is applied to the string to test for satisfaction (for example, some meta-symbol used in a piece of grammar to recognize dates might test if the next string of characters is a string of digits, and thus a number); or
- (c) by another transition tree which has this meta-symbol as its root node.

The above example is typical: A large amount of semantic information is embedded in the syntactic description of the application language. JOHN and HEIGHT are not defined as instances of the single meta-symbol <NOUN> as they would be in a more formal grammar, but rather are separated into the semantic categories indicated by the meta-symbols <PERSON> and <ATTRIBUTE>. The technique of embedding such semantic information in the syntax has been referred to as *semantic grammar* (Burton, 1976), and it greatly increases the performance of LIFER's automatic spelling correction, ellipsis, and paraphrase facilities, described below.

### Applications

LIFER has been used to build a number of natural language interfaces, including a medical database, a task scheduling and resource allocation system, and a computer-based expert system. The most complex system built with a LIFER interface involved a few man-months of development of the natural language front-end: The LADDER system (Language Access to Distributed Data with Error Recovery) developed at SRI, which provides real-time natural language access to a very large database spread over many smaller databases in computers scattered throughout the United States (Sacerdoti, 1977, and Hendrix, et al., 1978). Users of the system need have no knowledge of how the data is organized nor where it is stored. More importantly, from the point of view of this article, users do not need to know a data query language: They use English, or rather a subset that is "natural" for the domain of discourse and which is usually understood by the LIFER front-end. The interpretations of the inputs by LIFER are translations into a general database query language, which the rest of the LADDER system converts to a query of the appropriate databases on the appropriate computers (See Article Applications.F4 on AI in Information Retrieval systems).

Another interesting system to use a LIFER front-end was the HAWKEYE system (Barrow, et al., 1977), also developed at SRI. This is an integrated interactive system for cartography or intelligence, which combines aerial photographs and generic descriptions of objects and situations with the topographical and cultural information found in traditional maps. The user queries this database and cues image-processing tasks via a LIFER natural language interface. A unique feature of this interface is the combination of natural language and external referents. For instance, using a cursor to point to places within an image, the user can ask questions such as "What is this?" and "What is the distance between here and here?" The interpretation of such expressions results in requests for coordinates from the subsystem providing graphical input, which are then handed to subsystems that have access to the coordinates-to-object correspondences.



## Human Engineering

LIFER is intended as a system which both facilitates an interface builder in describing an appropriate subset of a language and its interpretation in his system, and also helps a non-expert user to communicate with the application system in whatever language has been defined. For this reason, close attention was paid to the human engineering aspects of LIFER. Experience with the system has shown that, for some applications, casual users of LIFER have been able to create usable natural language interfaces to their systems in a few days. The resulting systems have been directly usable by people whose field of expertise is not computer science.

**The interface builder.** Unlike PROGRAMMAR (in SHRDLU, Article F5), there is no "compilation" phase where the language specification is converted into a program to recognize the defined language. Instead, changes are made incrementally every time a call to the language specification functions is made. Furthermore, it is easy (by typing a prefix character) to intermix statements to be interpreted by the specification functions, statements to be parsed using the partially specified grammar, and statements to be evaluated in the underlying implementation language of LIFER, namely, INTERLISP (see Article AI Languages.C1). Thus, the interface builder can define a new rewrite rule for the grammar or write a predicate for some meta-symbol and test it immediately, which leads to a highly interactive style of language definition and debugging. A *grammar editor* allows mistakes to be easily and quickly undone. The ability to intermix language definition with parsing allows the interface user to extend the interface language to personal needs or taste during a session using the application system. This extension can either be done by directly invoking the language specification functions, or, if the interface builder has provided the facility, by typing natural language sentences whose interpretations invoke the same language specification functions.

**The interface user.** LIFER provides many features to ease the task of the user typing in sentences to be understood by the system. First of all, it provides feedback indicating when LIFER is parsing the input sentence and when the applications software is running. When LIFER fails to parse a sentence, it tries to give the user useful information of how it failed. It tells the user how much of the input was understood and what it was expecting when it got to the point where it could no longer understand. Interactions with the user are numbered and the user can refer back to a previous question and specify some substitution to be made. For instance:

12. How many minority students took 28 or more units of credit last  
quarter?  
PARSED! 87
13. Use women for minority in 12  
PARSED! 156

Notice the "PARSED!" printed by LIFER to indicate parsing success. This facility can be used to save typing (and more errors), both when similar questions are being asked and when errors in previous inputs are being corrected. The user can simply specify synonyms to be used. For instance:

## 28. Define Bill like William

will cause LIFER to treat the word BILL the same as WILLIAM. LIFER also allows for easy inspection of the language definition, which is useful for both interface builders and sophisticated users.

There are three more sophisticated aspects of LIFER designed to make interactions easier for the user--the spelling correction, ellipsis, and paraphrase mechanisms: When the parser is following along a branch of a transition tree and reaches a point where it can go no further, it records its failure in a failure list. If the input is eventually parsed correctly, the failure list is forgotten. However, if no successful parse can be found, the parser goes back to the last (rightmost) fail point and attempts to see if a misspelling has occurred. (Fail points to the left in the sentence are assumed not to be caused by spelling errors, since at least one transition using the word must have been successful to get to the fail point further to the right. This is not foolproof, however, and sometimes LIFER will fail on a spelling mistake). The INTERLISP spelling correction facility is used to find candidate words that closely match the spelling of the suspect word. The use of semantically significant syntactic categories (such as <PERSON>) above greatly restricts the allowable word substitutions and improves the efficiency of the spelling corrector.

While interacting with an applications system, the user may want to carry out many similar tasks (for example, in a database query system, one often asks several questions about the same object). The LIFER system automatically allows the user to type incomplete input fragments and attempts to interpret them in the context of the previous input (i.e., the interface builder need not consider this issue). For instance, the following three questions might be entered successively and understood by LIFER:

- 42. What is the height of John
- 43. the weight
- 44. age of Mary's sister

If an input fails normal parsing and spelling correction, LIFER tries elliptic processing. Again, because languages defined in LIFER tend to encode semantic information in the syntax definition, similar syntactic structures tend to have similar semantics. Therefore LIFER accepts any input string that is syntactically analogous to any contiguous substring of words in the last input that parsed without ellipsis. The analogies do not have to be in terms of complete subtrees of the syntactic tree, but they do have to correspond to contiguous words in the previous input. The elliptical processing allows for quite natural and powerful interactions to take place, without any effort from the interface builder.

The paraphrase facility allows users to define new syntactic structures in terms of old structures. The user gives an example of the structure and interpretation desired, and the system builds the most general new syntactic rule allowed by the syntactic rules already known. The similarity between the semantics and syntax is usually sufficient to ensure that a usable syntax rule is generated. The following example assumes that the interface builder has included a rule to interpret the construction shown to invoke a call to the language specification function PARAPHRASE with appropriately bound arguments. After typing

63. Let "Describe John" be a paraphrase of "Print the height, weight and age of John" ,  
the user could expect the system to understand the requests

- 64. Describe Mary
- 65. Describe the tallest person
- 66. Describe Mary's sister

even with a fairly simply designed LIFER grammar. (In the context of the earlier examples, this example assumes that "the tallest person" can correspond to the meta-symbol <PERSON>.) The method used to carry out paraphrase (which, as can be seen, is a much more general form of synonymic reference) is quite complex. Basically it invokes the parser to parse the model (the second form of 64.) that is already understood. All proper subphrases (i.e., subphrases that are complete expansions of a syntatic category) of the model that also appear in the paraphrase are assumed to play the same role. A new syntatic rule can then be written, and the actions involved by the model can be appropriately attached to the paraphrase rule.

### Conclusions

Although grammars constructed with LIFER may not be as powerful as specially constructed grammars, LIFER demonstrates that useful natural language systems for a wide variety of domains can be built simply and routinely without a large-scale programming effort. Human engineering features and the ability of the naive user to extend the system's capabilities are important issues in the usefulness of the system.

### References

Hendrix (1977a), Hendrix (1977b), and Hendrix (1977c) all describe the LIFER system. The LADDER information retrieval application is described in Hendrix, et al. (1978) and Sacerdoti (1977). Barrow, et al. (1977) describes the HAWKEYE system.

## References

- Abelson, R. The structure of belief systems. In R. Schank & K. Colby (Eds.), *Computer Models of Thought and Language*. San Francisco: W. H. Freeman, 1973. Pp. 287-339.
- Akmajian, A., Culicover, P., & Wasow, T. *Formal Syntax*. New York: Academic Press, 1977.
- Akmajian, A., & Heny, F. *An Introduction to the Principles of Transformational Syntax*. Cambridge: MIT Press, 1975.
- Bar-Hillel, Y. The present status of automatic translation of languages. In F. L. Alt (Ed.), *Advances in Computers* (Vol. 1). New York: Academic Press, 1960. Pp. 91-163.
- Bar-Hillel, Y. *Language and information*. Reading, Mass.: Addison-Wesley, 1964.
- Bar-Hillel, Y. *Some reflections on the present outlook for high-quality machine translation*. Mimeo, University of Texas, 1970.
- Barrow, H. G., Bolles, R. C., Garvey, T. D., Kremers, J. H., Lantz, K., Tenenbaum, J. M., & Wolf, H. C. Interactive aids for cartography and photo interpretation. *Proceedings ARPA Image Understanding Workshop, Palo Alto, Ca., October 1977*, 111-127
- Bartlett, F. C. *Remembering: A Study in Experimental and Social Psychology*. Cambridge: Cambridge University Press, 1977. (Originally published in 1932.)
- Bobrow, D. G. Natural language input for a computer problem-solving system. In M. Minsky (Ed.), *Semantic Information Processing*. Cambridge: MIT Press, 1968. Pp. 146-226.
- Bobrow, D. G., & Collins, A. (Eds.) *Representation and Understanding*. New York: Academic Press, 1975.
- Bobrow, D. G., & Fraser, J. An augmented state transition network analysis procedure. *IJCAI* 1, 1969, 557-567.
- Bobrow, D. G., & PARC Understander Group. GUS-1, a frame driven dialog system. *Artificial Intelligence*, 1977, 8, 155-173.
- Bobrow, D. G., & Winograd, T. An overview of KRL, a knowledge representation language. *Cognitive Science*, 1977, 1, 3-46.
- Boden, M. *Artificial Intelligence and Natural Man*. New York: Basic Books, 1977.
- Booth, A. D. (Ed.). *Machine Translation*. Amsterdam: North-Holland, 1967.
- Bresnan, J. A realistic transformational grammar. In M. Halle, J. Bresnan, & G. A. Miller (Eds.), *Linguistic Theory and Psychological Reality*. Cambridge, Mass.: MIT Press, 1978. Pp. 1-59.

- Bruce, B. Case systems for natural language. *Artificial Intelligence*, 1975, 6, 327-360.
- Burton, R. R. Semantic grammar: An engineering technique for constructing natural language understanding systems, BBN Report 3453, December 1976.
- Burton, R. R., and Brown, J. S. Toward a natural-language capability for computer-assisted instruction. In H. O'neil (Ed.), *Procedures for Instructional Systems Development*. New York: Academic Press, 1979, 273-313.
- Chafe, W. L. Discourse structure and human knowledge. In R. O. Freedle & J. B. Carroll (Eds.), *Language Comprehension and the Acquisition of Knowledge*. Washington, D. C.: Winston, 1972. Pp. 41-69.
- Charniak, E. Toward a model of children's story comprehension, AI-TR-266, MIT AI Lab, 1972.
- Charniak, E. A brief on case (Rep. 22). Lugano: Istituto per gli Studi Semantici e Cognitivi, 1975.
- Charniak, E., & Wilks, Y. *Computational Semantics: An Introduction to Artificial Intelligence and Natural Language Comprehension*. Amsterdam: North-Holland, 1976.
- Chomsky, N. Three models for the description of language. *IRE Transactions on Information Theory*, 1956, 2, 113-124. (Also in R. Luce, R. Bush, & E. Galanter (Eds.), *Readings in Mathematical Psychology* (Vol. 2). New York: John Wiley & Sons, 1965. Pp. 105-124.)
- Chomsky, N. *Syntactic Structures*. The Hague: Mouton & Co., 1957.
- Chomsky, N. On certain formal properties of grammars. *Information and Control*, 1959, 2, 137-167. (Also in R. Luce, R. Bush, & E. Galanter (Eds.), *Readings in Mathematical Psychology* (Vol. 2). New York: John Wiley & Sons, 1965. Pp. 125-155.)
- Chomsky, N. Formal properties of grammars. In R. Luce, R. Bush, & E. Galanter (Eds.), *Handbook of Mathematical Psychology* (Vol. 2). New York: John Wiley & Sons, 1963. Pp. 323-418.
- Chomsky, N. *Aspects of the Theory of Syntax*. Cambridge: MIT Press, 1965.
- Chomsky, N. Deep structure, surface structure, and semantic interpretation. In D. Steinberg & L. Jakobovits (Eds.), *Semantics*. Cambridge: Cambridge University Press, 1971. Pp. 183-216.
- Clippinger, J. H. Jr. Speaking with many tongues: Some problems in modeling speakers of actual discourse. *TINLAP-1*, 1975. Pp. 78-83.
- Codd, E. F. Seven steps to rendezvous with the casual user. In J. W. Klimbie & K. I. Koffeman (Eds.), *Data Base Management*. New York: North-Holland, 1974. Pp. 179-200.

- Cohen, P. R. On knowing what to say: Planning speech acts. Ph.D. thesis, Computer Science Department, University of Toronto, 1978.
- Colby, K., Weber, S., & Hilf, F. Artificial paranoia. *Artificial Intelligence*, 1971, 2, 1-25.
- COLING76. Preprints of the 6th International Conference on Computational Linguistics. Ottawa, Ontario, Canada, June 1976.
- Conway, M. E. Design of a separable transition diagram compiler. *CACM*, 1963, 6, 396-408.
- Feigenbaum, E., & Feldman, J. (Eds.). *Computers and Thought*. New York: McGraw-Hill, 1963.
- Fillmore, C. The case for case. In E. Bach & R. Harms (Eds.), *Universals in Linguistic Theory*. New York: Holt, Rinehart, & Winston, 1968. Pp. 1-88.
- Fillmore, C. Some problems for case grammar. *Georgetown University Monograph Series on Languages and Linguistics*, No. 24, 1971, 35-56. (a)
- Fillmore, C. Types of lexical information. In D. Steinberg & L. Jakobovits (Eds.), *Semantics*. Cambridge: Cambridge University Press, 1971. Pp. 370-392. (b) (Also in F. Kiefer, *Studies in Syntax and Semantics*. Dordrecht: Reidel, 1969. Pp. 109-137.)
- Firschein, O., Fischler, M., Coles, L., & Tennenbaum, J. Forecasting and assessing the impact of artificial intelligence on society. *IJCAI* 3, 1973, 105-120.
- Friedman, J. Directed random generation of sentences. *CACM*, 1969, 12, 40-47.
- Friedman, J. *A Computer Model of Transformational Grammar*. New York: American Elsevier, 1971.
- Goldman, N. Conceptual generation. In R. Schank, *Conceptual Information Processing*. Amsterdam: North-Holland, 1975. Pp. 289-371.
- Green, B. F., Jr., Wolf, A. K., Chomsky, C., & Laughery, K. BASEBALL: An automatic question answerer. In E. Feigenbaum & J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963. Pp. 207-216.
- Halliday, M. A. K. Categories of the theory of grammar. *Word*, 1961, 17, 241-292.
- Halliday, M. A. K. Notes on transitivity and theme in English. *Journal of Linguistics*, 1967, 3, 37-81, 199-244; *Journal of Linguistics*, 1968, 4, 179-215.
- Halliday, M. A. K. Functional diversity in language as seen from a consideration of modality and mood in English. *Foundations of Language*, 1970, 6, 322-361. (a)
- Halliday, M. A. K. Language structure and language function. In J. Lyons (Ed.), *New Horizons in Linguistics*. Harmondsworth: Penguin Books, 1970. Pp. 140-165. (b)
- Harman, G. (Ed.). *On Noam Chomsky: Critical essays*. Garden City, New York: Anchor Books, 1974.

- Harris, L. R. ROBOT: A high performance natural language processor for database query. *SIGART Newsletter*, 1977, No. 61, pp. 39-40.
- Heldorn, G. E. Automatic programming through natural language dialogue: A survey. *IBM Journal of Research and Development*, July 1976, 20(4), 302-313.
- Hendrix, G. G. Human engineering for applied natural language processing. *IJCAI* 5, 1977, 183-191.
- Hendrix, G. G. The LIFER manual: A guide to building practical natural language interfaces. SRI Tech Note 138, February, 1977.
- Hendrix, G. G., Sacerdoti, E. D., Sagalowicz, D. & Slocum, J. Developing a natural language interface to complex data. *ACM Transactions on Database Systems*, Vol. 3, No. 2, June 1978, 105-147.
- Hendrix, G. Speech understanding research. SRI International, Inc., Menlo Park, Calif., October 1976.
- Hendrix, G. LIFER: A natural language interface facility. *SIGART Newsletter*, No. 61, February 1977, pp. 25-26.
- Hendrix, G., Thompson, C., & Slocum, J. Language processing via canonical verbs and semantic models. *IJCAI* 3, 1973, 262-269.
- Hewitt, C. PLANNER: A language for proving theorems in robots. *IJCAI* 1, 1969, 295-301.
- Hewitt, C. Procedural embedding of knowledge in PLANNER. *IJCAI* 2, 1971, 167-182.
- Hewitt, C., Bishop, P., & Steiger, R. A universal modular ACTOR formalism for artificial intelligence. *IJCAI* 3, 1973, 235-245.
- Hopcroft, J. E., & Ullman, J. D. *Formal languages and their relation to automata*. New York: Addison-Wesley, 1969.
- Hudson, R. A. *English Complex Sentences: An Introduction to Systemic Grammar*. Amsterdam: North-Holland, 1971.
- Hudson, R. A. *Arguments for a non-transformational grammar*. Chicago: University of Chicago Press, 1976.
- Huffman, D. Impossible objects as nonsense sentences. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 6*. New York: American Elsevier, 1971. Pp. 295-323.
- Hunt, E. *Artificial intelligence*. New York: Academic Press, 1975.
- Jackson, P. *Introduction to artificial intelligence*. New York: Petrocelli, 1974.
- Kaplan, R. M. A general syntactic processor. In R. Rustin (Ed.), *Natural Language Processing*. New York: Algorithmics Press, 1973. Pp. 193-241.

- Katz, J. & Postal, P. *An integrated theory of linguistic descriptions*. Cambridge: The MIT Press, 1964.
- Kay, M. The MIND system. In R. Rustin (Ed.), *Natural Language Processing*. New York: Algorithmics Press, 1973. Pp. 155-188.
- Kellogg, C. A natural language compiler for on-line data management. *Proc. Fall Joint Computer Conference*, 1968. Pp. 473-492.
- Klein, S. Automatic paraphrasing in essay format. *Mechanical Translation*, 1965, 8, 3-4.
- Knuth, D. *The art of computer programming: Fundamental Algorithms (Vol. 1)*. Reading, Mass.: Addison-Wesley, 1973.
- Landsbergen, S. P. J. Syntax and formal semantics of English in PHLIQA1. In L. Steels (Ed.), *Advances in Natural Language Processing*. Antwerp: University of Antwerp, 1976.
- Lindsay, R. K. A program for parsing sentences and making inferences about kinship relations. *Proceedings of Western Management Science Conference on Simulation*, 1962.
- Lindsay, R. K. Inferential memory as the basis of machines which understand natural language. In E. Feigenbaum & J. Feldman (Eds.), *Computers and thought*. New York: McGraw-Hill, 1963. Pp. 217-236.
- Locke, W. N., & Booth, A. D. (Eds.). *Machine Translation of Languages*. New York: Technology Press of MIT and John Wiley & Sons, 1955.
- Lyons, J. *Introduction to theoretical linguistics*. Cambridge: Cambridge University Press, 1968.
- Lyons, J. *Noam Chomsky*. New York: Viking Press, 1970.
- Manna, Z. *Mathematical theory of computation*. New York: McGraw-Hill, 1974.
- Marcus, M. A computational account of some constraints on language. *TINLAP-2*, 1978, pp. 236-246.
- Matuzceck, D. *An implementation of the augmented transition network system of Woods*. University of Texas at Austin, Computer Science Department, CAI Lab, 1972.
- McCord, M. On the form of a systemic grammar. *Journal of Linguistics*, 1975, 11, 195-212.
- McDermott, D. *Assimilation of new information in a natural language-understander system*, MIT AI Memo TR-291, MIT AI Lab, February 1974.
- McIntosh, A., & Halliday, M. A. K. *Patterns of language*. Bloomington: Indiana University Press, 1966.
- Minsky, M. A framework for representing knowledge. In P. Winston (Ed.), *The Psychology of Computer Vision*. New York: McGraw-Hill, 1975.



- Minsky, M. (Ed.) *Semantic Information Processing*. Cambridge: MIT Press, 1968.
- Nash-Webber, B. *Semantics and speech understanding* (BBN Rep. 2896). Cambridge: Bolt Beranek & Newman, October 1974.
- Newell, A., & Simon, H. A. GPS, a program that simulates human thought. In E. Feigenbaum & J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963. Pp. 279-293.
- Norman, D., & Rumelhart, D. *Explorations in cognition*. San Francisco: W. H. Freeman, 1975.
- Oettinger, A. G. The design of an automatic Russian-English technical dictionary. In W. N. Locke & A. D. Booth (Eds.), *Machine Translation of Languages*. New York: Technology Press of MIT and John Wiley & Sons, 1955. Pp. 47-65.
- Paige, J. M., & Simon, H. A. Cognitive processes in solving algebra word problems. In B. Kleinmuntz (Ed.), *Problem Solving*. New York: John Wiley & Sons, 1966. Pp. 51-119.
- Perrault, C. R., Allen, J. F., & Cohen, P. R. Speech acts as a basis for understanding dialogue coherence. *TINLAP-2*, 1978, pp. 125-132.
- Petrick, S. R. Transformational analysis. In R. Rustin, *Natural Language Processing*. New York: Algorithmics, 1973. Pp. 27-42.
- Plath, W. REQUEST: A natural language question-answering system. *IBM Journal of Research and Development*, 1976, 20(4), 326-335.
- Postal, P. On the limits of context-free phrase structure description (RLE Quarterly Progress Rep. No. 64). MIT, January 1962.
- Postal, P. Limitations of Phrase Structure Grammars. In J. A. Fodor & J. J. Katz, *The Structure of Language*. Englewood Cliffs, N. J.: Prentice-Hall, 1964.
- Propp, V. *Morphology of the Folktale*. 2nd edition, translated by L. Scott. Austin: University of Texas Press, 1968.
- Quillian, M. R. The teachable language comprehender: A simulation program and theory of language. *CACM*, 1969, 12, 459-476.
- Quillian, M.R. Semantic memory. In M. Minsky (Ed.), *Semantic Information Processing*. Cambridge: MIT Press, 1968. Pp. 227-270.
- Raphael, B. A computer program which understands. *Proc. of the AFIPS Fall Joint Computer Conference*, 1964, 26, 577-589.
- Raphael, B. SIR: A computer program for semantic information retrieval. In M. Minsky (Ed.), *Semantic Information Processing*. Cambridge: MIT Press, 1968. Pp. 33-145.
- Raphael, B. *The Thinking Computer*. San Francisco: W. H. Freeman, 1976.

- Rieger, C. Conceptual memory and inference. In R. Schank, *Conceptual Information Processing*. Amsterdam: North-Holland, 1975. Pp. 157-288.
- Riesbeck, C. Conceptual analysis. In R. Schank, *Conceptual Information Processing*. Amsterdam: North-Holland, 1975. Pp. 83-156.
- Rumelhart, D. Notes on a schema for stories. LNR research memo, UC San Diego, 1973.
- Rustin, R. (Ed.). *Natural language processing*. New York: Algorithmics, 1973.
- Sacerdoti, E. D. Language access to distributed data with error recovery. *IJCAI* 5, 1977, 196-202.
- Samlowski, W. Case grammar. In E. Charniak and Y. Wilks (Eds.), *Computational Semantics*. Amsterdam: North-Holland, 1976. Pp. 55-72.
- Scha, R. J. H. A formal language for semantic representation. In L. Steels (Ed.), *Advances in Natural Language Processing*. Antwerp: University of Antwerp, 1976.
- Schank, R. Identification of conceptualizations underlying natural language. In R. Schank & K. Colby (Eds.), *Computer Models of Thought and Language*. San Francisco: W. H. Freeman, 1973. Pp. 187-247.
- Schank, R. *Conceptual Information Processing*. Amsterdam: North-Holland, 1975.
- Schank, R., & Abelson, R. P. *Scripts, Plans, Goals, and Understanding*. Hillsdale: Lawrence Erlbaum Assoc., 1977.
- Schank, R., & Colby, K. *Computer models of thought and language*. San Francisco: W. H. Freeman, 1973.
- Schank, R., Goldman, N., Rieger, C., & Riesbeck, C. MARGIE: Memory analysis, response generation, and inference on English. *IJCAI* 3, 1973, 255-261.
- Schank, R., & Yale AI Project Group *SAM -- A story understander*. Yale University Computer Science Research Report 43, August 1975.
- Schwartz, J. T. *On programming: An interim report on the SETL Project*. New York University, Computer Science Department, 1973.
- Searle, J. *Speech Acts*. Cambridge University Press 1970.
- Self, J. Computer generation of sentences by systemic grammar. *American Journal of Computational Linguistics*, Microfiche 29, 1975.
- Shapiro, S. A net structure for semantic information storage and deduction, and retrieval. *IJCAI* 2, 1971, 512-523.
- Simmons, R. F., Burger, J. F., and Schwarcz, R. M. A computational model of verbal understanding. *ACM FJCC*, 1968, 441-456.

- Simmons, R. F. Answering questions by computer: A survey. *CACM*, 1965, 8, 53-70.
- Simmons, R. F. Storage and retrieval of aspects of meaning in directed graph structures. *CACM*, 1966, 9, 211-214.
- Simmons, R. F. Natural language question-answering systems: 1969. *CACM*, 1970, 13, 15-30.
- Simmons, R. F. Semantic networks: Their computation and use for understanding English sentences. In R. Schank & K. Colby (Eds.), *Computer models of thought and language*. San Francisco: W. H. Freeman, 1973. Pp. 63-113.
- Simmons, R. F., Burger, J. F., & Long, R. E. An approach toward answering English questions from a text. *Proceedings of the Fall Joint Computer Conference*. New York: Spartan, 1966. Pp. 357-363.
- Simmons, R. F., & Slocum, J. Generating English discourse from semantic networks. *CACM*, 1972, 15, 891-905.
- Steinberg, D., & Jakobovits, L. *Semantics*. Cambridge: Cambridge University Press, 1971.
- Sussman, G., Winograd, T., & Charniak, E. *MICRO-PLANNER reference manual*, AI Memo 203, MIT AI Lab, July 1970.
- Taylor, B., & Rosenberg, R. S. A case-driven parser for natural language. *AJCL*, Microfiche 31, 1975.
- Thompson, F. B. English for the computer. *Proceedings of the Fall Joint Computer Conference*. New York: Spartan, 1968. Pp. 349-356.
- Thorndyke, P. Cognitive structures in comprehension and memory for narrative discourse. *Cognitive Psychology*, 1977, 9, p. 77.
- TINLAP-1. Schank, R., & Nash-Webber, B. (Eds.). *Theoretical Issues in Natural Language Processing: An Interdisciplinary Workshop in Computational Linguistics, Psychology, Linguistics, and Artificial Intelligence*. Association for Computational Linguistics. June 1975.
- TINLAP-2. Waltz, D. L. (Ed.) *Theoretical Issues in Natural Language Processing-2*. New York: Association for Computing Machinery, 1978.
- Waltz, D. L. Natural language interfaces. *SIGART Newsletter*, February 1977, No. 61, 16-64.
- Waltz, D. L. An English language question answering system for a large relational data base. In press, 1979.
- Weaver, W. Translation (1949). In W. N. Locke & A. D. Booth (Eds.), *Machine Translation of Languages*. New York: Technology Press of MIT and John Wiley & Sons, 1955. Pp. 15-23.

- Weizenbaum, J. Symmetric list processor. *CACM*, 1963, 7, 524-545.
- Weizenbaum, J. ELIZA. *CACM*, 1966, 9, 36-45.
- Weizenbaum, J. *Computer power and human reason: From judgment to calculation*. San Francisco: W. H. Freeman, 1976.
- Welln, C. W. *Semantic networks and case grammar* (Publ. 29). University of Stockholm, Institute of Linguistics, 1976.
- Wilensky, R. Understanding goal-based stories. Doctoral dissertation, Yale Research Report 140. September, 1978.
- Wilensky, R. Why John married Mary: Understanding stories involving recurring goals. *Cognitive Science*, 1978, 2, 235-266.
- Wilks, Y. Time flies like an arrow. *New Scientist*, 15 December 1977. (a)
- Wilks, Y. What sort of taxonomy of causation do we need for language understanding? *Cognitive Science*, 1977, 1, 235-264. (b)
- Wilks, Y. An artificial intelligence approach to machine translation. In R. Schank & K. Colby (Eds.), *Computer models of thought and language*. San Francisco: W. H. Freeman, 1973. Pp. 114-151.
- Wilks, Y. Natural language understanding systems within the AI paradigm: A survey and some comparisons, AI Memo 237, Stanford AI Lab, December 1974.
- Wilks, Y. Preference semantics. In E. L. Keenan (Ed.), *The formal semantics of natural language*. Cambridge: Cambridge Univ. Press, 1975. Pp. 329-350.
- Wilks, Y. A survey of syntactic analysis procedures for natural language. *AJCL*, Microfiche 47, 1976. (a)
- Wilks, Y. Processing case. *AJCL*, Microfiche 56, 1976. (b)
- Wilks, Y. Making preferences more active. *Artificial Intelligence*, December 1978, 11(3), 197-224.
- Winograd, T. Procedures as a representation for data in a computer program for understanding natural language, MIT Tech. Rep. TR-17, MIT AI Lab, 1971.
- Winograd, T. *Understanding natural language*. New York: Academic Press, 1972.
- Winograd, T. Five lectures on artificial intelligence, AI Memo 246, Stanford AI Lab, September 1974.
- Winograd, T. Parsing natural language via recursive transition net. In R. Yeh (Ed.), *Applied Computation Theory*. Englewood Cliffs, N.J.: Prentice-Hall, 1975. Pp. 451-467.

- Winograd, T. *Language as a Cognitive Process*. Book in preparation, Addison-Wesley, 1980.
- Wong, H. K. *Generating English sentences from semantic structures*, Tech. Rep. 84, University of Toronto, Computer Science Department, 1975.
- Woods, W. A. *Transition network grammars for natural language analysis*. *CACM*, 1970, 13(10), 591-608.
- Woods, W. A. *An experimental parsing system for transition network grammars*. In R. Rustin (Ed.), *Natural Language Processing*. New York: Algorithmics Press, 1973. Pp. 111-154. (a)
- Woods, W. A. *Progress in natural language: An application to lunar geology*. *National Computer Conference Proceedings*, 1973, pp. 441-450. (b)
- Woods, W. A. *What's in a link: Foundations for semantic networks*. In D. Bobrow & A. Collins (Eds.), *Representation and understanding*. New York: Academic Press, 1975. Pp. 35-82.
- Woods, W. A., & Kaplan, R. *The lunar sciences natural language information system* (BBN Rep. 2265). Cambridge: Bolt Beranek & Newman, 1971.
- Woods, W. A., Kaplan, R., & Nash-Webber, B. *The Lunar Sciences Natural Language Information System: Final report* (BBN Rep. 2378). Cambridge: Bolt Beranek & Newman, June 1972.
- Yngve, V. *Random generation of English sentences*. 1961 International Conference on Machine Translation of Languages and Applied Language Analysis. London: Her Majesty's Stationery Office, 1962.

## Index

- parsing strategies 248
- Abelson, Robert 287  
 ad hoc parsers 272  
 ad hoc representation 225  
 agreement 252  
 ALGOL 230  
 anaphoric references 277  
 application language 297  
 Artouni, G. B. 228  
 ATN 225, 228, 230, 250, 252, 254-255,  
 256, 258, 264-265, 276-278, 284,  
 285, 297  
 augmented transition network 254
- BABEL 264, 265  
 backtracking 248, 254, 259, 281  
 Bar-Hillel, Yehoshua 228, 230  
 BASEBALL 225, 230, 267-268  
 blackboard 248, 258  
 blocks world 263, 279  
 Bobrow, Daniel 227, 269  
 Booth, A. Donald 228  
 bottom-up processing 249, 258, 296  
 Britten, D. H. V. 228  
 Burton, Richard 298
- case 264, 283  
 case frame 226, 245  
 case grammar 244-246, 264  
 case grammars 242  
 causal chain 288  
 causal links 289  
 chart 250, 256-258  
 Chomsky, Noam 230, 233, 238  
 co-routining 259  
 Codd, E. 276  
 combinatorial explosion 250  
 competence vs. performance 238  
 computational linguistics 1, 228, 285  
 conceptual analyzer 284  
 conceptual dependency 227, 231, 262,  
 264-265, 283-284, 287-289  
 concordances 224  
 context-free grammar 228, 235-236, 238,  
 240, 250, 252, 261, 262, 267  
 context-sensitive grammar 228, 234-235,  
 238  
 control mechanisms 248  
 CONVERSE 225  
 cybernetics 228
- DEACON 225  
 declarative representation of  
 knowledge 226  
 deductive mechanism 225  
 deep structure 240, 254  
 demons 284  
 dependency grammar 263  
 derivation tree 236, 238, 239, 247, 254,  
 261, 276, 279  
 dictionary 228  
 discrimination net 265, 285
- early NL programs 224-225, 230, 248,  
 249, 267-273  
 ELIZA 225, 248, 249, 270-272  
 ellipsis 300  
 embedding 252  
 EPAM 264  
 expectations 227  
 extended discourse 265  
 extended grammar 238-246  
 extended grammar parsers 250
- Fillmore, C. 244  
 finite-state transition diagram 252  
 formal grammar 233-237  
 formal languages 233-237, 252  
 frame 227, 246, 289  
 Friedman, Joyce 256, 261  
 FSTD 252, 253

generative grammar 238, 240  
generative semantics 240  
goal 290, 292-293  
Goldman, Neil 264, 285  
grammar 1, 228, 233-246, 247, 250, 252,  
276, 280, 297  
grammarless parsers 249, 250  
Green, Bert 267  
GSP 256-260  
GUS 227

Halliday, Michael 242  
hashing 271  
HAWKEYE 298  
Hendrix, Gary 296  
heuristic 225, 249, 260, 264, 268, 270,  
276, 279, 280, 282  
heuristics 281  
human engineering 299  
human problem solving 270

ideational function 242  
inference 225, 230, 231, 246, 262, 264,  
267, 284, 285  
information retrieval 224, 262, 268, 276,  
298  
interlingua 229, 231, 273, 284, 285, 287  
INTERLISP 300  
interpersonal function 242  
interpretive semantics 240  
IPL-V 267-268  
island driving 249

Kaplan, Ronald 250, 256  
Katz-Postal hypothesis 240  
Kay, Martin 256  
Klein, Sheldon 262  
knowledge-based systems 225  
KRL 227

LADDER 298  
lexicon 240  
LIFER 227, 250, 251, 296-302  
limited logic systems 225  
Lindsay, Robert 267  
LISP 230, 268, 279, 284  
list-processing languages 267-273  
logic 226  
LUNAR 225, 255, 276-278

Machinese 229, 231, 273, 286  
MARGIE 227, 231, 246, 264, 283-286,  
287, 289  
mechanical translation 1, 224, 228-233,  
262, 265, 267, 273-276, 285, 287  
MICRO-PLANNER 279-280  
MIND 256, 259  
mood system 242  
morpheme 239  
multiple sources of knowledge 248, 281

named plan 295  
natural language 1  
nondeterminism 254, 281  
nonterminal symbols 233

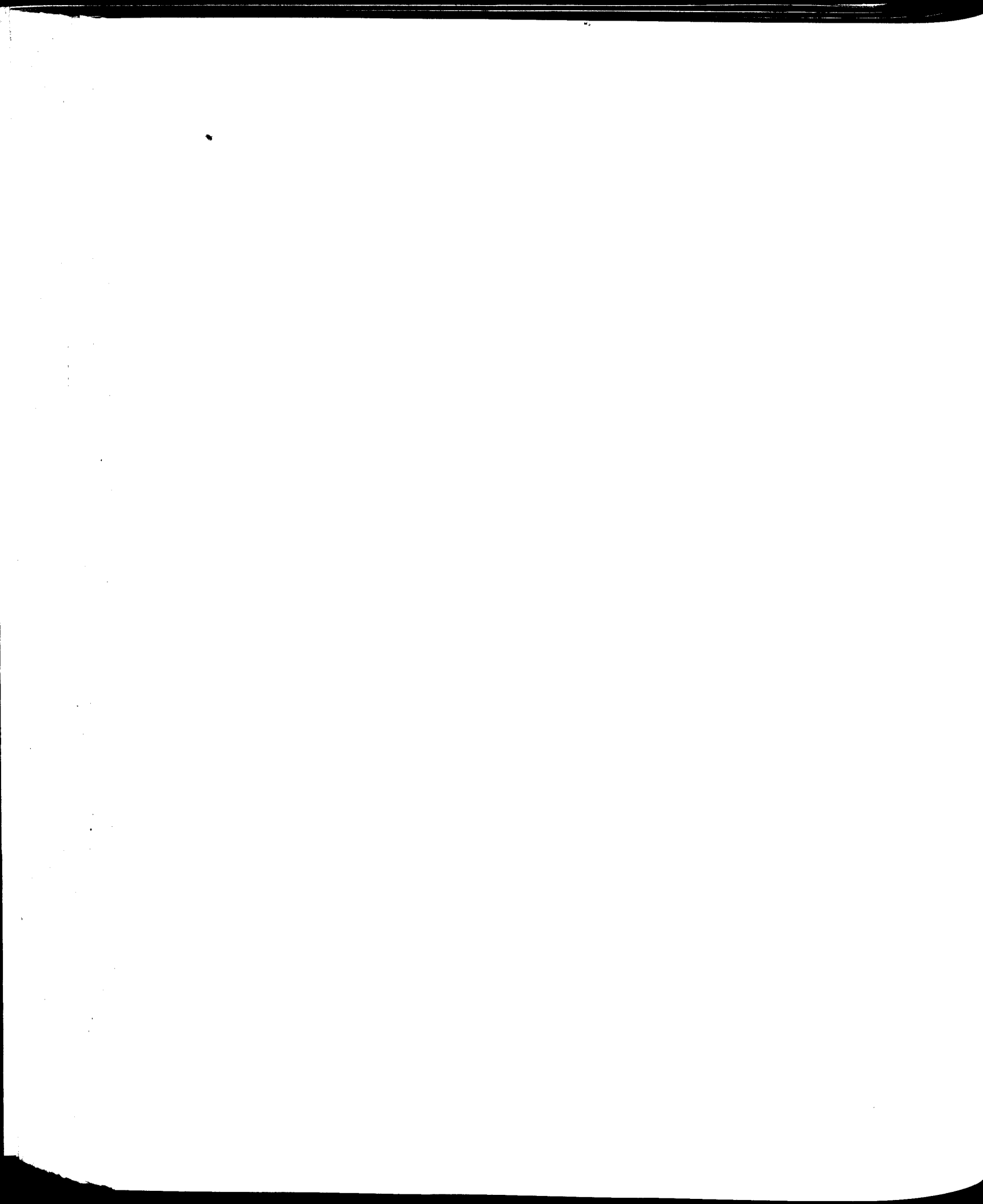
obligatory transformations 239  
Oettinger, Anthony G. 229  
optional transformation 240

PAM 287, 295-296  
parallel processing 248, 254, 281  
paraphrase 264, 300  
paraphrasing 224, 246, 262, 284, 285  
paraplates 265, 275  
PARRY 248  
parser 276, 281, 289, 297  
parsing 1, 233, 234, 236, 247-252, 268,  
270, 272, 279, 284  
pattern matching 249, 268-273

- Petrick, S. 250  
 PHLQA1 227  
 phonemes 239  
 phonological component 240  
 phrase marker 239  
 phrase-structure grammar 228, 234-237,  
     238-239, 250, 251  
 phrase-structure marker 261  
 plan 287, 291-292  
 PLANNER 262, 279-280  
 pragmatics 242, 260  
 predicate calculus 262, 276, 280, 282  
 problem solving 269, 279  
 procedural representation of  
     knowledge 274, 279-280, 284  
 procedural representation of knowledge 225  
 procedural semantics 226  
 procedural/declarative controversy 226  
 production systems 284  
 productions 233  
 PROGRAMMAR 281, 299  
 PROTOSYNTHESIS I 225  
 PROTOSYNTHESIS II 225  
 pseudo-language 228
- quantifier 276  
 query language 276  
 question answering 224, 262, 267, 273,  
     279, 284  
 Quillian, Ross 225, 262
- random text generation 228  
 random text generation 261  
 Raphael, Bertram 225, 268  
 recursive pattern matcher 247  
 recursive transition networks 253  
 regular grammar 236, 238, 252  
 Reifler, Erwin 229  
 Reisbeck, Chris 289  
 representation of knowledge 1, 224-227,  
     229, 231, 245, 261-266, 274, 279-  
     280, 283, 287-292
- rewrite rules 233, 250, 297  
 Richens, R. H. 228  
 Rieger, Chuck 284  
 Riesbeck, Chris 284  
 ROBOT 227  
 RTN 252-254
- SAD-SAM 225, 230, 250, 267  
 SAM 227, 231, 246, 264, 283, 287, 293-  
     294  
 Schank, Roger 227, 231, 245, 264, 283-  
     286, 287  
 schema 289  
 script 287, 289-291, 294  
 scripts 227  
 semantic component 240  
 semantic density 274  
 semantic grammar 250, 298  
 semantic markers 281  
 semantic net 226, 245, 262, 264, 285  
 semantic primitives 227, 231, 246, 264,  
     273, 283, 287, 288  
 semantics 1, 229, 272, 297  
 SHRDLU 225, 243, 248, 250, 263, 279-  
     282, 299  
 Simmons, Robert 225, 264  
 SIR 225, 230, 249, 268-269  
 SLIP 271  
 Slocum, J. 264  
 Smirnov-Troyansky, P. P. 228  
 SOPHIE 250  
 SOPHIE 248  
 speech acts 242  
 speech understanding 1, 226, 248, 249,  
     255, 276, 287  
 SPEECHLIS 255, 276  
 spelling correction 300  
 start symbol 234  
 stereotypes 274-276  
 story grammars 287  
 story understanding 227, 283, 287  
 STUDENT 225, 230, 249, 269-270  
 stylistics 265



- surface structure 240, 244, 261, 264
  - syntactic categories 233
  - syntactic component 240
  - syntax 1
  - systemic grammar 230, 242-243, 280
- 
- template matching 249
  - templates 245, 265, 273, 274-276
  - terminal symbols 233
  - text generation 1, 231, 246, 255, 256, 259, 261-266, 274, 285
  - text-based systems 225
  - textual function 242
  - theme 292-293, 295
  - theorem prover 280
  - TLC 225, 262
  - top-down processing 249, 287, 296
  - transformational grammar 228, 230, 238-242, 243, 244, 250, 255, 261, 284
  - transformational grammar parsers 250
  - transformational rules 239-242
  - transition trees 297
  - transitivity system 242
  - tree 256
  - Turing machine 254
- 
- verb sense 265
- 
- Weaver, Warren 224, 228, 231, 273, 285
  - Weizenbaum, Joseph 270
  - Wilks, Yorick 231, 265, 273-276
  - Winograd, Terry 225, 227, 243, 250, 263, 279-282
  - Wong, H. K. 264
  - Woods, William 225, 250, 252, 276-278
  - world knowledge 1, 226
- 
- Yngve, Victor 228, 261, 263



# Speech Understanding Systems

# Speech Understanding Systems

## Table of Contents

A. Speech Understanding Systems Overview . . . . .	319
1. The HEARSAY I Speech Understanding System . . . . .	326
B. Selected Early Speech Systems . . . . .	329
1. DRAGON Speech Recognition Program . . . . .	329
2. SPEECHLIS . . . . .	331
C. Recent Speech Systems . . . . .	335
1. The HARPY System . . . . .	335
2. The HEARSAY-II System . . . . .	338
3. The HWIM System . . . . .	342
4. The SRI Research on Speech Understanding . . . . .	345
References . . . . .	348
Index . . . . .	351



### The Problem: Understanding Connected Speech

Several early *isolated-word recognition* systems in the 1960s preceded the work on speech understanding systems of the early 1970s. The technique used by these isolated-word systems was to compare the acoustical representation of each word in a relatively small vocabulary to the speech signal and to select the best match, using a variety of "distance" metrics. The Vincens-Reddy system was one of the first isolated-word systems to be successfully demonstrated (Vincens, 1969). Until quite recently, these systems would cost in the tens of thousands of dollars and offer about 95% accuracy on a small vocabulary. This methodology has recently been refined to produce commercially available products for isolated recognition of up to 100 words, costing under \$1,000, although the utility of these products has yet to be demonstrated.

Unfortunately, *connected speech* signals could not be handled with the same techniques: Ambiguities in human speech and erroneous or missing data in the voice signal preclude obtaining a simple, complete set of steps for the direct transformation of acoustic signals into sentences. Non-uniformities are introduced into the acoustic signal in several ways: First, the microphone and background noise of the environment introduce interference in the recording of the spoken utterance. Second, a given speaker does not pronounce the same words quite the same way every time he speaks: Even if the program is "tuned" to one speaker, the matching process between the phonetic characteristics of words (the templates) and the actual utterance is inherently inexact. Third, the pronunciation of individual words change when they are juxtaposed to form a sentence. Lastly, the words are not separated in the voiced sentence. On the contrary, whole syllables are often "swallowed" at word boundaries. Taken together, these factors imply that the basic acoustic signal, which is the foundation for the rest of the processing, does not look at all like a concatenation of signals of individual words.

The difficulties introduced in recognizing "connected" speech required a new viewpoint on the methodology. Researchers speculated that *expectations* about the form of the utterances could be gleaned from contextual information, such as the grammar and current topic of conversation, and that these expectations could be used to help identify the actual content of the signal. Thus, the task came to be viewed as one of *interpretation* of acoustic signals, in light of knowledge about syllables, words, subject matter, and dialogues, and carried with it the problem of organizing large amounts of diverse types of knowledge. The way that the speech systems enable communication between the "sources" of these various types of knowledge is one of the most interesting aspects of their design.

### The ARPA Speech Understanding Research Program

In the early 1970s, the Advanced Research Projects Agency of the US Department of Defense decided to fund a five-year program in speech understanding research aimed at obtaining a breakthrough in speech understanding capability. A study group met in 1971 to set guidelines for the project (Newell et al., 1971). This group of scientists set specific performance criteria for each dimension of system inputs and outputs: The systems were to accept normally spoken sentences (connected speech), in a constrained domain with a 1,000-word vocabulary, and were to respond reasonably fast ("a few times real-time" on current high-speed computers) with less than 10% error. This was one of the few times that AI programs had had design objectives set before their development. Setting these

standards was important since they approximated the minimum performance requirements for a practical, connected speech understanding system in a highly constrained domain (although producing a practical system was notably not a goal of the ARPA program). The final performance of the ARPA SUS projects will be discussed later in this article.

A number of different designs were explored during the ARPA effort, from 1971 to 1976, including alternative control structures and knowledge representations for transforming the speech signal into a representation of the meaning of the sentence (from which an intelligent response could be generated). These systems employed several types of knowledge, including a vocabulary and syntax tailored to the task area. In a typical system, a series of complicated transformations are applied to the analog signal, which represents the intensity of the utterance as recorded by the microphone, resulting in a compact digital encoding of the signal.

### Some Important Design Ideas

Further processing of the digitized signal was the task of the systems developed in the ARPA program. These systems will be described in more detail later; but first some general design considerations should be illuminated. In many ways these ideas, which are ubiquitous in AI research, appear especially clearly in the context of the large speech understanding systems.

**Top-down and Bottom-up Processing.** As mentioned above, the ARPA systems all used "top-end" knowledge about "likely" utterances in the domain to help identify the contents of the speech signal. Knowledge of the possible "grammatical" sentences (syntax), "meaningful" sentences (semantics), and "appropriate" sentences (pragmatics) was used at each point in the dialogue. For example, consider the HEARSAY-I speech system (article B1), which played "voice chess" with the speaker by responding to the moves that he spoke into the microphone, using a *chess program*. (see Search.Heuristics in Games) to figure out the best response. Not only did HEARSAY-I use syntactic knowledge about the specific format of chess moves (e.g., "pawn to king-4") to anticipate the form of incoming utterances, but it also used its chess program's legal-move generator to suggest moves that were likely to be tried by the opponent--and then examined the speech signal for those particular moves.

The importance of top-down or "expectation-driven" processing has also been pointed out by workers in natural language understanding research (see Natural Language, especially the articles on MARGIE and SAM). Although there is no *word recognition* problem in understanding type-in sentences, determining the meaning of the input, so that an appropriate response can be evoked, requires the use of much knowledge about the world. Similarly, in vision research where, as in speech, the task is one of recognition as well as understanding, a strong model of the physical world, as well as knowledge about what things the camera is likely to find, are typically used to help figure out what is in the scene (see Vision).

It is generally agreed that this *constraining knowledge* is necessary for adequate performance in the speech understanding task: Without expectations about what to look for in the input, the task of identifying what is there is impossible. "Experiments" with several systems demonstrated the effect of removing syntactic and semantic constraints on the

processing of the speech signal. The HARPY system, which combined all of the phonetic, syntactic, and semantic knowledge into one integrated network, was 97% accurate in actually identifying the words in the utterance, even though it showed only 42% accuracy in phonetic segmentation of the utterance. In other words, since "top end" knowledge about what words were allowed to follow others was incorporated in the network, HARPY could often guess the right words even when it didn't have an accurate phonetic interpretation of the signal. In the HEARSAY-I system, where the phonetic, syntactic, semantic, etc., knowledge was separated into independent subsystems (called *knowledge sources*), a more convincing kind of experiment could be performed: the system was designed so that it could run with only some of the knowledge sources "plugged in." Compared with its performance with just the phonetics and lexicon knowledge sources operating, the performance of HEARSAY-I improved by 25% with the addition of its syntax knowledge source and by another 25% with the addition of the semantics knowledge source (Lea, 1978).

**Generality vs. Power.** The way that top-down processing is used to constrain the expected content of sentences reflects an important universal issue in AI systems design. The top end of the speech systems contains knowledge about a specific language (grammar) and a specific domain. In the development of all of the speech understanding systems, general grammatical knowledge gave way to grammars that were very specific to the task requirements (called *performance grammars*), making use of the structure of the stereotypical phrases used in a particular task domain (Robinson, 1975). The generality vs power trade-off, between domain independence and the use of domain specific knowledge to constrain hypothesis formation, is an important idea in all areas of AI research.

**Cooperating Knowledge Sources.** Another major design idea was to separate the various types of knowledge (phonetics, syntax, semantics, etc.) into independent *knowledge sources*. These, in turn, communicate with each other about the possible interpretations of the incoming signal. Typical of this design were the HEARSAY systems developed at Carnegie-Mellon University. In the HEARSAY model, the knowledge sources were theoretically to know nothing about each other, not even of each other's existence. They were thought of as independent processes that looked at the signal and generated hypotheses about words, phrases, etc., on the *blackboard*--a global data-structure accessed by all of the knowledge sources via an established protocol. Hypotheses generated by the bottom-end knowledge sources (about syllables and words) could be examined for feasibility by the syntactic and semantic knowledge sources; and these knowledge sources, in turn, could post suggestions about what words might be expected. Thus top-down and bottom-up processing found an explicit, natural implementation in this paradigm.

The advantages of the HEARSAY model were those generally associated with *modularization* of knowledge (see Representation): adding, modifying, or removing a knowledge source could theoretically be accomplished without necessitating changes in the other knowledge sources. In a multiprocessor environment, where the different knowledge sources are running as processes on different machines, such a system would be less sensitive to transient failures of processors and communication links--i.e. exhibit *graceful degradation*, as it is called.

**Compiled knowledge.** The other principal type of speech processing system is one where the knowledge about all of the sentences that are meaningful in the task domain are *precompiled* into one network. The nodes in the network are phonetic templates that are to



be matched against the voice signal. The links in the network are used to control the matching process: After a successful match at node N, only the nodes that are linked in the net to node N need be tried next--no other sound is "legal" at that point. The processing of the input signal starts at the beginning of the utterance, matching the first sound against the "start" nodes of the net, and proceeds left-to-right through the network, trying to find the "best path" according to some matching metric. Since all paths through the net are legal sentences, the best path corresponds to the system's best interpretation of the utterance. The DRAGON system, and its successor HARPY, both developed at Carnegie-Mellon, are examples of compiled knowledge systems.

### The Status of Speech Understanding Research

The HARPY system was the best performer at the end of the ARPA program. The performance requirements established by the working committee and the final results of the HARPY system are compared below (after Klatt (1977)):

GOAL (November, 1971)	HARPY (November, 1976)
Accept connected speech from many cooperative speakers in a quiet room using a good microphone with slight tuning/speaker accepting 1000 words using an artificial syntax in a constraining task yielding < 10 percent semantic error requiring approximately 300 million instructions of per second of speech (MIPSS).*	Yes 5 (3 male, 2 female) yes computer terminal room close-talking microphone 20-30 sentences/talker 1011 word vocabulary average branching factor = 33 document retrieval 5 percent  requiring 28 MIPSS  using 256K of 36-bit words costing \$5.00 per sentence.

\* The actual specs stated "a few times real-time" on a 100 MIPSS machine.

Comparing the performance of the various systems at the termination of the ARPA project (September, 1976) is complicated by several factors. A standard set of utterances was not prepared for testing, since each system used a different task domain. The task domains, which included document retrieval (HARPY, HEARSAY-II), answering questions from a database (SRI system and BBN's HWIM), and voice chess (HEARSAY-I), had a range of difficulties (as measured by branching factor) from 33 for HARPY to 196 for HWIM. Systems also varied according to the number of speakers and amount of room noise that could be accommodated and the amount of tuning required for each new speaker.

For example, in a variety of tasks, DRAGON recognized from 63-94% of the words and from 17-68% of the utterances with up to a 194-word vocabulary. This variation of results across domains displayed by DRAGON demonstrates the difficulty of specifying how well a

system performs. It appears that the number of words in the lexicon alone is an inadequate measure of the complexity of the understanding task. The CMU speech group proposes a measurement termed *average branching factor*, based on the number of sounds that can typically follow at each point in each legal sentence. For example, DRAGON's performance was better with a particular 194-word vocabulary than with another 37-word vocabulary (consisting of just the alphabet plus numbers), with a higher branching factor, due to the similarity in phonemic structure of the 26 letters.

The complexity of the speech understanding task is also demonstrated by the impact of the problem on other areas of AI. During the recent speech efforts, new research in natural language, representation of knowledge, search, and control strategies has been required to deal with the objective of recognizing continuous speech in the presence of ambiguity. For a more detailed comparison and discussion, see Lea & Shoup (1978b) and Klatt (1977).

### Summary

Considerable progress towards practical speech understanding systems has been made during the 1970s, and work in this area has generated ideas that influenced work in other areas of AI, such as vision and natural language understanding. Following is a summary of the conclusions of the same study group that established the requirements for the ARPA project at the beginning of the decade:

The gains go beyond empirical knowledge and engineering technique to basic scientific questions and analyses. A few examples: Network representations for speech knowledge at several levels have been created that have substantial generality. A uniform network representation for the recognition process has been developed. Rule schemes have been created to express certain phonological and acoustic-phonetic regularities.... Techniques have been found for measuring the difficulty and complexity of the recognition task. The problem of parsing (syntactic analysis) with unreliable or possibly missing words (so that one cannot rely on parsing left-to-right, but must be able to parse in either direction or middle-out from good word matches) has been successfully analyzed. New paradigms have been developed for many of the component analysis tasks and for the control structure of intelligent systems. Substantial progress has been made on understanding how to score performance in a multi-component system, how to combine those scores, and how to order search priorities. (Medress et al., 1977)

Most of the principal Speech Understanding Systems are described in the articles in this chapter. Many of the ARPA contractors produced multiple systems during this time period. Work at Bolt, Beranek and Newman, Inc., produced first the SPEECHLIS and then the HWIM system, using previous BBN research in natural language understanding (see Natural Language.LUNAR). Carnegie-Mellon University produced the HEARSAY-I and DRAGON systems in the early development phase (1971-1973) and the HARPY and HEARSAY-II programs in the later stage. SRI International also developed several speech understanding programs, partly in collaboration with the Systems Development Corporation. Additional systems, not reported in this chapter, include work at Lincoln Labs, Univac, and IBM. The current IBM system

utilizes the dynamic programming approach explored in the DRAGON system and is the most active speech understanding project under development since the end of the ARPA program (Bahl et al., 1978).

#### References

The recent book by Lea (1978) contains the best comparative overview of the ARPA speech systems, as well as detailed articles on the systems themselves written by their designers. Other good summary articles are Lea & Shoup (1978b) and Klatt (1977). For a popular account of the ambiguities inherent at the phonetic level of encoding, see Cole (1979). Descriptions of early speech research and the goals of the ARPA program are in Newell (1975) and Reddy (1975).

### A1. The HEARSAY I Speech Understanding System

The HEARSAY I speech understanding system was developed at Carnegie-Mellon University (Reddy, Erman, Fennell, & Neely, 1973). The design of HEARSAY I was based on a theory that the best approach to a complicated domain such as speech understanding was to divide the problem into a group of competing *knowledge sources*. Each of these is an expert at some small part of the task, such as combining syllables to make words. The theory also assumes that each of these knowledge sources can be used when appropriate, responsive to changes in the interpretation (understanding) of the utterance. HEARSAY I was the first system to break up a problem into small components, to utilize an evaluation of what the program accomplished to guide what would be done next, and to include knowledge sources reflecting the content of the domain of expertise from which the sentences would be taken. These techniques allowed HEARSAY I to become the first system to recognize connected speech.

The domain of voice-chess was used to provide a semantic base for the HEARSAY I program. Voice-chess pits the computer against a person in a game of chess where the person speaks his moves to the computer. The fact that only particular phrases make sense in the world of chess was utilized to limit the searching required by the program. The following demonstrates the relationship between domain knowledge and understanding. If a person heard the incomplete utterance "Pawn to King [*missing word*]" and knew a little about the way chess moves are stated, he could deduce that the missing word would have to be a small number corresponding to the rows of the chess board. Moreover, if a person knew the positions of all the pieces and the rules for making moves, guessing at the missing word would be limited to just the few places that pawns could move. And, if a person knew a lot about chess strategy and this was the first move of the game, he could guess that it was *likely* that this was one of the standard moves--e.g., Pawn to King-four. This level of semantic constraint could be available in very specialized domains (e.g., air traffic control). HEARSAY I tied directly into the legal move generator of a chess-playing program that provided the *syntactic* and *semantic* constraints mentioned above. The syntactic constraints utilized a grammar based on the domain--e.g., piece takes/moves to position instead of the subject-verb-object grammar--which became the standard representation for later systems. Furthermore, the fact that only a few legal moves were possible put semantic constraints on which moves "made sense"--roughly corresponding to the situation where a person doesn't quite hear a sentence but comes up with an idea of what is being said, which is completely wrong but which still makes complete sense relative to the conversation in progress. No attempt has been made to directly tie the design of speech systems to psychological research, but several of the intuitive psychological concepts have been borrowed and explored.

#### Special Features

**Segmented acoustical signal.** The acoustic signal was divided into regions based on features in the waveform. The acoustic labeling process was on time-varying segments that (hopefully) matched the syllables in the original utterance. The typical method in use when HEARSAY was designed divided the speech signal into fixed time intervals and labeled these intervals for future processing, which introduced artifactual labelings where syllable boundaries had been crossed.

Separate, independent, anonymous knowledge sources. Isolating the knowledge along functional lines allowed efficient modification of the problem-solving structure of the program, by allowing a free substitution of modules. Substitution was possible since each KS was not dependent on the methodology behind, or the existence of, any other KS.

### Representation of Knowledge

Three knowledge sources were provided in HEARSAY I (compared to the 12 in HEARSAY II): (a) speaker- and environment-dependent knowledge and acoustic-phonetic knowledge, (b) syntactic knowledge, and (c) semantic knowledge. The speaker and environmental knowledge was used to adjust the phonetic analysis to match the variations that arose between individuals and different user settings. The acoustic-phonetic knowledge source contained the relation between features in the acoustic signal and their associated syllables. The semantic component was tied directly into the chess-playing program to provide the type of constraints explained above.

### Control Structure

The main task of the control structure was to utilize the appropriate knowledge sources in a sequence that would lead to the recognition of all the words in the sentence. The application of knowledge sources was a three-phase process: Poll the knowledge sources, hypothesize, and test. The poll phase determined which knowledge sources could contribute to the sentence hypothesis and determined an estimate of the efficiency of each contribution. The hypothesis phase invoked the knowledge source with the most confident proposal and made the results--e.g., a few new words--available for further processing. The test phase evaluated (verifies) each of the hypothesized words with respect to all of the existing hypotheses. Unverified hypotheses were stored away, in case the system had to backtrack and attempt a different path to recognize the utterance, using an alternate set of hypotheses.

### Limitations

There were several problems with this implementation of the HEARSAY design, which led to the HEARSAY II implementation. Most of these limitations were overcome during the ARPA speech project.

The hypothesize-and-test paradigm was used only at the word level. Subwords (such as syllables and phonemes) and superwords (phrases and concepts) were not directly used in the dynamic portions of the recognition task.

Different phases of the speech recognition task could not be done in parallel because of the lock-step nature of the control structure.

The hypotheses were not represented as competing hypotheses in the global database; therefore each hypothesis was processed separately, which caused redundant computation.

There was no explicit system structure to implement policy decisions; hence it was hard to create policy algorithms that controlled the order of evaluation of knowledge sources, in order to create or test new hypotheses.

### Summary

HEARSAY I was the first demonstrable system for the recognition of nontrivial connected speech. This system marked a radical departure in both knowledge representation and control structure from previous speech recognition systems. The use of semantics to help judge the relevance of words derived from the acoustic processing was a very useful idea. It is believed (Lea & Shoup, 1978a) that had current acoustic processing techniques been available in the early 1970s, HEARSAY I, using the chess domain, might have met the 1976 speech goals. Many of the concepts used in this system have been adapted for the HEARSAY II and HARP systems developed at CMU.

### References

See Reddy, Erman, Fennell, & Neely (1973), Reddy (1975) and Reddy (1976).

## B. Selected Early Speech Systems

### B1. DRAGON Speech Recognition Program

The DRAGON connected speech recognition system was developed at Carnegie-Mellon University (Baker, 1975; Lowerre, 1976). The recognition problem was represented as a network that incorporated paths corresponding to each of syntactic combinations of words defined by the grammar. Each word was represented in the network by each of its possible pronunciations. Transition probabilities were associated with each arc in the network to represent the likelihood that a phone (phonetic unit) would follow another phone for each legal sentence. A dynamic programming approach considered all pathways in the network "in parallel" across the entire utterance. The program stored the state with the highest probability (and how to get there) corresponding to each 10 ms. slice of acoustical waveform information. These best states were connected together to select the path through the net that best matched the sequence of sounds in the utterance. This design demonstrated the capability of a mathematically based, nonheuristic approach to speech recognition.

#### Special Features

**Delayed decision.** A global best path through the network was found by considering the entire utterance before collecting the local best path segments; so no backtracking was needed. The computation time for finding the best path grows linearly with the length of the utterance.

**General theoretical framework/generative form of the model.** A uniform mathematical framework was used to represent the speech processing task in DRAGON. The model assumed a probabilistic relation between the internal variables and the observed variables (e.g., from all phrases to the acoustic waveforms), that is, that each of the observed variables (e.g., acoustic waveform) could be considered probabilistically linked to a series of internal (as yet unknown) variables (e.g., the basic phones in the sentence). For a given sequence of phones, a procedure existed to generate the corresponding acoustic representation. Bayes's Theorem was used to reverse the computation and generate all possible phones from the given acoustic signal. An additional constraint was placed on this mapping process, the Markov assumption, which limited how much past history of the variables was used in the computation. The DRAGON system assumed that only the current state and the immediately previous state were relevant to the mapping process.

**Hierarchical system/integrated network.** The same mathematical framework for considering the relationship between acoustic signals and phones was used between phones and words and between words and their position in the grammar. These levels nested well, with each higher level state (e.g., a single word) composed of a collection of lower level states (e.g., the phones that compose that word). This hierarchy was stored as a single large network with each level of processing embedded in the next most complex level. The implication of this technique was to merge all of the knowledge sources into one integrated unit, such that cooperation between knowledge sources was maximized (see HEARSAY II article for alternative methods of cooperation).

### Representation of Knowledge

The knowledge sources for DRAGON were: (a) integrated networks composed from the syntax and lexical/phonetic spellings, (b) user-dependent acoustic-phonetic templates, (c) a heuristic matching routine to rate the templates against actual acoustic signals, and (d) a table of conditional probabilities to relate internal variables to external observations, calculated from training data.

### Control Structure

The control structure for DRAGON was very simple since most of the work was done before the program was run on any speech input. A major section of the program was the generation of the finite-state network that represented the legal discourse for a particular task. When the program operated on the input, it traversed the entire graph using the results of the acoustic parameterization to determine likelihoods that a particular phoneme existed at the corresponding point in the utterance. When the graph was filled out, the overall best route through the graph was selected as the result of the program. Since all paths were covered, the globally optimal path was selected.

### Limitations

The major difficulties with this approach were long computation times, network size that grew combinatorially with the size of the language, the difficulty of adding knowledge that didn't fit the representation scheme, and unnatural segmentation of the acoustic signals. The major computational expense came from the requirement that for every 10 ms. segment of signal, all acoustic-phonetic templates had to be matched and all state probabilities updated. The decision to break the acoustic signal at a fixed interval, rather than have "naturally" occurring (but difficult to pinpoint) segmentation, introduced several artifacts into the system. These included a proliferation of templates (168 templates for 33 phones) and unusual phonemic spellings in the dictionary.

### Summary

The DRAGON system stressed the importance of a mathematically *admissible* algorithm. This system design was utilized as the basis for the HARPY system. While DRAGON computed for a significantly longer time than its contemporary heuristic-based system, HEARSAY I, its recognition percentage was higher.

### References

See Baker (1975), Lowerre (1976), and Reddy et al. (1977).



## B2. SPEECHLIS

SPEECHLIS (Woods, 1975) was a speech understanding system developed at Bolt, Beranek, and Newman. The task domain was queries about the chemical analysis of the Apollo 11 moon rocks, based on previous work on the LUNAR system, a natural English question-answering system. An "incremental simulation" approach was taken in the design of SPEECHLIS; that is, experiments with it were begun before the complete system was built, using combinations of computer programs and human simulators. The system used a vocabulary of approximately 250 words and a subset of the complete LUNAR grammar. SPEECHLIS was organized around the following knowledge sources:

1. **Acoustic feature extraction:** first-pass segmentation and labeling of the acoustic signal into partial phonetic descriptions.
2. **Lexical retrieval:** retrieval of words from the lexicon, to be matched against the input signal. Candidates are retrieved on the basis of knowledge of the vocabulary and partial phonetic descriptions.
3. **Word verification:** determination of the degree to which a particular word matches the acoustic signal at a particular location in the signal.
4. **Syntax:** judgment of the grammaticality of a hypothesized interpretation of the signal, and proposal of words or syntactic categories to extend a partial interpretation.
5. **Semantics:** notation of coincidences between semantically related words found at different locations in the signal; a judgment of meaningfulness of a hypothesized interpretation and a prediction of the particular words or classes of words to extend a partial interpretation.
6. **Pragmatics:** use of knowledge of the current context, situation, or speaker.

### Special Features

**Parser driven system.** The parser was the main controller of the system, so that the flow of control is from the syntax and semantics down to the acoustic processing components. This flow is quite different from the early speech systems, which started at the acoustic signal and were not guided by the possible meanings of words derived from a partially completed analysis.

**Lattice representation of results.** Both words and acoustic segments were represented as a lattice of the significant ways to divide up the utterance for a given level of abstraction.

**Levels of representation.** SPEECHLIS subdivided the problem into many different

levels of interpretation: acoustic, lexical retrieval, word verification, syntax, semantics, and pragmatics. Each of these levels used a different type of object--e.g., phoneme, words, and concepts--to store the hypotheses at each level of understanding.

### Representation of Knowledge

**Acoustic-phonetics.** The analysis portions of the system were based on the results of the encoding of the acoustic waveform. In SPEECHLIS, the technique known as *linear predictive coding* was used to obtain a concise digital representation of the utterance. This method utilized a model of the vocal tract to help distinguish the speech sounds from the acoustic properties of the vocal tract. **Syntax.** A parser for speech, unlike a parser for text, must be able to deal with ambiguity and error in the words that it is presented. Thus, it must not only detect syntactic ambiguities and inconsistencies, but it must also aid in selecting a syntactically well-formed sequence of words from the many possible sequences in the word lattice. To this end, the SPEECHLIS parser uses an ATN grammar (Natural Language.ATN). When given a hypothesis about the existence of a sequence of "reliable" words (known as an *island*) to process, it attempted to create the ways in which the island might be accepted by the grammar if surrounded by some suitable context. The combinatorial problems were combatted by the use of the semantics component. **Semantics.** The semantics module was used to propose additional words that might have occurred in the original utterance but which were missing from the initial word lattice because of poor match quality. It constructed meaningful sets of word matches from a lattice of possible matches, evaluated the consistency of syntactic structures and semantic hypotheses, and transformed the best hypothesis about an utterance into a formal procedure that operated on its database in order to answer questions or absorb new information.

Two principle data structures were used in the SPEECHLIS semantics processor to represent the semantic knowledge so as to be easily accessed (see corresponding articles):

1. **A semantic net:** a directed graph that represents associations among words and concepts.
2. **Case frames:** descriptions of the ways in which the relationships between concepts (as indicated in the semantic net) hold and how they might be expressed in an utterance.

### Control Structure

The task of the control structure was to organize the application of the various knowledge sources in the analysis of the speech signal to infer the correct interpretation. The control structure carried out its task through the manipulation of "theories" that represented alternative hypotheses about the utterance being interpreted. A theory contained the words hypothesized to be in the utterance, together with syntactic, semantic, and pragmatic information, and scoring information, which represented the likelihood of the theory. The theories generally represented only partial hypotheses, and the control structure had to manage their creation and refinement, devoting its resources to those theories estimated as having the greatest likelihood of validity.

Monitors played a central role in exploring theories. A monitor is a trap set by a hypothesis for new information which, if found, would result in a change, or extension, to the monitoring hypothesis. Such traps do not cause immediate interruptions. Instead, the system maintained a queue of "events" to be processed, and a monitor caused a new event to be created and placed in the queue. In addition to this event-driven mechanism, the control structure also contained a mechanism for actively seeking out new information by making "proposals," that is, requests to match a particular word or set of words at some point in the utterance. In an example from the lunar rocks understanding domain, hypothesizing the word "concentration" set up monitors for the concepts SAMPLE and CHEMICAL ELEMENT located in the semantic net. If some element, such as HELIUM, was hypothesized, then the evaluation of the related concepts in the semantic net, in this case CHEMICAL ELEMENT, would trigger the monitor set by "concentration" and strengthen both hypotheses (Nash-Webber, 1975).

Central to the success of the control structure was a good evaluation mechanism, able to assess the likelihood that a particular theory will yield the correct result. The control structure used scores resulting from theory evaluation by the various sources of knowledge to allocate its resources.

The general flow of control in SPEECHLIS was (briefly) as follows:

- (a) Acoustic-phonetic analysis returns a "segment lattice." This lattice represents all possible segmentations of the utterance and alternative identities of the individual segments. The lattice is also analyzed by a phonological rule processor.
- (b) Lexical retrieval and word matching return a "word lattice." These are words with three or more phonemes that score above a matching threshold.
- (c) The Semantics module constructs theories about the consistency of the concepts implied by each word. It notes words that could be semantically related to the given word and notes coincidences between two or more semantically related words or concepts. The semantic coincidences are sorted according to their order of likelihood, which is used for the construction of larger theories. This process iterates until the semantics module has nothing more to add.
- (d) The Syntax module performs syntactic evaluation of the hypothesized words. It picks up words from the word lattice and proposes new words (e.g., a, of, the) to fill in gaps in the theory. Syntax is closely coupled to semantics when making such proposals.

### Limitations

The outstanding problems were (a) no clearly articulated control mechanism, (b) recognition of a thrashing state in which no one hypothesis emerged as a good candidate for the whole utterance, (c) the need for a more rigorous measure of confidence in the evidence, together with (d) ways of combining such confidence measures.

**Summary**

This program demonstrated the application of previously developed natural language processing techniques, such as the augmented transition net grammar, to speech understanding research. The HWIM system is the successor to SPEECHLIS, and a great majority of these ideas were utilized in that system. The use of levels of representation and a top-down control of the system have become standard features of later systems.

**References**

See Woods (1975) and the whole special issue on speech recognition IEEE (1975).

## C. Recent Speech Systems

### C1. The HARPY System

The HARPY speech recognition program was designed at Carnegie-Mellon University (Lowerre & Reddy, 1978; Lowerre, 1976). The system was designed after an extensive evaluation of two previous speech understanding systems at CMU:- DRAGON and HEARSAY I. Important concepts were drawn from both of these previous systems with an emphasis on the operations research approach taken by the DRAGON system. This approach views the speech recognition problem as the traversal of a network of paths through the search space. Each of these paths corresponds to one pronunciation of one of the possible sentences formed from the grammar. Since there may be several possible pronunciations of each word and many ways to put words together to form sentences, the number of paths is quite large. To limit the size of the network several paths may share common subnetworks.

The design decision to combine the various speech levels: acoustic, phonetic, and syntactic into one "pre-compiled" data structure is probably the most significant aspect of the program. One knowledge source in the HEARSAY II system adopted this technique for verification at the word level and used a HARPY-like search of the network corresponding to each word in the lexicon.

With such a large network to traverse in order to determine the correct path, search techniques had to be exploited. Lowerre and Reddy developed the concept of beam search, which examined the best few paths at each node encountered. The decision to include a path was based on the spread of likelihoods of success for the paths leaving a node in the network. If many equally likely paths are available, then a large portion of the paths are searched or, in less ambiguous cases, a few of the best paths are followed up.

All speech programs use some description of the acoustic signal as the basis for recognizing words and then sentences. HARPY used the HEARSAY concept of segmenting speech by characteristic pauses and breaks in the waveform. The DRAGON system used fixed time intervals to section the acoustic waveform and had to introduce several artificial acoustic labels, because meaningful segments were subdivided.

#### Special Features

**Precompiled network.** HARPY creates a large network comprised of all phonetic spellings for each syntactic path in the grammar. This network includes word junction phenomena, which are the adjustments made in the pronunciations of words due to those preceding and following them in continuous speech.

**Beam search.** The best paths in the recognition network are selected for further evaluation. This pruning is determined by the comparison of the likelihood of success with a variable threshold. This strategy eliminates evaluation of possible sentences that start correctly but contain one or more incorrect words. HARPY keeps about 1% of the states at each step in the evaluation of the network. Experiments showed that neither a fixed number nor a fixed fraction worked well for this process. Finally, a fixed range of likelihood from the best state was settled upon (Newell, 1978).

**Processing segmented speech.** The decision to use a flexible division of the acoustic signal according to acoustic events, rather than according to a fixed time interval, allows for a single acoustic template per phone. However, since the network is composed of a sequential phonetic representation, the system is very sensitive to missing or poorly labeled segments.

**Heuristics to limit search time and size of network.** The computation time of the program is drastically reduced by the compilation of the speech recognition knowledge into the network representation. The network is condensed by removing redundant states or by recognizing common groupings of states. The number of states is slightly increased, but the number of connections (i.e., pointers) can be markedly decreased, by introducing special states at common intersections in the network.

### Representation of Knowledge

HARPY uses only acoustic, phonetic, and syntactic knowledge sources. These sources are initially represented as a BNF grammar specification, a dictionary, and interword juncture rules, which are then compiled into the finite-state transition network structure. Speaker-dependent acoustic-phonetic templates and a table of data-dependent transition probabilities are other data structures.

### Control Structure

Unlike the DRAGON system, which searched every path in the network, the HARPY system uses a threshold to limit the active states to only those states whose probability is within a threshold based on the highest state probability for that time segment (the "beam"). Thus, if the probabilities are well separated, only a few states will be considered, and conversely, if the probabilities are bunched together, many states will be pursued.

### Limitations

The extension to much larger vocabularies must be examined in future research efforts, since the explicit creation of the network of possibilities can have a large memory requirement. The design of the current system cannot easily accommodate the pragmatics of the utterance, which may be needed to constrain search in an expanded speech domain. HARPY is also sensitive to missing acoustical segments and missing words.

### Summary

A case study of two dissimilar speech systems led to a first in system design: a system that met the 1976 goals of the speech community (see Newell et al., 1971, and the overview article). This objective was accomplished by the combination of dynamic programming techniques with useful heuristics, such as beam search. A current topic of research is to determine the maximum capabilities of this type of algorithmic approach. This system has been singled out as an example of how far speech recognition can be pursued

without the use of semantic and pragmatic constraints. The fact that this system was able to meet the 1976 goals demonstrates the distance that speech understanding has come since the early single-word recognition systems available in 1970.

#### References

See Lowerre (1976), Lowerre & Reddy (1978) and Reddy et al. (1977).

## C2. The HEARSAY-II System

The HEARSAY-II speech understanding system was developed at Carnegie-Mellon University (Reddy et al., 1977). HEARSAY-II was designed to provide a general framework for AI problems with the following characteristics: a large search space, knowledge at different levels of abstraction (in this case varying from acoustical signals to semantic networks), and ambiguous or noisy data. The design of HEARSAY-II incorporates lessons from experiences with previous CMU systems: DRAGON, HEARSAY I, and HARPY (see respective articles).

The HEARSAY model divides the speech understanding domain into many levels of representation--from the acoustic signal data at the "bottom," through phonemes, words, phrases, and concepts, to the relationship of a particular utterance to the entire ongoing dialog at the "top." The key concept of HEARSAY is the creation of "knowledge sources" that take information at one (or more) level and create, confirm, or deny a hypothesis at another level or at the same level. Each of these knowledge sources (KS) is designed to be separate and to work independent of other knowledge sources; but each is also meant to cooperate with the complement of KSs in verifying hypotheses. Each hypothesis is stated in terms of a location in time (calculated from the start of the utterance) of a particular word, phrase, etc., gleaned from the acoustic signal; for example, concluding the likelihood that the word "give" is located at the beginning of the sentence. Restrictions are placed on the implementation of the KSs to allow for an easy reconfiguration of the modules without necessitating complementary changes in other components of the system.

Each knowledge source accesses a global data structure, the "blackboard," for storing current hypotheses and for inter-KS communication. In the September 1976 version of the system, the blackboard was divided into 6 hierarchical levels--parameter, segment, syllable, word, word sequence, and phrase--upon which 12 KSs operated. Example KSs are: (a) the syllable-based word hypothesizer, whose function is to generate hypotheses at the word level in a time area near the location of the syllable; or (b) a predictor of possible time-adjacent grammatical word extensions. These 6 levels and 12 KSs were selected from a considerable number of other possibilities developed over the course of the project.

The great flexibility of this design is exhibited in its method for utilizing knowledge sources. After several words are postulated for each segment of the acoustical signal, knowledge sources are activated to work on the most promising hypotheses first, both throughout the levels and across the span of the utterance. Thus the program utilizes the "higher" levels of knowledge to control the "lower" levels. For example, after a given phrase has been derived by acoustic means, the syntactic knowledge source predicts words that often occur immediately following the original phrase: These predictions are then used to examine the acoustic signal for possible instances of the hypothesized word, for verification. Thus, processing works bidirectionally among the levels of representation until a conceptual translation, spanning the utterance, is formed. Each knowledge source adds some partially plausible information about what can be deduced in one section of the problem space; and when these hypotheses are integrated with the results of other knowledge sources, they produce a coherent interpretation of the utterance.



### Special Features

These special features are based on the original HEARSAY concept. Some of these design goals were modified in the demonstration version.

**Separate, independent, anonymous knowledge sources.** Isolating the knowledge along functional lines allows efficient modification of the problem-solving structure of the program, by allowing a free substitution of modules. Substitution is possible since each KS is not dependent on the methodology behind, or the existence of, any other KS. Of course, a minimal number of KSs must exist to provide sufficient knowledge content for any analysis to take place. In particular, there must be enough KSs to span all the levels of representation.

**Self-activating, asynchronous, parallel processes.** The KSs can be viewed as individual knowledge-based programs that respond to patterns in the database as necessary. No temporal relationship between the execution of the KSs is explicitly required. A parallel processor version of portions of the HEARSAY design has been built to exploit these features.

**Globally accessed database.** The blackboard acts as a structure on which the hypotheses and their support criteria can be stored. The data structure is fixed for each information level on the blackboard. This feature allows the creation of kernel accessing routines, used in common by each KS for manipulating the global store at each level. A snapshot of the blackboard during HEARSAY execution reveals a partial analysis of the utterance as a three-dimensional network consisting of the levels of representation, time, and the possible alternatives--with the contextual and structural support for each alternative explicitly marked in the network.

**Data-directed knowledge invocation.** The knowledge sources react to changes in the blackboard and criticize or create hypotheses wherever practical. This procedure sets up a new pattern over the blackboard, to which other KSs may be able to respond. This activity continues until no knowledge source can respond or until the time and space limits of the program are exceeded.

### Representation of Knowledge

The HEARSAY knowledge framework for speech processing is broken into many fixed levels (e.g., acoustic segments, words, etc.), to provide the appropriate representations for designing knowledge sources. The majority of the knowledge is stored in individual processes that monitor the appropriate levels of the framework. The monitoring is accomplished by "attaching" to the knowledge source both a list of the patterns in the database to which the KS is designed to respond and a summary of the kinds of hypotheses it creates. Thus, each knowledge source acts like a large production rule (see article on production rules).

### Control Structure

The HEARSAY II implementation provides a general framework for the manipulation of knowledge sources. The following description is based on the set of interactions established for the September 1976 configuration of knowledge sources.

The acoustical processing and segmentation knowledge sources are invoked to establish enough seed hypotheses to start the interpretation process. Possible syllables are postulated from the segmented signal. Possible words are derived from the syllables. Words are rated by a HARPY-like KS that contains a network representation for all the possible pronunciations of each word in the vocabulary. At present, about 75% of the correct words are hypothesized, but a large number of incorrect words are also being considered (Erman, 1977).

The next task is the conversion of word hypotheses into short sequences called *islands*, by utilizing legal word pairs (abstracted from the grammar) and acoustic and phonetic criteria related to word juncture. The highest rated single words are "expanded" by the addition of words to which they relate temporally or syntactically. The best of these multi-word islands are hypothesized at the next level in the blackboard. This string of words is checked for syntactic consistency and may be used to generate surrounding words that, if verified, would extend the island. This process continues until at least one complete sentence is recognized and all other possible interpretations have lower credibilities, or until the system runs out of resources.

**Focus of Attention.** Since the island-building and verification activities can take place anywhere in the utterance, a typical search problem develops. Each potential action is rated by a statistically based scheduler (Hayes-Roth & Lesser, 1977) that estimates the utility of a given operation against the goals for recognition of the entire utterance. In particular, the scheduler tries to make sure that good hypotheses are being generated in each time segment of the utterance.

### Limitations

The major problem with the design of HEARSAY II is that there is the trade-off between generality and processing time. The cost of generality is increased computational load. In the HARPY system, which figures out a priori how to process a given utterance, a substantial time savings has been demonstrated. Some of the design goals for completely asynchronous and functionally separate KSs had to be relaxed in order to get the performance demonstrated in the September 1976 system (see Article Overview on Design Considerations). The major reason for decreasing the asynchronous activity is the unreliability of the results at the lower levels, thus making it dangerous to proceed based on only partial results.

### Summary

HEARSAY II has achieved the ARPA five-year specifications for a "successful" speech recognition program. The generality of this design for knowledge-based systems has been demonstrated beyond speech understanding; currently, vision and protein crystallography research. The system design is currently being refined and coordinated with other approaches, particularly the HARPY design, as well as with a multi-processor configuration.

**References**

See Erman (1977), Hayes-Roth & Lesser (1977), Lesser & Erman (1977), and Reddy et al. (1977).

### C3. The HWIM System

The HWIM ("Hear What I Mean") speech understanding system was designed and constructed during the period 1974-1976 at Bolt, Beranek, and Newman (Woods et al., 1974-1976). The HWIM system is the successor to the SPEECHLIS speech understanding system. The domain of discourse of the HWIM system is the role of a travel budget manager that answers questions about trips and the status of the travel budget. Typical sentences that the system could recognize are: "List all trips to California this year," "What is the registration fee for the next ACL conference?", and "The registration fee is twenty dollars." Each of these types of statements requires a different kind of interaction with the database and with the pragmatic information about planning trips. In particular, the statement about a registration fee only makes sense after the context of a specific conference has been established.

The goal of the HWIM design was to form a global hypothesis or *theory* about the input (stimulus) presented to the system. This complete theory is often accomplished by a refinement and integration of partial theories, until all of the input data is accounted for. The process is divided into several steps: (a) Form a seed hypothesis from elementary constituents (e.g., acoustic wave phenomena), (b) form the set of possible constituents, given a partial theory (called *theory-driven* or *predicted hypothesization*), (c) measure the match between a hypothesis and the input stimuli, and (d) rate the content and reliability of created hypotheses. In the speech setting, a typical theory is a set of compatible word hypotheses, with possible gaps between words, and partial syntactic and semantic interpretations.

The perceptual theory uses two computational devices to create and verify hypotheses: the *monitor* and the *proposal*. The monitor is a marker in the knowledge structure that starts processing when the appropriate partial theories concerning a piece of knowledge have been created. The monitor helps relate different partial theories by stating what interpretations can be drawn, if enough supporting evidence has been supplied, for the existence of a particular entity in the utterance. The monitor concept eliminates the requirement that the system must always go from lowest levels of processing ("raw data") to the highest (interpretation of the sentence). The proposal results in a comparison, of the input and the current elementary hypotheses. The monitors are passively looking for verification information, and the proposals actively attempt verification. Triggered monitors, known as events, are ranked and queued for later action. At any point in the operation of the program, the next action of the system is determined by selecting from the best ranked event in the queue. One major research goal of the BBN speech project was understanding the control structure issues of when (and how) to create new hypotheses, as opposed to verifying older hypotheses.

#### Special Features

**Uniform scoring policy.** All hypotheses from all knowledge sources are rated on the same scale, as opposed to individual ad hoc ratings.

**General linguistic consultant using ATN grammars.** Syntactic, semantic, and pragmatic information are combined into a single linguistic knowledge source.

**Bidirectional parsing.** The system can work outwards from "islands" of strongly hypothesized words. The parser can extend the hypotheses by suggesting words occurring just before or after the island.

**Improved hypothesis verification strategies.** The HWIM verification technique generates a synthetic acoustic signal based on hypothesized words and then compares this signal with the original acoustic features at the parametric level.

**Control strategies with proven characteristics.** Several control strategies were developed, some of which can be guaranteed to find the best possible interpretation of an utterance without systematically enumerating all possibilities.

### Representation of Knowledge

Several knowledge structures exist in HWIM that correspond to the different levels of processing. The basic structures are: (1) dictionary, which contains the words known to the system; (2) segment lattice, which contains an exhaustive list of segments that match the acoustic signal; and (3) general linguistic consultant that is built around the augmented transition net parsing technique.

The dictionary contains the phonetic "spelling" for each word and the usual variations in pronunciation. The dictionary is algorithmically expanded to include all predictable pronunciation deviations, using the application of phonological rules.

The segment lattice allows for the representation of alternate segmentation paths where the acoustic evidence is sufficiently ambiguous to make unique segmentation decisions impossible. Segment labels are selected from the list of phoneme's and include the likelihood of a phoneme's existence in the acoustic information.

The linguistic consultant is based on a "pragmatic" grammar that includes the constituents of the domain--e.g., "trips" and "budgets"--in place of the typical linguistic subdivisions of "subject" and "verb," etc.

### Control Structure

HWIM has the ability to select from one of many different control strategies, all based on the concept of finding *islands* of reliability and working out towards the point of understanding the rest of the query. All the strategies perform an initial scan of some region of the utterance and form initial *seed events*, consisting of single words at individual points in the utterance. For left-to-right strategies, the initial scan considers only words that could begin the utterance. Combination strategies then examine the beginning of the utterance and, next, switch to the island driven method to complete the task.

These control strategies drive the three major components of the system: (a) the lexical retrieval component, (b) the verification component, and (c) the syntactical component. The lexical retrieval component is used to match dictionary word pronunciations against the segment lattice in two ways: (a) without regard to context, or (b) "anchored" off a previously found word match or set of word matches, to take advantage of the effects of

known adjacent words. In either case, this component can seek either all words or only those in a specified set. It uses the tree-structured dictionary in order to do matching and across-word acoustical analyses.

The verification component takes a word pronunciation and the available context and generates a spectral representation (a synthesized acoustic signal) for words that are hypothesized by other knowledge sources. The actual acoustic signal is compared to this synthesized signal and a distance metric is calculated. This feature allows an independent verification method for hypothesized words, since all other methods are using the same acoustic source as a basis for making conclusions. This method uses a large amount of computational time.

The syntactical component utilizes syntactic, semantic, pragmatic, and factual knowledge to determine the likelihood of hypothesized words. The parser is used: to test the grammaticality of a sequence of words, to predict possible extensions of words, and to build up a semantic representation of the utterance for analysis and response.

#### Summary

This system has received recognition for several important features based on the word/acoustic matching and retrieval components and the extension of the ATN parsing methodology. The HWIM designers performed extensive research on control strategies, which will be very important to later system designs.

#### References

See Woods et al. (1974-1976).

#### C4. The SRI Research on Speech Understanding

One segment of the ARPA Speech Understanding Project was the joint workings of SRI International (Walker, 1978) and the Systems Development Corporation (SDC). The signal-processing, acoustics, and phonetics portions of the system were derived from the original SDC system, known as the Vocal Data Management System. SRI provided the parsing, syntax, semantics, pragmatics (commonsense knowledge about the world), and discourse analysis sections of the system. At different stages of development, domains of discourse included: the Blocks World of Winograd, the repair of small appliances, and the areas of data management concerning naval fleet movements (based on SDC experience) and electromechanical equipment maintenance (in conjunction with the SRI Consultant system). As in the other articles in this section of the handbook, we will concentrate on the non-acoustical processing levels, therefore on the work done at SRI.

##### Special Features

**Top-down control.** The higher level processing (e.g., the phrase level) controls the sentence interpretation, as opposed to acoustic considerations. Top-down control allows the semantic information to guide the search of the potentially ambiguous acoustic information. The system utilizes an increased unit size for hypothesis generation, *the phrase*, for guiding the top-down processing. The phrase consists of a sequence of words. Bottom-up control is also utilized in the system.

**Flexibility of processing.** Linguistic capabilities include: starting processing anywhere in an utterance, mapping from the phrase to the phonemic representation, and handling ellipses (implied information based on previous utterances) and anaphoric references (pronoun reference to recent phrases).

**Focus of attention.** Certain areas of each information structure are singled out as most relevant to the current discourse. Focus of attention can dramatically impact the amount of processing and help to increase the understanding of utterances with multiple meanings by considering the most recent topics of interest.

##### Representation of Knowledge

**Semantic Nets.** The SRI system uses a modified form of semantic nets with case frames to encode the concepts of the domain and previous utterances. The network is divided into task-domain knowledge and a scratch area that represents current utterances. The network is separated into spaces that are utilized to handle quantification, to separate real and hypothetical worlds, and to distinguish different levels of abstractions. The semantic processor, for example, contains a series of functions that map between the surface and the deep case structures; these functions facilitate suggestions of which surface features to examine, based on semantic knowledge. Functions also exist to use the semantic representation to answer simple questions. Versions of the system designed for use with the SRI consultant task contain a process model, or script, that encodes the course of normal construction of some object.

**Language Definition System.** The language definition system contains the complete specification of that portion of the English language to be accepted by the system. All words known to the system form the lexicon. The lexicon is divided along syntactic boundaries into categories for nouns, verbs, etc. Associated with each word and category are attributes based on information from each knowledge source (e.g., semantics, phonology, grammar, etc.). Each category is provided with instructions for calculating attributes for each member-word that is found in the current utterance. Redundancy rules contain attributes in common for the entire language definition. Composition rules are available that can generate plausible phrases out of words or smaller phrases; these are usually based on grammatical considerations. These rules also specify which attributes can be assigned to the new phrase; for example, they determine the focus of the phrase or relate the semantics of a particular word to the whole phrase. Composition rules designate which grammatical factors--such as case, mood, number, and person--should be used to judge the phrase when considered by the parser. Also included is prosodic information, for example, the expected change in pitch at the end of an utterance that is a question.

### Control Structure

The system is controlled from the parsing mechanism that tries to form words, and eventually phrases, out of the acoustic signal. The parser uses the *language definition system*, which is a structured listing of the legal discourse (domain-specific) and its attributes, to integrate all knowledge sources in a coherent way. Thus, the main program representation is at the phrase level, with the meaning and ramifications--e.g., likelihood--of each phrase recorded. Semantic nets are used to show the relations between concepts in both the discourse area and in the representations of the contents of previous utterances. Pragmatic information is used to predict likely utterances, based on previous utterances. Acoustic processing represents the input speech as formants, which is a categorization of the frequency content, together with special features, such as rising pitch in prosodic analysis.

**Parser.** The parser controls system performance and integrates all knowledge sources in its attempt to produce the best of several interpretations of the utterances. The main data structure for the parser is the parse net, which includes phrases generated by the language definition system. A task queue holds a selection of operations waiting to be performed to expand the parse net. Requests for checking the acoustic information for a specific word or phrase are typical examples of waiting tasks. The parse is completed when the queue is empty or when limits on resources are exceeded. Tasks are scheduled to run based on priorities determined by the language definition system and are moderated by expectations of interpretations likely to be generated and how close the task is to the focus of attention. The parsing strategy is called a "best-first" approach. To help coordinate the choice of which phrases to expand next, a particular part of the parse net is singled out as being the most relevant direction to take, and this decision biases future decisions until another area looks more promising.

The parse net is made up of nodes representing phrases generated and predictions to be examined (usually against acoustic data). Predictions are of the form: Category X, location Y? (e.g., look for a verb at some start time). The nodes are linked by connections between unverified phrases and the predictions they spawn. The parse net also contains attribute information such as the expected start and stop time of the candidate phrases.



False steps are avoided in the parsing process by storing past mistakes, but the generated mistaken phrases can be reused if they are recognized as appropriate in another context.

**Discourse Analysis and Pragmatics.** An important capability is the use of information gleaned from previous utterances to help disambiguate the current (possibly incomplete) phrase. The system is designed to handle both anaphoric references and ellipses by looking at the recent dialogue. Anaphoric reference replaces a pronoun in the current phrase with a noun from a preceding piece of discourse. For example, "What is the depth of the Ethan Allen? What is *its* speed?" This substitution process is accomplished by examining the case frame of the current utterance and using those constraints to search previous phrases for a match. Elliptic references deal with filling out incomplete phrases using terms already mentioned. For example, "What is the depth of the Ethan Allen? The Lafayette"?, where the second question is assumed to be the same order as the first. The capability to make assumptions of this kind is accomplished by determining the attributes of the elliptic phrase (i.e., "The Lafayette") and attempting a match of these against segments or phrases of the preceding utterance. Discourse analysis is also used to suggest what type of questions might be expected in subsequent utterances.

### Summary

The SRI system stressed the natural language understanding aspects of the speech recognition problem. Detailed symbolic models of the domain of discourse were available for assisting in the interpretation and response phases of the program. Since the system was never fully tested, it must be evaluated on the application of the processing techniques to natural language understanding research.

### References

See Paxton (1976), Reddy (1976), and Walker (1978).

## References

- IEEE Transactions on Acoustics, Speech, and Signal Processing, February 1975 (a special issue on Speech Understanding).
- Bahl, L. R., Baker, J. K., Cohen, P. S., Cole, A. G., Jelinek, F., Lewis, B. L., & Mercer, R. L. Automatic recognition of continuously spoken sentences from a finite state grammar. *Proceeding of the 1978 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Tulsa, Oklahoma, 1978, pp. 418-421.
- Baker, J. K. The Dragon System -- An Overview. *IEEE ASSP*, February 1975, ASSP-23(1), 24-29.
- Cole, Ronald A. Navigating the slippery stream of speech. *Psychology Today*, 12:11, April 1979, 77-87.
- de Mori, R. On speech recognition and understanding. In K. S. Fu & A. B. Whinston (Eds.) *Pattern recognition, theory and application*. Leyden: Nordhoff, 1977. Pp. 289-330.
- de Mori, R. Recent advances in automatic speech recognition. *Proc. of the 4th Int. Joint Conf. on Pattern Recognition*, Kyoto, Japan, November 1978.
- Erman, L. D. A functional description of the Hearsay-II speech understanding system. *Speech Understanding Systems, Summary of Results of the Five-year Research Effort at Carnegie-Mellon University*, CMU Computer Science Tech. Report, August 1977. (Also: *IEEE Conf. of Acoustics, Speech, and Signal Processing*, Hartford, Conn., May 1977.)
- Erman, L. D., & Lesser, V. R. The HEARSAY-II speech understanding system: A tutorial. In W. A. Lea (Ed.) *Trends in Speech Recognition*. Englewood Cliffs, N.J.: Prentice-Hall. In press, 1978. (a)
- Erman, L. D., & Lesser, V. R. System engineering techniques for artificial intelligence systems. In A. Hanson & E. Riesman (Eds.), *Computer vision systems*. New York: Academic Press. In press, 1978. (b)
- Hayes-Roth, F., & Lesser, V. *Focus of Attention in the Hearsay-II Speech Understanding System*, CMU Computer Science Tech. Report, Carnegie-Mellon University, January 1977.
- Klatt, D. H. Review of the ARPA Speech Understanding Project. *Journal of the Acoustical Society of America*, 1977, 62, 1345-1366.
- Lea, W. A. (Ed.) *Trends in Speech Recognition*. Englewood Cliffs, N.J.: Prentice-Hall. In press, 1978.
- Lea, W. A., & Shoup, J.E. Review of the ARPA SUR Project and survey of the speech understanding field (Final report on ONR Contract No. N00014-77-C-0570). *Speech Communication Research Laboratory*, Santa Barbara, CA, 1978. (a)

- Lea, W. A., & Shoup, J. E. Specific contributions of the ARPA SUR Project. In W. A. Lea (Ed.), *Trends in Speech Recognition*. Englewood Cliffs, N.J.: Prentice-Hall. In press, 1978. (b)
- Lesser, V., & Erman, L. D. A Retrospective View of the HEARSAY-II Architecture. *IJCAI* 5, 1977, 790-800.
- Lowerre, B., & Reddy, R. The Harpy Speech Understanding System. In W. A. Lea (Ed.), *Trends in Speech Recognition*. Englewood Cliffs, N.J.: Prentice-Hall, 1978.
- Lowerre, B. The Harpy Speech Recognition System, Doctoral dissertation, CMU Computer Science Tech. Report, Carnegie-Mellon University, 1976.
- Medress, M. F., et al. Speech understanding systems: Report of a steering committee. *Sigart Newsletter*, 1977, 62(April), 4-8. (Also in *Artificial Intelligence*, 1977, 9, 307-316.)
- Nash-Webber, B. L. The role of semantics in automatic speech understanding. In D. Bobrow & A. Collins (Eds.), *Representation and Understanding: Studies in Cognitive Science*. New York: Academic Press, 1975.
- Newell, A. A Tutorial on Speech Understanding Systems. In D. R. Reddy (Ed.), *Speech Recognition: Invited Papers Presented at the 1974 IEEE Symposium*. New York: Academic Press, 1975. Pp. 3-54.
- Newell, A. HARPY, Production Systems and Human Cognition, CMU-CS-78-140. Dept. of Computer Science, Carnegie-Mellon University, 1978.
- Newell, A., Barnett, J., Forgie, J., Green, C., Klatt, D. H., Licklider, J. C. R., Munson, J., Reddy, D. R., & Woods, W. A. *Speech Understanding Systems: Final Report of a Study Group*, Carnegie-Mellon University, 1971 (reprinted by American Elsevier, Amsterdam, North-Holland, 1973).
- Newell, A., Barnett, J., Forgie, J., Green, C., Klatt, D., Licklider, J. C. R., Munson, J., Reddy, R., & Woods, W. *Speech Understanding Systems: Final Report of a Study Group*. Amsterdam: North Holland/American Elsevier, 1973 (originally published in 1971).
- Paxton, W. H. A Framework for Language Understanding, SRI Tech. Note 131, AI Center, SRI International, Inc., Menlo Park, Calif., June 1976.
- Reddy, R., Erman, L., Fennell, R., & Neely, R. The HEARSAY Speech Understanding System: An Example of the Recognition Process. *IJCAI* 3, 1973, 185-193. (Reprinted in *IEEE-Transactions Computers*, 1976, C-25, 427-431.)
- Reddy, R., et al. *Speech Understanding Systems, Summary of Results of the Five-year Research Effort at Carnegie-Mellon University*, CMU Computer Science Tech. Report, Carnegie-Mellon University, August 1977.
- Reddy, R. (Ed.) *Speech Recognition: Invited Papers of the IEEE Symposium*. New York: Academic Press, 1975.

- Reddy, R. Speech Recognition by Machine: A Review. *Proceedings of the IEEE*, 1976, 64, 501-531.
- Robinson, Jane J. *Performance Grammars*. In D. Raj Reddy (Ed.), *Speech Recognition: Invited Papers of the IEEE Symposium*, New York: Academic Press, 1975, 401-427.
- Smith, A. R. *Word hypothesization for large-vocabulary speech understanding systems*. Doctoral dissertation, Carnegie-Mellon University (also available as Tech. Rep.), 1977.
- Vincens, P. *Aspects of Speech Recognition by Computer*. Ph.D. Thesis, Computer Science Department, Stanford University, 1969.
- Walker, D. E. (Ed.) *Understanding Spoken Language*. New York: North Holland, 1978.
- Woods, W. A. SPEECHLIS: An Experimental Prototype for Speech Understanding Research. *IEEE ASSP*, February 1975, ASSP-23(1), 2-10.
- Woods, W. A., et al. *Speech Understanding Systems, Final Report* (BBN Report 3438). Cambridge: Bolt, Beranek, & Newman, November 1974-October 1976 (Vols. 1-5).

## Index

ARPA speech research 320, 324

blackboard 322  
bottom-up processing 1, 321  
branching factor 323, 324

case frames 345  
chess program 321  
compiled knowledge 322  
connected speech 320

DRAGON 323

events 342

graceful degradation 322

HARPY 323  
HEARSAY 322

isolated-word recognition 320

knowledge sources 320, 322

legal-move generator 321

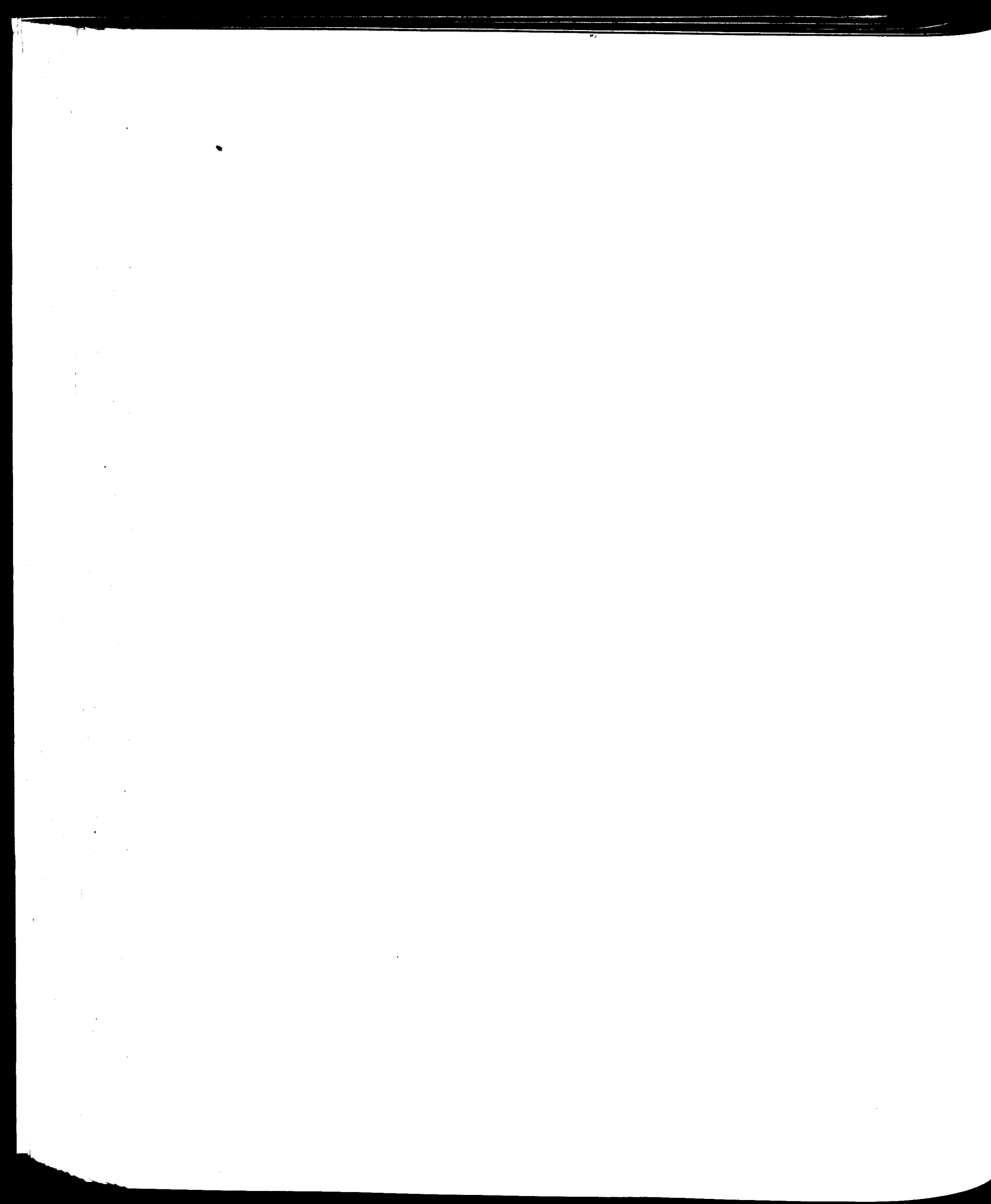
modularization 322  
Monitors 333

performance grammars 322  
phonetics 1  
pragmatics 321  
prosodic 1

prosodics 1

semantic nets 345  
semantics 321  
signal 1  
speech recognition 1  
speech understanding 1  
syntax 321  
system performance 323

template 1  
top-down processing 1, 320, 321



# AI Programming Languages

# AI Programming Languages

## Table of Contents

Historical Overview of AI Languages	355
AI Language Features	
Overview of Language Features	370
Data Structures	376
Control Structures	388
Pattern Matching	399
Environment	407
Important AI Programming Languages	
LISP	416
PLANNER and CONNIVER	433
OLISP	435
SAIL	436
POP-2	437



## AI Languages

### An Historical Overview of Artificial Intelligence Programming Languages

Stephen Westfold

Artificial Intelligence programming languages have had a central role in the history of Artificial Intelligence. They have two important functions. First, they allow convenient implementation of programs demonstrating and testing AI ideas. Second, they provide vehicles of thought; as with other high-level languages, they allow the user to concentrate on higher level concepts and avoid being distracted by low-level implementation details. In this article we distinguish between the more general-purpose languages widely used in AI, such as LISP and POP-2, and the higher level AI languages, such as PLANNER and QLISP. Using the concepts of the higher level languages imposes a structure on the way one thinks, which can be restrictive, but without some such structure it is very difficult to approach the problems of AI. Frequently, new ideas in AI are accompanied by a new AI language in which it is natural to apply these ideas. Usually, such a higher level language is built on an existing high-level AI language so that the desirable features of the host language do not have to be re-implemented in the new language.

Figure 1 gives a rough indication of the directions in which AI languages developed and the major influences on the languages.

IPL was developed around 1956 by Newell, Shaw, and Simon as the first programming language specifically for AI. Its design was guided by ideas from psychology, especially the intuition of association. The primary elements of the language were symbols as opposed to numbers, around which all other languages of the time were built. To form associations of these symbols, list processing was introduced. The objective was to enable programs to build data structures of unpredictable size and shape conveniently. The problem of unpredictable shape was solved by using data elements consisting of two fields, each of which could hold either a symbol or a pointer to another such data element. This simple arrangement allows arbitrary binary trees or *list structure* to be built. The problem of unpredictable size is handled by having a *free list* of data elements that are allocated to the various data structures as required.

A major advantage of list structure is that elements can be inserted and removed very simply from existing structure. However, it is clearly desirable that elements deleted from all structures be available for reuse in new structure. In IPL the user is responsible for returning cells to the free list when they are no longer required. For programs that build complex structures that include some sharing of substructures, however, it is difficult to determine, in general, when an element is deleted from one structure whether it, or any structure it points to, is part of any other structure. Later list processing systems have therefore taken responsibility for reclaiming elements no longer used. One method for doing this is to maintain a *reference count* for each element showing how many other elements point to it. Every primitive list operation that deletes a pointer must decrement the count of the element pointed to and, if the count is zero, reclaim it and recursively delete any pointers from the element. An alternative method for reclaiming elements is *garbage collection*. Periodically, such as when the free list is empty, the garbage collector *traces* all the pointers to elements accessible to the program. Tracing a pointer consists of *marking* the element pointed to and recursively tracing the pointers of this element. This process ensures that all elements still in use are marked, so the unmarked elements can be collected and added to the free list. (This algorithm also serves as an example of how desirable it is to allow recursive procedures in list processing). It turns out that there are some combinations of list operations where it is not feasible to maintain reference counts correctly [MOSE 1970], whereas garbage collection can still be applied.

## AI Languages

Another feature of IPL is the *generator*, which is a procedure for computing a series of values. It produces one value each time it is called and is then suspended, so that it starts from where it left off the next time it is called. [see Control Structures section for more information] This idea was to turn up later in CONNIVER and similar languages.

Most of the programs of the early history of AI were written in IPL, in particular the version IPL-V. Many of these are described in *Computers and Thought* [FEIG 1963] including: the Logic Theorist, General Problem Solver, the Newell-Shaw-Simon Chess Program, EPAM, Feldman's two-choice decision model, SAD SAM, and Tonge's Assembly Line Balancing program. Some more recent programs written in IPL are Quillian's [QUIL 1968] and REF-ARF [FIKE 1971].

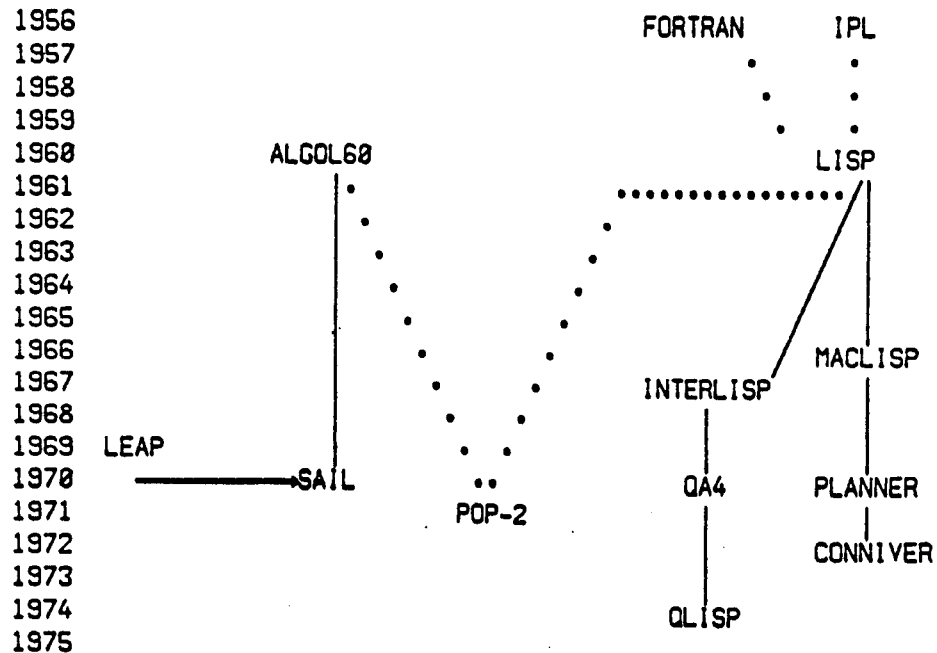


Figure 1. Historical chart of the development of AI languages. The dotted lines indicate major influences. The bold lines indicate stronger relationships.

Many of the ideas of IPL went into LISP. Unlike IPL, a rather low-level language with a *sequence-of-instruction* style although it was considered quite a high-level language at the time of its invention, LISP is a high-level language with an algebraic style inspired by, but different from, the algebraic notation pioneered by FORTRAN. During the design phase of LISP, some of the LISP ideas were implemented within FORTRAN leading to the language FLPL. FLPL was created so that some of the features of LISP would be in a working system that could be used for writing a plane geometry program. FLPL will not be discussed further. The first dialect of LISP to be widely used was LISP 1.5. In the late 1960s a number of new dialects of LISP were developed to meet the demand for sophisticated programming aids. The most important of these dialects are MACLISP and INTERLISP (derived from 940 LISP through BBN LISP). Another significantly used dialect is UCI LISP, an extension of STANFORD LISP 1.6 (itself an extended version of LISP 1.5) developed at the University of California at Irvine. UCI LISP will not be discussed further because

## AI Languages

its advanced features are basically a subset of those of INTERLISP. Other languages developed around this time were SAIL at Stanford and POP-2 at Edinburgh. SAIL is a version of ALGOL60 augmented by LEAP, an associative retrieval formalism. It was intended especially for those AI applications, such as vision, that require fast arithmetic. POP-2 is closer to LISP but with an ALGOL-like syntax and was designed to be suitable for implementation on medium-sized machines.

Toward 1970 a number of AI researchers developed formalisms to allow convenient expression of special purpose procedural knowledge for controlling problem-solving and deduction - PLANNER and QA4. CONNIVER embodied a reaction against PLANNER, giving the AI programmer lower level control while still maintaining many of the ideas of PLANNER. QLISP bore a similar relation to QA4, thought to a lesser extent, with an additional motivation being the desire to fit cleanly into the new version of INTERLISP.

A number of features first introduced in AI programming languages have been found to be useful in Computer Science more generally and have been included in many programming languages. It is difficult to trace the source of ideas with much certainty; frequently the same or similar ideas occur independently at different times and places, so some people probably would not agree with some of the items listed below. Also, the list is not intended to be complete.

Probably the most important idea so far has been list processing. This is one of the basic ideas of the area of data structures, with applications in many fields of Computer Science including complexity theory, databases, operating systems, and compilers. An early addition of list processing primitives to a more generally available language yielded SLIP, a subroutine package for FORTRAN [WEIZ 1963]. SLIP was used for some AI work but mainly for other fields. List Processing facilities were included in PL/I, based more on the primitives of IPL than the slightly higher-level ones of LISP. Most recent general-purpose programming languages, including PASCAL, have flexible data structures such as records which owe much to the ideas of list processing. Garbage collection is typically used to manage the storage of such structures.

The designs of LISP and ALGOL overlapped in time, with McCarthy being involved with both. McCarthy was influential in the decision to include both conditional expressions and recursion in ALGOL60, having already decided to include them in LISP.

Symbol manipulation has been critical for the field of algebraic manipulation. Indeed, many of them, including MACSYMA [MART 1971] and REDUCE [HEAR 1971], have used LISP as a base language. Some symbol manipulation facilities were included in the language COMIT developed around 1960. COMIT is primarily a string handling language with some features specially designed to help in the analysis of natural language text. It was used for some AI work. It has been superseded to a large extent by SNOBOL.

The applicative style of programming, pioneered in LISP (also used in APL), has been suggested by a number of people, including Backus [BACK 1978] to be a more appropriate style than the von Neumann machine-oriented languages that currently dominate.

## AI Languages

### 1. LISP

LISP was conceived around 1958 by McCarthy as a programming language for AI applications, in particular, for those based on the use of formal logic. The development of the precise form of LISP was determined by an interplay of practical concerns and of theoretical and aesthetic considerations.

The particular planned programs that motivated the language were a plane geometry theorem prover and McCarthy's *Advice Taker* [MCCA 1959]. Following is a list of some of the important features of LISP, along with reasons they were included. The reasons revolved around what was required for the originally planned programs and for other smaller programs that were considered during the design process, in particular, a program for symbolic differentiation.

1. List Processing. The originally proposed programs required the representation of formal language sentences and the ability to perform logical inferences for which the list processing techniques of IPL were well suited.
2. Algebraic Notation. The algebraic notation of FORTRAN was seen to be convenient because arbitrary subexpressions of sentences could be extracted using composition of primitive extraction functions.
3. Prefix Form. For internal use it is convenient to use prefix form rather than infix form, because for most applications it is necessary to know the main connective before deciding what to do with an expression. Thus, unlike almost all other symbolic systems, LISP is based on prefix rather than infix notation. For example, the infix expression  $4 * A * X + 5 * B$  is represented in the prefix notation of LISP as (PLUS (TIMES 4 A X) (TIMES 5 B)). It is useful for a symbolic differentiation program to have the "PLUS" first because it needs to know that it is dealing with a sum before it can know how to deal with the subterms involving "TIMES".
4. Conditional Expression. McCarthy introduced the conditional expression (*if then else*) when specifying legal move routines for Chess in FORTRAN.
5. Recursive Functions. The recursive nature of the differentiation of algebraic expressions made it natural to use recursive functions. Thus, differentiating the expression (PLUS 4 A (TIMES B X) X) can be done by recursively applying the function that does differentiation to each of the arguments of "PLUS," returning the sum of the results achieved as the answer.
6. Functional Arguments. For expressing the above differentiating process it was convenient to introduce the *maplist* function, which applies a given function to every element of a list. This required the introduction of functional arguments and a notation for functions; the  $\lambda$ -notation invented by Church [CHUR 1941] was chosen.
7. Garbage Collection. It was difficult to include explicit erasure of abandoned list structure in the natural definition of differentiation, so it was desirable for the system to collect abandoned list structure automatically. Reference counts could not be used because of hardware limitations, so the idea of garbage collection was invented.

## AI Languages

It became clear during the design stage that LISP provided a neater way of describing computable functions than did Turing machines or other methods used in recursive function theory. McCarthy considered it useful to have LISP expressions obey convenient mathematical laws to allow proofs of properties of programs to be constructed using normal mathematical methods. This objective was made possible for pure LISP although for practical programming reasons, functions with side effects were also included in the language implemented.

The *universal LISP function eval* was written to show that it was briefer and easier to understand than the description of a Universal Turing machine. Writing *eval* required a representation of LISP functions as LISP data. When it was realised that *eval* could be used as an interpreter for the language, it was hand coded in machine language, and LISP was thereby available. A consequence was to freeze the specification of the language, in particular, the list format of LISP programs, even though lists had never been the intended external format.

Following mathematical notation, LISP programs consist only of applicative expressions built up from variables and constants using functions. Thus, LISP is strongly function oriented. A function call is written as a list with the first element being the function symbol or object and the other elements the arguments to the function. This orientation gives LISP a distinctly different flavor than other programming languages. In fact, people who know other programming languages frequently have difficulty learning LISP, although many users coming from a mathematical background find it at least as easy a first language to learn as other programming languages.

Over the two decades since 1959, LISP has been the dominant language in AI, being used in diverse systems, some of which are far removed from the type of system for which it was originally envisaged. There are many reasons for this dominance, and it is unclear which were the most important. Some of the reasons are those described above, which influenced the design of LISP, but others were somewhat incidental to the design and are discussed below.

LISP was the first high-level language that provided convenient symbol manipulation and list processing facilities (IPL provided these facilities but was a fairly low-level language). This fact suggests that LISP may have become entrenched because of the costs of people adjusting to new programming languages, even when a newer language is generally recognized to be superior, as has happened with FORTRAN. The costs of changing to a new language have become more important with the introduction of large LISP programming systems, such as INTERLISP, which have many user facilities. A competing system would need to provide comparable facilities to lure users away from such a system, and supplying such facilities would involve considerable expense.

LISP has proved an excellent language for interactive programming. The basic LISP statement, a call to a function, is a convenient and very powerful command language. LISP lends itself to writing programs incrementally:— functions can be defined incrementally, without declarations; data structures and variable names (atoms) can be created dynamically. Using the interpreted form of functions, it is simple to make *try-and-see* modifications to functions. The implementors of LISP have tended to stay in fairly close contact with the users; the result has been that the LISP systems have tended to develop to meet the needs of the users. The simple framework of LISP has been accommodating to such developments down diverse dimensions.

The ability of LISP programs to obtain functions at run-time and to execute them using the LISP interpreter *eval* is a powerful facility, as is the ability to pass around and store functions. The many user facilities provided in current LISP systems, which will be described in the following section, also rely heavily on these techniques and the ability to manipulate user programs easily.

### AI Languages

The simple correspondence between internal and external forms of LISP functions has proved to be an advantage particularly when it is desired to manipulate functions by other LISP functions, for example, in list-oriented editors or in "smart" compile macros. This convenience has encouraged system builders to provide powerful tools to help users manipulate and manage their programs and has allowed users to tailor the system to their individual requirements.

As well as being good for interactive programming, LISP has proved itself a good, convenient language for programming in general. It is a powerful, concise language for expressing processes on arbitrarily sized and shaped list data structures, through recursion and conditional expressions. The program need not take account of such details as memory management because these are handled by the LISP system through such features as the garbage collector. Typically, distinct functions are used for distinct subproblems, which leads to modular programs. By defining functions to represent the operations that a programmer thinks of as basic, the main procedures can be written using these functions as primitives and thus reflect the steps that the programmer has in mind. In this way LISP can effectively be used to build higher level languages. Alternatively, it is easy to write interpreters of specially designed higher level languages represented as list structure. On the other hand, the basic functions are at a low enough level so that the user does not often feel limited by the language, and most implementations have allowed fairly convenient access to the machine and operating system facilities.

## AI Languages

### 2. Current General AI Languages

The medium-level, general-purpose programming languages commonly used in AI are SAIL, POP-2, and several versions of LISP, particularly INTERLISP and MACLISP. A characteristic of AI programs is that they tend to be large and complex. This fact has led to a demand for programming systems that provide extra support for developing and debugging programs. [See section on Environment] The various LISP systems have led the way in this direction because LISP has been the dominant AI language and because it lends itself to interactive use. LISP programs are written as list expressions and so are easily manipulable by support programs written in LISP, such as EDITORS and programs that temporarily redefine functions or expressions in functions, which are useful when debugging. The presence of these types of features in LISP systems has increased the demand for them in other systems where they were not such a natural development. POP-2 was developed by AI researchers in the United Kingdom because there were no good implementations of LISP on the machines they were using because they disagreed with some of the features of LISP and thought that others could be improved. It is still the most used AI language in the United Kingdom but it has never been used much in the United States.

#### INTERLISP

The system that has gone furthest in the direction of including user facilities is INTERLISP [TEIT 1978], which evolved from a series of LISP systems at BBN, notably 940 LISP in 1967 and BBN LISP in 1970. As well as highly developed versions of the facilities described above, it has the following features: a uniform error-handling system that allows some types of automatic error correction, such as spelling correction, entry to a special, flexible debugging facility, and handling of particular error conditions by user functions; CLISP - an alternative extensible expression syntax; PROGRAMMERS ASSISTANT - which keeps track of a user's commands (which are lists) and so allows selected commands to be undone, redone or edited, and redone; MASTERSCOPE - a cross-referencing facility that creates a database of a user's program and uses it to answer user queries; FILE PACKAGE - which keeps track of the state of a user's program, such as storing functions that have been altered, on a permanent file at the end of a debugging and editing session. These facilities have been written so as to enable the user to have a large amount of control over the way they behave, made possible because the user can specify general LISP functions to be executed at important points of the system, as well as being able to set many global variables which help determine the system's behavior.

In 1976 an implementation of Bobrow and Wegbreit's *spaghetti stack* [BOBR 1973] was incorporated into INTERLISP to provide convenient primitives for building very general control structures. [See section on Control Structures for further description].

The features that have been added to INTERLISP over the years give a strong demonstration of how receptive the LISP framework is to the incorporation of new ideas unrelated to the design considerations for LISP. One recent example of this is the introduction of an optional package for including type declarations in programs. Characteristically for a LISP system feature, the user can use as much of this as his personal style and particular program require. Type checking can be done when the program is running interpretively to aid in debugging, but need not be included in the compiled version of the same program if efficiency is considered more important than caution. On the other hand the compiler may take advantage of type declarations to produce more efficient code. General addition of types is allowed in a hierarchical manner, with the power of specifying arbitrary LISP functions for such things as type checking, or what to do when assigning to a variable of a given type.

## AI Languages

### MACLISP

MACLISP has been under continuous development since it was begun in about 1966 at MIT. MACLISP developed in parallel with INTERLISP. Fairly frequently, a feature in one that was found to be useful was implemented in the other. There has, however, been more of an emphasis on efficiency in time and space utilization in MACLISP. Some of the main forces pushing in this direction were the increasing requirements of MACSYMA and, to a somewhat lesser extent, of higher level AI programming languages such as MICROPLANNER and CONNIVER, which were implemented in MACLISP. In contrast to INTERLISP, where most of the user facilities are resident in the system's core image, the user facilities in MACLISP are in separate programs or are automatically loaded when required. This feature is acceptable because it has a fast loader and the operating system has good facilities for skipping between one program, such as an editor, to another, such as MACLISP. The code produced by the MACLISP compiler is very efficient, on the order of twice the efficiency of INTERLISP compiled code. In particular, effort was directed at producing efficient compiled arithmetic; and by 1974 this objective had been accomplished to such an extent that compiled arithmetic was comparable to that of a FORTRAN compiler.

### SAIL

In 1969 at Stanford, SAIL resulted from combining LEAP, an associative retrieval formalism, with a version of ALGOL 60. It was designed for those AI systems, such as vision and speech understanding systems, that required fast arithmetic and some of the facilities available in LISP. Some of the facilities that make SAIL convenient to use are its string-handling capabilities, a macro facility, and convenient I/O. The associative database consists of attribute-object-value triples, which were used to do the same sort of things that LISP had done with properties of atoms. However, they are more powerful than LISP properties in that any of the keys can be used to retrieve values for the other slots. This and other features they shared with the PLANNER-like languages discussed in the next section.

Around 1973 some more facilities were added, prompted by other languages, especially LISP. A restricted type of list was introduced that did not allow lists as its elements. This deficiency was overcome by the introduction of records. However, unlike LISP, routines for manipulating the lists and records and for performing I/O on them must be supplied by the user. Another facility added was a very powerful interactive debugging system: BAIL. General control structures, including multiple parallel processes, were also added.

### POP-2

POP-2 is a language with many of the properties of LISP but with an ALGOL-like syntax. It was developed around 1971 by Popplestone, Burstall, and others at the University of Edinburgh. Like LISP it is interactive and allows general manipulation of functions. It was intended for efficient implementation on the medium-sized machines available to the authors. Thus, it relies on packages that have to be explicitly loaded when required, to supply user facilities. It allows partially applied functions to be created that can be used to do the same sort of things as FUNARGs. It also has dynamic lists, which are lists with a function at their end that is automatically called when more list elements are required. This feature can be used to implement *generators* and allows such things as I/O devices to be treated as lists. The stack can be explicitly pushed and popped, which allows functions to return multiple values fairly cleanly.



## AI Languages

### 3. Higher Level AI Languages

At the end of the 1960s and the beginning of the 1970s, a number of languages were developed that were designed to provide primitives useful for programs that require problem-solving and deduction facilities. An important idea behind them was that techniques that were thought to be generally useful should be generally available, so that new systems could be built more easily.

#### PLANNER

PLANNER was designed by Hewitt from about 1967 to 1971 as a foundation for problem-solving [HEWI 1971]. It was built around the idea that the essential thing about knowledge was how it was used in achieving purposes. Hewitt tried to develop a formalism whereby any piece of knowledge could be naturally expressible in a procedure that would use it where appropriate. This approach was partly in reaction to the uniform proof procedures that were then widely used, where domain knowledge is represented as propositions that are manipulated by a domain independent theorem prover to produce the desired result. Some of the ideas introduced by Hewitt are as follows. Goal-directed invocation of procedures, whereby each procedure has a pattern that represents the goals that it can achieve, so that it can be called whenever it is relevant to achieving a desired goal. He emphasized the importance of backtrack control, whereby choices can be made and their implications explored, while still maintaining the capability to reject these choices together with all their consequences, in order to try other choices based on the reasons for rejection [see article on control structures for further description of backtracking]. Rather than having an explicit goal tree, as in previous problem-solving systems such as STRIPS or GPS [ERNS 1969], the goal tree is implicit in the control structure of the program.

Only a portion of PLANNER was actually implemented as MICRO-PLANNER in 1972 [SUSS 1971]. It included pattern-directed invocation of procedures and automatic backtracking. Each procedure is indexed in an associative database by a pattern indicating what it is useful for. There are two types of procedures: *consequent theorems* and *antecedent theorems*. A consequent theorem is called when its pattern matches a subgoal to be solved. An antecedent theorem is called when its pattern matches an assertion added to the database (there is another kind of antecedent theorem for assertions deleted from the database). Antecedent theorems can be used for forward reasoning and for keeping the database consistent. Another use is for setting up expectations. For example, suppose that it is important when one assertion added to the database is followed by the addition of a certain other assertion to the database. This can be checked by having the first assertion trigger an antecedent theorem that sets up an antecedent theorem to be triggered if the second assertion is, in fact, added. The reason for calling them *theorems* comes from analogy with the way that theorems were used (and are still used) in AI inference systems. Thus, a theorem of the form "A  $\supset$  B" is used in two ways: if it is desired to show "B" then this theorem allows you to set up the subproblem of proving "A", which if true will allow you to conclude "B" is true also; alternatively, if you find that "A" is true the theorem allows you to conclude that "B" is true.

An important program written in MICRO-PLANNER is SHRDLU [WINO 1972].

#### CONNIVER

CONNIVER resulted from a reaction by Sussman and McDermott to what they considered excesses in MICRO-PLANNER [SUSS 1972, MCDE 1974]. It retained many of the ideas of

## AI Languages

PLANNER, but at a lower level so that fewer of the mechanisms were imposed on the user. In particular, Sussman and McDermott objected to the use of automatic backtracking. In MICRO-PLANNER, most of the information gained from following a wrong choice was lost on failure. The result was that programs tended to get bogged down in blind back-tracking.

CONNIVER introduced the idea of a tree of contexts in the database. This tree was used to represent the different situations for different choices of actions at the same time. A powerful pattern matcher was implemented that allowed flexible associative access to the database. A version of the *spaghetti stack* was also implemented, using lists rather than a stack, so it was rather inefficient. Together, the spaghetti stack and context tree allow, for example, a problem-solver to suspend a process working on one subproblem and to continue work on another subproblem that seems more promising, and then, at some later point, to resume the process working on the original subproblem if necessary.

HACKER [SUSS 1975] was written in CONNIVER.

### QA4 and QLISP

QA4 was developed by Rulifson and others [RULI 1971] around the same time as MICROPLANNER. It was preceded and influenced by QA1, QA2 and QA3, which were all developed by Cordell Green. QA1 was an attempt at formalizing the ideas in Raphael's SIR program, that Raphael had seen to be too ad hoc and as a result difficult to extend. Before this was completed it was seen by Raphael and Green that the ideas of resolution theorem proving were suitable for the task. This led to the development of the resolution theorem based system of QA2. QA2 introduced the now standard method for extracting answers from the process of proving theorems with existentially quantified variables - i.e. values for the variables which satisfy the theorem. QA3 was an improved implementation of QA2 in which Green explored how to tackle various types of problem using a resolution theorem prover [GREE 1969], including program synthesis, verification, and problem-solving. QA3 was also used in STRIPS and ZORBA.

QA4 was intended to overcome certain problems found in using QA3 - specifically the difficulties of guiding search. It was difficult to specify procedural and domain dependent knowledge in a way that the theorem prover could use. It was seen to be useful to have theorems or procedures indexed by their purpose, an idea which led to the implementation of pattern directed invocation similar to that of PLANNER. QA4 was the first language to develop the idea of representing assertions uniquely, in the way that atoms are represented uniquely, so that properties can be associated with assertions. It used a context mechanism like that of CONNIVER and also had a general control structure. To make it more widely available and to take advantage of new facilities in INTERLISP, a cleaner implementation was embedded in INTERLISP and called QLISP. When the *spaghetti stack* was implemented in INTERLISP, QLISP was modified to take advantage of it. Unlike MICRO-PLANNER and CONNIVER, which are interpreted languages on top of LISP, QLISP is basically a subroutine package for INTERLISP. This makes it much easier to mix QLISP with INTERLISP.

Additional features of QLISP are extra data types like sets, tuples, and bags, together with procedures for handling them, and the ability to use them in pattern matching. Pattern matching plays a larger role in QLISP than in CONNIVER, being used for such things as constructing data. QLISP also makes the distinction, blurred in PLANNER, between finding the truth of an assertion by doing a simple look-up in the database and finding it by deduction using consequent procedures (PLANNER theorems). Everywhere that a pattern-directed procedure call may occur, QLISP allows

## AI Languages

the user program to specify a *team* of procedures that should be considered. A team consisting of a single procedure corresponds to the traditional style of programming, whereby a "subproblem" is solved by calling a specific function. On the other extreme, a team consisting of all the goal-achieving procedures corresponds to the nondeterministic style of PLANNER, whereby the subproblem is announced and any procedure whose purpose matches the subproblem description may be called.

### POPLER

Developed by Davies in 1973 at the University of Edinburgh, POPLER is a language based on PLANNER and embedded in POP-2. It has an implementation of the *spaghetti stack*. It makes the same distinction as QLISP between testing the database and calling consequent procedures, and it further distinguishes between those procedures that deduce the value of an assertion in a world model from those that achieve the assertion as a goal, producing a new world model from the old model.

## AI Languages

### 4. Conclusions

The higher level AI languages have had an important influence on AI. The most striking thing about them, though, is that none are being used much now. Most were used to write a few important systems that demonstrated the utility of the *FEATURES* in the system, and sometimes the limitations of the system, and then were not used again. Now most of the programs of the type that were written in these languages are written in LISP. Frequently, however, some of the features of the higher level AI languages, such as the associative database or backtracking, are implemented in the new programs, although in a form specialized for the particular task.

There are two main reasons for this disuse of higher level languages. First is the problem of time and space efficiency. Second, the concepts built into the languages have not coincided closely enough with those that researchers later wanted for AI programs. Although researchers typically want to use some of the features of the AI languages, they may not want to pay the price of having all the unwanted features around; and the modularity of the systems is not sufficiently good to allow the extraction of the desired features (which has also been a criticism of INTERLISP). If what is wanted is not quite what is provided, then the user may have to struggle with the system to get what he wants. It has been quite difficult for an embedded system such as MICROPLANNER or CONNIVER (QLISP is not embedded but is a set of functions callable from INTERLISP) to communicate cleanly with the host language, mainly because the control and variable stacks are not shared. Also, for a specific task, there is often a more efficient way of implementing a feature than the fairly general method used in the programming language. Occasionally a commonly used feature has been implemented in a lower level language, such as in the case of the *spaghetti stack* in INTERLISP.

This situation has resulted in some wasteful duplication of work. Such waste might be avoided in the future by having more packages with smaller scope and with fairly specialized application. Then a program builder could select those required. Another possibility is that some intermediate level language might be designed that allows higher level constructs to be defined easily.

#### Some Current Directions

A number of programming languages for AI have been developed since those described above. None of these have had widespread use and it is too early to know their significance. Three categories are considered:— new LISP dialects, languages based on logic, and knowledge representation languages.

A number of LISP machines are being developed using dialects similar to current dialects such as MACLISP and INTERLISP. Different machines are being developed at M.I.T. AI Laboratory, BBN, and Xerox PARC. The standard use of lexical binding rather than the usual dynamic binding for LISP has been proposed and implemented in SCHEME [STEE 1978], a variant of MACLISP. Steele and Sussman argue that lexical binding is more natural for commonly used AI control structures such as coroutines and generators, and that the full generality of the spaghetti stack is not necessary for AI applications. The latter is supported by the fact that AI systems have not so far taken much advantage of the full generality offered by the spaghetti stack. A major new LISP implementation, NIL, for new large address space machines uses SCHEME as a model, in particular, its lexical binding.

Two languages based on first order predicate calculus are PROLOG and FOL [WEYH 1979].

## AI Languages

PROLOG programs consist of "axioms" in first order logic together with a theorem to be proved. The axioms are restricted to be implications with the left and right hand sides in *horn clause* form. If the theorem contains existentially quantified variables the system will return instantiations of these which make the theorem true (if such exist) using methods developed from those of QA3 [GREE 1969]. The style of programming is similar to that demonstrated in QA3 and, to a lesser extent, PLANNER. Automatic backtracking is used but the programmer may add annotation to control which order clauses and axioms are considered. A compiler has been implemented for PROLOG that enables programs in a variety of domains to execute in terms of time comparably to corresponding compiled LISP programs. Another direction of logic has been explored through FOL: the uses of meta-theory. FOL is primarily a proof checker that accepts logic statements and proof step commands that can be carried out and tested for correctness. However, it provides a powerful theory manipulating structure that allows the building of arbitrary meta-theory. Thus, for example, a theorem may be proved not only within a theory but also with the help of the meta-theory. Proving a theorem by going through the meta-theory corresponds closely with executing a procedure to produce the theorem.

The representation languages have been based on the idea of frames discussed in the Representation Chapter. These include KRL and FRL, which are built on top of LISP. These languages provide frame data structures and procedures for manipulating them, including pattern matchers. Procedures can be attached to slots in the frames to be called in particular circumstances involving the slot, such as when the slot is filled, changed or examined. The languages do not provide a separate language for specifying procedures; the host language being used for this.

### REFERENCES

Bobrow, D. G., "Requirements for Advanced Programming Languages for List Processing Applications", *Communications of the ACM*, July, 1972, pp. 618-627.

Bobrow, D. G. and Raphael, B., "New Programming Languages for Artificial Intelligence Research", *ACM Computing Surveys*, September, 1974, pp. 153-174.

Bobrow, Daniel G., and Raphael, Bertram. "A Comparison of List-Processing Computer Languages," *Programming Systems and Languages*, ed. Saul Rosen. New York: McGraw-Hill, Inc., 1967.

Daniel G. Bobrow and Bertram Raphael. *A Comparison of List-Processing Computer Languages*, *CACM*, 7:4, April, 1964, 231-240.

D. G. Bobrow and B. Wegbreit, "A Model and Stack Implementation of Multiple Environments", *CACM*, Vol. 16, No. 10, pp.10-39, October, 1973.

Burstall, R.M., et al, *Programming in POP-2*, Edinburgh University Press, Edinburgh, 1971. Contains a primer and the POP-2 reference manual.

Davies, D. Julian M. *POPLER 1.5 Reference Manual*, University of Edinburgh, TPU Report No. 1, May 1973.

Ernst, George W., and Newell, Allen, *GPS: A Case Study in Generality and Problem Solving*, Academic Press, New York, New York, 1969.

Feigenbaum, E. and Feldman, J. (eds.), *Computers and Thought*, McGraw-Hill, 1963.

## AI Languages

Feldman, J.A. and Rovner, P.D. An ALGOL-based Associative Language. CACM, 1969, 12:8, 439-449.

Green, C. C. "The Application of Theorem Proving to QA Systems", Stanford Technical Report CS 138, SAIL Memo AI-96 1969.

C. Hewitt, "Description and Theoretical Analysis (using schemas) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot", Phd Thesis, MIT, Feb., 1971.

C. Hewitt, "Procedural Embedding of Knowledge in PLANNER", 2nd IJCAI, 1971.

Martin and Fateman, The MACSYMA System, in (S. Petrick, ed.) 2nd Symposium on Symbolic and Algebraic Manipulation. NY: ACM SIGSAM. pp 59-75, 1971.

McCarthy "History of LISP", ACM Sigplan Notices, 1978, 13(8), 217-223.

McCarthy "Recursive Functions of Symbolic Expressions and their Computation by Machine", CACM 3, 4, April 1960, pp. 184-195.

Moses, J., "The Function of FUNCTION in LISP", MIT AI Memo No. 199, June 1970.

D. V. McDermott and G. J. Sussman, "The Conniver Reference Manual", MIT AI Memo No. 259A, January 1974.

Popplestone, R.J. "The design philosophy of POP-2", in Machine Intelligence 3, pp. 393-402. D. Michie, ed. Edinburgh University Press, 1968.

Pratt "LISP: An Amicus Curiae Brief"

R.Reboh, E. Sacerdoti, "A Preliminary QLISP Manual" Stanford Research Institute AI Tech. Note 81, 1973

Reiser, J. SAIL Users Manual, Stanford A. I. Memo AIM 249. Stanford, Calif.: SAIL, Stanford University, 1976.

Rulifson, J. F., Waldinger, J. A., Derkson, J. A., "A language for Writing Problem Solving Programs," Proc. IFIP Congress '71, Ljubljana, Yugoslavia, 23-28 August 1971.

Sandewall, E. "Programming in an Interactive Environment: the LISP Experience", ACM Computing Surveys

Smith, R.L. TENEX SAIL, (Technical Report No. 248) Stanford, Calif.: IMSSS, Stanford University, 1975.

Smith, Douglas K. "An Introduction to the List-Processing Language SLIP," Programming Systems and Languages, ed. Saul Rosen. New York: McGraw-Hill, Inc., 1967.

G. J. Sussman, T. Winograd, and E. Charniak, "MICRO-PLANNER Reference Manual", MIT AI Memo 203A, DEcember, 1971.

**AI Languages**

Sussman, G. J., A Computer Model of Skill Acquisition, American Elsevier Publishing Company, New York, New York, 1975.

G. J. Sussman and D. V. McDermott, "Why Conniving Is Better Than Planning", MIT AI Memo No. 255A, April 1972.

W. Teitelman, "INTERLISP Reference Manual", Xerox Palo Alto Research Center, December, 1978.

W. Teitelman, "Toward A Programming Laboratory", Proceedings IJCAI3, Stanford, California, August, 1973, pp.1-8a.

Weizenbaum, J. "Symmetric List Processor," Communications of the ACM, VI (September, 1963).

Weyhrauch, R. "Prolegomena to a theory of mechanized formal reasoning", Stanford A.I. Memo, 1979.

White, J.L., "A Historical Perspective on MACLISP"

Winograd, T, Understanding Natural Language, Academic Press, Inc., New York, New York, 1972.

## I. OVERVIEW OF LANGUAGE FEATURES

This overview article attempts to explain the motivation for having special AI languages in the first place. Then, in the next four articles we will look more closely at various aspects of general-purpose AI languages. The purpose of the articles is twofold: to present some powerful AI programming techniques, and to show how they are embodied in some major AI languages.

The aspects covered are data representations, control structures, pattern matching capabilities and the programming environment. Each aspect will be introduced by a few pages outlining important general features, for instance, data types, data bases and context mechanisms in the case of the data structure aspect. An attempt will be made to motivate the inclusion of specific features in AI languages, by discussing some of the applications they find.

Six AI languages are covered: basic LISP, PLANNER, CONNIVER, QLISP, SAIL and POP-2. In each aspect article, the introduction is followed by a survey highlighting interesting points about each language that fall under the that aspect. While the approach will be largely comparative, the comparisons are intended to bring out significant tradeoffs and alternatives rather than to tell the reader which language is "best" or even best for a particular purpose. The latter sort of comparison would be difficult and controversial, and of limited usefulness since most of the languages are of very limited availability.

No previous knowledge of the languages is assumed here. Nor is any attempt made to help the reader learn how to program in them. Experience in programming and familiarity with basic concepts (e.g. data type, variable) will be very helpful but not absolutely essential to understanding the discussion. At certain points features of AI languages will be contrasted with those generally found in other computer languages.

Readers wishing to do some AI programming would be best advised to consult with some knowledgeable person (assuming there is more than one AI language available to the prospective programmer, to begin with!) In general, the languages with less in the way of fancy AI features - LISP, SAIL and POP-2 - are more accessible, better supported, and faster running than the more advanced PLANNER, CONNIVER and QLISP. It would probably be best to start with one of them.

### I.1. Why Have AI Languages?

The Historical Overview article traces the development of the early AI languages in response to the fundamental needs for list processing, recursion and symbolic (as opposed to purely numeric) computation. This early development took place in the late 50's and early 60's, about a decade before the developmental period of the languages covered here (with the exception of LISP, which is an early language but has remained in general use.) LISP, PLANNER, CONNIVER, QLISP, SAIL and POP-2 all offer list processing, recursion and symbolic computation as basic facilities; indeed these features have "spun off" and are included in many general-purpose computer languages developed since the early 1960's.



Our focus in these articles is on more advanced features, built on the basic features just mentioned. These more advanced features represent in a sense a "raising of expectations" concerning what the AI programmer should be able to expect from a programming language. In writing their programs, AI system builders tend to use certain general methods such as, for instance, depth-first search. If the language had such methods embodied as language primitives, then the programmer would be spared the burden of coding them and could concentrate on higher-level issues. Also, a well-chosen set will tend to lead the programmer to structure his/her thinking and programs in a good style. PLANNER, CONNIVER and QLISP especially are attempts to integrate a powerful set of language primitives.

There has been a tendency for use of the more advanced languages to fall off in recent years, after considerable initial enthusiasm, and most AI programs are still written in the old standby LISP. This phenomenon seems largely due to the *restrictions* that inevitably accompany the codification of an (advanced AI) technique into a particular language, together with the *inefficiency* that results from trying to make the language constructs as general as possible. As you can see, these two constraints work against each other; the more general and powerful a language is made, the slower it will run, and if generality is sacrificed in the name of efficiency, users are likely to find that the language can't do what they want. Sandewall [sand78] comments that PLANNER, CONNIVER and QLISP are clumsy; the particular offerings of each satisfied few users, and they are not used much anymore.

A programmer building up his/her own high-level constructs can hand-tailor them to do just what is required for the idiosyncratic needs of his/her system, and no more, and optimize efficiency within that range. It seems in principle impossible for an advanced AI language to match this. People respond to this situation in various ways.

Use an advanced AI language anyway, judging its convenience to be worth the inefficiency. (Though as said before, these languages are often inconvenient as well as inefficient.)

Write one's own special-purpose AI language, or borrow someone else's. Special languages have been developed for application in robotics, production systems [rych76], natural language parsing [yon parser language from NL chapter], and other areas. Many rule-based systems, for instance MYCIN [yon applications] or DENDRAL [yon applications], use an *ad hoc* domain-specific language to express the rules.

Develop a new advanced AI language (and hope it will not be too inefficient to use.)

Wait patiently, expecting that in a few years, computers will be fast enough that inefficient languages can be tolerated.

Work on an automatic programming system with the idea of eventually being able to automatically choose an efficient implementation of advanced AI constructs. [yon AP chapter of Handbook]

## **I.2. How AI Programs Differ from Other Programs**

The purpose of this section is to make general remarks about the peculiar characteristics of AI programs, contrasting them with "ordinary" programs, in an attempt to provide perspective for the special language features discussed in the four articles that follow.

From the start, it must be admitted that the class of "AI programs" is not clearly defined.

It is a matter of opinion whether MACSYMA [yon applications], for instance, is an AI program. Here we will evade the issue by sticking to things that are clearly AI.

### **Data Structure**

We begin with remarks about data structures in AI. Some feeling for the difficulties of designing data structures for an AI program may be gained by examining the Knowledge Representation chapter of this Handbook [yon rep]. The common sort of structure known as a "semantic net" will serve to illustrate.

As a first cut a semantic net can be represented by a set of records, one per node of the net, having a field for each kind of link that can emanate from the node. But a host of difficult issues soon arise, exemplified below.

Perhaps there will be a large number of kinds of link, so that records are impractical. In that case a property list, i.e. a list of property name-property value pairs, should be used instead.

Suppose we need to find all nodes having a certain kind of link; should there be an index?

How is inconsistency in the net to be avoided as new facts are added? E.g. if "Francis isa man" is already encoded, what happens when "Francis isa woman" is asserted?

Given the difficulties with AI data structures, we would like our languages to help us as much as possible, by offering a wide range of data types (each with appropriate operations and semantics), facilities for dealing with large data bases, and the like. But to the extent that an AI language offers standard data facilities, it will have to embody choices on difficult issues such as those listed above, and it is not to be expected that the choices will be optimal for all applications.

### **Control Structure**

Let us turn now to the control structure needs of AI programs. AI programs are generally quite large and like most large systems, consist of many modules each carrying out a certain kind of

subtask, and the subtasks have to be sequenced together so as to perform the whole task. How the task can be divided up, and how the sequencing can be effected, are highly dependent upon the control structure facilities offered by the language used.

In a typical block-structured language, modules must obey a strict calling hierarchy. At any point in the execution there are very limited options as to what to do next, namely options that can be accommodated by conditional statements (if-then statements, case statements, loop exit tests.) This discipline becomes a Procrustean bed when applied to most AI programs. As an extreme case consider HEARSAY-II [your article in Speech chapter.] There, the action of one expert module modifies the database (blackboard) and leads in unpredictable fashion to the activation of some other module. The system is data-driven.

In HEARSAY-II each activation of a given module is independent of its last activation. In other systems it is common for modules to have an internal state which survives from activation to activation. In control structure terms, several procedures must be suspended while another one runs, then it suspends and control passes to one of the suspended procedures (selected somehow.)

### Pattern Matching

So far we have talked about the divergence of AI programs from other programs in two major aspects: data structure and control structure. Individually each tends to be looser and more complicated in AI programs, and there is also a greater complexity in their interaction. On the one hand the control structure becomes increasingly "data-driven". On the other hand more powerful operations for accessing data are introduced. A striking fact about recent AI languages is the extensive use of pattern matching to mediate both directions of interaction, and so we have taken up pattern matching as a separate subject.

In our control structure example, HEARSAY-II, each module has a set of *production rules* [your somewhere where production rules are explained] whose condition part is a pattern. If some group of syllables, words or whatever in the database match the pattern, the rule can activate.

In our data structure example, the semantic net, it is common to think of the net as a graph (with labelled edges), the retrieval pattern a graph of the same type, and the pattern matching process as a subgraph isomorphism problem. That is all very well, but the best known algorithms for subgraph isomorphism require exponential time so it is not practical as a pattern matching method. Adjustments must be made, to get along with less powerful pattern matchers.

### Programming Environment

From the discussion so far it should be apparent that AI programs are not easy to design and debug. This brings up the fourth aspect on which we will compare AI languages, their programming support facilities. Because they usually are developed as a research project, AI programs tend to undergo drastic revisions. They are also quite large so the programmer cannot keep everything

straight just in his/her head. AI languages have pioneered in the interactive mode of programming. Considerable effort has gone into the development of extensive programming environments for some of the major AI languages, providing editors, trace and debugging packages, and other aids in the construction of large complex systems.

### 1.3. Languages Covered

High-level computer languages tend to fall into two broad classes. The "ALGOL-like" or block structured languages are commonly recognizable by the many "begin" and "end" statements, delimiting blocks, which their programs contain. These languages usually allocate space for variables, arrays and other data at compile time, so that once a program is compiled the space available for its data is fixed. The nested structure of blocks is used to define the *scope* of program variables, that is, the region of the program in which they are accessible, and similarly defines which procedures can call which other procedures. Only one of the six languages, SAIL, is of the block-structured type.

SAIL (Stanford Artificial Intelligence Language) resulted from extending ALGOL 60 with an associative database and real-time and multiprocessing capabilities. Its applications are chiefly in vision and speech research (plus a good deal of non-AI application.) SAIL includes some list-processing data types and these have dynamic allocation (see below.)

The "LISP-like" or characterized by dynamic allocation and dynamic scoping. Dynamic allocation means that the space to be used by a data object is not fixed ahead of time, but is allowed to grow and shrink as needed. Dynamic allocation is almost essential for list-processing. Dynamic scoping means that any procedure can call any other, and variable values are passed down the control chain rather than being determined by the static block structure. That is, once a variable is declared or otherwise allocated in, say, procedure A, it can be accessed from within any procedure B that A calls, or C that B calls, etc., regardless of where A, B and C appear in the actual program text. LISP-like languages are usually interpreter-based and may have a compiler also. LISP, PLANNER, CONNIVER and QLISP are all LISP-like languages, indeed they are all built upon LISP itself in some way.

LISP is one of the very oldest AI languages, dating back to about 1959. It introduced many ideas fundamental to later AI language designs, and in various dialects it is still the most widely used AI language. To avoid confusion of dialects, we speak of "basic LISP" which corresponds roughly to the LISP 1.5 defined in 1962 [mcca62] [weis67] [yon LISP article].

PLANNER is a problem-solving language that carries out automatic deductions from a database of assertions and theorems to satisfy some goal. The PLANNER ideas were developed by Carl Hewitt at MIT, and a subset of them were actually implemented in 1971.

CONNIVER was developed at MIT shortly after PLANNER and largely in reaction to it. CONNIVER is basically a problem-solving language, very similar to PLANNER in some respects but intended to correct the perceived excesses of PLANNER. Its data structure and control structure are more flexible. Neither PLANNER nor CONNIVER are much used any more.

QLISP is also a problem-solving language, developed at SRI independently of PLANNER and CONNIVER. It relies heavily on data base methods and a powerful pattern matcher to solve problems.

POP-2 shares characteristics of both classes, in most respects closer to LISP than to ALGOL. It features a small and flexible core language, reminiscent of basic LISP but with more data types, that can be extended in different directions by library packages. POP-2 was developed at Edinburgh and is the major AI language used in Britain.

With the exception of LISP all six languages were developed in the late sixties and early seventies and have reached a more or less mature form as of now. Significant AI programs have been written in each of the six languages and those not still in active use have at least been highly influential.

By no means is our list of six exhaustive. Rather than try to give a complete survey and comparison of languages used in AI we restrict this discussion to the small group of major ones intended for general application in AI. We have not included any of the newer languages that are still in process of development or have not seen sufficient use to make clear what is most important about them.

We will now take up the four chosen aspects of AI programming technique one by one, in more concrete detail, and show how each language provides some of the techniques as standard language features. Keep in mind that the language features are seldom at the level of global system structure, e.g. heuristic search or cooperating expert modules. They are more like building blocks. An exception is PLANNER, which provides the global structure of goal-driven backtracking; but as we shall see, that leads to problems.

## II. DATA STRUCTURES

### II.1. MOTIVATION

The general goals of a data representation are

1. To mirror certain entities which the programmer has in mind, in a natural and convenient way.
2. To be efficient in storage space and in the time required to operate on the data.

These goals often come into conflict. In artificial intelligence programs, the entities concerned tend to be rather large and complicated when represented as data structures. But complex data structures are inefficient, so there is a tendency to sacrifice some naturalness and convenience in order to make do with simpler data structures. The data structure offerings of various languages may be viewed as compromises seeking some kind of optimum. In the context of AI languages we will discuss three data structure issues: data types, problems of storage and retrieval in large data bases, and division of the data base into contexts.

### II.2. Data Types.

Every computer language offers a selection of data types. Since the beginning, AI languages have included some "list" data type or types because of the fundamental importance of list processing in AI programs. We won't say much about list data here, since all six languages have quite similar list structures and operations (with the exception of SAIL which is somewhat deficient.) The reader is referred to [yon LISP article], [yon Hist Overv. article] for further discussion of lists in AI.

Here, our main concern will be with new data types that some of the languages have introduced, including

set

bag (like a set, but with repeated elements)

tuple (basically a list of fixed length. A tuple with 2 components is an ordered pair, for instance.)

record (like a tuple but its components are accessed by name, so the programmer need not keep track of their actual position.)

function (a procedure treated as a piece of data. See below.)

Some languages also allow the user to define his/her own data types.

The function (or procedure) data type is of particular importance. The reader may be puzzled what it means to say a function is a datum. Basically, a the datum used is a pointer to some internal representation of the code of the function. In LISP, that internal representation would be list structure. Programs can manipulate the function in various ways, such as assigning a variable to the pointer value, passing the pointer as a parameter, or modifying the function's code. And when desired, the function can be applied to some arguments and executed. The many uses of function datums will be illustrated below. All six languages offer them in some form.

#### Data Base storage and retrieval.

In the last decade there has been a shift away from AI systems built around general problem-solving methods to systems that rely on a large knowledge base and relatively weak problem-solving or inference methods. [ref Feigenbaum's Knowledge Engineering paper in 5IJCAI.] Accordingly, there is a growing concern with questions of how to organize and access large data bases. The term "knowledge base" indicates that these AI data bases are not merely big files of uniform content.

The ideal from the programmer's point of view would be an AI language and supporting system able to retrieve data according to any specification that might be constructed by his/her program. Thus a detective program might request a list of all white males in Los Angeles between 5'9" and 5'11" in height with red hair who work at an auto garage. The programmer would also like the language system to perform certain updating and bookkeeping tasks, even to the extent of automatically detecting inconsistencies in the data base. Most of this remains a dream. There are severe limits on how much a language can do without becoming hopelessly inefficient (or perhaps too complicated to design in the first place.) The six languages covered here have all stopped at some compromise set of data base features which are supposed to be of considerable utility, yet feasible to implement. If a programmer wishes his/her program to have a data base in the form of, say, a semantic net (which none of the languages offer), [yon Representation chapter] then he/she will have to construct the semantic net primitives from whatever data base primitives the language does provide.

The selection of a good set of data base primitives is crucial. There is still much debate on the relative utility of alternative schemes. One common scheme is multiple indices. Incoming data are indexed by several of their attributes, for instance author, title and subject if the data are books, and the retrieval process is simple and fast. An attribute and its value are specified; the system looks in the appropriate index, finds the entry (if any) for that value, and retrieves all items listed for that entry.

McDermott [mcde75] argues that multiple indexing can be implemented efficiently even when the data base is split into contexts (see below) and proposes a detailed scheme for doing it.

A more advanced scheme which was introduced by Carl Hewitt in his PLANNER language, and has spread widely since then, retrieves according to a pattern - a sort of structural sketch of an item, with some pieces left undefined. The system retrieves all items that fit the general pattern.

Implementation must be done carefully, though. The simpleminded approach of scanning the entire data base testing each item against the pattern - the British Museum algorithm - is out of the question for really large data bases.

Beyond pattern-directed retrieval one can speak of "semantically-directed retrieval" in which items are somehow selected by their meaning, rather than their explicit structure. One can also raise legitimate objections to the whole idea that data bases should consist of a set of separate items. We will leave these issues aside, since none of our six AI languages have attained such a level of sophistication. [yon Knowledge Representation chapter.] There is, however, one quite standard technique which starts to move away from the crude notion of a set of items. This is the division of a data base into *contexts*, which will be introduced at some length now.

### Contexts

The basic idea is to replace the global data base with a tree of distinct data bases called *contexts*. See the figure. The reason for arranging contexts in a tree, rather than a line or graph or whatever, has to do with the usual interpretation placed on them, that they each represent a distinct state of the world or set of assumptions about it. As the world changes or new assumptions are made, a context naturally gives rise to "descendant" contexts which differ slightly from each other and from their common parent. Conceptually, each context is a full data base in its own right; in reality most of the information in a given context will be the same as in the (parent) context just above it, so to save space only the differences are actually stored. The root of the tree, of course, must really be a full data base.



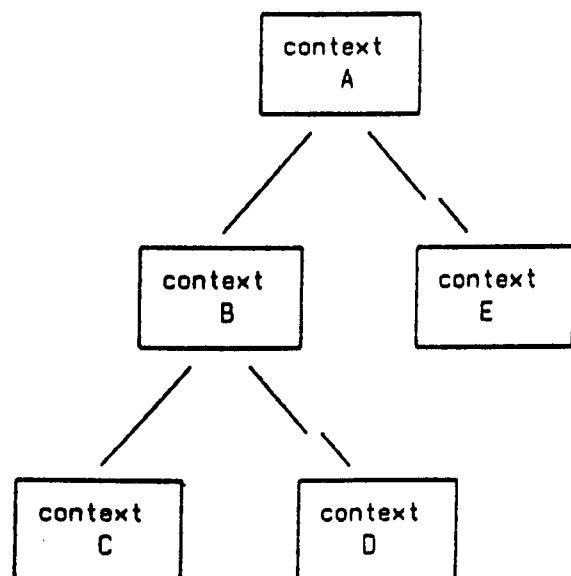


Figure n.

Contexts are very useful in hypothetical reasoning. One instance is robot planning; here the robot can simulate alternative sequences of actions, creating contexts to represent the states of the world that result. If the consequences of an action are undesirable, the robot planner can delete the context and try something else. (The robot has to *simulate* the actions, because the real world does not allow you to "take back" an action once you do it.) We need to introduce some terminology for operations on contexts. To begin with, a process at any given time is using one specific context as its data base. This is called its current context and it is said to be "in" the current context. Referring to the figure, assume the current context is A and no other contexts exist yet. We will illustrate each term by its effects on the tree in the figure.

**PUSH** Create a new context as a descendant of the current context, and enter it. Initially the contents of the new context are identical to its parent. If we PUSH, in the figure context B is created and becomes the current context. Suppose we PUSH again; now C is created and is the current context.

**SPROUT** <cxt> Like PUSH, except that the new context is a descendant of the designated <cxt> and is not automatically entered. SPROUT A creates E; and SPROUT B creates D. We are still in C.

**POP** The inverse of PUSH. Move up the context tree to the parent of the current context. POP may be destructive, i.e. the current context may be deleted. POP puts us back in B.

SWITCH <cxt> Leave current context and enter the designated context <cxt>. SWITCH E makes E be the current context, i.e. puts us in E.

DELETE <cxt> Delete the context <cxt> and all its descendants. DELETE B deletes B, C and D.

We now discuss the data structure features of the individual languages.

### II.3. LISP

Basic LISP has little in the way of sophisticated data facilities, but provides a flexible base on which they may be built.

As described in the Handbook article on LISP [yon LISP article], the original LISP had only one data type, namely list structure built up out of *cons cells* and *atoms*. This makes for a simple and elegant language, but at considerable cost in efficiency and readability. A list-structure representation of arrays, for instance, is highly inefficient; and the access of its elements by a chain of *car* and *cdr* list selector functions is somewhat opaque to the reader of a program. To supplement lists, later versions of LISP have tended to add more and more data types. Integer, real, string, array and record types are more or less standard. Many versions also allow the user to define new data types.

LISP was the first computer language to represent functions (procedures) as data in a practical way. Functions are denoted by LISP lists of a particular format called the "lambda expression." [yon LISP article.] Treating functions as data was one of the major pioneering ideas of LISP, and its influence will be seen in most of the other AI languages here discussed. [yon LISP article for more on importance of functions-as-data in LISP.] As an example of the power it gives, a robot planner could take its functions for moving and for picking up, extract the LISP code for MOVE(here, A), PICK-UP(OBJ), and MOVE(A, here), and combine the pieces of code to form a single function FETCH(A, OBJ). Thus the robot has added to its own repertoire of actions.

LISP provides a database facility of a crude sort through the *property list*. Every atom has attached to it a list of property-value pairs, for instance, the property list of atom K2 might include property RANGE paired with value HIMALAYAS. The main deficiency of property lists as a database mechanism is that they are indexed only one way; given atom K2 we can retrieve all its property-value pairs (or its value for any given property), but we cannot efficiently retrieve all atoms having RANGE property HIMALAYAS.

Pattern-matching data retrieval and context mechanisms do not exist in basic LISP, though as we shall see they are relatively easy to include in languages built on top of LISP. The recursive structure of LISP is especially well suited to implementing pattern matchers [yon LISP article on this score.]

## II.4. PLANNER

Simple PLANNER data types include all of those supported by MACLISP.

PLANNER was the first language to implement a general format associative database. The particulars of PLANNER's database stem from PLANNER's theorem-proving semantics, in which a theorem that can prove a fact is just as good as having the fact explicitly stored. The global associative database can store two semantically different classes of data items. The first class is 'assertions' which are LISP lists. Their elements may be of any MACLISP type. All assertions in the database are assumed to be true for purposes of PLANNER's automatic deduction control structure. But beyond that, assertions are treated merely as lists with no inherent semantics. Thus, either (RED MARS) or (MARS RED) or even (MARS IS RED) could have the same meaning, if the PLANNER program is written so as to manipulate them appropriately. Some example assertions are:

```
(COLOR GRASS GREEN)    (HOT SOUP)    (LOCATION PUMP-BASE (10 4 50))
(THIS IS A LONG ASSERTION)
```

The second class is procedural data in the form of 'theorems' which typically state when they can be invoked and what their effects will be. (theorems are further explained in the Control Structure and Pattern Matching sections).

Assertions are explicitly added to the database by an ASSERT operation, and deleted by ERASE. They may also be added implicitly: whenever an assertion is added to or deleted from the database it may trigger off certain PLANNER theorems to perform updating operations or consistency checks. More will be said about these "antecedent theorems" or *demons* in the Control Structure section. PLANNER theorems can also be added and deleted in the database, but this is seldom actually done by a PLANNER program.

For some time the only context mechanism in PLANNER was the one used by the automatic backtracking mechanism (see the Control Structures section). This restores the context of a failure point each time it is returned to. Later implementations of PLANNER context mechanisms of the kind introduced by CONNIVER.

## II.5. CONNIVER

It is sometimes helpful to view CONNIVER as a restructured and extended PLANNER. Both are built on MACLISP.

All the PLANNER data types are carried over to CONNIVER, with three important additions.

1. Possibility lists
2. Tags
3. Context tags.

Possibility lists and tags are discussed in the Control Structure article of this chapter. Possibility lists are used during fetch operations on the database, and tags are used for jumping from one process to another in CONNIVER's generalized control structure. Context tags are used to designate contexts, which we will discuss shortly.

The data base facilities of CONNIVER are virtually identical to those of PLANNER. Again, the data base contains assertions and theorems (called methods in CONNIVER) which are all assumed to be true, and which are accessed by pattern-match.

The only important difference between the PLANNER and CONNIVER databases is that CONNIVER's database is organized into contexts.

The global variable CONTEXT holds the context tag of the current context. By storing tags for other contexts and then reassigning CONTEXT to one of them, the program switches its context. Context tags are generated by standard functions:

(PUSH-CONTEXT <context tag>)	sets up a new context as a direct descendant of <context tag> and returns a context tag representing it.
(POP-CONTEXT <context tag>)	returns a context tag representing the direct ancestor of <context tag>.

Other context functions are not described here.

CONNIVER has the most fully-developed context mechanism of our six languages. Some context mechanisms were implicitly present in PLANNER, but were tied to the backtracking control structure and inaccessible to the programmer. CONNIVER made them explicit and also permitted general switching from any node on the context tree to any other, more general than the strict heirarchical movement done during PLANNER deductions. In terms of search methods, CONNIVER programs can execute a breadth-first or best-first search (or any other kind; [you Search chapter]) whereas PLANNER programs are constrained to a depth-first search.

## II.6. QLISP

QLISP has a wide range of data types. All the data types of INTERLISP are available in QLISP. These include integers, reals, strings, list, pointers, arrays of any of the previous types, and records with fields of any of the previous types. The user can define new data types, also.

The unique data types that QLISP provides are TUPLE, VECTOR, BAG, and CLASS. The names are a bit confusing. VECTORS correspond to what we called "tuples" in the introduction to this chapter. TUPLES are just like VECTORS except that their first component is a function. A CLASS is a set. A BAG is a bag. For two QLISP objects to be equal they must be of the same data type and have the same components. In the case of CLASS and BAG, the order of the components does not matter. Here are some examples:

```

(TUPLE F A)      ≠ (VECTOR F A)
(VECTOR A B C)  ≠ (VECTOR B A C)
(CLASS B C A)   = (CLASS A B C A)
(BAG A A B C)  = (BAG C A B A) ≠ (BAG A B C)
(BAG C A)      ≠ (CLASS C A)
(BAG B (CLASS C A B C) A A) = (BAG B A (CLASS C A B) A)

```

Every datum stored in the database is first transformed to a canonical form, such that any two datums which are theoretically the same, like (BAG B A A C) and (BAG C A B A), will map onto the same canonical representation, in this case (BAG A A B C). Pattern matching is greatly simplified. In theorem-proving, for example, a bag is the natural representation for the operands of functions such as PLUS where repetition is important but order is not. QLISP easily can prove that (PLUS (BAG A B C D)) is equal to (PLUS (BAG C D B A)) by canonicalizing the BAGs, since the two expressions then become identical. Canonical representation also allows QLISP to use two unique database methods.

First, the entire database is stored as a discrimination net. [yon Representation chapter to explain what a discrim. net is. If it does.] Only the canonical forms are put in the net, making it feasible to search the database in a uniform top-down manner. Retrieval in general, and especially pattern matching, is thereby simplified.

QLISP unfortunately stores every subexpression of an item as a separate item in the discrimination net, and suffers from excessive use of space.

Second, since every object is uniquely represented, a property list just like that attached to atoms in LISP (atoms are all uniquely represented in LISP!) can be attached to *every* object in the database. In LISP, only atoms are uniquely represented and only atoms can have a property list. Here is a QLISP property statement.

```
(QPUT (TUPLE PHONE-NUMBER-OF MIKE) LENGTH 7)
```

The example statement adds a TUPLE representing Mike's phone number into the database, if it isn't already there, and says that the LENGTH property of the number has value 7. Object properties provide an elegant solution to the problem, from which PLANNER and CONNIVER suffer, of being unable to distinguish the mere *presence* of a statement in the database from its *truth*. In QLISP, truth or falsity is indicated by setting the MODELVALUE property of a statement to be T or NIL. Thus, to claim it is *true* that grass is green, execute

```
(QPUT (VECTOR COLOR GRASS GREEN) MODELVALUE T)
```

For automatic inference and updating of the database, if-added and if-removed demons similar to those of PLANNER and CONNIVER are available in QLISP.

Context mechanisms were developed in QA4, ancestor of QLISP, independently of CONNIVER.

## II.7. SAIL

SAIL has three categories of data types. First there are the types inherited from ALGOL, namely integer, real, Boolean, string, and arrays. Second, the user can define record types, in which the components of a record are specifiable as any of the above types or as (the same or different) record type. Records are especially important in AI applications because they can serve as building blocks for list structures. The cons cell of LISP is essentially a record with two components (car and cdr), whereas SAIL records can have any number of components. More general list structures can be built. However, the LISP approach has the advantage that standard functions are available for searching lists, deleting from lists, appending lists etc. In SAIL these must be defined by the user. Some LISP dialects offer a record package, too. The third data category is the most interesting, the items of SAIL's associative database mechanism.

Items were the major feature of the earlier LEAP language, which was incorporated in SAIL. The motivation for LEAP was to implement an efficient software scheme for associative processing, or the accessing of data by partial specification of its content rather than by the address of the storage location it happens to reside in.

An item is either a primitive identifier (atom), or a triple of items. Triples are formed by the association of three items, like so:

Attribute • Object ■ Value

Here are examples of associative triples:

COLOR • MARS ■ RED  
 CLASS • RED ■ COLOR  
 SATELLITE • MARS ■ PHOBOS  
 SATELLITE • MARS ■ DEIMOS

Notice that items are not restricted to any one position in associations, nor does the Value have to be unique. In fact the labels "Attribute", "Object" and "Value" are mere conveniences with no special significance. Associations can themselves be items:

DISCOVERER • [COLOR • MARS ■ RED] ■ ANONYMOUS

Associations are created and removed from the global database by the statements:

MAKE <item1> • <item2> ■ <item3>  
 ERASE <item1> • <item2> ■ <item3>

For efficiency reasons the internal representation of each item is kept down to one PDP-10 word, causing difficulties when one wishes to have an item that is, say, a large array. As a somewhat awkward solution, each item can have an attached "DATUM", which is stored separately and can be of any ALGOL type. A further difficulty comes from the fact that associative triples can themselves be items. At 36 bits per word, this implies 12 bits to address an item, and an allowable total of only  $2^{12}$  or 4096 items. For large data bases that would not be enough.

Associations in the SAIL data base are triply indexed and can be retrieved quite efficiently. Retrieval statements will be explained in the Pattern Matching section. There are no automatic updating or consistency checking facilities for the SAIL data base.

## II.8. POP-2

POP-2 has a quite rich collection of data types. Many of them are standard special cases of more general types, as indicated in the table below.

integer	
Boolean	(0 means false, anything else means true)
real	
strip	(sequence)
string	(strip of characters)
record	
pointer	
atom	
ordered pair	(like the LISP cons cell)
list	
word	(it has 8 components which are all characters)
function	
array	

Data types are classified as "simple" and "composite". Integer, Boolean, real, pointer and atom are simple, the rest are composite. User-defined data types are allowed.

POP-2 has the helpful (to clarity and ease of programming) property of treating all data types in a uniform way. Data of any type can be used:

as the actual parameters of functions

returned as the results of functions

assigned to variables of their type

tested for equality.

Components of composite data types are always accessed by four kinds of function, illustrated below for the type "list". In the case of user-defined types, the user must supply the four kinds of function. Note that the destructor function shown here produces two outputs. In general, POP-2 functions can produce multiple outputs. Also note that the selector and the updater have the same name. This is quite common.

Kind of function	For lists
constructor	<code>cons(x, (y z)) = (x y z)</code>
destructor	<code>dest( (x y z) ) = x, (y z)</code>
selectors	<code>hd( (x y z) ) = x,       tl( (x y z) ) = (y z)</code>
updaters	<code>hd(MY_LIST) ← x,       tl(MY_LIST) ← (y x)</code> <code>Now MY_LIST = (x y x).</code>

Associative database and context facilities are not part of the basic POP-2 language but there exist libraries which provide them. LIB HBASE provides for two-place associations rather similar to SAIL's except for being only pairs, not the more flexible triples. Pattern-matching retrieval may be done in HBASE; see the Pattern Matching section. Limited context facilities are found in LIB CONTEXT, which is compatible with LIB HBASE.

## II.9. SUMMARY

Under the heading of data structure, we have discussed three main subtopics: data types, large database facilities, and context mechanisms. We now take these up one by one, for an overall view.

### Data Types

The languages vary considerably in the number and kind of data types which they offer. Basic LISP is at one extreme: it began with exactly one data type, and a few supplementary ones were added later. Advantages of a sparse set of data types accrue mostly to the writers of compilers and interpreters, whose job is simplified because there are fewer operations and less need for type-conversion. Also, the relatively small compilers and interpreters produced help conserve available core space. From the *user's* point of view there is little to recommend such a small set of data types, except that it (almost) removes the need for type declarations.

Later versions of LISP such as INTERLISP and MACLISP, and to an even greater extent the languages QLISP and POP-2, have provided rich sets of data types and the access functions that go with them. Programming is easier because the data structures can more closely mirror the programmer's ideas, and type-checking becomes available. Efficiency is enhanced because the standard data types can be implemented closer to the machine level than equivalent structures built of more primitive units. For example, a SAIL or POP-2 record uses fixed-offset fields and avoids the overhead of the pointers needed in an equivalent LISP list structure.

A related issue is whether to allow user-defined data types. The advantages are similar to those discussed in the previous paragraph. However when user-definability is made available, as in POP-2, it tends to be little used. Probably the main reason is simply the extra effort required from the user (who has to define all the primitive operations on his/her new data type, remember!) User-definability also results in "unique" programs that other people may have difficulty understanding.



### Large Data Base Facilities

Data base facilities were present in a crude form in the LISP property list feature. The next "level of complexity" is represented by the multiple index scheme of SAIL associations. Next above the use of fixed-form associations is the structural pattern-match as a general retrieval mechanism, found in PLANNER, CONNIVER and QLISP. Appropriate retrieval is somewhat hampered in PLANNER and CONNIVER because they cannot attach properties to assertions as QLISP can. PLANNER also keeps the possibilities lists hidden from the programmer, who has always to operate at the goal level even though it may be inefficient to do so.

A higher level of complexity is indicated in the various "frame" proposals, focus-of-attention mechanisms, and other techniques in an experimental stage. [yon Representations chapter], [yon some individual lang. articles.] Our six languages do not include any of these.

### Context Mechanisms

Context mechanisms may also be arranged in a loose ordering of complexity, starting with the scoping rules which virtually all languages have. These provide a new "context" whenever a new function or block is entered, and restoration of the previous "context" upon exit. This basic level of contexting is extremely useful in programming practice. Next comes the ability of a program to push and pop contexts on demand, whenever they are needed, rather than in rigid correspondence to the structure of the program. QLISP [In both these forms, the contexts existing at any given time are simply those in the direct line from the global context down to the current context. CONNIVER is more advanced and allows a whole tree of contexts to exist, with freedom for the program to sprout new contexts below any existing context and to jump around arbitrarily between contexts.

To sum up very briefly, LISP introduced a simple and flexible data type and a way to represent functions as data. PLANNER introduced the general associative database, and CONNIVER improved it by the addition of contexts. QLISP increased the power of the database by defining special data types and putting everything in a discrimination net. SAIL went in another direction, developing an efficient multiple index scheme. POP-2 showed how a wide range of data types can all be treated in a very clear and uniform way.

### III. CONTROL STRUCTURES

#### III.1. MOTIVATION

The most important of the AI control structures we will discuss is *coroutining*. The central idea here is to relax the strict control heirarchy found in most languages (including LISP), in which every procedure runs for a while, then returns control to the procedure that called it, and vanishes. Coroutines are procedures that can pass control (note we do not say "return control") to any other coroutine. Furthermore, the coroutine need not vanish; instead it can suspend itself, and later continue from where it suspended.

We now need to introduce a lot of terminology. First of all, it is speaking loosely to talk about procedures or coroutines suspending or vanishing; a procedure is a piece of code. To draw the distinction, an instantiation or execution of a coroutine will be called a *process*. A process can *suspend*, so its current *state* must be saved - the values of its variables, and which statement to resume execution at. Initially there is only one process, and it can *create* new processes. This corresponds to a procedure call in the heirarchical regime. Once it is created, a process can be in one of three states: running, suspended or terminated. If control is transferred to the process when it is created, then it is running; otherwise it is suspended. A running process can *resume* or *activate* some other process, i.e. pass control to the other process, causing itself to suspend. While a process is suspended its state is preserved but nothing happens. A running process can also *terminate* as it passes control to another process. *Terminated* processes usually vanish as in the strict heirarchy scheme, but in certain cases their state will linger on. (Cases in which some other process is still using the variable bindings of the terminated process.)

Bobrow and Wegbreit [bobr73] provide a formal and clear model for general coroutining in terms of *stack frames*, which also can be efficiently implemented. Their model unified the various types of procedural interconnections seen in previous AI languages (e.g. recursion, coroutines, generators, funargs, funvals). Each process is represented by one stack frame, which contains the process state and links indicating where control was passed from and where the values of free variables are to be obtained. We will not discuss control at the implementation level, but curious readers should start with this paper.

The *generator* is one common special kind of coroutine. Generators are used when a potentially large set of data may be produced by a function (like matching the database) but only a few at a time are needed. A generator produces one item and then suspends. When a process needs another piece of information it resumes the generator. The generator returns a new piece of information and suspends again. A generator always returns control to the routine that activated it. This continues till there are no more items to produce, and the generator terminates.

In a coroutining regime, only one process is running at any given time. There can be a considerable amount of communication and cooperation between processes, but it is rather awkward since Process A must suspend itself in order to pass information to Process B. In a *multiprocessing* regime, many

processes can run at once, and freely pass messages back and forth. (In actuality, there may be only one process executing, with rapid switching among the set of "running" processes. Or the processes may in fact execute simultaneously, on different processors.) SAIL is the only one of our six languages that allows for multiprocessing; with the declining cost of computers there is increasing interest in languages for multiprocessing and some of the newer AI languages reflect this

Coroutining and multiprocessing only provide more flexibility in the flow of control, without assisting in the problem of determining *where* control is to flow next, during a process. The usual method for this is a conditional statement (*if* or *case* statement) that chooses one of a few predetermined directions. Several of the six languages provide a much more powerful method, *pattern-directed invocation*, described in the Pattern Matching section of this article.

Another important control concept is the *demon*. Implementations differ, but in concept a demon is a kind of suspended process, that is waiting for a certain kind of event to occur (for instance, a certain kind of update operation on a database). When such an event occurs, the demon is activated automatically, performs its job, whatever that may be, and either terminates or suspends in wait for the next event. Typically, demons are used to make inferences as new information comes in, or perform bookkeeping tasks of some kind, or recognize significant occurrences.

### III.2. LISP

Basic LISP completely lacks coroutining or multiprocessing facilities. LISP control structure is suited to AI programming mainly by its emphasis on recursion. Recursive procedures/functions are often nearly essential in AI tasks; examples are pattern matching, tree searching, and parsing. For further discussion of recursion and its importance in AI programming, see [yon AIH article on LISP].

Recursion allied with LISP's dynamic scoping rule allows functions to be used by other functions in a "context-free" manner: a function's behavior depends only on its arguments and the values which its free variables have when it is called; the context in which the function was originally defined is not significant. Furthermore, any function is allowed to call any other function. This freedom makes it easier in many cases to put together separate modules to form an AI system.

The "FUNARG" mechanism of LISP permits functions to be used in a context-dependent or history-dependent way when desired. A FUNARG consists of a LISP function together with an environment (an environment is like a context; see [yon Data Structures article]) that the function, so to speak, carries along with it. Various things can be done with FUNARGS; for example, they can be used to implement generators. Suppose we tried to write a generator as an ordinary LISP function. We would define a function that generates just one item each time it is called and changes the state of the generation process by updating a free variable. Here is such a "generator" for square numbers. (If you are unfamiliar with LISP programs you may wish to refer to [yon article on LISP].)

```

NextSquare: (LAMBDA NIL
             (PROG NIL
              (SETQ N (ADD1 N))
              (RETURN (TIMES N N)) ) )

```

NextSquare uses the free variable *N* to hold the state of the generation. If initially *N*=3 and we call NextSquare repeatedly, we will get the sequence 16, 25, 36, ... . But NextSquare is not really a generator, an independent process, because if any function happens to use a variable called *N* locally, and calls NextSquare, NextSquare will use the local value of *N* and the sequence will be disrupted.

A true generator results when we construct a FUNARG which consists of a function (NextSquare) and a pointer to an environment (in which *N* has some initial value, say 3.) Call the FUNARG GenSquare. If we repeatedly evaluate (GenSquare) we get 16, 25, 36, ... as before, only GenSquare always uses that *N* which is in its specified environment, and so is unaffected by local versions of *N*. We could construct another FUNARG with NextSquare and a *different* environment, in which *N* has a different initial value, say 8. Call it GenSquare2. GenSquare2 will generate its own sequence 81, 100, 121, ... totally independently of GenSquare or anything else. In general, FUNARGs act much like coroutines; they "activate" when their function is called, "suspend" when it exits. Unlike general coroutines, FUNARGs always return control to the function that called them.

CONNIVER has a control structure very much like LISP's, with the addition of full corouting facilities. PLANNER extends the LISP control structure in a completely different direction. PLANNER is somewhat like a backtracking version of LISP - functions call each other recursively as in LISP, but when a function fails (returns NIL) it is *automatically called again* using the next possible argument value. This continues until either the function succeeds or all arguments fail.

### III.3. PLANNER

The control structure of PLANNER is very interesting and we will have a lot to say about it. The fundamental point to keep in mind is that PLANNER is *always* goal-directed. Functions are invoked because it looks like they may cause a goal to be satisfied, *not* because some other function called them. PLANNER functions simply do not call each other; they just state a goal and the system chooses a function to apply towards that goal.

A rather strong analogy may be drawn to the control structure of GPS (yon Search chapter GPS article). Both systems completely separate domain-dependent information from the control structure and use a uniform depth first search to solve problems. In both systems the problem is represented by a goal to be reached or established. The starting point is different, however; GPS is given a single initial object (which may be thought of as the state of the world) which it tries to transform into the goal object. PLANNER is given a whole database of facts and tries to prove that the goal follows from them. Thus GPS's paradigm is actions on the world, PLANNER's is theorem-proving. But the actual methods they use are more similar than this might suggest.

Each first will check to see if the goal is equal to the initial state (or some fact in the database.) If not, GPS looks for an operator that will reduce the difference between the goal state and the initial state, and PLANNER looks for a "theorem" which can prove the goal from the database.

Each then sets up subgoals. For GPS, these are to transform the initial state into something that the chosen operator can apply to, and to transform the result of the operator into the goal. The whole GPS method is applied recursively to these two subgoals. PLANNER theorems can set up any number of subgoals, namely whatever facts must be established before the theorem will imply the goal. Again, the whole PLANNER method is applied to each subgoal.

If at any point a subgoal fails, both GPS and PLANNER will backtrack and try to apply a different operator or theorem.

The net result is a depth-first search of an AND-OR tree (OR nodes wherever there is a choice of operators or theorems; AND nodes for the subgoals induced by each operator or theorem.)

Please note that both the GPS and PLANNER control structures were simplified for expository purposes. The closing paragraph of this section mentions a few complexities of the PLANNER control structure.

In addition to the goal-directed or "consequent" theorems PLANNER programs can have "antecedent theorems" which are a species of demon and trigger when facts are added to the database or deleted. (GPS has nothing analogous.) They can modify the database in arbitrary ways, generally filling out implications of a new fact or doing some kind of cleanup. Deductions which would have to be made anew each time a goal called for them can be done just once by an antecedent theorem, as in the following example. Suppose a common goal is to establish that sb-and-so is mortal. A consequent theorem for this would read

```
(CONSE (MORTAL ?X) (GOAL (HUMAN ¬X)) )
```

I.e., it will prove that something is mortal if you can satisfy the subgoal of proving that it is human. Alternatively, write

```
(ANTE (HUMAN ?X) (ASSERT (MORTAL ¬X)) ).
```

Now, if an assertion such as (HUMAN Me) is added to the database, the antecedent theorem activates and causes (MORTAL Me) to also be added.

Here is a more extensive example of PLANNER in action. Suppose we have a consequent theorem giving one set of conditions under which an object can be deduced to be broken:

```
(CONSE (BROKEN ?X)
  (GOAL (FRAGILE ¬X))
  (GOAL (HEAVY ?Y))
  (GOAL (ON ¬Y ¬X)) )
```

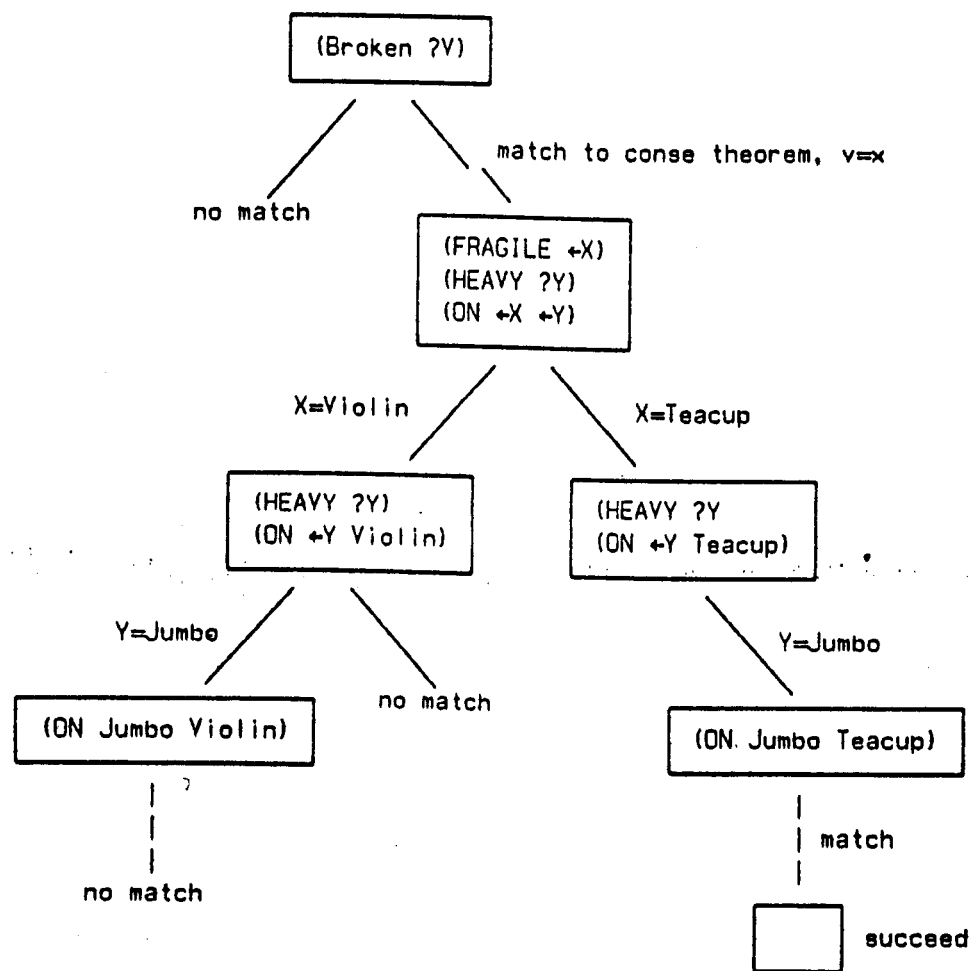
And a database containing the following assertions, among others:

(HEAVY Jumbo) (FRAGILE Violin) (ON Jumbo Teacup) (FRAGILE Teacup)

Now let's see what happens when a PLANNER program executes:

(GOAL (BROKEN ?V))

It will first scan the database looking for broken things, then use consequent theorems to deduce that other things are also broken. The backtracking search produces the tree below.



PLANNER does pure depth first search [yon Search chapter]. In the tree shown above, search

proceeds top to bottom and left to right. Initially the goal is to match or prove (BROKEN ?V). This fails to match any assertion in the database, however it does match to the consequent theorem. That theorem sets up three subgoals and they are matched one by one, with backtracking whenever the match fails. When all three eventually get matched, control returns up to the original goal and V is bound to the value of X, i.e. Teacup. The goal (BROKEN ?V) has been satisfied by the deduced fact (BROKEN Teacup).

You will notice how PLANNER wastefully re-matches (HEAVY ?Y) to (HEAVY Jumbo) after choosing a different fragile object. PLANNER unfortunately makes no provision for two choices being independent.

PLANNER control structure is actually not quite so rigid as pictured here. There is a command which finalizes a certain portion of the control tree, so that actions are not undone during backtracking. Also there is a mechanism to fail back to an arbitrary failpoint (as opposed to the closest one). But the fundamental control structure is indeed a rigid depth first search.

### III.4. CONNIVER

PLANNER control structure was characterized by the use of three different kinds of "theorems", and automatic backtracking. CONNIVER has exactly the same three "theorems", only it calls them "methods": if-needed, if-added and if-removed methods. Instead of automatic backtracking the language provides a set of more primitive control elements out of which a programmer can construct, if desired, a backtracking scheme identical to PLANNER's. Normally the programmer will find alternative schemes more appropriate. Here, the consequent theorem from the Broken example is rewritten as a CONNIVER if-needed method. The reader who finds the CONNIVER code opaque may benefit from reading [yon article on LISP], since CONNIVER looks a lot like LISP.

```
(IF-NEEDED (BROKEN ?X)
  (PROG (Fragiles Y Heavies Copy-Heavies)
    (CSETQ Fragiles (FETCH '(FRAGILE ?X)))
    (CSETQ Heavies (FETCH '(HEAVY ?Y))))

  OUTERLOOP:
    (TRY-NEXT Fragiles '(ADIEU))
    (CSETQ Copy-Heavies Heavies)

    INNERLOOP:
      (TRY-NEXT Copy-Heavies '(GO 'OUTERLOOP))
      (IF (FETCH '(ON ←Y ←X)) THEN (NOTE '(BROKEN ←X)))
      (GO 'INNERLOOP) ) )
```

Some explanation is required.

A problem with the PLANNER theorem was its repetition of the search for heavy objects, every time a new fragile object was found. This was forced because the

automatic backtracking has to assume that later choices are dependent on earlier ones, even when, as in this case, they are not. In CONNIVER the list of heavy objects can be fetched once and re-used. Accordingly, the first section of the program finds all fragile objects and all heavy objects.

In the second section the generator statement (TRY-NEXT FRAGILE\_LIST (ADIEU)) grabs the next fragile object in the list and binds X to it. The (ADIEU) is executed when there are no more fragile objects; it causes the method to terminate.

In the third section the inner loop scans through all heavy objects in the same fashion, and each pair of a fragile X and heavy Y is tested to see whether Y is on X, and if so, the method NOTES this and moves on. Noting an item consists in adding it to a "possibilities list".

That possibilities list becomes the result of the if-needed method after (ADIEU) is executed. Thus, the effect of the method is to return a list of all deducible instances of (BROKEN ?V), which will be just ((BROKEN Teacup)) assuming the same database as in the PLANNER version.

Possibilities lists are implicitly present in PLANNER too, but are not accessible to the user. Whenever a choice-point is reached during automatic backtracking, and any of several assertions or theorems might be invoked to satisfy a goal, they are all placed in a possibilities list. Then one by one they are removed from the list and tried out until one of them succeeds. All of this is automatic. In CONNIVER possibility lists are a separate data type and may be manipulated by a user program. In many respects CONNIVER is like a PLANNER in which implicit control and data structures are made explicit and put under control of the programmer. Greater flexibility results, at the cost of longer and messier programs.

CONNIVER allows general corouting. As one application, a program can be written to do breadth-first or best-first search [yon Search chapter] [by suspending the process associated with one node and reawakening a suspended process at some other node. PLANNER, you will recall, always does a depth-first search. Another use of corouting is the implementation of generators. The broken-object finder above can be turned into a generator simply by adding (AU-REVOIR) after (NOTE (BROKEN ←X)). AU-REVOIR is the CONNIVER function that suspends the current process and resumes whichever process called it. Now, each time the method is called it finds one broken item, produces it and then suspends. The advantage of this technique is that the calling process can ask for broken items only as it needs them, and the generator may not have to find them all.

### III.5. QLISP



The GOAL statement in QLISP works like PLANNER's, first searching the database to try to match the pattern there, then invoking consequent theorems (if-needed methods) to try to derive it. Automatic backtracking occurs whenever one of the theorems fails. The main difference is that consequent theorems to try must be listed in the GOAL statement itself, for instance:

```
(GOAL (MORTAL Socrates) APPLY Grim_Reapers )
```

Grim\_Reapers would be the class of theorems, similar to the PLANNER consequent theorem we saw before, (CONSE (MORTAL ?X) (GOAL (HUMAN ←X))), relevant to establishing that Socrates is mortal. It is called an *APPLY team*. One could circumvent the need to set up teams by putting all theorems in one giant team and applying it everywhere (thus simulating PLANNER.) The point, however, is to avoid such inefficient means.

Database operations may also have APPLY teams, which take the place of the if-added and if-erased demons of PLANNER and CONNIVER.

### III.6. SAIL

The most interesting control structure feature of SAIL is its multiprocessing capability, something none of the other five languages covered here possess. Designers of SAIL [feld72] wanted to allow parallel exploration and cooperation by problem-solving processes. There was also the need, for memory-space reasons, to split large hand-eye systems into multiple communicating processes.

Multiprocessing is implemented within the ALGOL-like block structure of SAIL. Any process (invocation of a SAIL procedure) can SPROUT, or create, another process. The standard ALGOL scope rules are followed just as if the new process were started by a procedure call; thus related processes are automatically able to share a common database. The JOIN primitive suspends a process until the processes named in the JOIN have terminated, at which point the first process resumes execution. JOIN is used as a synchronization primitive. Thus we might see the example:

```
SPROUT(P1, GRAB(HAND1, HAMMER));
SPROUT(P2, GRAB(HAND2, NAIL));
...
JOIN(P1, P2);
POUND(HAMMER, NAIL);
...
```

The variables P1 and P2 hold the (unique) names assigned to the new processes. The names are used to identify processes in later commands.

Interprocess communication is handled by a mail system implemented as a message queueing system. The mail is delivered (whenever possible) by the process scheduler. Processes can simply check for messages as they go along, or can place themselves in a queue while waiting for an appropriate message.

Using the first method, the grab-hammer process could inform the grab-nail process about the movements of the arm and hammer, so the grab-nail process could avoid a collision with the other arm.

Demons can be implemented using the second method. To set up a demon, create a process which immediately places itself in a queue waiting for a certain kind of message. And arrange that whenever a database update is done, a message is sent out. Demons that are waiting for such a message will be activated, simulating a direct activation of demons by the database update.

Coroutining is a special case of multiprocessing, in which only one process is active at any time. The coroutining primitives create, terminate, activate and suspend can all be implemented using SAIL's message-passing mechanism.

Generators are a special case of coroutining, in which the generator process always resumes the process that activated it (rather than some third process), and terminates itself when nothing is left to generate.

### III.7. POP-2

POP-2 control structure is much like that of LISP. The basic language is very simple, partly in the interests of fast compilation, and contains none of the specialized AI control structures found in PLANNER, CONNIVER and QLISP.

Here is a summary of the POP-2 libraries offering control features.

LIB CASES merely provides a case statement, which POP-2 lacks.

LIB COFUNS provides "coroutines, parallel processes, semaphores and synchronizing primitives. It appears that these are not used for building AI programs; the LIB COFUNS writeup says they are "most useful for the simulation and investigation of the behaviour of truly parallel programs." Care was taken to make the process scheduler fair and *unpredictable*.

LIB PICO-PLANNER seems to have nothing significant that LIB POPLER doesn't have.

LIB POPLER provides a "POP-2 PLANNER". A "substantial subset" of PLANNER. Implemented by Julian Davies who is now at London, Ontario; POPLER is in use there. [ref: POPLER: A POP-2 PLANNER"] POPLER includes database demons and pattern matching like those of PLANNER and CONNIVER. It provides automatic backtracking but also "uses the Bobrow & Wegbreit frame structure model and allows general multiprocessing to be programmed with primitives similar to" those of CONNIVER and QLISP.

LIB BACKTRACK, LIB SOFT BACKTRACK appear to be superseded by POPLER.

The basic POP-2 language does not include general coroutinging. It does provide for the use of a generator, to construct a *dynamic list*. The programmer defines a function of no arguments, say *F*, and applies the standard function FNTOLIST to *F*. The result is the list *LF* of values which *F* produces on successive calls. Of course, *F* has to read and change a free variable(s), representing the state of the generation, or else every element of *LF* would be the same. Now comes the interesting part. The program can operate on *FL* just as on ordinary (static) lists. But in fact, *FL* is not all there; it starts empty and new elements are added onto the back of it only as needed. This means that *FL* can be potentially very large or even infinite, and it does not matter so long as only the initial part of it is used.

The dynamic list allows the programmer to abstract away from the details of how a list is produced; whether it is computed once and for all, or is extended when needed. Similarly, the *memo function* allows abstraction from the details of how a function is computed. Memo functions are provided by one of the POP-2 libraries. The name comes from the notion that a function, if it is called several times with the same argument, would do better to "make a memo of it" (the answer) and store it in a look-up table. As an example of the utility of memo functions, consider a chess end-game program. It will tend to evaluate the same board position several times during its search, because of the limited number of possible positions. If we make the evaluation function into a memo function, then each position's value will be stored in a table as the search proceeds, and no position will be evaluated more than once.

### III.8. SUMMARY

In the (chronological) sequence PLANNER, CONNIVER, QLISP we observe an increase in the variety of control structures and in the programmer's access to them. This development seems to address two major needs: to concisely express powerful control techniques, and without making one's programs grossly inefficient. PLANNER took a giant step in the direction of power by shifting the entire domain of discourse in which programs are written from the imperative level (do this, then do that, etc.) to the level of goals. The programmer formulates problems in terms of goals that are to be established, writes "theorems" that reduce a goal to subgoals, and sets the automatic deduction mechanism going. This approach suffers from the inefficiency of its central method, automatic backtracking, and the inability of the user to remedy this by expressing himself at the imperative level on occasion. Automatic backtracking in PLANNER is convenient if the user has absolutely no good idea of how to guide the program.

CONNIVER was developed largely in response to these problems. Sussman and McDermot (72) argue the following points against PLANNER and for CONNIVER:

- (1) Programs which use automatic backtracking are often the worst algorithms for solving a problem.

(2) The most common use of backtracking can almost always be replaced by a purely recursive structure which is not only more efficient but also clearer, both syntactically and semantically.

(3) Superficial analysis of problems and poor programming practice are encouraged by the ubiquity of automatic backtracking and by the illusion of power that a function like GOAL gives the user merely by brute force use of invisible failure-driven loops.

(4) The attempt to fix these defects by the introduction of failure messages is unnatural and ineffective.

CONNIVER retreats to the imperative style of programming with some automatic deduction primitives kept on, namely the 3 kinds of "theorem" or "method". In addition it gives the user the flexibility implied by coroutines, which enable techniques like breadth-first or depth-first search, and others, to be expressed conveniently.

QLISP re-introduces automatic backtracking but as optional and under restrictions (that the "consequent theorems" to try must be named.) Overall, it includes practically every control structure found in PLANNER or CONNIVER.

All three of these LISP-based languages rely heavily on pattern matching to guide the flow of control. The basic technique is pattern-directed invocation, in which the function to execute next is chosen by its match to some datum that is held "up for grabs" so to speak. A problem here is that if one function fails to do anything useful with the datum, that function is simply flushed and another one tries. It seems plausible that cooperation and sharing of information between the different functions attempting a goal might be better. This is part of the idea behind SAIL's multiprocessing primitives, through which processes can access a shared data base and send messages to each other as they work on a problem. This is reminiscent of the "cooperating experts" or "blackboard" architectures of some major AI systems [yon HEARSAY-II],[yon maybe others], which, however, construct their own control facilities out of LISP. Cooperative interaction of expert sub-systems is still at a relatively primitive level in SAIL.

Randy Davis [ref] discusses several control schemes and when one might want to use each of them, in the context of his own program, TEIRESIAS [yon AIH ref to TEIRESIAS], which uses strategy rules to decide (at run time) how to sequence other rules. One major dimension along which control schemes vary is the degree of data-drivenness. In the heirarchy scheme, conditional statements (and loop tests) interrogate a data value and select from a few predetermined paths depending on the value. In a strategy scheme, complex "strategic reasoning" (whatever that is) would be done, taking into account various data values of the moment, and the result of the reasoning would be a decision on what to do next. Most of the six AI languages allow selection of what to do next by a pattern match. This allows selection from a wide range of options that need not be laid out explicitly beforehand, according to the structure or "pattern" exhibited by some datum. Sequencing according to strategy is a very high-level method. The AI languages we will discuss are somewhere between that level and the rigid heirarchy scheme.

## IV. PATTERN MATCHING

### IV.1. MOTIVATION

AI languages use pattern matching for two major tasks: accessing data, and choosing a procedure to execute.

Pattern matching can be a form of *content addressing*. In most computer languages data items have to be accessed by their name, for instance the name of a variable or the particular array location that holds them. The general idea of content addressing is to eliminate this need for arbitrary names of things, and rely on their content (meaning) instead. Here is how it can be done by pattern matching. A pattern such as ((FATHER-OF ?X) ?Y) is a kind of sketch of a data item, and will match and retrieve any stored item like ((FATHER-OF MARY) (FLIES KITES)). Thus no name or address has to be remembered for a stored item. This is known as *pattern-directed retrieval* or retrieval by pattern match.

Similarly, programs need not call procedures by name, if each procedure has a pattern associated with it. In *pattern-directed invocation*, a data item is tested against the pattern of each procedure, and the (first) one that matches is called. All this is done implicitly, replacing the tests and branches on data that one would otherwise have to use in choosing a procedure. Pattern-directed invocation is a special case of content-directed invocation, which is discussed in

Both methods are highly useful in AI programming because they take such a burden off of the programmer. Modularization of programs is much easier to accomplish because the programmer doesn't have to specify who calls whom, or what access path is taken to get to an item in the database. A price must be paid, in the overhead of running the pattern matcher.

Typically, a pattern is a structure with variables embedded in it, and it matches another structure if it could be made identical, by replacing its variables with some values. Occurrences of the same variable must be replaced by the same value. For example, (A ?X C ←X) will match (A B C B) but not (A B C D).

The power (and overhead!) of a pattern matching mechanism is partly related to the type of pattern variables which can be handled. Here are some common variable types, expressed in terms of matching on lists.

- 1) An 'open' variable (?X) matches to any element of a list, and binds X to that element.
- 2) A 'closed' variable (←X) must have already been bound, and matches only the value of X.
- 3) A 'restricted' variable may have arbitrary restriction placed on it. These restrictions are procedurally attached to the variable in some way; in the simplest case, as a Boolean function which must return "true".

4) Segment variables. These match to a sublist of any length, rather than to an element. Open and closed segment variables are denoted ??X and ↔X, respectively.

To avoid confusion the different kinds of var will always be written ?X, ↔X, etc. ignoring the varying notations of the languages.

Some examples may clarify the ideas. Suppose this is the object we want to match to.

(A B (C A) B)

Here are some patterns, with an indication whether each matches the object. No restricted variables are shown.

Pattern	Match?	Bindings
?X	yes	X=(A B (C A) B)
(?X ↔X (C A) B)	no	
(?X B (C ↔X) B)	yes	X=A
(?X ?Y (C A) B)	yes	X=A, Y=B
(?X ?X (C A) B)	illegal pattern	
(A B ?Z B)	yes	Z=(C A)
(??X)	yes	X=(A B (C A) B)
(??X (↔↔X) B)	no	
(??X (↔↔Y) B)	yes	X=(A B), Y=(C A)
(A ??Z)	yes	Z=(B (C A) B)
(B ??Z)	no	

Typically patterns are matched against objects that contain no variables. Some systems, notably QLISP, allow for patterns to be matched against patterns.

## IV.2. LISP

LISP has no pattern matching constructs but lends itself well to implementing them, and since the next three languages are LISP-based it is appropriate to say a little about this.

Patterns typically take the form of nested structures, standing for (for instance) a mathematical formula, an assertion, or some kind of structured object. These can all be uniformly represented as LISP lists, as described in [yon LISP article.] Uniformity is a big advantage because it means the *same* pattern matching algorithm can be used for these different kinds of object. Pattern matching on nested structures is inherently a recursive process, so the free use of recursion in LISP helps a lot.

The reader will observe that pattern matching in PLANNER and CONNIVER always operates on ordinary LISP list structure and in a recursive manner. The QLISP matcher operates on list structure, and also on the special QLISP data types like CLASS and BAG.

### IV.3. PLANNER

PLANNER's big innovation was simply to *use* pattern matching in a thorough way such as no previous language had done. Access to the PLANNER database is exclusively by pattern-directed retrieval. Thus, the goal pattern

(?X Socrates)

will match to a database assertion

(MORTAL Socrates)

and X will be bound to Socrates.

As another application of pattern matching, PLANNER functions (theorems) are always called through pattern-directed invocation. Consequent theorems have an attached pattern which is matched to the current goal, and if the match succeeds the theorem is invoked. For example,

```
(CONSE (MORTAL ?Y)
        (GOAL (HUMAN ←Y)) )
```

With this consequent theorem, the goal (?X Socrates), meaning roughly "tell me something about Socrates" will match to the theorem's pattern (MORTAL ?Y) meaning "I can tell you when things are mortal", with the upshot that the theorem establishes a subgoal (MAN Socrates). If it succeeds in proving this goal, it returns (MORTAL Socrates).

Antecedent and erasing theorems [yon Control Structure article] similarly have attached patterns, but these are matched against assertions being added to or deleted from the database, rather than goals. The same pattern matching method is used in all cases.

There is nothing particularly interesting about PLANNER's actual pattern matcher. The implementors of micro-PLANNER deliberately chose the simplest scheme that would be adequate for their needs. The matching is only one level deep; thus (?X (?Y Plato)) is not a valid PLANNER pattern. There is no requirement that PLANNER *assertions* be single level but the matcher will treat elements of the top level as simple units, no matter what their internal structure may be. In practice most PLANNER assertions have only one level of structure. One might guess this is to accommodate the pattern matcher's limited powers.

Full PLANNER, as opposed to the implemented subset micro-PLANNER, uses general tree matching, which is the same as multilevel list matching.

For an extensive example of how PLANNER's pattern matching facility is used, see [yon SHRDLU].

#### IV.4. CONNIVER

CONNIVER uses pattern matching for the same purposes as PLANNER, except that functions can be invoked directly as well as via pattern. The pattern matcher is more powerful, first because it handles multilevel patterns like (?X (?Y Plato)), which it can match to (MORTAL (TEACHER-OF Plato)). There is no restriction on the level of nesting. Secondly, recall that PLANNER and CONNIVER patterns are LISP list structures. CONNIVER in fact allows patterns to be general s-expressions, using the dot notation of LISP. Whatever comes after a dot stands for the rest of the list, not its next element. For example, the pattern (PLUS 0 . ?X) will match to (PLUS 0 1 2 3 4), binding X to the list tail (1 2 3 4). This would be useful in an algebraic simplifier.

These two advances over PLANNER used in combination provide moderately powerful pattern matching capability. The matcher can be useful on a stand-alone basis, as a way of analyzing the structure of data; CONNIVER allows this with the MATCH statement. Execution of

```
(MATCH ((FATHER-OF ?WHO) . ?WHAT) ((FATHER-OF FRED) WHISTLES DIXIE) )
```

binds ?WHO to FRED and ?WHAT to (WHISTLES DIXIE). In another case, ?WHO might get bound to (OWNER-OF (LOUD TRUMPET)) and ?WHAT to (GRIMACES).

So far all the patterns we have shown contain just two kinds of thing: variables that must match to something, and constants like FATHER-OF, PLUS. CONNIVER adds flexibility by allowing variables that are assigned some constant value before the match is done; the effect is as if the variable were replaced by its value before the match begins.

#### IV.5. QLISP

QLISP has by far the most powerful pattern matching facilities of the six languages covered. There are three contributing factors:

1. The special data types BAG, CLASS, TUPLE and VECTOR when they are used in patterns and data.
2. Segment variables.
3. A powerful unification algorithm is used.

Recall one of the examples for CONNIVER pattern matching, that the pattern (PLUS 0 . ?X) will match to (PLUS 0 1 2 3 4), binding X to the list tail (1 2 3 4). Unfortunately that pattern will not match to (PLUS 1 0 2 3 4) or (PLUS 4 3 2 1 0) even though they are meant to be equivalent. LISP list structure forces a spurious ordering of the arguments. In QLISP, one can write the pattern as

```
(PLUS (BAG 0 ??X))
```



where ??X is a *segment variable*. A segment variable matches to any number of items, rather than a single one. Furthermore, the order of the items following BAG is irrelevant. So, the pattern will match to any of the following objects.

```
(PLUS (BAG 0 1 2 3 4))      X = (BAG 1 2 3 4)
(PLUS (BAG 1 0 2 3 4))      X = (BAG 1 2 3 4)
(PLUS (BAG 4 3 2 1 0))      X = (BAG 4 3 2 1)
(PLUS (BAG 0))              X = (BAG)
```

The so-called QLAMBDA functions in QLISP have a pattern instead of an argument list, much like PLANNER consequent theorems. The QLISP programmer can exploit the power of QLISP pattern matching to express computations very concisely, as in these one-liner examples:

```
PlusZero:      (QLAMBDA (PLUS (BAG 0 ??X)) (PLUS (BAG $$X)) )
PlusMinus:    (QLAMBDA (PLUS (BAG ??X ?Y (MINUS ?Y))) (PLUS (BAG $$X)) )
BothSets:     (QLAMBDA (TUPLE (CLASS ?X ??OTHERS) (CLASS ?X ??YETOTHERS)) $$X)
```

PlusZero, if its pattern matches, has found a 0 in the arguments to a PLUS and will return the PLUS without the 0. PlusMinus finds E and -E in the arguments, for some expression E, and removes them. BothSets finds an element common to the two sets (CLASSES) given to it, and returns that element. The reader may enjoy writing equivalent LISP functions and comparing them for length and clarity.

PLANNER and CONNIVER use pattern matching only in the contexts of pattern-directed retrieval and pattern-directed invocation. The QLISP programmer can call the pattern matcher directly, as well as use it implicitly in retrieval and invocation.

In most pattern matching applications only the pattern contains variables; it is matched to a constant object. Unification generalizes this to a "merge" or *unification* of two patterns both containing variables. The patterns unify if there is some substitution of values for variables such that they (the patterns) become identical. (A ?X) unifies with (?Y B) under the binding X = B, Y = A. Further discussion of unification may be found [yon AIH chapter on theorem proving]

QLISP uses a unification algorithm extended to handle the special data types and segment variables. Most commonly one of the patterns is all constant and unification reduces to a standard match. For some tasks, notably theorem proving, it is necessary to manipulate objects (like mathematical formulas) that contain variables, and then unification is very useful. It is significant that the first major program written in QLISP was a program verifier.

## IV.6. SAIL

In SAIL pattern matching is done only in the context of database retrieval. The FOREACH statement is the major way of doing this.

In the FOREACH statement patterns, instead of being list structures, are composed out of four types of expression:

1. Associative triples such as  $X \bullet Y \text{ fi } \text{GREEN, MOTHER} \bullet \text{ADRIAN fi } Z$ .
2. Restricted variables, using any Boolean function.
3. Expressions of the form  $X \text{ in } A$ , where  $A$  is a set.
4. *Matching procedures*, which are essentially generators.

These interact in a manner best shown by example.

```
FOREACH X, Y SUCH THAT X IN AnimalSet AND Gregarious(X) AND
Desert(Y) AND Range • X = Y DO PRINT(X);
```

The conjunctive conditions in the FOREACH are scanned left to right. Here, a set expression is first so some  $X$  in  $\text{AnimalSet}$  is chosen. Then  $X$  is tested to determine whether it satisfies the Boolean function  $\text{Gregarious}$ . If not, another  $X$  in  $\text{AnimalSet}$  is chosen. If so, the matching procedure  $\text{Desert}$  generates a  $Y$  and then the database is checked to determine whether  $\text{Range} \bullet X \text{ fi } Y$ . If not, another  $Y$  is generated. Pairs  $X$  and  $Y$  that meet all the conditions are passed on to the action part which in this case consists merely of printing  $X$ . The net effect of this FOREACH is to print out all gregarious animals that live in deserts.

In general, a FOREACH can have any number of variables and any number of conjoined conditions. The ordering of the conditions is critical; for instance if we had put the triple  $\text{Range} \bullet X \text{ fi } Y$  first, the FOREACH would find all matching triples in the associative database and try each pair of  $X$  and  $Y$  on the other conditions. (A matching procedure, when its variable is already bound, behaves as a Boolean function.) If the database includes a large amount of information concerning animal ranges, this would be highly inefficient. Note that PLANNER consequent theorems have a similar property, that the order in which subgoals are listed can enormously affect the size of the search space.

## IV.7. POP-2

Basic POP-2 does not have pattern matching. As with the AI control structure features, the pattern matching features are provided by various library packages, which consist of pattern matching procedures written in POP-2 which can be compiled so it is as if the user's program contained their

declarations. POP-2 is relatively well suited for doing pattern matching, but not so well as LISP because of the greater range of types POP-2 data may come in.

Four incompatible libraries offer significant pattern matching features.

LIB PICO-PLANNER has features similar to those of PLANNER.

LIB POPCORN and LIB POPLER have features similar to CONNIVER's; those of POPLER are actually a little more powerful than CONNIVER's.

LIB HBASE provides an associative database reminiscent of SAIL associative triples, except that two instead of three items are associated.

#### IV.8. SUMMARY

In the sequence PLANNER, CONNIVER, QLISP we saw a general upwards trend in sophistication of pattern matching and the range of uses it could be put to. PLANNER patterns had only one level of list structure and so would not be much help in analyzing the structure of data items, even if the language allowed that. They were quite effective for fetching assertions from the data base and choosing a function (theorem) to invoke, however. QLISP patterns are by comparison very general, and much, perhaps most, of the language's power depends on them. They serve as a major method for analyzing data, not just in the sense of extracting parts but of performing quite complicated tests and searches and returning the results.

Not surprisingly, pattern matching is expensive.

In almost any particular case, the pattern match algorithm will be more general than is really required, implying that replacing it with *ad hoc* code would yield a speedup. When the pattern contains segment variables (as in QLISP) the slowdown is especially severe. In this connection it is interesting to note that QLISP is termed by its designers [sace76] a "language for the interactive development of complex systems" with the explicit intention that once a QLISP program works, the user can cut it over to pure INTERLISP.

The user can even do so in stages, because QLISP and INTERLISP can be freely mixed.

None of the languages offer all the pattern matching power one could wish. For one thing, they all do exact structural matches. A very desirable feature would be "best match" capability: instead of matching exactly or failing, the matcher would do the best it could and return information on points of disagreement. Or, perhaps a "semantic match": match according to the *meaning* of pattern and object, whatever that means, instead of their syntactic structure.

QLISP's use of BAGs and CLASSES in which element order is irrelevant is a crude approach to semantic matching.

It is not clear whether pattern matching is somehow an essential or fundamental notion in data/control interaction. Randy Davis' TEIRESIAS (yon TEIRESIAS) is an example of an AI system in which control sequencing is guided by a set of strategy rules that take into account various data values and the current state of the process (and the application of these strategy rules can be guided by yet higher strategy rules, too.) This scheme bears no particular resemblance to pattern matching. Ideas about "focus of attention" as a way of selecting which data to operate on (attend to) have been floating around for several years. At any given point in a process there is a "focus" on a certain class or kind of data, yielding advantages both in efficiency, since the rest of the data base can be ignored, and in making the program's behavior more consistent.

Neither strategy rules, nor focus of attention, have been incorporated into any AI language yet. This is in accordance with a general rule, that before an advanced programming method is made a standard feature of any language, it will be experimented with by various people who develop it for their own programs.

## V. ENVIRONMENT

### V.1. MOTIVATION

The purpose of a programming environment is to aid the user in all phases of program development, from the initial writing of program text through modification, debugging, assembling of modules, and documentation, not necessarily in that order. The limitation is feasibility. As will become more or less apparent, designing and implementing the environment can easily be as much work as for the language itself. Given that people don't have infinite time to write environment support programs, nor the computer infinite space to hold them, the most crucial (useful) features need to be identified. The scope and quality of environment that is developed will depend greatly upon how strong the pressure is to do it. The amount of work spent on the six languages varies greatly and this tends to show up most strongly in the less central regions.

Of all computer languages, AI languages generally have the best environment facilities. As major reason, we cite the special difficulties of writing and improving AI programs.

AI programs tend to have certain characteristics that greatly influence the style of programming used. Most obviously, perhaps, they are big. As with other large programs, designers and programmers usually try to break the system down into several discrete modules which can be written separately. (By "module" we do not necessarily mean a single procedure—often many procedures which together perform a subtask.) It often happens in AI that the modules will interact strongly no matter how the breakdown is done; the programmer then has no choice but to write each module with the necessary flexibility for interactions. Finally, since the development of an AI program is usually a research effort, the program tends to be always in a state of flux, subject to very frequent modification and occasional major restructuring.

One style frequently used by AI programmers emphasizes incremental development, submodule by submodule and module by module. Each new module is added to what already exists and the expanded configuration is tested to see how it behaves with the new module added. An extreme case of incremental development occurs in rule-based systems where rules are added one or a few at a time until there are hundreds of them. During incremental development, the missing modules may have to be simulated by a person who interacts with the program. In other cases the system will run with only some of its modules. The speech-understanding system Hearsay-II [yon Speech] is an example of the latter; if some of its major modules (syntax expert, semantics expert, etc.) are unplugged it will decay gracefully, still running, but with more recognition errors.

Another programming style, "structured growth", is described in [sand78] (p. 60):

"... an initial program with a pure and simple structure is written, tested, and then allowed to grow by increasing the ambition of its modules. The process continues recursively as each module is rewritten. The principle applies ... to the flexibility of the

data handled by the program, the sophistication of deduction, the number and versatility of the services provided by the system, etc."

The classical "structured programming" method of stepwise refinement is not so much used in AI. For discussion of this point see [sand78] p. 60.

To sum up briefly, AI programmers have to impose some workable organization upon a large set of interacting modules, that is flexible enough to allow constant modification, correction and growth of the system, and have to do all this without bogging down in utter confusion. The support provided by a good programming environment is essential. One might succeed in writing a big AI program without it, but there is no reason one would want to.

Further on the question of why AI languages tend to have highly-developed environments, it is also significant that the environment system of a language often resembles an AI system in its own right. It may rely on a big database describing the program, and consists of several modules (i.e. the editor, debugger etc.) which can interact strongly. Consider then the advantages of writing environment programs in an AI language - presumably the one being supported, because it is there anyway. In general it is advantageous to write environment support in the language being supported, so that the support programs can be worked on within that environment.

Partly, also, AI environments are better because of closer contact between language users and implementors. Often, they are the same people.

One could make an endless list of desirable environment features. The most basic features are listed below.

1. An interactive language, i.e. one in which statements can be typed in as commands and are executed immediately. Compiler-based languages are generally not interactive.
2. A good editor. If possible, one that can deal with a program according to its structure (not just as a text of characters.)
3. Interactive debugging facilities including breaks, backtraces and examining and changing program variables.
4. I/O routines. The most common or ordinary I/O actions should be specially supported by standard system I/O functions, so that the programmer is not burdened with such details.

Some of our six languages are limited to the four features just listed, or even fewer. In the individual sections which follow, keep these basic features in mind - not all six languages have all the basic features - and also observe how some languages go beyond them.

## V.2. LISP

As with pattern matching, basic LISP lends itself quite well to use with various environment facilities. Basic LISP included no pattern matching, we saw, but pattern matching appeared in each of the specialized AI languages built on top of LISP. The situation here is not so extreme. All dialects of basic LISP have some environment features, notably that the language itself is interactive. PLANNER, CONNIVER and QLISP add more.

I will first list some advantages of LISP from the environmental support point of view, then mention the few environment features which are present in almost any LISP dialect you might run across. See also the LISP article [yon LISP article] where the former point is also discussed. With regard to the points below, keep in mind this idea: it is advantageous to implement the environment system in the same language it supports, so that environment facilities may be used to help develop and maintain their own code. Thus, a LISP environment should best be written in LISP.

1. LISP is an interpretive language. This means LISP is readily used in the interactive mode, which is a very important environment feature all by itself. The support programs will also be in LISP and so they, too, will be used interactively. Interpreted languages are also easier to debug, because in compiled languages a rather opaque the source program and the object program which actually executes. However, once the program works a compiled version tends to execute about 10 times faster. Most LISP systems have a compiler as well as the interpreter.
2. The basic language is simple and uniform (see LISP article [yon LISP article]), obviously a help to (environment) programs that have to examine or change LISP user programs.
3. Further along the lines of (2), there is a standard representation of LISP programs as list structures which environment programs can exploit.
4. The flexibility of function-calling in LISP makes it easy for environment programs to call one another, e.g. the debugger could call the editor to insert or remove printout statements.
5. LISP possesses an elementary database facility - the property lists and association lists - which is useful for storing information about user programs.

It is hard to think of anything about LISP that works *against* environmental support. In fact, all I could think of was its ugly (prefix) notation. If you make things easier for the user by introducing a nicer-formatted external form, as some LISP systems have done, you also complicate the job of your environment programs to a surprising degree. Some complications also accrue to the user. *Limited* use of external syntax seems to be a viable compromise, as in the CLISP feature of INTERLISP [teit??]

Almost any LISP system includes some debugging facilities and a scheme for printing LISP programs with indentations for easier reading (this is much more significant than you might

suppose!) Beyond that, they vary widely. In this article we will discuss the two major dialects, MACLISP and INTERLISP.

### V.3. PLANNER

PLANNER is implemented in MACLISP so we start with a brief description of the environment features of MACLISP.

The MACLISP system has no residential text editor, so the user has to leave MACLISP, edit programs with a separate editor, then return to MACLISP to try them again. Most computer languages, for that matter, are like this. Under the ITS timesharing system at MIT where micro-PLANNER was implemented, it is easy to switch back and forth between parallel editor and MACLISP jobs.

PLANNER was implemented by writing an interpreter for it in MACLISP. Little of the MACLISP environment

Only a minimal environment was produced because micro-PLANNER's implementors wanted to get a running system quickly and without too much effort. The environment also was written in MACLISP. We need to qualify the point made earlier, that it is best to write environment support programs for a language in that language itself. Since PLANNER's interpreter was written in MACLISP, the MACLISP system had to be around anyway and *its* environment could be used for work on the PLANNER environment. Indeed, given PLANNER's automatic backtracking control structure, it would be a very bad idea to write environment support programs in it.

Like MACLISP read-evaluate-print loop. If an error break occurs, PLANNER sets up a lower read-evaluate-print loop and enters it recursively. Inside such loops, the user can examine and change variables, or even run PLANNER programs it was stopped.

Instead of backtraces from break loops, PLANNER offers a trace-as-you-go facility. By saying (THTRACE <object>) the user requests a trace of some particular function, or goal, or all goals, or various other "objects"; then as programs run the system prints out information every time the goal or whatever is activated. Considering that PLANNER's control structure results in an extensive tree of goals and actions, it does make sense to trace the whole tree as it develops rather than backtracing from one particular node - which would reveal only one branch of the tree.

No new editor had to be provided; the same editor used for MACLISP was available.



#### V.4. CONNIVER

CONNIVER differs from PLANNER by including MACLISP as a subset. CONNIVER functions can freely call LISP functions, but CONNIVER is not just an extended LISP; it includes coroutines and other control regimes falling outside LISP's domain. The MACLISP environment will not work for CONNIVER. As with PLANNER, a minimal environment was implemented.

CONNIVER has read-evaluate-print loops very much like PLANNER's. Backtraces from breaks are done, rather than a trace-as-you-go. The editor is the same.

#### V.5. QLISP

QLISP derives from the problem-solving QA4 which was embedded into INTERLISP to make it more efficient and more accessible. Care was taken to preserve the extensive INTERLISP environment while extending the language with QLISP constructs. QLISP was specifically intended [sace76] as a "language for the interactive development of complex systems" and emphasizes environment features far more than the other five languages we are discussing. Since almost all these features are carry-overs from INTERLISP, we will first discuss the INTERLISP environment.

#### INTERLISP

The name INTERLISP, as the reader may have guessed, stands for "interactive LISP". This LISP variant has steadily grown in size for several years, both in the language and its supporting environment. For lack of space we will describe only some major environment facilities here. See the manual [wilb76] for more. Note that INTERLISP is a "residential" system: the facilities reside in core and can be called without the user leaving LISP. Another important general point is that the facilities are well integrated with each other. On occasion this integration can lead to unpredictable interactions and consequent difficulties [sand78] (p. 51).

The editor operates directly on the list structure of programs and data. The user can define composite editor commands as macros or even define macros that call arbitrary LISP programs. Conversely, INTERLISP programs can call the editor and give it commands.

The debugging package is quite ordinary. Upon an (error) break a new read-evaluate-print loop is entered, in which the user can ask for a backtrace, or evaluate any LISP statement (including statements to examine or change variables.) The trace-as-you-go (like PLANNER's) is also available.

Both a compiler and an interpreter are available.

An extensive I/O file package exists.

The "Programmer's Assistant" monitors and records all user inputs and retains a history of the computation. The user can ask the Assistant to repeat a command, undo its effects, or other things. The intended sense (to the user) is of a loyal flunkie who carefully watches what you do and can take over a lot of the dirty work if you ask.

The DWIM (Do What I Mean) package will attempt to figure out what you really meant to say, and do that, rather than follow orders literally. As a major part, DWIM corrects spelling errors. For instance, the user defines a function FLATTEN and then says (FLATEN A). DWIM will correct FLATEN to FLATTEN and proceed. DWIM causes annoyance at times, when it "corrects" something that wasn't wrong.

CLISP (Conversational LISP) is a set of convenient surface syntax constructs that translate into LISP. It includes various abbreviations, a record package, a pattern match compiler, and other things. It is implemented through the error mechanism of INTERLISP: anything not recognized as LISP is checked to see if it valid CLISP, otherwise it is an error.

### QLISP

QLISP is implemented through the error mechanism of INTERLISP just as CLISP is. When the interpreter comes to a QLISP construct it translates it into INTERLISP and executes that. It also stores the translated version so retranslation need not be done. In programs, LISP and QLISP can be freely mixed. QLISP is thus a surface extension of INTERLISP, whereas PLANNER and CONNIVER were distinct language systems built on top of MACLISP. Implementation of QLISP constructs through the error mechanism implies, you will note, that execution of pure LISP constructs is not slowed down at all by the presence of QLISP.

Some difficulties arose in trying to treat QLISP constructs as if they were on a par with INTERLISP constructs. For instance, because of the translation of QLISP into INTERLISP, QLISP functions would not show up on a trace or backtrace. A special QTRACE facility had to be added to get around this problem. And some "unfortunate interactions" resulted from the clever adaptation of some of the INTERLISP support facilities to implement certain QLISP language features. For instance, the CLISP pattern match feature becomes unuseable because QLISP uses it for its own pattern matching. The QLISP manual [wilb76] contains many warnings about how to avoid running into these painful effects.

For the most part, though, INTERLISP environment features carry over directly to QLISP.

QLISP also offers a standard function NEWSTATEMENT and an execution routine for some new statement. This extensibility feature is typically used to "provide alternative control structures for invoking the standard QLISP statements, or to provide special syntax for user-defined QLAMBDA functions."

## V.6. SAIL

SAIL emphasizes good access by user programs to the operating system. That is not an "environment" feature in the sense of a program-writing environment, but discussion of it fits better here than anywhere else.

It has many standard functions which are basically monitor calls, and "complete" access to PDP-10 facilities. The language was designed with real-time or close to real-time hand-eye applications in mind.

The run-time facilities include especially I/O facilities such as interrupt handling. There is also provision for linking SAIL programs to hand-coded assembly language algorithms (for efficiency.)

Since SAIL is a compiler-based language and not interactive, the experience of writing and debugging SAIL programs is considerably different from the other languages. Programs are written separately using whatever text editor is convenient, then compiled and run. Unlike the other five languages in which each (small) function is written separately, SAIL programs are block-structured and usually there will be many procedures nested inside the main program. Testing of individual procedures (functions) is less easy, because they cannot (usually) be run in isolation.

There is a quite good interactive debugger, however, The user can modify the error handling.

## V.7. POP-2

POP-2 has an incremental compiler, allowing an interactive style of programming similar to LISP. The user can type in any statement and have it executed immediately, or can define functions and edit them. The functions tend to be even smaller than LISP functions.

You will recall that POP-2 relies on library packages to keep the basic language as small as possible. The editor (which incidentally is not especially powerful and is not oriented to POP-2 syntax) resides in core all the time facilities must be brought in as packages. The packages cannot be integrated tightly like the environments features in INTERLISP, but compared to INTERLISP the core space occupied by POP-2 environment code is very small.

Some of the available packages are listed below.

LIB TRACE supports tracing for debugging purposes.

LIB STACKCH helps track down the source of stack errors.

LIB TIME measures execution time of individual functions.

LIB AUDIT reads the text of a POP-2 program, analyzes it, and prints out information about the functions defined and the variables used.

LIB CALL AND EXPLAIN allows package writers to attach explanatory messages to their packages. Package users are then provided with tutoring in the use of the packages.

## V.8. SUMMARY

The programming environment is the least AI-specific of the four AI language aspects we have examined. At the same time, it is an area in which AI languages have pioneered. We may expect, therefore, to see a great deal of influence upon other computer language environments in the future.

The key word is "interaction". In the environments we have examined, one can distinguish several different kinds of feature that all fall under the rubric of "interaction". The most important, perhaps, is for the language itself to be interactive. Programs can be built up of small modules (procedures, functions) which are each tested immediately, right after typing them in, rather than having to embed them inside a complete program and run it all through a compiler. Variables affected by the module can be examined and set by typing in appropriate statements of the language (e.g. assignment statements.)

As you can see, an interactive language provides much of what is ordinarily provided by an interactive debugger. LISP, PLANNER, CONNIVER and QLISP all do the same thing when an error break occurs during program execution: they set up a read-evaluate-print loop just like the one at top level. The user now has the full facilities of the language to figure out what went wrong.

In a break loop, these languages supplement the regular language features with special debugging facilities such as back-tracing and commands to abort or resume execution.

SAIL, which is the only non-interactive language of the six, has a quite extensive interactive debugger (BAIL) which includes special commands to examine and set variables.

Interactive editors are extremely useful. Many AI language environments have no special editor and the user uses a general text editor to prepare programs. SAIL and many LISPs including MACLISP have no resident editor. PLANNER and CONNIVER follow MACLISP in this respect. INTERLISP and POP-2 have resident editors; INTERLISP's editor operates on programs as LISP list expressions rather than as character strings. It has commands to move up and down in the list structure and can manipulate entire lists as single objects.

The Programmer's Assistant feature of INTERLISP exemplifies another kind of interaction, at the meta level in a sense. User commands to the Assistant refer not to LISP objects, but to the user's manipulations of LISP objects. The Assistant may be told to repeat an action, or show the recent history of user commands, for instance.

For smooth interaction it is important that the various components of an environment be well integrated with each other. INTERLISP is the only one of our six languages that (by and large)

accomplishes this integration. Almost any component can be called from within almost any other. In the other five languages, components can only be called from the top level generally. POP-2 carries this to the extreme: the components exist in distinct library packages and have to be loaded into core and compiled when you want to start using them.

In the introduction we said that "no matter how good an environment may be, one will have little difficulty thinking of further desirable features to add." To prod the reader's imagination, here is a list of a few desirable features that no language environment, to our knowledge, has.

Automatic testing of program modules with sample data. Automatic re-testing whenever a module is changed.

Automatic documentation

Program optimization

Transition from interpreted code to compiled code done automatically based on how much the program is being used.

So far we have treated the language environments as givens. But surely no one environment could please all users. Why not let the user tailor the environment to suit his or her own preferences? (Why not tailor the language too? That could lead to Babel. Tailoring the environment does not cause the Babel problem, because once a program works, anyone can use it without concerning themselves with how it grew.) INTERLISP actually allows user modification of its environment facilities. At the simplest level, there are a great many environment parameters which can be set as convenient. More advanced, and less common, a user can redefine environment functions.

What are the future directions of AI environments? At present the state of the art is represented by the INTERLISP environment. A few experimental systems are beginning to surpass it

## LISP

## INTRODUCTION

The name LISP stands for LISt Processing. In the late 50's and early 60's several list-processing languages were developed by AI researchers. Almost all AI computations may be characterized as symbol-processing, with the symbols embedded in some kind of list structure. Numeric processing and the use of "rigid" data structures such as arrays are less common. The early languages included IPL, SLIP, FLPL, and also LISP which soon superseded the others. See [ref: Overview article and perhaps others] for more about the first three.

The beginnings of LISP date from 1956. LISP, as recounted by its inventor John McCarthy [McCarthy 78], was originally motivated by his desire for a practical list processing language for artificial intelligence work on the IBM 704 computer. As the language developed, it became apparent that LISP could be mathematically elegant as well, and some care was taken to ensure this. McCarthy's first paper on LISP [McCarthy 1960] described it both as a practical programming language and as an idealized model of computation suitable for use in recursive function theory.

As described in [McCarthy 78] the key ideas that were embodied in LISP include:

- Computing with symbolic expressions rather than numbers. Bit patterns in a computer stand for arbitrary symbols, not just those of arithmetic.
- List processing. Representing data as linked-list structures in the machine and as multi-level lists on paper.
- Control structure based on the composition of functions to form more complex functions.
- Recursion as a way to describe processes and problems.
- Representation of LISP programs internally as linked-lists and externally as multi-level lists, i.e. in the same form as the data.
- The function EVAL serves as an interpreter of LISP and as a formal definition, within LISP, of the language itself.

All these points will be touched on in what follows. We will concentrate on LISP as a practical programming language, ignoring most of the theoretical issues. After describing its fundamental design, we define several of the most important language constructs with the intent that readers should then be able to understand small LISP programs. The Appendix contains some sample programs, of an "Alish flavor". After the definitions, some general implications of the structure of LISP for the way it is used are discussed, and some of its applications are mentioned. Lastly we list some disadvantages and difficulties of LISP.

We begin by describing the data and control structures of basic LISP, making no effort to be complete. For fuller introduction to the language see [Weissman 67], [Friedman 74], [Siklosy 76], or [Allen 78].

## DESCRIPTION OF LISP

**Data Structure** In basic LISP there is only one data type, generically referred to as "list structure". In practice, almost all LISP data takes the specific form of a *list* or else an *atom*, and we shall lose little by ignoring the other variations. Atoms are identifiers such as I-AM-AN-ATOM, 3, XYZ and NIL. They have no component parts, hence the name, but various properties or attributes can be attached to individual atoms. The most important attribute an atom can have is a *value*, in the sense of a variable having a value, and certain atoms have standard values--NIL has the value NIL, T has the value T, and any number atom such as 12, 1.732, -1.066E3 has the corresponding integer or floating-point number as value. Other atoms can have values bound to them. Lists may have atoms as elements, as in the four-element list (A B I-AM-AN-ATOM 3), or may have other lists as elements, as in the two-element list ( (A) (B I-AM-AN-ATOM 3) ), or any mixture of atoms and lists, for instance ( (AM (A (MORE COMPLICATED) LIST) ) ). In general, a list is defined recursively as

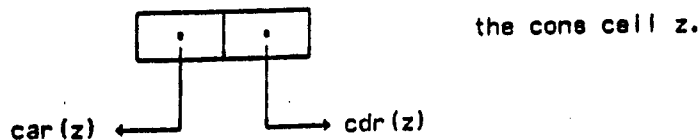
( element1 ... elementN )

where  $N \geq 0$  and each element is either an atom or a list. When  $N = 0$ , we have the *null* or *empty* list, written as () or NIL. As mentioned before, NIL is also an atom, and in fact it has the distinction of being the only datum that is both an atom and a list. Other special properties of NIL will appear later.

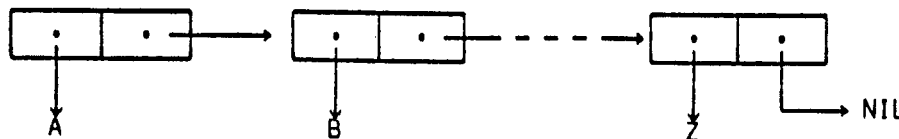
The recursive structure of lists is very flexible and allows one to represent many kinds of information. Below are examples of information represented with LISP lists.

(2 3 5 7 11 13 17 19)	A set of numbers.
((- B) + (SQRT ((B * B) - (4 * A * C))))	An algebraic expression.
(I (saw ((that (gasoline can)) explode)))	A parsed sentence.
(GREEN GRASS)	An assertion. ("Grass is green")
((ON A B) (ON A C) (NOT (TOUCH B C)) )	A set of assertions describing an arch.
((A C R) (A R C) (C A R) (C R A) (R A C) (R C A))	Permutations of three atoms.

Internally, LISP lists and all other list structures are represented by means of *cons cells*. Each cons cell is a pair of pointers (occupying the left and right halves of one computer word, usually) to other cons cells or to atoms, which do not point anywhere. A typical cons cell *z* is diagrammed below. What its left half points to is called the *car* of the cell *z*, and what its right half points to, the *cdr* (pronounced could-er.)



Using cons cells, the list ( A B ... Z ) is represented as follows:



In mathematics, sets are taken as the fundamental objects and then ordered pairs, sequences, tuples, relations and other objects are built up from sets. LISP data may be regarded as an alternative formalism in which the *ordered pair*, represented by the *cons cell*, is fundamental. Sequences and sets are then represented by LISP lists, an n-tuple by a list of length n, and a relation by a list of tuples. The list structure of LISP can be used to model essentially any data structure; thus a 2-dimensional array may be represented as a list of rows, each row in turn as a list of elements. The point, however, is not so much to model standard data structures (since this is often clumsy, as when one wishes to index into a two-dimensional array, and often unnecessary, because many LISP systems are augmented with special numeric, record, array and string data types) as to model the complicated and often unpredictable data structures that arise in symbol-processing tasks.

**Control Structure** LISP's control structure is basically *applicative*. That is, the flow of control is guided by the application of functions to arguments which may in turn be functions applied to *their* arguments. This contrasts with the sequential control structure of most languages, in which separate statements are executed one after another. As an example of the contrast, compare the ALGOL-like and LISP-like versions of a program to compute the two square roots of a nonnegative number.

```
ALGOL-like:  procedure ROOTS( value X: real; R1, R2: real );
              begin
                R1 ← SQRT(X);
                R2 ← -R1
              end
```

```
LISP-like:  ROOTS(X):
            BOTHSIGNS( SQRT(X) )

            where BOTHSIGNS(Y):
                LIST( Y MINUS(Y) )
```

The "LISP" function ROOTS computes the positive root and then applies a second function, BOTHSIGNS, to it. ROOTS(3.0) = BOTHSIGNS(1.732) = ( 1.732 -1.732 ). In LISP, procedures are not differentiated from functions and each function, whether it is part of the language or is user-defined, returns a (single) value.

Applicative structure similar to LISP's should be familiar to readers already, from the composition of functions in mathematics, or the structure of arithmetical and logical expressions in various languages. Indeed, LISP makes no distinction between a statement and an expression. The generic term "expression" covers both, and statement-like operations such as assigning a value to a variable can be freely intermixed with expression-like operations such as taking a square root. The term "evaluation" covers both evaluation of an expression and execution of a statement. Thus, evaluation of ROOTS(3) gives the answer ( 1.732 -1.732 ).

**Syntax** LISP syntax reflects the uniform control structure. A LISP expression is defined recursively as either an atom, which is treated as a variable and has a value, or a list of the form

```
( function argument ... argument )
```

with zero or more arguments, each of which is an expression. Such a list denotes the



application of "function" (which is given usually by name, but sometimes in the form of a LAMBDA-expression, to be discussed later) to the arguments. The function may be a standard function of the language, or may be user-defined. Thus, the body of ROOTS would actually be written (BOTHSIGNS (SQRT X)). X is a variable, BOTHSIGNS is the name of a user-defined function, and SQRT is the name of a standard function.

**Scoping** The scoping rule of LISP is also closely tied to the applicative control structure. Purely dynamic scoping is used: during the evaluation of a function F, a variable Z that is not local to F will have the value given to Z by whatever function G it was that called F; if G gave no value, then the value given Z by whatever function H called G; and so on. Scoping, then, depends only upon the calling order of functions ("dynamic") and in no way depends upon when or where they were declared ("static".)

**Recursion** Dynamic scoping allows the free use of recursive functions, i.e. functions that call themselves (with different argument values, one presumes!) The usual example of a recursive function is Factorial,  $N! = N*(N-1)!$  for  $N > 0$ , and  $0! = 1$ . In the section "Sample LISP Programs" we show a LISP version of Factorial and illustrate recursion with some other LISP programs also. In contrast to many languages, LISP places absolutely no restrictions on the use of recursion and in fact it is rare to write a LISP program that does not rely on it. (By "program", incidentally, we mean not necessarily a single function, but also a system of functions that together accomplish some task.) Recursion tends to make for concise and powerful programs. Complex programs that use recursion, especially *mutual recursion* in which e.g. F calls G and G calls F, would often be very messy and difficult to write without it.

The control structure of LISP, like the data structure, is uniform and based on a recursive definition. These characteristics can make the language quite confusing at times. They also have the distinct advantage of combining simplicity, flexibility and power.

**Some LISP Functions** There are a small number of basic LISP functions, in terms of which most other LISP functions can be defined. With the exception of the function CONS which causes a cons cell to be made, none of these basic functions have any side effects, so we describe them by the value they return.

Most LISP functions evaluate their arguments before computing anything from them--here, only QUOTE does not--and it is essential to pay attention to the distinction between an expression and the value of that expression. For instance, if B has been bound to the atom A, then  $\text{value}(\text{QUOTE } A) = A$ , but  $\text{value}(A) = B$ . If nothing had been bound to A,  $\text{value}(A)$  would be undefined.

In the table below, the symbols "e", "e1", "p1", etc. stand for any expressions given as arguments to the functions.

Expression -----	value( Expression ) -----	Comments -----
(QUOTE e)	e	QUOTE is essential in order to manipulate an expression itself, rather than its value (if it has any!) $\text{value}(\text{QUOTE } A) = A$ . From now on, we abbreviate (QUOTE e) as 'e.
(CAR e)	first element of the list value(e).	Not defined if value(e) is an atom. $\text{value}(\text{CAR } '(A B)) = A$ .
(CDR e)	what is left when the first element of value(e) is deleted.	Not defined if value(e) is an atom. $\text{value}(\text{CDR } '(A B)) = (B)$ . (CAR (CDR e)) is abbreviated (CADR e), (CDR (CDR (CAR e))) is abbreviated (CDDAR e), etc.
(CONS e1 e2)	list that results from prefixing value(e1) onto the list value(e2).	$\text{value}(\text{CONS } 'A '(B)) = (A B)$ . $\text{value}(\text{CONS } \text{NIL } \text{NIL}) = (\text{NIL})$ . Has side effect of setting up a cons cell.
(EQUAL e1 e2)	T if value(e1) is equal to value(e2), NIL otherwise.	Note how NIL is used to mean "False". $\text{value}(\text{EQUAL } '(B) (\text{CONS } 'B \text{NIL})) = T$ . $\text{value}(\text{EQUAL } e 'e) = \text{NIL}$ , usually, but $\text{value}(\text{EQUAL } \text{NIL } '\text{NIL}) = T$ .
(ATOM e)	T if value(e) is an atom, NIL if it is a list.	$\text{value}(\text{ATOM } \text{NIL}) = T$ . $\text{value}(\text{ATOM } (\text{CAR } '((A) B))) = \text{NIL}$ .
(COND (p1 e1) ... (pn en))	value(ei), where pi is the first of the p's whose value is not NIL. If all the p's are NIL, the value is NIL.	This is the basic "branching" or conditional function of LISP. $\text{value}(\text{COND } (\text{NIL } 'A) (T 'B)) = B$ . Some LISP systems have the abbreviation (IF p1 THEN e1 ELSE e2) for (COND (p1 e1) (T e2)). $\text{value}(\text{IF } (\text{CONS } \text{NIL } \text{NIL}) \text{ THEN } 1 \text{ ELSE } 2) = 1$ .
(EVAL e)	value(value(e))	EVAL is the opposite of QUOTE; $\text{value}(\text{EVAL } 'e) = \text{value}(e)$ . Suppose A is bound to a list of variable names, then $\text{value}(\text{EVAL } (\text{CADR } A)) = \text{value}$ of the second variable in the list.

Using only these functions, any computation can be done in LISP; i.e. these seven functions have the power of a Turing machine. However, they do not allow one to write programs in the style common to most programming languages, as a sequence of statements which operate by causing side effects. The prime example of this kind of statement is *assignment*. In LISP, the SET and SETQ functions perform assignment.

There are other LISP functions that operate by side-effect, but we need not describe them here.

(SET e1 e2)	value(e2)	Like all LISP functions, SET returns a value, but its real purpose is to assign value(e2) to the atom which is value(e1). Thus, after (SET 'A 'B), value(A) = B.
(SETQ e1 e2)	value(e2)	(SETQ e1 e2) is identical in value and effect to (SET 'e1 e2), and is the most common form.

Another construct common to most languages is block-structuring. In LISP, this is provided by the function PROG (meaning "Program".) It is really stretching terminology to call PROG a "function", since it is basically a Begin-End block with local variables, however PROG does have the standard LISP function syntax and does always return a value. Here is what a PROG block looks like.

```
(PROG (atom1 ... atomM) e1 e2 ... eN)
```

PROG does not evaluate its first argument, which has to be a list (possibly empty) of atoms; instead each atom is a local variable with, initially, no defined value. The subsequent expressions e1, e2, etc. are evaluated in that order, and if any one of them causes the special RETURN function to be evaluated,

```
(RETURN e0),
```

then immediately evaluation of the PROG stops and value(e0) is returned. If no RETURN is encountered, all of e1 ... eN are evaluated and the default value NIL is returned.

Here are some miscellaneous functions which are commonly used, defined in terms of the basic functions:

(NULL e)	"	(IF e THEN NIL ELSE T)
or (NOT e)		
(OR e1 ... en)	"	(COND (e1 T) ... (en T) (T NIL) )
(AND e1 ... en)	"	(COND ((NOT e1) NIL) ... ((NOT en) NIL) (T T) )
(LIST e1 ... en)	"	(CONS e1 (CONS ... (CONS en NIL) ...))

Example: value( (LIST 'A A) ) = (A value(A) ).

User-defined functions in LISP are written in a notation derived from the  $\lambda$ -calculus of Church [Church 41]. The LISP LAMBDA corresponds to a procedure declaration in an ALGOL-like language, in the loose sort of way that PROG corresponds to a Begin-End block.

```
(LAMBDA (atom1 ... atomM) e)
```

denotes a function of M parameters (note that no type declarations are necessary!) When a function is called, giving M expressions as actual parameters, all actual parameters are evaluated and their values are bound to the formal parameter atoms. So, if the user defines

```
Exchange: (LAMBDA (X) (LIST (CADR X) (CAR X) )
```

```

then value( (Exchange '(A B)) ) = value( (LIST (CADR '(A B)) (CAR '(A B))) )
      = ( value( (CADR '(A B)) ) value( (CAR '(A B)) ) ) = (B A).

```

But `value( (Exchange (A B)) )` is not defined unless A is a function and B has a value.  
 We have now introduced enough of LISP to cover the sample programs in the Appendix.

**Allocation and Garbage Collection.** LISP relies completely upon dynamic allocation of space for data storage. During execution of a program, each evaluation of the CONS function causes one cons cell to be made, and one computer word is allocated for the cell. Gradually, the program's core space is filled up and if nothing were done, it would soon be jammed full of cons cells. Fortunately, once the program has used a cons cell it often "forgets" all about it--i.e. retains no direct or indirect pointer to it, and can never access it again-- so the storage locations of these old cons cells can be recycled. When core begins to get full, the user program is suspended and the so-called Garbage Collector (a system program) is called. The Garbage Collector locates all the forgotten cells and makes their storage locations available to the user program, which is then resumed.

Comparing this scheme to static allocation, in which each variable or array or whatever has a fixed amount of storage reserved for it before the program executes, it is clear that static allocation involves less overhead. But for LISP, in which list structures grow unpredictably, it would be hopelessly restrictive. The time spent in garbage collection is part of the price paid for the flexibility of LISP's data structure.

## SOME MAJOR POINTS ABOUT LISP

In the previous section, it was shown how in LISP, data and programs are both highly recursive, and are represented as nested lists. There are closer connections between them.

**Programs mirror data** In LISP, functions may be written so as to mirror the structure of the data they operate upon, best shown by an example. The function `Substitute` takes any list "Object" and generates a copy of it in which every occurrence of a given atom "Old" is replaced by another list or atom "New".

```
Substitute: (LAMBDA (Object Old New)
            (IF (ATOM Object)
                THEN (IF (EQUAL Object Old) THEN New ELSE Object)
                ELSE (CONS (Substitute (CAR Object) Old New)
                          (Substitute (CDR Object) Old New))))
```

Suppose we evaluate `(Substitute '(PLUS (TIMES A X) X) 'X (PLUS 2 3))`. `Substitute` is called a total of 13 times. If one draws out the internal representation of `(PLUS (TIMES A X) B)`, one finds it to contain 6 cons cells and 7 atoms. `Substitute` is called exactly once for each of these pieces, and the tree-structure of instances of `Substitute` is isomorphic to the structure of `Object`. There are two cases: either `Object` is an atom, and the appropriate value is returned without recursion, or `Object` is a list, in which case `Substitute` is applied recursively to the `CAR` and `CDR` of `Object`, and the results `CONSD` together. Note that if `Old` does not occur in `Object`, then the body of `Substitute` simplifies to

```
(IF (ATOM Object)
    THEN Object
    ELSE (CONS (CAR Object) (CDR Object))).
```

As the reader can imagine, this parallelism of control structure to data structure is an enormous help in dealing with complex nested data. It is very characteristic of programming in LISP that when writing functions one need only concern oneself with the recursive definition of the data and not with what it looks like in the large. Pattern-matching and parsing are cases in point. In parsing, the input data is linear but the output is tree-structured. The grammar used in the parse may be regarded as a recursive definition of the structure of this output.

**Programs are data** The internal representation of a LISP program (assuming it has not been compiled) is the same as that of any other multi-level list, i.e. cons cells and atoms. LISP is unique among programming languages in storing its programs as *structured* data (of course many languages store them unstructured, as bit strings or sequences of tokens.) This property is very important, for several reasons.

First, it is particularly easy to write LISP programs that generate LISP expressions and programs, as in automatic programming [\*\*\*\*ref: handbook chapter on automatic programming] and other AI applications.

Second, functions can be passed as parameters to other functions. (Just as any list structure can.) For instance, suppose we had a function to do minimax game-tree search. [ref Search chapter.] We could pass it an evaluation function as a parameter; changing this parameter, under control of some strategy, would cause different moves to be chosen by minimax.

Third, there is procedural representation of knowledge. LISP procedures to find facts can be stored in a database as if they were facts. For instance, evaluation of either expression below would return the value `(7 -7)`. `.BEGIN ALG; (QUOTE (7 -7))` or `'(7 -7)`

`(ROOTS 49) .END;` In more complicated cases the procedural representation could do arbitrary deductions before returning a value `v`, yet from the outside it is just as if one evaluated `'v`.

Of more fundamental importance is the manner in which LISP can be a foundation for more advanced languages. The method is to write an interpreter, in LISP, for a new LISP-like

language. Syntactic constructs of the new language are represented as multilevel lists just as in LISP itself, making the interpretation relatively easy to do. The special AI languages micro-PLANNER [Sussman 71], CONNIVER [McDermott 72], and QLISP [Sacerdoti 76] were all implemented in this way, incorporating powerful features such as automatic backtracking and pattern matching. Of course there is a severe slow-down inherent in the use of an extra level of interpretation, but in a research environment that is usually acceptable.

**LISP is interpretive** Originally LISP was to be a compiler-based language. While design of the first compiler was underway, it was suddenly noticed that the LISP function EVAL was in essence a LISP interpreter. Before, EVAL had been of theoretical interest only and existed only on paper. It was soon hand-coded, and long before a compiler, a LISP interpreter became available.

Interpretive execution (or evaluation) has strong advantages during the program development phase, chiefly that it permits interactive programming, and so it remains generally used. Once a program is fully operational it is usually compiled for greater speed.

An important consequence of interpretation is greater flexibility of the language itself. An interpreter is much more accessible to change than a compiler, especially when it is written in a high-level language--like LISP. Most of the existing dialects of LISP have appeared when heavy LISP users (researchers) at an installation modified and extended LISP to suit their particular needs.

**LISP is interactive** Any interactive language system must be interpreter-based rather than compiler-based. LISP interpretation is far easier than for most languages, because of the uniform syntax, and other features such as dynamic allocation and the absence of type declarations also suit it to interactive use. Essentially all existing LISP systems are interactive.

Since LISP encourages the composition of large programs out of many small functions, large programs (like those in AI) can be developed incrementally by writing and debugging the component functions one at a time.

There exist a few quite large LISP systems that provide not only the direct language support but an entire "environment" for interactive LISP programming, including editors, debugging and tracing facilities, and alternative syntactic forms more convenient than the pure list notation. These LISP systems also extend the basic language with additional special-purpose functions and sometimes new data types.

One such system is INTERLISP [Tietelman 75] which has over 600 functions; string, record, array and hash-array data types; a special stack for coroutines and other advanced control structures; and an extensive environment. Another is MACLISP [ref ??], with a much smaller environment, but excellent arithmetical facilities.

## APPLICATIONS

**Applications in AI** Throughout its history the major application of LISP has always been in artificial intelligence research. During the initial implementation period 1958-1962 several AI programs were written in the new language [Berkeley 66].

The great majority of AI programs since 1960 have been written in LISP, or in a language built on top of LISP. The table below lists a few such programs, their task domain and LISP-related features that are prominent in each program.

Program -----	Task domain -----	LISP-related -----
Analogy [Evans 68]	solving geometrical analogy problems similar to those in intelligence tests	complex, unpredictable data structures representing the figures and analogies
SHRDLU [Winograd 72]	natural language understanding and robot planning	recursive planning; written in micro-PLANNER and in a parser language, PROGRAMMAR.
[Boyer and Moore 75] [Manna 77]	proving theorems about LISP functions, program synthesis	LISP functions treated as data
MYCIN [Davis 77]	medical diagnosis	based on production rules

The books by Allen [Allen 78] and Winston [Winston 77] have some good examples of how LISP programs are written for various applications.

**Other applications** LISP is widely applied in the mathematical theory of computation and in program verification work (both because of its uniform structure and clean semantics). MACSYMA [The Mathlab Group 77], a powerful mathematics program for algebra, integration, etc. is written in MACLISP. In the larger realms of general scientific or commercial programming, LISP is seldom used.

Some claim LISP is not only the language of choice for AI and a few other fields, but should be used in many other areas too, and so far has not due to historical and sociological causes [Pratt]. It is argued that top-level syntax can be added so that LISP will include most features of ALGOL-like languages, and that better compilers would make LISP at least as efficient, in both symbolic and numeric computations, as any other high-level language.

### Disadvantages of LISP

**Ugly syntax.** A common complaint about the list format of LISP programs is that they are difficult to read. The only syntactic items are separators such as spaces, and parentheses which provide most of the structure. This way of representing structure is convenient for the machine reading it but inconvenient for humans, so in practice programs are printed so that the structure is also indicated by indenting. People have also tried to implement alternative input languages for LISP programs. None of these, however, have caught on, and list structure program input is still predominant. The main exception, perhaps, is CLISP provided by INTERLISP as an alternative input syntax which is designed so that it can be mixed with ordinary LISP. Not all INTERLISP users like this though. One advantage of LISP input syntax is that the relation to its internal representation is clear for a user. This is useful, for example, where programs manipulate other programs such as in a list oriented function editor.

**One data type.** The lack of distinct data types is harmful when it prevents type-checking at interpretation-time or compile-time, which would often detect bugs. Many

LISP systems do support a few additional data types, and QLISP has a wide range of types that are useful in symbolic computation- bags, tuples, relations, etc.

**Inefficiency.** As with any language, LISP is quite slow when executed interpretively, especially if additional levels of interpretation are interposed by languages "built on" LISP, e.g. PLANNER. Speed is traded for convenience and extensibility. LISP can be compiled and is then of moderate efficiency.

**Lack of a standard.** Unlike FORTRAN and other well-known languages, there has never been an attempt to agree on a standardized LISP. The lack of a language standard and the proliferation of incompatible versions make LISP badly suited to be a production language, and in AI research work there are severe difficulties in transporting LISP programs to machines using a different LISP. A partial cure is to leave the program on its home machine and use it there via a computer network such as the ARPA net

**Historical problems.** The last common ancestor of the diverse LISPs now in existence is LISP 1.5, defined in 1962 [McCarthy 62]. Some of its limitations are still widespread in later LISPs, e.g. weak arithmetic and inconvenient input/output formats.

These disadvantages are mostly inherent in the language and can only partially be alleviated. Even the lack of a language standard is inherent to the extent that LISP's flexibility works against standardization. Some of the disadvantages listed are less bothersome in a research setting (such as that of almost all AI work) than they would be elsewhere. The notorious slowness of arithmetic in most LISPs is not an inherent problem; as MACLISP demonstrates, if the demand exists LISP arithmetic can be made quite efficient.

## CONCLUSION

LISP is the second oldest programming language still in widespread use. John McCarthy [McCarthy 1978] attributes its longevity to two facts. First, LISP "occupies some kind of local optimum in the space of programming languages" through its combination of recursion, the representation of symbolic information by list structures, and representation of programs in the same way. Second, its uniform syntax and semantics make LISP a good target language into which to interpret or compile from a higher-level language.

The major application of LISP has always been in Artificial Intelligence research, where it is currently the dominant language and shows no particular signs of being replaced by the newer AI languages (PLANNER, CONNIVER, SAIL, POPLER, and others--see [\*\*\*\*ref AI Langs. Historical Overview, Comparison of AI Langs].) LISP is not specialized to artificial intelligence applications in the same sense as newer languages with such features as pattern-invoked procedures or automatic backtracking. It is first and foremost a list-processing language, and the basic operations of CAR, CDR and CONS are quite close to the machine level. So far, though, it has generally been thought of as an AI language.

The basic concepts of LISP serve as a good introduction to features found in most or all AI languages, such as representation of information in list structures, recursion, easy extensibility of the language, and the potential to be used interactively.



## APPENDIX: SAMPLE PROGRAMS

**Factorial - a simple recursive program.** This example was chosen for the sole purpose of showing how recursion works in LISP. Even so, it is not unrealistically simple; LISP functions of comparable simplicity are by no means rare.

```
Factorial: (LAMBDA (N) (IF (EQUAL N 0) THEN 1
                          ELSE (TIMES N (Factorial (SUB1 N))))))
```

The evaluation of (Factorial 3) proceeds as follows: N is bound to 3 and the expression (IF (EQUAL N 0) THEN ... ELSE ...) is evaluated. Since (EQUAL N 0) evaluates to NIL, i.e. "false", the expression (TIMES N (Factorial (SUB1 N))) is to be evaluated. Its value is 3\*value( (Factorial 2) ), and so Factorial is called recursively. A separate instance of the variable N is allocated, bound to 2 and the process continues.

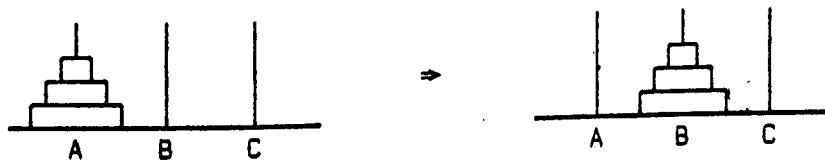
```
value( (Factorial 3) ) = 3 * value( (Factorial 2) )
value( (Factorial 2) ) = 2 * value( (Factorial 1) )
value( (Factorial 1) ) = 1 * value( (Factorial 0) )
value( (Factorial 0) ) = 1.
```

The fourth call, with N = 0, does not recurse. Control returns to the third call and then to the second and first.

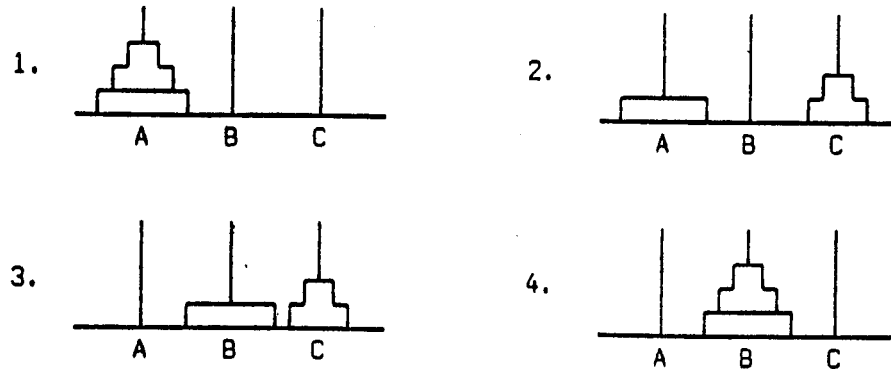
```
value( (Factorial 1) ) = 1 * 1 = 1,
value( (Factorial 2) ) = 2 * 1 = 2,
value( (Factorial 3) ) = 3 * 2 = 6.
```

Recursion can be a method of great power when some complex task is to be performed. If the programmer can think of a way to reduce it to one or more smaller tasks of the same kind, then there is no need to explicitly state (in a program) how the task as a whole is to be done. In the Factorial example, knowing only the reduction  $N! = N*(N-1)!$  (and the base case  $0! = 1$ ), we wrote a recursive program that implicitly computes  $N! = N*(N-1)*(N-2)*... *2*1*1$ . A more impressive example follows.

**Towers of Hanoi- problem solving by recursion.** In the Towers of Hanoi puzzle, a tower of (generally eight) disks must be transferred from Peg A to Peg B, moving one at a time, and never placing a larger disk on top of a smaller. Peg C may be used.



To move a tower of two disks, one would move Disk 1 from A to C, Disk 2 from A to B, and then Disk 1 from C to B. Now suppose we wish to transfer a tower of N disks, this problem can be reduced to three steps: transfer the tower of Disks 1 through N-1 from A to C, move Disk N from A to B, then transfer the tower from C to B.



Of course if  $N = 1$ , just move the disk from A to B. Now immediately we can write a recursive program to print out the solution for any number of disks.

```
MoveTower: (LAMBDA (DiskList PegA PegB PegC)
            (IF DiskList THEN
              (PROG (
                    (MoveTower (CDR DiskList) PegA PegC PegB)
                    (PRINT (LIST 'Move (CAR DiskList) 'from PegA 'to PegB))
                    (MoveTower (CDR DiskList) PegC PegB PegA))))))
```

Here is the printout resulting from evaluation of (MoveTower '(Disk1 Disk2 Disk3) 'A 'B 'C):

```
(Move Disk1 from A to B)
(Move Disk2 from A to C)
(Move Disk1 from B to C)
(Move Disk3 from A to B)
(Move Disk1 from C to A)
(Move Disk2 from C to B)
(Move Disk1 from A to B)
NIL
```

Observe the following points:

1. NIL is returned as the value of the PROG.
2. Each instance of the function MoveTower calls two other instances. The reader is invited to draw the tree of recursive calls to MoveTower and calculate how many moves it takes for N disks. By the way, it is not hard to see that the solution is optimal.
3. One may regard the tree of calls to MoveTower as a problem-reduction tree consisting entirely of AND nodes. Solution of problems by problem-reduction and recursion is very common in AI, [\*\*\*\*refs to thm-proving, pattern-matching, block-stacking; perhaps to some general discussion in the Problem-Solving chapter]. LISP is well suited to most such methods.

**Manipulating facts and rules.** Recall from the data structure section how assertions about objects may be represented as LISP lists. The next example program makes simple logical deductions from a database of assertions. The assertions are of two kinds: a fact that a certain predicate is true of a certain object, e.g. (MAN Socrates), and a general rule that one predicate implies another. A suitable *ad hoc* representation for the rules is (ALL predicate1

predicate2), e.g. (ALL MAN MORTAL). Given a statement such as (MORTAL Socrates) and a database, the program Prove will return T if the statement can be deduced from its database, NIL if not.

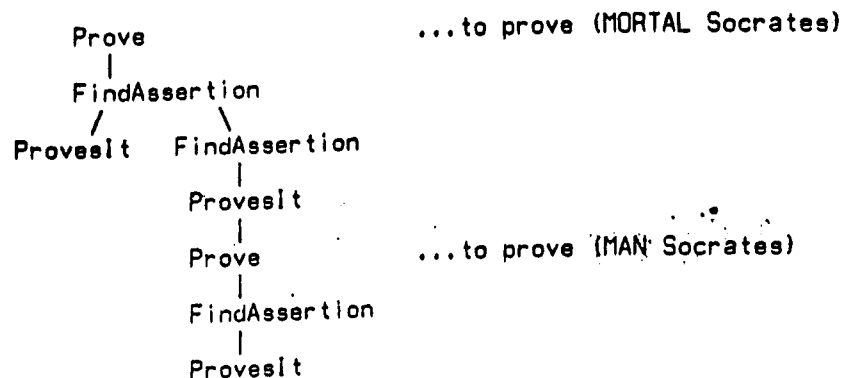
```
Prove:      (LAMBDA (Statement DataBase) (FindAssertion DataBase))
```

```
FindAssertion: (LAMBDA (RestOfDataBase)
                (IF (NULL RestOfDataBase)
                    THEN NIL
                    ELSE (OR (ProvesIt (CAR RestOfDataBase))
                            (FindAssertion (CDR RestOfDataBase))))))
```

```
ProvesIt:   (LAMBDA (Assertion)
            (OR (EQUAL Statement Assertion)
                (AND (EQUAL (CAR Assertion) 'ALL)
                    (EQUAL (CADR Assertion) (CAR Statement))
                    (Prove (CONS (CADR Assertion) (CDR Statement))
                          DataBase))))
```

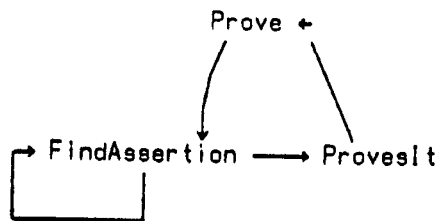
We would describe this system of three functions in English by saying, "A statement can be proven from a database if the first assertion in the database proves it, or it can be proven from the rest of the database. An assertion proves a statement of the form (Predicate Object) if either it is identical with the statement, or it is of the form (ALL Predicate2 Predicate) and the new statement (Predicate2 Object) can be proven from the database."

Evaluation of (Prove '(MORTAL Socrates) '((MAN Socrates) (ALL MAN MORTAL))) builds up the following tree of function calls before returning T:



Observe the following points:

1. Clarity is enhanced by splitting the prover, which *could* be written as a single function, into three smallish functions that each perform an identifiable task.
2. The three functions are purely applicative - no assignments or PROG statements. LISP programs tend to be most elegant when written in this style, but for MoveTower it would have been clumsy.
3. Though they do not directly call themselves, the functions Prove and ProvesIt are still considered recursive, since the system of three functions is mutually recursive - from each function, there is a chain of calls leading back to that function.



4. Of the three mutually recursive functions, only FindAssertion is recursive as it stands. FindAssertion illustrates a very common scheme for recursion down a list, which takes the general form:

```
F: (LAMBDA (x) (IF (NULL x) THEN NIL ELSE (G (H (CAR x))
                                             (F (CDR x))))))
```

In FindAssertion, G = OR, making

$$\text{value}(\text{Find } '(e1 \dots eN)) = \text{value}(\text{OR } (H e1) \dots (H eN)),$$

where H is Provesit. If G were CONS instead of OR, another very useful function would result:

$$\text{value}(F '(e1 \dots eN)) = (\text{value}(H e1) \dots \text{value}(H eN)).$$

5. The variable Statement is not defined as a parameter to Provesit, and so the dynamic scoping rule of LISP applies. Does FindAssertion, which called Provesit, define a value for Statement? No, but Prove, which called FindAssertion, did. So that value is used. If Provesit now calls Prove and passes it a newly-constructed statement as parameter, this becomes the new value of Statement and is used by the next lower instances of Provesit.

## LISP

## REFERENCES

- Allen, John. *Anatomy of LISP*.  
New York: McGraw-Hill, 1978.
- Berkeley, Edmund C. and Bobrow, Daniel G. (eds.), *The Programming Language LISP: Its Operation and Applications*, 2nd ed.  
Cambridge, Mass.: The MIT Press, 1966.
- Boyer, Robert S. and Moore, J. Strother. "Proving Theorems about LISP Functions", *JACM* Vol. 22 No. 1, 1975. pp 129-144.
- Church, Alonzo. *Calculus of Lambda Conversion*.  
Princeton, N.J.: Princeton Univ. Press, 1941
- Davis, Randall and Buchanan, Bruce. "Production Rules as a Representation for a Knowledge-Based Consultation Program",  
*Artificial Intelligence* Vol. 8, 1977. pp. 15-45.
- Evans, Thomas G. "A Program for the Solution of Geometric-Analogy Intelligence Test Questions", In Minsky, Marvin (ed.)  
*Semantic Information Processing*.  
Cambridge, Mass.: MIT Press, 1968.
- Friedman, Daniel P. *The Little LISP*.  
Chicago: Science Research Associates, 1974.
- Manna, Zohar and Waldinger, Richard. "Synthesis: Dreams => Programs",  
SAIL Memo AIM-302, Computer Science Dept., Stanford, 1977.
- Maurer, Ward D. *The Programmer's Introduction to LISP*.  
London: Macdonald, New York: American Elsevier, 1972.
- McCarthy, John "Recursive Functions of Symbolic Expressions and their Computation by Machine", *CACM* 3 (1960), pp. 184-196.
- McCarthy, John; Abrahams, Paul W.; Edwards, Daniel J.; Hart, Timothy P. and Levin, Michael I. *LISP 1.5 Programmer's Manual*.  
Cambridge, Mass.: The MIT Press, 1962.
- McCarthy, John "History of LISP" *ACM SIGPLAN Notices*,  
Vol 13 No 8 August 1978 pp. 217-223.
- McDermott, V. et. al. "The CONNIVER Reference Manual",  
AI-M-259, MIT, Cambridge, Mass. 1972.
- Pratt, Vaughn R. "LISP: An Amicus Curiae Brief". \*\*\*\*\* where? when?
- Sacerdoti, Earl et. al. "QLISP: A language for the interactive development of complex systems",  
SRI International, Menlo Park, Calif. 1976.
- Sandewall, Erik. "Programming in an Interactive Environment: The "LISP" Experience", *Computing Surveys*, Vol. 10 No. 1  
March 1978, pp. 35-71.
- Siklossy, Laurent. *Let's Talk LISP*.  
Englewood Cliffs, N.J.: Prentice-Hall, 1976.

Sussman, Gerald; Winograd, Terry and Charniak, Eugene. "Micro-PLANNER Reference Manual", AI-M-203A, Artificial Intelligence Lab, MIT, Cambridge, Mass. 1971.

Teitelman, Warren. "INTERLISP Reference Manual".  
Xerox - Palo Alto Research Center, Palo Alto, Calif. 1976.

The Mathlab Group. *MACSYMA Reference Manual*.  
Laboratory for Computer Science, MIT Version 9,  
Cambridge, Mass. 1977.

Weissman, Clark *LISP 1.5 Primer*.  
Belmont, Calif.: Dickenson Publishing Company, 1967.

Winograd, Terry. *Understanding Natural Language*.  
New York: Academic Press, 1972.

Winston, Patrick H. *Artificial Intelligence*.  
Addison-Wesley, 1977.

## I. PLANNER and CONNIVER

### I.1. PLANNER

Like LISP, PLANNER was developed at MIT. Carl Hewitt of the MIT AI Lab began development of the PLANNER concepts in about 1967 and they continued to be developed for several years after that. [refs] Strictly speaking PLANNER is a loose (?) framework for expressing problem solving systems, not an actual computer language. Only a smallish subset of the ideas have ever been implemented, those included in the Micro-PLANNER language defined in 1971 [Sussman etc 1971.] That implementation was made largely because Terry Winograd wanted to write part of his natural-language dialog robot [ref] in PLANNER. Unless stated otherwise, when we refer to "PLANNER" in this article we mean Micro-PLANNER. This loose terminology is very common.

PLANNER introduced three important ideas.

Automatic backtracking as a control structure

Pattern-directed search of a (global) database

Pattern-directed invocation of procedures

Two of these have been extremely influential, the other less so. They are combined into a system oriented towards achieving goals (by solving problems.) In the typical use of PLANNER, the database represents the state of the world and a certain goal is set up. If pattern-directed database search finds that the goal is already achieved (current state of the world satisfies the goal) then the program halts. Otherwise pattern-directed invocation fires off an appropriate procedure that reduces the goal to subgoals, and the process recurses. Automatic backtracking comes into play if that procedure fails; another one is chosen. The PLANNER process is described in more detail in Section III.

The PLANNER interpreter is written in LISP and close analogues of most basic LISP functions are present in PLANNER. There is no PLANNER compiler [perhaps that would be impossible.] Since the LISP itself is not especially fast, with the additional layer of interpretation PLANNER becomes quite slow, and this is severely aggravated by the automatic backtracking which can easily lead to very big search trees. The upshot is that PLANNER can be grossly inefficient. This, along with other problems associated with the automatic backtracking, seem to be two major reasons why PLANNER is not used very much any more. However that may be, it is of considerable historical importance as the first language to allow programming at the level of goals and methods, leaving the details of choosing which procedure to call at which point, etc., up to the system.

## 1.2. CONNIVER

CONNIVER was largely inspired by the successes - and the defects - of PLANNER. Sussman and Drew McDermott, both at MIT (recall that Sussman was one of the Implementors of Micro-PLANNER), studied PLANNER and decided that its basic control structure was wrong:

*"automatic backtracking ... is the wrong structure for the domain for which PLANNER was intended, that is, Artificial Intelligence."*

The new language was an integration of ideas from PLANNER and elsewhere. As Sussman and McDermott say,

*"Conniver embodies few original ideas but is hopefully an original combination of the good ideas of others."*

While rejecting (wholeheartedly) the control structure of PLANNER the developers of CONNIVER retained essentially all of its data structuring and pattern matching ideas. A more general control structure was adopted, based on the now standard techniques given in [ref Bobrow and Wegbreit]. The control primitives of CONNIVER are at a lower level than those of PLANNER but enable the programmer to construct a wide variety of control regimes, including the backtracking regime if so desired. A major innovation

of CONNIVER was the subdivision of the global data base into separate "contexts" (more about contexts later.)

CONNIVER is interpreter based like PLANNER, the interpreter again being written in LISP. CONNIVER is integrated quite nicely into LISP; to the CONNIVER programmer LISP appears as a subset of CONNIVER. Considered as an extension of LISP, CONNIVER has three main additions:

A system-maintained data base organized into a tree of contexts

Pattern matching for database search and procedure invocation

Extended control facilities allowing generators, coroutines etc.



## I. QLISP and INTERLISP

The roots of QLISP go back about as far as PLANNER's, to a deductive question-answering system known as QA3 [ref to ccg's '69 thesis], and through QA3 to QA4 and then QLISP. All three language systems were developed at SRI in Menlo Park. QA3 stored a database of facts expressed in first order predicate calculus and, when given a question, would obtain an answer to it by resolution with the data base [ref to fuller descr. of QA3 method in AIH].

QLISP started as an expedient for more effective use of QA4. QA4 users suffered from a lack of good debugging and other environmental facilities; this was overcome by embedding QA4 into the new version of LISP called INTERLISP, giving access to the strong environmental features of that language. The result was called QLISP. Users had also suffered from the slowness of QA4, and by sacrificing some generality in QLISP a factor of about 30 speedup was obtained. Extensions beyond QA4 were then added, notably a quite sophisticated unification pattern matcher and, after the "spaghetti stack" mechanism was added to INTERLISP, corouting and other generalized control structures. QLISP reached mature form by 1976, latest of the advanced LISP-based languages PLANNER, CONNIVER and QLISP. It has been implemented on the PDP-10 in TENEX INTERLISP and on the IBM 360 and 370 in INTERLISP-370. A good many of QLISP's interesting features come from or depend upon INTERLISP, so we will have more to say about it in Section III.

As indicated in the title of [ref "QLISP: a ..."], QLISP is intended [was it from the start?] as "a language for the interactive *development* of complex systems." (Italics mine.) Its designers and implementors recognized that compared to straight LISP their higher-level language would inevitably be less efficient, and intended that once a user's program was working in QLISP he could cut over to straight INTERLISP. This sort of method is quite commonly done in AI, for instance the CONGEN part of the DENDRAL system [ref] was originally written in INTERLISP but once the algorithm was perfected it was completely rewritten in PDP-10 assembler. In the case of QLISP the transition to INTERLISP is relatively easy, because free intermixing of the special QLISP constructs with INTERLISP code is allowed. From the start those parts of a program for which INTERLISP is enough can be written in it, and the rest can be translated in stages. Both this advantage, and the excellent QLISP environment, spring from QLISP's tight integration with the underlying INTERLISP.

Two QLISP features of particular interest are its organization of the data base into a "discrimination net", some new data types, and the use of unification pattern matching.

## I. SAIL

SAIL (Stanford Artificial Intelligence Language) is an extension of ALGOL 60 developed at Stanford (We will assume the reader has some familiarity with ALGOL or a similar language.) The first version came out in 1969 and development continued for several years after that. As a compiler-based language and with a more restricted set of AI features than PLANNER, CONNIVER and QLISP, SAIL provides a programming experience much closer to the general programming tradition.

The orientation of SAIL is very different from the other languages so far introduced. It was intended chiefly for AI research in the areas of vision, speech and hand-eye systems, all domains where there is a large quantity of "real-world" data coming in from real devices that must be processed, including a large component of numeric processing. (Yet high level tasks like recognition and planning must also be done.) In SAIL, sacrifices of expressiveness and generality were made so that high efficiency could be maintained. In consequence of this orientation, SAIL, unlike the other five AI languages we are discussing, is used considerably outside AI as well as inside.

The most significant extension of SAIL over ALGOL is the incorporation of an associative-processing formalism known as LEAP, which we will now describe. LEAP is itself an extension of ALGOL 60, developed by Paul D. Rovner at MIT [ref Feldman and Rovner] as a language in which to do associative processing of the sort known to be useful in operating systems, computer graphics, computer aided design, data base management systems, and to be sure, AI. LEAP came into active use in early 1967, mostly in computer graphics applications.

The motivation for LEAP was to implement an efficient software scheme for associative processing, or the accessing of data by partial specification of its content rather than by the address of the storage location it happens to reside in.

LEAP was swallowed whole into the developing SAIL language and the problem of ill-fit between associations and ALGOL types persists. This problem will be explained in the Data Structure subsection of Section III.

Besides LEAP quite a few other features were put into SAIL. Among them are set, list and record data types, primitives for multiple processes, a large number of functions providing excellent access to hardware devices and operating system utilities, and an interactive debugger. The features are not well integrated with each other, partly for efficiency reasons, making the language something of a hodgepodge.

SAIL is implemented only on the PDP-10 under several operating systems. It is used at several installations in the U.S. though not a great many, still primarily for vision, speech and robotics work, plus much non-AI work.

## I. POP-2

POP-2 was designed by R. M. Burstall and Robin J. Popplestone, based on Popplestone's POP-1 language.

POP-2 arose out of the following situation. AI researchers at the University of Edinburgh and elsewhere in Britain were faced with the question of what language to use. There were basically two alternatives: use LISP, or create a new language. There were some new ideas, such as partial application of functions, and perceptions of things wrong with LISP, like its type structure. [or lack of same?] In addition, the researchers wanted a language suited to relatively small machines like the ICL-4000. The tendency in LISPs, then as now, was for the language (and core space needed to run it) to grow monotonically as various extensions and facilities are added on. The extreme case of this at present is INTERLISP. An alternative approach was adopted for POP-2, of designing a small, clean basic language of high flexibility which is used in conjunction with library packages that provide extensions. The user calls in only those packages whose features are needed for the particular program.

The language has been quite successful. It is currently the major AI language used in Britain and is used at installations in various other countries. POP-2 has been implemented on the ICL-4000, ICL-1900, ICL System 4, IBM 360 and the PDP-10. A new version POP-2.5 is scheduled to come out soon after the time of this writing.

The syntax of POP-2 is ALGOL-like, yet the underlying structure is more like the LISP family, with dynamic scoping, dynamic allocation and extensive use of the Lambda construct for treating functions as objects. Other features fall into neither category.

The basic POP-2 language is like basic LISP in having no features that are peculiar to AI applications. Each provides a flexible framework on which such features may easily be built, though in characteristically different manners. For LISP it tends to be done by superimposing another language through an interpreter written in LISP. In the case of POP-2 AI features have been put in library packages which when compiled and loaded define functions which extend the basic language in a certain direction, e.g. with data base retrieval functions. There are advantages and disadvantages to both techniques, of which some will come out in later discussion.

Two important facets of the basic language are:

1. Thorough treatment of functions as objects. In particular, the notion of evaluation of a function is generalized to partial evaluation. This and other POP-2 devices tend to blur the distinction between data and control in a useful way.
2. Extensible data types. It is also easy to change the internal representation used for a data type without performing major surgery on parts of the program that use it.

## References

- Ballard, D. H., Brown, C. M., & Feldman, J. A. An approach to knowledge-directed image analysis. *IJCAI* 5, 1977, 664-670.
- Bobrow, D. G. Requirements for advanced programming systems for list processing. *CACM*, 1972, 15(7), 618-627.
- Bobrow, D. G., & Wegbreit, B. A model and stack implementation of multiple environments. *CACM* 1973, 16(10), 691-602.
- Davies, D. J. **POPLER: A POP-2 PLANNER**, Research Memo MIP-R-89 (plus amendment), Edinburgh University, School of Artificial Intelligence, 1971.
- Davies, D., et al. **POPLER 1.5 Reference Manual**, University of Edinburgh, Edinburgh, Scotland, 1973.
- Derksen, J. A. **The QA4 Primer**, SRI Proj. 8721, Draft Memo 15, June 1972.
- Feldman, J. A. & Rovner, P. D. An ALGOL-Based Associative Language. *CACM*, 1969, 12(8), 439-449.
- Feldman, J. A., Low, J. R., Swinehart, D. C. & Taylor, R. H. **Recent Developments In SAIL**, STAN-CS-308, Stanford University, Computer Science Dept.; Artificial Intelligence Laboratory, AIM-176, 1972.
- Floyd, R. W. **Notes on Programming in SAIL**, Computer Science Dept., Stanford University, 1977.
- Foster, J. M. Programming language design for the representation of knowledge. In E. W. Elcock & D. Michie (Eds.), *Machine Intelligence 8*. New York: John Wiley & Sons, 1977. Pp. 209-222.
- Golomb, S. W., & Baumert, L. D. Backtrack Programming. *JACM*, 1965, 12(4), 516-524.
- Grief, I., & Hewitt, C. Actor Semantics for PLANNER-73. *Conference Record of the 2nd ACM Symposium on Principles of Programming Languages*, Palo Alto, CA, January 1975, pp. 67-77.
- Hewitt, C. **Description and theoretical analysis of PLANNER**, MIT-AIM-258, MIT, AI Dept., 1972.
- Hewitt, C. **PLANNER: A language for manipulating models and proving theorems in a robot**. *IJCAI* 1, 1969, 295-301.
- MacDermott, D., & Sussman, G. **CONNIVER Reference Manual**, MIT-AIM-259, MIT, AI Dept., 1972.
- McCarthy, J. History of LISP. *ACM Sigplan Notices*, 1978, 13(8), 217-223.

## AI Handbook

- McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., & Levin, M. I. **LISP 1.5 Programmer's Manual**. Cambridge, Mass.: The MIT Press, 1962.
- McDermott, D. V. **Very Large PLANNER-type Data Bases**, MIT-AIM-339, MIT, AI Dept., 1976.
- Moon, D. A. **MACLISP Reference Manual**. Project MAC, MIT, Cambridge, Mass., 1974.
- Newell, A. **Heuristic Programming: Ill-structured problems**. In J. S. Aronofsky (Ed.), **Progress in Operations Research** (vol. 3). New York: Wiley, 1969. Pp. 362-414.
- Popplestone, R. J. **The Design Philosophy of POP-2**. In D. Michie (Ed.) **Machine Intelligence 3**. Edinburgh: Edinburgh University Press, 1967. Pp. 393-402.
- Pratt, V. R. **The Competence/Performance Dichotomy in Programming**. 4th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Santa Monica, Calif., 1977, pp. 194-200.
- Prenner, C. J., Spitznagel, J. M., & Wegbreit, B. **An implementation of backtracking for programming languages**. SIGPLAN Notices, 1972, 7(11), 36-44.
- Reiser, J. (Ed.) **SAIL**, STAN-CS-76-674, Stanford University, Computer Science Dept.; Artificial Intelligence Laboratory, AIM-289, 1976.
- Reiser, J. F. **BAIL: A Debugger for SAIL**, STAN-CS-75-270, Stanford University, Computer Science Dept.; Artificial Intelligence Laboratory, AIM-270, 1975.
- Rivest, R. L. **Partial-Match Retrieval Algorithms**. SIAM Journal of Computing 5, 1976, pp. 19-50.
- Rulfson, J. F., Waldinger, R. J., & Derksen, J. A. **QA4: A procedural calculus for intuitive reasoning**, SRI AI Center Tech. Note 73, SRI International, Inc., November 1973.
- Rychener
- Sacerdoti, E. D. **A structure for plans and behavior**, SRI AI Center Tec. Note 109, SRI International, Inc., August 1975.
- Sacerdoti, E. D., et al. **QLISP: A language for the interactive development of complex systems**, SRI AI Center Tech. Note 120, SRI International, Inc., March 1976.
- Sandewall, E. **Ideas about Management of LISP Data Bases**. IJCAI 4, 1975, 585-592.
- Sandewall, E. **Programming in the interactive environment: the LISP experience**. ACM Computing Surveys, 1978, 10(1), 35-71.
- Sandewall, E. **Some observations on conceptual programming**. In E. W. Elcock & D. Michie (Eds.), **Machine Intelligence 8**. New York: John Wiley & Sons, 1977. Pp. 223-265.

## AI Programming Languages

- Smith, D. C., & Enea, H. J. Backtracking in MLISP2: An efficient backtracking method for LISP. *IJCAI* 3, 1973, 677-686.
- Smith, N. W. SAIL Tutorial, STAN-CS-76-576, Stanford University, Computer Science Dept.; Artificial Intelligence Laboratory, AIM-290, 1976.
- Sussman, G. J., & McDermott, D. V. From PLANNER to CONNIVER: A genetic approach. *AFIPS*, 1972, 1171-1180.
- Teitelman, W. A display oriented programmer's assistant. *IJCAI* 5, 1977, 905-916.
- Teitelman, W., et al. INTERLISP Reference Manual, Xerox PARC, Palo Alto, CA, October 1978.
- Tesler, L. G., Enea, H. J., & Smith, D. C. The LISP70 pattern matching system. *IJCAI* 3, 1973, 671-676.
- Waldinger, R. J., & Levitt, K. N. Reasoning about Programs, *Artificial Intelligence*, 1974, 5(4), 235-316.
- Waterman, D. A., & Hayes-Roth, F. (Eds.) *Pattern-Directed Inference Systems*. New York: Academic Press, 1978.
- Weissman, C. *LISP 1.5 Primer*. Belmont, Calif.: Dickenson Publishing Co., 1967.
- Wilber, M. B. *A QLISP Reference Manual*, SRI AI Center Tech. Note 118, SRI International, Inc., March 1976.

# Applications-oriented AI Research

Part 1

# Research on Applications of AI

## Table of Contents

A. Overview of Applications of Artificial Intelligence . . . . .	443
1. TEIRESIAS . . . . .	449
2. MACSYMA . . . . .	462
3. AM . . . . .	468
B. Miscellaneous Applications of Artificial Intelligence . . . . .	482
1. The SRI Computer-based Consultant . . . . .	482
2. PROSPECTOR: An Expert System for Mineral Exploration . . . . .	487
3. RITA . . . . .	495
4. Artificial Intelligence in Information Retrieval . . . . .	500
References . . . . .	508
Index . . . . .	514



## A. Overview of Applications of Artificial Intelligence

Within the past decade Artificial Intelligence (AI) techniques have been applied to the development of *expert systems*, computer systems intended to assist researchers solve complex problems in their scientific or medical speciality. These systems are most strongly characterized by their use of *domain knowledge* gleaned from experts that enables them to perform their problem-solving tasks.

While reading this chapter the reader should be aware that nearly all the systems described here were originally designed to be applications in the intended communities, and all but a few are in consistent use. Most of the systems are still being researched and developed. Thus, perhaps a more appropriate title for this chapter might be "Applications-Oriented Artificial Intelligence Research." The emphasis here is a description of the applications and research of AI techniques to real-world problems.

It has been observed that what distinguishes a layman or general researcher from a specialist in a scientific or technical domain is the vast amount of *empirical* knowledge about the domain that the expert has amassed during the course of his profession. This task-specific knowledge is, of course, based on any *conceptual* or theoretical knowledge that underlies problem solving in the domain. Any so-called *knowledge-based* system designed to assist users in the domain at this expert level requires both the empirical and the theoretical knowledge as well. Developing representational vehicles that are able to encode this partly public, partly private knowledge of the domain has occupied the AI researchers during the construction of all these systems.

Using representations of domain-specific knowledge, artificial intelligence research has yielded systems with significant problem-solving abilities, at times even going beyond the abilities of the human experts. In addition to developing adequate representations of this domain-specific knowledge, research has emphasized the development of various reasoning and explanation procedures that manipulate this knowledge. In particular, much emphasis has been placed on the development of methods of *inexact reasoning* since for many of these domains, notably medicine, the experts appraisal of the problem situation cannot always be completely certain.

The major domains of expertise that have been developed as applications systems include: the diagnosis and treatment of various diseases (see section *Medicine.Overview*), the design of computer assistants for both the analytic and synthetic aspects of organic chemistry (see section *Chemistry.Overview*), interactive tutoring systems in education (see section *Education.Overview*), and assistants for performing advanced mathematics (see article E2). A number of other notable applications have been developed including applications of AI to database information retrieval problems (see article F4) and a geological assistant (see article F2). There are a host of recent applications as well that do not have articles in this chapter, such as SACON, a system for advising structural engineers in the use of a large finite-element analysis program used to model various mechanical structures (Bennett et al., 1978); PUFF, a system for diagnosing a patient with various pulmonary dysfunctions (Feigenbaum, 1977); and HEADMED, a system for diagnosis and treatment of psychiatric patients (Heiser, 1977, 1978).

Typically, these systems will be considered intelligent if they meet the following criteria: The system gives correct answers or useful advice, and the concepts and reasoning processes that the system uses to solve the problem resemble those that the user might employ. These last concerns have motivated the design of systems capable of explaining their reasoning about a case, capable of maintaining a focused dialogue with a user when pursuing relevant facts and inferences about the user's case, and capable of using knowledge at the conceptual level of the user when solving and explaining both the problem and the system's solution. Achieving these primarily *human-engineering* concerns has required many advances in artificial intelligence. These abilities and developments are detailed for each system in the following articles.

### Evolution of Expert Systems

Work in AI during the 1960s identified and explored general-purpose problem-solving techniques that would be applicable in a large number of problem-solving situations. This research introduced and refined the concept of heuristic search (see chapter Search, Search) as a mechanism of problem solving. These ideas and developments were embodied in such systems as GPS, REF-ARF, QA4, PLANNER, etc. These systems dealt with problems in domains such as chess, robot planning, and blocks world manipulations, as well as the classic problem-solving situations found in puzzles such as the Tower of Hanoi and The Missionaries and Cannibals.

During the mid 1960s, the first expert systems were developed, and these included DENDRAL and MACSYMA.

During 1965, the Heuristic Programming Project at Stanford University began to apply these search techniques to the design of an intelligent assistant to aid chemists in elucidating the structure of unknown chemical compounds. Motivated by interest in modeling the thought process of research scientists, Edward Feigenbaum and Joshua Lederberg of the DENDRAL project began to emphasize and use large amounts of domain-specific knowledge in the solution of this major *real-world problem*.

These systems were designed to aid researchers by being able to manipulate and explore large symbolically expressed problems that were known to be difficult for human researchers to solve. These problems were characterized by the fact that as their specification grew in complexity, so did the number of solution possibilities that had to be examined. The larger the size of the problem specification (e.g., size of the molecule in atoms/bonds or complexity of the expression to be integrated), the more difficult it was for human researchers to discover solutions or be confident that all valid solutions had been found. This *combinatorial explosion* in the solution search space easily outstripped the abilities of most human researchers. The ability of these applications systems to deal with these larger solution spaces extended the limit on the types of problems capable of being solved with the present conceptual tools.

More recently, the motivation for constructing these knowledge-based systems has come to include a number of other factors. These expert systems promise to have significant economic (SYNTHESIS, PROSPECTOR) and social impact. For example, the organic synthesis systems are used actively by drug and chemical manufacturing companies to uncover

inexpensive methods of synthesizing various compounds. In medicine, these systems have the capability to examine all possible diseases that might be afflicting a patient. In addition, the ability to codify the expertise in a domain makes these systems potentially available for tutoring and assessment purposes.

For a system to achieve broad applicability within a speciality and remain complete and correct in its search for problem solutions, large amounts of domain-specific knowledge have had to be represented and handled. Thus, while heuristic search management is still a major concern in the construction of any expert system, the large amounts of expert knowledge required to achieve an adequate, efficient solution to these problems have fostered problems in the construction and maintenance of these knowledge bases. The concerns of effective representation and management of the large, domain-specific knowledge bases have shifted attention away from development of programs designed to solve large combinatorial problems, such as those that prompted the DENDRAL programs, to those that require more empirical knowledge for their solution. Current research emphasizes not only the representational adequacy of the existing formalisms but also such issues as the appropriate *grain size* of the knowledge (see article Representation.Issues) and improved explanation, inference, and acquisition abilities (see article C6).

### Dimensions of Applications

Most of the application systems described in this chapter can be viewed as *consultants* that formulate opinions or as models about *cases* that give advice to their users. The tasks these consultants are designed to perform are typically repetitive and sometimes beyond human abilities--problems that require knowledge of facts and relationships known only by specialists.

The notion of a consultant implies that the user and the system interact during the problem-solving task. The current systems emphasize the cognitive abilities that support this interaction such as the ability to explain lines of reasoning and to interactively acquire new domain knowledge. This is especially true for the medical and educational systems where much research has gone into the design of well-engineered, responsive, user interfaces.

The AI research conducted for these application systems is different from other mainstream AI research such as that on speech or vision. Applications research does not concentrate on developing models of the various physiological functions that are of interest in these other areas. The cognitive abilities required by the current applications are primarily conceptual in nature and do not depend on sophisticated perceptual capabilities in order to be performed. Research concentrates instead on the requirements for systems to utilize *developed* human expertise. This expertise is typically at a high conceptual level and is easily encodable in the symbolic representational formalisms that have been developed.

**Representational adequacy.** Applications research has proved a valuable testing ground for the techniques developed in other areas of AI research. In addition to the augmentation of heuristic search methods by domain-specific knowledge, representation formalisms developed for modeling psychological aspects of cognition--such as semantic nets (see article Representation.B2) and production systems (see article Representation.B3)--have been used ubiquitously in the applications described in this chapter. Techniques developed in the course of natural language research (see chapter Natural Language, Natural

Language) have been used to achieve the effective man-machine interface required of these interactive consultant systems.

**Domain-independence of the systems.** As part of the research on the adequacy of these representational formalisms, a number of these systems have attempted to maintain a strict separation between the domain-specific knowledge supplied by the expert and the domain-independent knowledge and capabilities of problem solving that the systems intrinsically possess. The task of determining what abilities and what knowledge constitutes an effective domain-independent system occupies much of the AI research in applications. For example, the EMYCIN system consists of the basic control structure found in the MYCIN system (see article C1) with the infectious disease knowledge base removed; this "empty MYCIN" system retains the capability to interact with the user during a case, to explain its reasoning, and to answer questions about a case in a new domain of expertise. This system has been used successfully to develop the applications in pulmonary dysfunction, structural analysis, and psychiatric diagnosis mentioned in the beginning of this overview. Numerous other systems similar to the EMYCIN system are being developed, such as the IRIS system (see article C7); these domain-independent consultation systems are a major product of this applications research.

**Explanation and the opacity of knowledge.** As mentioned previously, a major design issue for some of these systems, for the consultants in particular, is whether the system needs to explain its reasoning to a user. This capability is implemented primarily in the effort to convince users that the system's reasoning is appropriate and that its conclusions about a case are reasonable. In some cases, however, the problem-solving expertise used by the system is in a form that is not at all similar to the expertise that an expert user would use to obtain the solution. For example, in the case of the DENDRAL programs, the generator of chemical structure solutions uses a procedure for exhaustively producing solutions based on various graph theoretic notions that the average organic chemist using the system is unlikely to know or care about. Thus a major portion of the DENDRAL expertise resides in a procedure that is conceptually *opaque* to the normal user. The generator was developed because it was discovered that the method the chemist uses to generate solutions is incomplete and the method the DENDRAL program uses has been mathematically proven complete. A similar situation exists in the MACSYMA system, which uses the Risch algorithm for evaluating various types of integrals. While mathematically correct, it is rarely employed by human mathematicians because of its complexity. The correctness and continued success of the programs serve as their primary form of explanation: The user community is thus convinced that the performing system is both acceptable and useable.

In contrast, systems such as MYCIN and PROSPECTOR have been designed to represent and explain the reasoning process used by the system in a manner that is directly understandable by the knowledgeable user. These systems required a representational formalism capable of supporting the reasoning and explanation abilities that would closely approximate the conceptual structure of expert and user. Since most of these scientific and technical domains have a well-defined set of concepts that its practitioners use consistently, the systems designers have capitalized on this consistency and have designed the programs to accept and reason with knowledge using these concepts.

Assuming a system uses domain-specific knowledge that is in an explainable form and that the system is capable of explaining its reasoning, the system designer faces another

issue: Should the system reason and apply the expertise in a manner that resembles the methods employed by the human expert? In MYCIN, for example, no claim is made by the designers that the simple back-chaining reasoning methodology has any strong resemblance to the methods actually employed by human physicians performing infectious disease diagnosis. Although the medical concepts employed by the system are familiar to most physicians, the method of inferring the infections and causal organisms, while understandable by a physician, bears little resemblance to a doctor's normal diagnostic reasoning. By contrast, the PIP and INTERNIST systems emphasize the similarities of their diagnostic procedure to that used by physicians.

**Knowledge acquisition.** During the development of the knowledge base, the expert is unlikely to present all of the relevant facts and relationships that are required for expert performance in the domain. Being human, experts tend to forget or simplify details about their knowledge, requiring the system to be able to augment its knowledge at a later time. Since the knowledge imparted to the system is largely empirical and the domains are themselves rapidly developing, it is necessary that the system be able to perform these changes easily and in an *incremental* or modular fashion. Thus, most of the recent applications systems have emphasized the use of representation vehicles that allow for the incremental construction of the knowledge base.

Many researchers use *production rules* to perform this incremental construction. Each rule and rule set represents a "chunk" of domain expertise that is communicable to the user and that can be added or extracted with relative ease. Thus the performance of the system can be improved by modifying the knowledge base with new rule sets that deal with new domains or subdomains. Furthermore, the production rule formalism can directly accommodate the concepts of the domain expert and thus is more easily communicable to the user and expert.

### The Future

A primary research activity in the near future will be the development of facilities for acquiring the domain concepts and the empirical knowledge that these systems require. At present this is a painful process involving many individuals including both domain experts and computer scientists who together construct the knowledge base. The design of more efficient interfaces for acquiring this domain-specific knowledge, along the lines of the TEIRESIAS system (see article C6) and the methods used by the Meta-DENDRAL system (see article B3), need to be developed before significantly larger expert systems can be constructed.

While the domains and methods that have been developed are interesting and challenging in their own right, they, in fact, represent only a small fraction of the total cognitive or even conceptually cognitive abilities that a human possesses. These abilities are for the most part as yet undefined in current cognitive research; if they were, they would be the subjects of further AI research.

The size of current systems is typically given in terms of some convenient measurement of the domain-specific knowledge the system contains. For example, the MYCIN system contains approximately 450 *rules* and a similar number of clinical parameters that it

uses to diagnose and prescribe treatments for patients with bacteremia, cystitis, and meningitis. The SYNCHEM system contains approximately 390 transforms that it uses to construct plausible organic synthesis routes. The order of magnitude of expert knowledge has been primarily a function of expert involvement and effort. These systems can potentially support larger knowledge bases but there has been no effort yet to construct these larger, more comprehensive systems. At present, only selected subdomains are actually represented and used.

While the design of current knowledge-based systems appears adequate for a certain amount of this scaling of the knowledge base, without large degradation in efficiency or ease of use, it is clear that AI and computer science will have to develop new techniques for handling the truly large-scale knowledge bases that will exist in the future. Steps in this direction have been taken with the development by Davis (1976a) on representing knowledge about knowledge, or *meta-level knowledge* about the domain. This domain-specific knowledge is used to determine the consistency and appropriateness of various knowledge sources developed and used by the system. The use of meta-knowledge is one of the ways knowledge can be organized both dynamically and statically so that it is comprehensible not only to the machine but also to the human user and expert.

#### **An Application Article**

An article on the individual applications systems in this chapter will attempt to cover the following topics:

A description of the problem domain (e.g., chemistry, infectious disease, etc.), the particular problem task the application system was designed to perform (e.g., elucidate chemical structures, diagnose and treat a patient with an infectious disease, etc.), and the major system design motivations, both for AI and for the task domain.

A description of the task-specific knowledge that had to be represented to perform the problem-solving task (e.g., knowledge about probable bond breaks for a compound in a mass-spectrometer, knowledge about possible infections and their causal organisms, etc.).

A description of the particular AI methods that were used to represent this knowledge and a description of how the represented knowledge is used to reason about a particular case. This description sometimes includes an annotated sample interaction between a user and the system.

An indication of the current level of expertise of these systems and an indication of their present status and possible future development.

Throughout these articles, emphasis has been placed on illuminating the major issues and contributions to Artificial Intelligence made by the design of these systems.

## A1. TEIRESIAS

TEIRESIAS is a system for facilitating automatic acquisition and maintenance of the large knowledge bases used by expert systems. Although TEIRESIAS is not itself an application of AI to some domain, it deals with many important issues in expert systems design that are relevant to all of the programs described in this chapter. The system was developed by Randall Davis as part of his doctoral research at the MYCIN project at Stanford, and this article assumes some familiarity with MYCIN's rule-based knowledge representation scheme and its *backward-chaining* control structure (see Article C1). However, the ideas and techniques TEIRESIAS uses are not necessarily limited to MYCIN's domain of infectious diseases or to the production-rule formalism used by MYCIN.

### Knowledge-based Programs

As discussed in the Applications Overview, systems that achieve expert-level performance in problem-solving tasks derive their power from a large store of task-specific knowledge. As a result, the creation and management of large knowledge bases and the development of techniques for the informed use of knowledge are now central problems of AI research. TEIRESIAS was written to explore some of the issues involved in solving these problems.

Most expert programs embody the knowledge of one or more experts in a field, like infectious diseases, and are constructed in consultation with these experts. Typically, the computer scientist *mediates* between the experts and the program he is building to model their expertise. This is a difficult and time-consuming task, because the computer scientist must learn the basics of the field in order to ask good questions about what the program is supposed to do.

TEIRESIAS's goal is to reduce the role of the human intermediary in this task of *knowledge acquisition*, by assisting in the construction and modification of the system's database. The human expert communicates, via TEIRESIAS, with the *performance program* (e.g., MYCIN), so that he can discover, with TEIRESIAS's help, what the performance program is doing and why. TEIRESIAS offers facilities for modifying or adding to the knowledge base to correct errors: Using TEIRESIAS, the human expert can "educate" the program just as he would tutor a human novice who makes mistakes. Ideas about how this "debugging" process is best carried out are at the core of TEIRESIAS's success.

TEIRESIAS also recognizes the inexact, experiential character of the knowledge that is often required for knowledge-based systems and (as examples below will illustrate) offers the expert some assistance in formulating new "chunks of knowledge" of this sort. Another major aim of the system was to provide a mechanism for embodying strategic information. *Meta-rules* (discussed below) are used to direct the use of object-level rules in the knowledge base and to provide a mechanism for encoding problem-solving strategies.

### Interactive Transfer of Expertise.

It is an established result that an expert knows more about a field than he is aware, or capable of articulating completely. Thus, asking a broad question like "Tell me everything

you know about staph-infections" will yield only a fraction of his knowledge. TEIRESIAS's approach is to present the expert with some errors made by an already established, but still incomplete, knowledge-based program and to ask a *focused* question: "What do you know that the program doesn't know which makes your expert diagnosis different in this case?"

This interaction is called *transfer of expertise*: TEIRESIAS incorporates into the performance program the capabilities of the human expert. TEIRESIAS does not attempt to derive new information on its own, but instead tries to "listen" as attentively and intelligently as possible, to help the expert augment or modify the knowledge base.

Interactive transfer of expertise, between an expert and an expert program, begins when the expert identifies an error in the performance of the program and invokes TEIRESIAS to help track down and correct the error. Errors are manifest as program responses that the expert would not have made or as "lines of reasoning" that the expert finds odd, superfluous, or otherwise inappropriate. The first kind of error might be, for example, a wrong conclusion about the identity of a bacteria. On the other hand, the performance program may just ask the expert, during a consultation, a question that, in the expert's opinion, does nothing to resolve the identity of the bacteria. This is an example of the "line of reasoning" type of error.

Both kinds of error are assumed, by TEIRESIAS, to be indicative of a deficit, or "bug," in the performance program's knowledge base. Transfer of expertise begins when TEIRESIAS is called upon to correct the deficit. TEIRESIAS fixes bugs in the knowledge base by:

1. Stopping the performance program when the human expert identifies an error.
2. Working backwards through the steps in the performance program that led to the error, until the bug is found.
3. Helping the expert fix the bug by adding or modifying knowledge.

To identify faulty reasoning steps in the performance program, the expert can use the WHY and HOW commands to ask TEIRESIAS to back up through previous steps, *explaining* why they were taken. The same explanatory abilities can also be used when there is no bug, to help the user follow the system's line of reasoning. Since many large performance programs carry out very complex inferences that are essentially "hidden" from the person using the program, this is a valuable facility.

### Meta-level Knowledge

One of the principal problems of AI is the question of appropriate representation and use of knowledge about the world (see Representation). Numerous techniques have been used to represent domain knowledge in various applications programs. A central theme of the research on TEIRESIAS is exploring the use of *meta-knowledge*. Meta-level knowledge is simply the representation in the program of knowledge about the program itself--about how much it knows and how it reasons. This knowledge is represented using the same representation techniques used to represent the domain knowledge, yielding a program containing *object-level* representations describing the external world and *meta-level* representations that describe the internal world of the program, its self-knowledge. For



example, many AI programs use the notion of a *frame* to represent the knowledge used by the system (see Article Representation.B7). One can imagine a meta-level frame that describes the structure of all frames in the system, or one that denotes the different classes of frames used in the system. One of TEIRESIAS's representations is very close to this notion, the *schema* described below.

Meta-level knowledge has taken several different forms as its uses have been explored, but it can be summed up as "knowing about what you know." In general, it allows the system to both use its knowledge directly and to examine it, abstract it, and direct its application. The capabilities for explanation, knowledge acquisition, and strategic reasoning in TEIRESIAS inspired the incorporation of explicit meta-level knowledge, and these capabilities are based on the use of that knowledge.

### Explanation

There are two important classes of situations where expert systems should be able to explain their behaviour and results. For the user of the system who needs clarification or reassurance about the system's output, the explanation can contribute to the *transparency* and thus the *acceptance* of the system. The second major need for explanation is in the debugging process described above, where a human expert uses the system's explanations of why it has done what it has done in order to locate some error in the database. The first of these applications of explanation has been explored in the question-answering facility of the MYCIN system; the explanation capability in TEIRESIAS has explored both uses but has concentrated on the latter.

The techniques used in TEIRESIAS for generating explanations are based on two assumptions about the performance program being examined, namely a) that a recapitulation of program actions can be an effective explanation, as long as the correct level of detail is chosen, and 2) that there is some shared framework for viewing the program's actions that will make them comprehensible to the user. In the MYCIN-like expert systems that use production-rule knowledge bases, these assumptions are valid, but it is easy to imagine expert systems where one or both are violated. For example, the first assumption simplifies the explanation task considerably, since it means that the solution requires only the ability to record and play back a history of events. This assumption rules out, in particular, any need to simplify those events. However, it is not obvious, for instance, that an appropriate level of detail can always be found. Furthermore, it is not obvious how this approach of recapitulation, which often offers an easily understood explanation in programs that reason symbolically, would be applied to expert systems that perform primarily numeric computations.

A simple recapitulation will be an effective explanation only if the level of descriptive detail is constrained. It must be *detailed* enough that the operations the system cites are comprehensible; the conceptual level must be *high* enough that the operations are meaningful to the observer, so that unnecessary detail is suppressed; and it must be *complete* enough so that the operations cited are sufficient to account for all behavior.

The second assumption concerns the user's comprehension of the expert system's activity, which depends on the fundamental mechanism used by the program and the level at which it is examined. Consider a program that does medical diagnosis using a statistical approach based on Bayes's Theorem. It is difficult to imagine what explanation of its actions

the program could give if it were queried about computed probabilities. No matter what level of detail is chosen, such a program's actions are not (nor were they intended to be) a model of the reasoning process typically employed by physicians. Although they may be an effective way for the computer to solve the diagnosis problems, there is no easy way to interpret these actions in terms that will make them comprehensible to humans unacquainted with the program.

Thus, the lack of mechanisms for simplifying or reinterpreting computation means that TEIRESIAS's approach is basically a first-order solution to the general problem of explanation. But, in the context of a MYCIN-like expert system, for which TEIRESIAS was designed, the simple AND/OR goal tree control structure offers a basis for explanations that typically needs little additional clarification. (The operation of TEIRESIAS's explanation facility is illustrated in the sample protocol at the end of this article.) The invocation of a rule is taken as the fundamental action of the system. This action, within the framework of the goal tree, accounts for enough of the system's operation to make a recapitulation of such actions an acceptable explanation. In terms of the constraints noted earlier, it is sufficiently detailed--the actions performed by a rule in making a conclusion, for instance, correspond closely enough to the normal connotation of that word--that no more detailed explanation is necessary. The explanation is still at a high enough conceptual level that the operations are meaningful and the explanation is complete enough--there are no other mechanisms or sources of information that the observer needs to know in order to understand how the program reached its conclusions.

#### Knowledge-acquisition: Rule Models and Schemata

When the expert has identified a deficit in the knowledge base of the performance program, TEIRESIAS questions him in order to correct the deficit. This process relies heavily on meta-level knowledge about the performance program, encoded in *rule-models* and *schemata*. In other words, TEIRESIAS knows about what the performance program knows.

The meta-level knowledge about *objects* in the domain includes both structural and organizational information and is specified in *data structure schemata*. Acquisition of knowledge about new objects proceeds as a process of instantiating a schema--creating the required structural components to build the new data structure and then attending to its interrelations with other data structures. By making inquiries in a simple form of English about the values of the schema's components, this knowledge acquisition process is made to appear to the expert as a natural, high-level inquiry about the new concept. The process is, of course, more complex, but the key component in the system's description of its own representation.

TEIRESIAS's *rule models* are empirical generalizations of subsets of rules, indicating commonalities among the rules in that subset. For example, in MYCIN there is a rule model for the subset of rules that conclude affirmatively about *organism category*, indicating that most such rules mention the concepts of *culture site* and *infection type* in their premise. Another rule model notes that those rules that mention *site* and *infection type* in the premise also tend to mention the *portal of entry* of the organism.

This knowledge about the contents of the domain rules is used by TEIRESIAS to build *expectations* about the dialogue. These expectations are used to facilitate the process of translating the English statements into the performance program's internal representation

and to identify information missing from the expert's entry. An example of TEIRESIAS's use of rule models in its knowledge acquisition dialogue is given in the sample protocol below.

### Meta-rules and Performance Strategies

In performance programs with sufficiently small knowledge bases (like MYCIN's), exhaustive invocation of the relevant parts of the knowledge base during a consultation is still computationally feasible. In time, however, with the inevitable construction of larger knowledge bases, exhaustive invocation will prove too slow. In anticipation of this eventuality, *meta-rules* are implemented in TEIRESIAS as a means of encoding strategies that can direct the program's actions more selectively than can exhaustive invocation. The following meta-rule is from MYCIN's infectious disease domain:

#### METARULE 001

- If
- 1) the infection is a pelvic-abscess, and
  - 2) there are rules which mention in their premise enterobacteriaceae, and
  - 3) there are rules which mention in their premise gram positive rods,
- Then There is suggestive evidence (.4) that the rules dealing with enterobacteriaceae should be evoked before those dealing with gram positive rods.

This rule suggests that since enterobacteriaceae are commonly associated with a pelvic abscess, it is a good idea to try rules about them first, before the less likely rules mentioning gram positive rods. Note that this meta-rule does not refer to specific object-level rules. Instead it specifies certain attributes of the rules it refers to, for example, that they mention in their premise enterobacteriaceae.

### An Example: TEIRESIAS in the Context of MYCIN

We will now illustrate TEIRESIAS's operation in affiliation with the MYCIN system (see Article C1), paying particular attention to TEIRESIAS's explanation and knowledge acquisition facilities. MYCIN provides the physician with advice about the diagnosis and drug therapy for bacterial infections. The system asks questions about the patient, the infection, the cultures grown from specimens from the patient, and any organisms (bacterium) growing in the culture. (Typically, of course, the exact identity of the organism is not yet known.)

MYCIN's database is composed of rules that specify a situation (involving information about the patient, culture, and organism) and the conclusions that can be drawn in that situation. For example, to conclude whether a patient suffers from a bacteria in the enterobacteriaceae category, MYCIN invokes Rule 95 of its knowledge base:

## Rule 095

IF the site of the culture is blood, and  
 the gram stain is positive, and  
 the portal of entry is gastrointestinal tract, and  
 [A-the abdomen is the locus of infection, or  
 B-the pelvis is the locus of infection]  
 THEN there is strongly suggestive evidence that  
 enterobacteriaceae is the class of organisms for  
 which therapy should cover.

MYCIN asks questions of the physician to determine which rules are applicable in the current case--i.e., to determine the "situation." To determine if the organism is of type enterobacteriaceae, MYCIN would inquire whether each clause in Rule 95 were true.

The procedure of working backwards from the conclusion to the situation that would make the conclusion true is called *backward chaining* and is MYCIN's fundamental control structure. Backward chaining can go "several levels deep," meaning that the clauses in a situation that must be true (for the conclusion to be true) might themselves be conclusions of other rules. For example, a condition on the identification of enterobacteriaceae is that the infection be localized in the abdomen or the pelvis; but this condition is itself a conclusion, and it is only true if its conditions (that the culture from these places is recent, and is therapeutically significant) are true. Furthermore, the condition of therapeutic significance has its own conditions, and so on. Here is a protocol showing MYCIN's basic style of interaction, the human expert's responses are prefixed with \*\*:

## -----Patient 281-----

- 1) Patient's name:  
\*\* Albert Hall
- 2) Sex:  
\*\* Male
- 3) Age:  
\*\* 34
- 4) Have you been able to obtain positive microbiological information about a possible infection of Albert Hall?  
\*\* Yes

## -----Infection 1-----

- 5) What is the infection?  
\*\* Primary-Bacteremia
- 6) Please state when signs and symptoms of infection-1 first appeared:  
\*\* 15-Dec-75 12:00

## -----Culture 1-----

- 7) From what site was the specimen for culture-1 taken  
\*\* Blood
- 8) Please give the date and time when this blood culture was obtained:  
\*\* 16-Dec-75 12:00

The first significant organism from this blood culture will be referred to as:

-----Organism-1-----

9) Enter the identity of ORGANISM-1

\*\* Unknown

10) Is ORGANISM-1 a rod, or coccus, (etc.):

\*\* Rod

11) The gram-stain of ORGANISM-1:

\*\* Gram negative

12) Did ORGANISM-1 grow in the aerobic bottle?

\*\* Yes

13) Did ORGANISM-1 grow in the anaerobic bottle?

\*\* Yes

:

:

:

My therapy recommendations will be based on the following probable infection(s) and potential causative organism(s):

INFECTION-1 is PRIMARY-BACTEREMIA

<Item-1> E.COLI [ORGANISM-1]

<Item-2> PSEUDOMONAS-AERUGINOSA [ORGANISM-1]

FIGURE 1. PART OF A DIALOGUE WITH MYCIN.

In sum, MYCIN constructs a *goal tree* (see Article Search.Problem Reduction) of questions that must be resolved in the course of the consultation to conclude the identity of a bacteria. When it cannot resolve a question by inference from what it knows already, it asks the respondent to provide an answer. Each node of the goal tree has subnodes that result from the application of a rule:

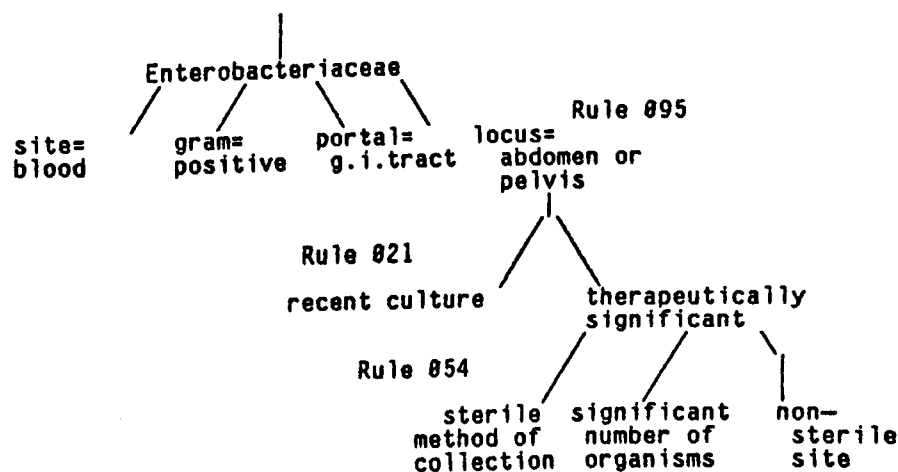


Figure 1. A MYCIN Goal Tree

**Explanation: HOW and WHY.** Suppose that during a consultation with MYCIN the clinician wants an explanation of the reasoning behind some question MYCIN has asked him. The explanation given by TEIRESIAS in response to the physician's question, WHY?, involves backing up the goal tree to the previous node and presenting the rule used to derive the subnode from its ancestor:

MYCIN: What is the suspected portal of entry of ORGANISM-1 into this sterile site?

Doctor: WHY

TEIRESIAS: [i.e. why is it important to determine the suspected portal of entry of ORGANISM-1 into this sterile site?]

[3.0]...in order to find out about an organism or class of organisms for which therapy should cover.

It has already been established that

[3.1] the site of the culture is blood

[3.2] the stain of the organism is gram positive

Therefore if:

[3.3] the portal of entry of the organism is g.i.

[3.4] [Abdomen is the locus of infection, or  
Pelvis is the locus of infection]

Then

There is strongly suggestive evidence (.9) that enterobacteriaceae is the class of organisms for which therapy should cover.

[Rule 096]

In contrast to WHY, which spells out "higher" nodes in the goal tree, the HOW command asks TEIRESIAS to explain the nodes that have been or will be expanded lower down the tree. For example, in the following exchange, the expert asks how MYCIN will determine whether the abdomen or pelvis is the locus of infection:

Doctor: HOW[3.4]      *note: 3.4 is the prefix of the statement of interest in the exchange above*

TEIRESIAS: [i.e. how were you determining that the Abdomen is the locus of infection, or that the Pelvis is the locus of infection?]

[4.0] At that point Rule 021 was being used.

If the doctor now wants to see Rule 021, he asks TEIRESIAS to explain HOW it was used, referring to the prefix (4.0) of the statement in which it was mentioned:

Doctor: HOW [4.0]

TEIRESIAS: [i.e. how was Rule 021 used?]

[5.0] It has already been established that

[5.1] the culture is recent.

Therefore if:

[5.2] there is therapeutically significant disease  
associated with the occurrence of this organism

then

It is definite (1.0) that the site of the culture is  
the locus of infection in the patient.

The explanation facilities of TEIRESIAS involve climbing up and down MYCIN's goal tree explaining WHY questions are asked and HOW they are answered. If MYCIN makes a misdiagnosis, these same facilities are used (either by the expert or TEIRESIAS, at the choice of the expert) to back up through the rules that achieved the diagnosis until the expert has found the faulty one or determined that one is missing. TEIRESIAS is then used to expedite the alteration or construction of rules by the expert.

**Knowledge acquisition.** TEIRESIAS is able to help an expert formulate rules, in English, because it has expectations about what rules "look like." The expectations are stored internally in the form of *rule models* that are formed automatically by examining all the rules in the knowledge base. TEIRESIAS knows, for example, that there is a class of rules that have as their conclusions the probable identity of a bacteria and that these rules typically have as conditions (a) the site of the culture, (b) the stain of the culture, and (c) the probable portal of entry of the bacteria. These rule models help TEIRESIAS translate English versions of new rules into a machine representation acceptable to MYCIN.

To illustrate TEIRESIAS's debugging and correction facilities, we will continue with the consultation from Figure 1. Suppose that after MYCIN presents the expert with its diagnosis, TEIRESIAS steps in and checks whether the expert agrees:

MYCIN: My therapy recommendation will be based on the following probable infection(s) and potential causative organism(s):

INFECTION-1 is PRIMARY BACTEREMIA  
<Item-1> E.COLI [ORGANISM-1]  
<Item-2> PSEUDOMONAS-AERUGINOSA [ORGANISM-1]

*Now TEIRESIAS steps in.*

Are all these identities correct?

*The expert, marked with a \*\*, says No.*

\*\* No.

Are there any identities missing?

\*\* Yes.

Which ones?

\*\* KLEBSIELLA-PNEUMONIAE, PROTEUS-NON-MIRABILIS

Are there any identities which shouldn't have appeared?

\*\* yes

Which on(s)?

\*\* PSUEDOMONAS-AERUGINOSA

Do you want to

1] track down the problem yourself, or

2] allow me to guide the debugging, or

3] ignore it for now and fix it another time.

Type 1,2, or 3.

\*\* 2

*If the expert had typed 1, he would have used the HOW and WHY commands to debug the problem, as discussed above.*

We'll start with PSEUDOMONAS-AERUGINOSA first:

*TEIRESIAS starts debugging by trying to find out why an inappropriate diagnosis occurred. Frequently solving a problem will make others "go away" as well.*

The following rules were used in deducing that the identity of ORGANISM-1 is PSEUDOMONAS-AERUGINOSA:

Rule 184

Since [1.1] the category of ORGANISM-1 is not known

[1.2] the gram stain of ORGANISM-1 is gram negative

[1.3] the morphology of ORGANISM-1 is rod

[1.4] the aerobicity of ORGANISM-1 is facultative

There is weakly suggestive evidence (.3) that the identity of ORGANISM-1 is PSEUDOMONAS-AERUGINOSA.

Is this rule correct

Note that this rule, which resulted in the wrong conclusion, is not necessarily an incorrect rule. It might be (and in fact is) the case that it was incorrectly invoked because one or more of its conditions are true when they should be false.

\*\* Yes



*The expert says the rule is correct.*

Should its premise have failed for this case?

\*\* Yes

Which clause of the premise should have failed?

\*\* Clause 1.

The expert believes that MYCIN should have known the category of ORGANISM-1. MYCIN failed to establish the category, causing Rule 184 to be used.

Should it have been possible to conclude a category?

\*\* Yes

Which one?

\*\* Enterobacteriaceae

*Our old friend. But recall that MYCIN's rule 095 about enterobacteriaceae requires a positive gram stain, but it can be seen from Fig. 1 that it has a negative gram stain. Thus, Rule 95 was never evoked. In the course of more debugging, it transpires that MYCIN has a couple other rules dealing with this category of bacteria, but they weren't evoked either. The expert decides that another rule is required, and we pick up the dialogue where he is proposing one to TEIRESIAS:*

If: 1 \*\* THE PATIENT'S INFECTION IS PRIMARY-BACTEREMIA

2 \*\* THE SITE IS ONE OF THE STERILE SITES

Then: \*\* THERE IS EVIDENCE (.8) THAT THE CATEGORY IS ENTEROBACTERIACEAE

*TEIRESIAS parses this rule into its internal representation and then translates it back into English, so the expert can check it. We pick up the dialogue after the expert has agreed on the parse. TEIRESIAS is applying its rule model.*

I hate to criticize, Dr. Davis, but did you know that most rules about what the category of an organism might be, that mention-  
the site of a culture

the infection  
ALSO mention-  
[A]- the portal of entry of the organism  
Shall I try to write a clause to account for [A]?  
\*\* Yes.

*Here we see one of the utilities of the rule model. TEIRESIAS prompts the expert to add a bit of knowledge that is typically found in rules like these and offers to fix the rule itself. This completion is accomplished by looking at other rules that fit the same rule model, to find the most likely portal of entry clause.*

how about -  
[A] The portal of entry is gastrointestinal. Ok?  
\*\* Yes.

*TEIRESIAS now does some finishing up: checking the complete rule with the expert for final approval and asking the expert to write a brief description (for bookkeeping purposes) of why the rule was needed. Finally, it reruns the consultation internally, using the responses from Fig. 1, which it has stored. It turns out that adding the rule above did, in fact, cure the other problems with the first consultation, and this time the diagnosis is satisfactory to the expert.*

#### Summary: TEIRESIAS and Expert Systems

TEIRESIAS aids a human expert in monitoring the performance of a knowledge-based system. When the human expert spots an error in the program's performance, either in the program's conclusions or its "line of reasoning," TEIRESIAS assists in finding the source of the error in the database by *explaining* the program's conclusions--retracing the reasoning steps until the faulty (or missing) rule is identified. At this point, TEIRESIAS assists in *knowledge acquisition*, modifying faulty rules or adding new rules to the database. *Meta-level knowledge* about the kinds of rules and concepts in the database is used to build expectations in TEIRESIAS's *model-based understanding* process. Meta-level knowledge is also used to encode problem-solving *strategies*, in particular, to order the invocation of rules so that those that are most likely to be useful (given the current knowledge of the program) are tried first.

**References**

The principal reference on TEIRESIAS is the doctoral dissertation by Davis (1976a). Uses of meta-knowledge in expert systems are discussed in Davis & Buchanan (1977). Also see Davis (1977) and Davis & Buchanan (1978).

## A2. MACSYMA

MACSYMA is a large, interactive computer system designed to assist mathematicians, scientists, and engineers in solving mathematical problems. It has a wide range of algebraic manipulation capabilities, all working on symbolic inputs and yielding symbolic results, as well as an extensive numerical subroutine library (IMSL) and plotting package.

MACSYMA is used extensively by hundreds of researchers from government laboratories, universities, and private companies throughout the United States. Many of these users spend a substantial portion of every day logged in. Currently, the system runs exclusively on a Digital Equipment Corporation KL-10 at M.I.T. and is accessed through the ARPA Network; however, there are plans to distribute it to other sites in the near future. MACSYMA's funding is supplied almost exclusively by its user community.

The original design for MACSYMA was laid out by Carl Engleman, Bill Martin, and Joel Moses in 1968. They built on their previous experience with the Mathlab '68 system and the theses of Martin and Moses. Martin had constructed an algebraic manipulation system to solve certain problems in applied mathematics. Moses had produced a program that was able to do indefinite integration about as well as a typical graduate student. The system had its first users in 1971 and has undergone continuous development since then, a total of about 45 man-years of effort.

The implementation of MACSYMA is based on the belief that the way to produce a high-performance program for general mathematics is to "build in" a large amount of knowledge. The corresponding approach to system construction is often called "knowledge-based programming." MACSYMA is an extremely large system, as algebraic manipulation systems go; at present, it can perform at least 600 distinct mathematical operations, including differentiation, integration, solution of equations and systems of equations, Taylor series expansions, matrix operations, vector algebra, order analysis, etc. The current system consists of about 230,000 words of compiled LISP code and an equal amount of code written in the MACSYMA programming language. About one half of this code was written by MACSYMA staff members; the rest was contributed by various users.

The primary goal of algebraic manipulation research has been the invention and analysis of new mathematical algorithms and the extension of previously known numerical algorithms to symbolic manipulation.

While most of the algorithms incorporated in MACSYMA were known to mathematicians prior to its construction, a substantial number came about as a result of this research. The last decade has brought the discovery of new algorithms for finding the greatest common divisors of polynomials (Brown & Traub, 1971; Moses & Yun, 1973), factoring rational expressions (Musser, 1975), (Wang & Rothschild, 1975), sum simplification (Gosper, 1977), symbolic integration (Moses, 1971; Norman, 1975; Risch, 1969; Rothstein, 1977; Trager, 1978), and asymptotic analysis (Fateman, 1976; Norman, 1975; Zippel, 1976). The nature of this work has been largely mathematical; and, although Artificial Intelligence was instrumental in providing the environment in which MACSYMA was created, it has made little direct contribution since then.

Knowledge-based programming does, however, engender a number of difficulties for

which AI techniques offer partial answers. Two general types of difficulties are discussed here, namely user education and the handling of mathematical problems not amenable to algorithmic solution.

### Non-algorithmic procedures in MACSYMA

One of the most pressing problems in algebraic manipulation is that of simplification. Symbolic algorithms often generate large, unwieldy expressions that must be simplified into smaller, more meaningful forms. (Generally, the size of expressions is the most important criterion for simplicity, with standard formats and particularly revealing forms taking precedence.) To help users simplify their results, MACSYMA provides a variety of explicit expression transformation commands (such as expansion, factorization, partial fraction decomposition, etc.) and a simplifier that automatically applies a set of mathematical "rules" to every new expression as it is constructed. Examples of these rules are:

$$\begin{aligned} x*x &\rightarrow x^2 \\ \sin(x+\pi/2) &\rightarrow \cos(x) \\ \log(a*b) &\rightarrow \log(a)+\log(b) \end{aligned}$$

The user can, of course, define new commands and new rules.

### Semantic Pattern Matching

In applying a simplification rule, MACSYMA utilizes a "semantic pattern matcher" (RJF) to find instances of the rule's pattern. The matcher is "semantic" in that it uses knowledge about the operators and constants in an expression to find nonsyntactic matches. For example, the pattern  $a*x^2 + b*x + c$ , where  $a$ ,  $b$ , and  $c$  are pattern variables free of  $x$ , will match the expressions  $4*x^2 + 4*x + 1$ ,  $x^2 + x + 1$ ,  $x^2$ , and  $(x + 1)^2$ . In defining a rule, the user may specify arbitrary conditions (in the form of procedural predicates) on the pattern variables. For example, in determining whether an expression matches the above pattern, MACSYMA would call a user-specified function to check that any tentative assignments for  $a$ ,  $b$ , and  $c$  are free of  $x$ . As a result, the pattern would not match  $4*x^2 + 3*x + \sin(x)$ .

One problem with this pattern matcher is that the user is unable to control how much "semantics" the system uses in finding a match. In the very near future, a new pattern matcher will be released, in which the user will be able to specify a set of identities to use in attempting to identify instances of patterns. For example, while it is often desirable that the matcher use inverses, in some situations a user might prefer a simpler matcher, lest the rule  $a*b \rightarrow c$  apply to every lone  $a$  and  $b$ , as in  $b \rightarrow c/a$ . With the new pattern matcher, the user will be able to specify when he wants the inverse axioms to be used.

### Simplification by Hillclimbing

While size is not the sole criterion for simplicity, it is a useful guideline. For those applications in which the user desires the smallest possible form for an expression, MACSYMA provides a search-oriented simplifier called SCSIMP. Given an expression and a set of rules, SCSIMP applies each of the rules to the expression in turn and retains the smallest result. If

any such substitution leads to an expression smaller than the original, the process is repeated. For example, given the identities below, SCSIMP will convert the first expression into the last.

$$\begin{array}{l}
 K^2 + L^2 = 1 \qquad N^2 - M^2 = 1 \\
 \text{First expression: } K^2 N^2 + K^2 M^2 N^2 - K^2 L^2 N^2 - K^2 L^2 M^2 N^2 \\
 \text{Intermediate: } K^4 M^2 N^2 + K^4 N^2 \qquad \text{substituting for } L \\
 \text{Final Expression: } K^4 N^4 \qquad \text{substituting for } M
 \end{array}$$

Note, however, that SCSIMP, being a hillclimbing algorithm, is not guaranteed to produce the smallest answer. For example, it would not perform the simplification shown below.

$$\begin{array}{l}
 \text{First expression: } K^2 N^2 + L^2 M^2 \\
 \text{Intermediate form: } K^2 N^2 - K^2 M^2 + M^2 \qquad \text{substituting for } L \\
 \text{Simplest form: } K^2 + M^2 \qquad \text{substituting for } N
 \end{array}$$

The reason for not performing the simplification is that in order to arrive at the simplest form, a larger intermediate expression would have to be generated. Due to the *combinatorics* involved in generating arbitrarily large intermediate forms, this technique has not been incorporated in the current version of SCSIMP.

### The Relational Database and Inference

In certain problems, the symbols in mathematical expressions have restrictions on their ranges or on other properties useful in simplification. In order to allow the user to specify such properties, MACSYMA maintains a *relational database* of facts about symbols, stored in the form of a *semantic network*. For example, a user can declare (via the DECLARE command) that the symbol  $n$  is restricted to integer values, and MACSYMA can then simplify  $\cos((2^n + 1)\pi)$  to 0. Similarly, one can specify (via the ASSUME command) that  $x \leq y$ ,  $y \leq z$ , and  $z \leq x$ , and MACSYMA can then deduce that  $x = y = z$  (using the algorithm described below).

The database retrieval routines are supplemented by a fast but limited inference algorithm called CPM (Genesereth, 1976), which performs taxonomic deductions, property inheritances, set intersections, and other simple inferences. For example, given the facts that  $X$  is an integer, that integers are rational, and that the real numbers are partitioned into rationals and irrationals, CPM automatically deduces that  $X$  is not an irrational. Given the fact that a rational can be written as an integral numerator over an integral denominator,

CPM automatically deduces that  $X$  can be so written. The CPM inference algorithm was developed to enhance the retrieval capabilities of a high-level database system organized as a semantic network. It is an elaboration of Grossman's work (Grossman, 1976) on *constraint expressions* but has been carefully restricted so as to be susceptible to implementation on parallel hardware. The algorithm is a highly "compiled" form of *domain-independent constraint propagation* in which constraints, represented by "labels" on the nodes of the network, propagate across links to other nodes according to the laws of logic. It can perform certain inferences much more efficiently than their straightforward implementation in procedural problem-solving languages like CONNIVER. For further details on the CPM algorithm, the reader should consult Genesereth, 1976. In addition, Fahlman (1977) has described how such a constraint propagation algorithm can be implemented in parallel hardware for even greater efficiency.

### Heuristic Problem Solving

MACSYMA also includes a number of specialized procedural problem solvers; for example, the first phase of the integration routine (Moses, 1971), the commands for performing root contraction and logarithmic contraction, the inequality theorem prover, and others.

### User Education

The advantage of a large knowledge-based system like MACSYMA over a smaller, sparser system like REDUCE (Hearn, 1973) is that MACSYMA has more mathematical knowledge built in (i.e., it is larger and can do more). As a consequence, the user is not forced to communicate as much mathematical knowledge to the system. The disadvantage is that MACSYMA can be more difficult to understand and to use. The user might, for example, be unaware of the capabilities available or not know the names of commands, or he might get an unexpected result that he cannot explain.

To minimize these difficulties, MACSYMA offers a wide range of on-line user aids (Genesereth, 1977; Lewis, 1977), including a frame-oriented interactive primer (similar to PLATO), an information network, and an automatic program for searching the reference manual. In addition, some of MACSYMA's commands are able to explain their progress in a fashion that can be comprehended by the user. For example, if the VERBOSE option is selected, the POWERSERIES command prints out the goals and subgoals that it generates while working on an expansion.

Even with these provisions, users occasionally encounter difficulties due to their lack of knowledge of the system. Furthermore, such users are often unwilling to learn more about MACSYMA than is necessary to solve the immediate problem. The simplest way for such a user to acquire just the information he needs is to ask a consultant for help. Then, armed with the consultant's advice, he may surmount the difficulty and solve the problem.

Consultation is a method widely used in computer centers as well as in domains like business, law, and medicine; and, as computer technology becomes more pervasive and computer systems become more complex, the need for consultation grows. Unfortunately, human consultants are a scarce resource and quite expensive. Currently, work is underway

on an automated consultant for MACSYMA novices, called the Advisor. It is a program distinct from MACSYMA, with its own database and expertise. The Advisor accepts a description of a difficulty from its user and tries to reconstruct the user's "plan" for solving his problem. Based on this plan and its knowledge of MACSYMA, the Advisor then generates advice tailored to the user's specific need. For a description of the Advisor's operation, the user should see Genesereth, 1978.

### Future Plans

In addition to the features described above, several other AI-related capabilities are under development. Two of these are mentioned here, namely a new representation for algebraic expressions using data abstractions and a knowledge-based, plan-based mathematician's (or physicist's or engineer's) "apprentice."

Recently, David Barton has designed a radically new scheme for representing algebraic expressions. MACSYMA has two major representations, the general representation that uses LISP's traditional prefix format and the rational representation that uses a canonical form for polynomials and rational functions. The rational representation has become unwieldy over the years as extensions to the system changed its specifications. For example, coefficients of polynomials were originally assumed to be integers and were later generalized to include floating point numbers. A new representation was desired to handle "Taylor series," which contains rational number exponents, since the former representation, while relatively close to the rational representation, could not be retrofitted onto the rational representation. Barton's approach alleviates these difficulties and provides a capability for future generalization. The approach used is, furthermore, a natural one for abstract algebra.

Consider, for example, a  $2 \times 2$  matrix whose elements are Laurent series in  $y$  (truncated at  $y^2$ ), whose coefficients are polynomials in  $x$ , whose coefficients are rational numbers. In order to add such a  $2 \times 2$  matrix to another  $2 \times 2$  matrix, one needs to know how to add the elements. One approach would be to design a general addition routine that would check the types of each argument and finally perform the appropriate addition. This is similar to the approach previously taken by the rational function representation. In a symbolic system, and, in fact, in most applications, the type of object is intimately related to a set of operations that can be performed on it. In the MACSYMA context, these operations include addition, subtraction, multiplication, division, differentiation, substitution, coefficient extraction, and GCD computation. Barton's approach is to attach a tree of vectors to each expression. The tree corresponds to the gross structure of the expression. Each subexpression, for example, an element in the matrix, has a vector corresponding to it. The vector's elements are in a fixed order and contain pointers to the procedures that perform the corresponding operation on the type of the subexpression.

The approach indicated above permits expressions to be composed of arbitrarily nested types. This is a critical requirement in an interactive symbolic system. Preliminary testing of expressions represented in this manner indicates that common manipulations are not much slower and often faster than in the existing implementation. The reason for a speed-up is that less type-testing is needed in this approach.

Work has also begun on the design of an "apprentice" for the MACSYMA user. At present, MACSYMA is used mostly as a "symbolic calculator," with the user directing its



actions line by line and keeping track of the meaning of each result. The goal of the apprentice is to relieve the user of much of this drudgery. The approach being taken involves two components, namely knowledge about the user's domain and the use of a high-level problem-solving plan formalism.

Currently, most symbols in MACSYMA have no special meaning, and they can take on arbitrary values. In particular problem areas, however, certain symbols have particular interpretations and range restrictions. For example, the symbol MASS has a very special meaning to physicists and an obvious range restriction (nonnegative). A physicist's apprentice should know this range restriction and be able to use it, for example, in discarding negative roots or performing integrations. Similarly, practitioners in certain fields like to see their expressions written in standard formats, determined by the interpretation of the constituent symbols. For example, electrical engineers usually prefer resistance (Ri) and capacitance (Ci) expressions written as  $f(R_1, R_2, \dots, R_n) * g(C_1, C_2, \dots, C_n)$  rather than having the Ri and Ci intermixed.

Another way in which an apprentice could be of use is by keeping track of the user's *plan* for solving his problem. If the apprentice knows the steps involved and the significance of various results, it could inform the user of potential errors, make suggestions, and in many cases carry out steps by itself. The apprentice can gain familiarity with the user's plan in various ways: It may be a well-known mathematical procedure (e.g., some standard technique for solving partial differential equations or perturbation problems), or the user may describe his intentions before beginning his MACSYMA session, or the user may re-apply some previous plan. It is expected that this notion of a problem-solving plan will play an extremely important role in the next generation of algebraic manipulation systems.

### References

Unfortunately, there is no good introductory reference on the structure of MACSYMA. The reader is referred to the MACSYMA manual (Mathlab Group, 1977) and the primer (Moses, 1975) for an introduction to its use.

See also Brown & Traub (1971), Fahman (1977), Fateman (1972), Genesereth (1976), Genesereth (1977), Genesereth (1978), Gosper (1977), Grossman (1976), Hearn (1973), Lewis (1977), Moses (1971), Moses & Yun (1973), Musser (1975), Norman (1975), Risch (1969), Rothstein (1977), Trager (1978), Wang & Rothschild (1975), and Zippel (1976).

### A3. AM

AM is a computer program written by Douglas Lenat that explores elementary mathematics, enlarging its vocabulary of objects and operators by defining new ones, gathering empirical data about the concepts it possesses, and making conjectures to connect some of these mathematical entities.

The design of AM is a blending of four powerful methodologies: *heuristic search*, *production systems*, *best-first search*, and *frames*. AM is guided in its exploration by over two hundred heuristics, which act like plausible move generators; they suggest new definitions and conjectures, and they give hints as to how to gather more empirical data. The tutelage of so many heuristics is coordinated by representing each as a production rule, an expression of the form "IF *some condition* is satisfied THEN do *some appropriate action*" (see article Representation.B3). The space that AM is exploring is so immense that some kind of chunking is required; the grain of operators like "conjoin two existing definitions" is too fine. The third idea woven into AM, that of best-first search, enters in the form of an *agenda*, a job-list of fairly large, meaningful tasks--each of which is plausible. The tasks are at a much higher level, such as "generalize 'Prime numbers'", and each one is supported by a set of symbolic reasons. Together, the reasons give an overall numeric rating to the task, which determines its place on the agenda. Only a small fraction of the tasks placed on the agenda will ever be worked on: At each moment, AM expends its energy on the best task, the one with the highest rating. Each concept is represented as a frame, a structured entity with slots (Generalizations, Examples, Definition, etc.). This enables tasks--and heuristics--to talk about a particular slot of a particular concept.

The program began to run with a collection of one hundred concepts of finite set theory and relations and data structures, and in a couple hours it had defined about three hundred new concepts, half of which were quite well known in mathematics. One of the synthesized concepts was equivalent to natural numbers. AM rated this highly and spent much time developing elementary number theory, including conjecturing the fundamental theorem of arithmetic (each number has a unique prime factorization). This is of course much better behavior than one could expect from a blind sort of search through the space of legal mathematical definitions and propositions. At any moment, AM could justify its current efforts merely by printing out the symbolic reasons for the top-rated task, the one it was working on.

The significance of the project lies both in the architecture of the program (a blend of three now-popular methodologies) and in the fact that the program behaved well: An existence proof that open-ended math research--theorem *proposing* not theorem proving--could be adequately represented (and automated) as a heuristic search. Finally, it is worth noting that the ultimate impediment to AM's progress was its inability to discover new heuristic rules, as it had discovered new mathematical concepts. Only by constructing and experimenting with the program were we thus able to learn where the next research thrust should be, along the direction of automating the discovery and evaluation of heuristics.

### THE TASK

### Rationale

Few doubt the ubiquity of *inductive inference*, but the archetypical such enterprise is scientific research and, more specifically, mathematics research. To be more concrete:

1. In math research, one needn't cope with the uncertainties and fallibility of testing equipment; there are no uncertainties in the data (compared to, e.g., molecular structure inference from mass spectrograms).
2. Reliance on experts' introspections is one of the most powerful techniques for codifying the judgmental criteria necessary to do effective work in the field. By limiting a program to areas of mathematics in which the programmer is competent, reliance on external sources to guide formulating heuristic rules can be minimized. There are, however, several excellent sources available (Poincare, 1929, Polya, 1954, Lakatos, 1976, Knuth, 1974, Brotz, 1974, and Hadamard, 1945).
3. The more formal a science is, the easier it is to automate. For a machine to carry out research in psychology would require more knowledge about human information processing than now is known. Also, in a formal science, the *languages* used to communicate information can be quite simple, even though the *messages* themselves are sophisticated.
4. Since mathematics can deal with any conceivable construct, a researcher in this field is not limited to explaining observed data: He can also *create*. Related to this is the freedom to investigate--or to give up on--whatever the researcher wants to investigate. There is no single discovery that is the "goal," no given problem to solve, and no right or wrong behavior.
5. Unlike "simpler" fields, such as propositional logic, there is an abundance of heuristic rules available for the picking.

The limitations of math as a domain are closely intertwined with its advantages. For example, having no ties to real-world data can be viewed as a limitation, as can having no clear goal. There is always the danger that AM will give up on each theory as soon as the first tough obstacle crops up.

### Plausible Reasoning in Mathematics

The inadequacy of formal deductive methods in mathematics has long been argued (Poincare, 1929, Polya, 1954, Lakatos, 1976) and often lamented in the conclusions to resolution theorem proving articles (Bledsoe, 1971, Wang, 1960). An early use of analogic models in geometry theorem proving was quite successful (Gelernter's program [Gelernter, 1963] "drew" a diagram meeting all the premises of the theorem, and [among other things] presumed that segments or angles which turned out to be equal were not so coincidentally; all such equalities could then be drawn upon as lemmata). Later attempts at the introduction of heuristics into resolution theorem provers have been made by Bledsoe (in, e.g., nonstandard analysis), Pitrat (in propositional calculus), and a few others (Bledsoe, 1971, Kling, 1971, Brotz, 1974). It is hypothesized that the limited successes of these schemes is due to the relatively small number and variety of heuristics they possessed and to the fact

that resolution was the dominant driving force in each program, using heuristics merely to modulate the resolution search process. The reason for the small number of heuristics is simply the genuine paucity of informal rules-of-thumb that exist for the general environment in which the vast majority of those programs operate: domain-independent (asemantic) predicate calculus theorem proving.

Lenat's thesis (Lenat, 1976) was concerned with the mechanization of the "other half" of mathematical activity (apart from proving): the definition of new concepts and the recognition of plausible conjectures. His "AM" system has no proof capabilities, and this is one part of the AM model of Mathematics Research. Below is the model of math research that AM was based upon, pieced together from the writings of Poincare, Polya, Lakatos, Hadamard, and others:

1. The order in which a math textbook presents a theory is almost the exact opposite of the order in which it was actually discovered and developed. In such a text, new definitions are stated with little or no motivation as they are needed to state the next big theorem, whose proof then magically appears. In contrast, a mathematician doing research examines some already known concepts and tries to find some regularity in experimental data involving them. The patterns that he notices are the conjectures that he must investigate further, and these relationships directly motivate him to make new definitions.
2. Each step the researcher takes while developing a new theory involves choices from a large set of "legal" alternatives. The key to keeping this from becoming a blind, explosive search is the proper use of evaluation criteria. Each mathematician uses his own personal heuristics to choose the "best" alternative available at each moment.
3. Non-formal criteria (aesthetic interestingness, inductive inference from empirical evidence, analogy, and utility) are much more important than formal deductive methods in developing mathematically worthwhile theories, and in avoiding barren diversions.
4. Progress in *any* field of mathematics demands much non-formal heuristic expertise in *many* different "nearby" mathematical fields. So a broad, universal core of knowledge must be mastered before any single theory can meaningfully be developed.
5. It is sufficient, and pragmatically necessary, to have and use a large set of informal heuristic rules. These rules direct the sequence of the researcher's activities, depending on the current situation that he is in. These rules can be assumed to superimpose: The combined effect of several rules is just the sum of the individual effects.
6. The necessary heuristic rules are virtually the same in all branches of mathematics and at all levels of sophistication. Each specialized field will have some of its own heuristics; those are normally much more powerful than the general-purpose heuristics.
7. For true understanding, the researcher should grasp--that is, have access to, relate

to, store, be able to manipulate, be able to answer questions about, etc.--each concept in several ways: declaratively, abstractly, operationally, knowing its relevance and examples of it.

8. There are common metaphysical assumptions about nature and science: Nature is fair, uniform, and regular. Coincidences have meaning. Statistical considerations are valid when looking at mathematical data. Simplicity and symmetry and synergy are the rule, not the exception.

### Discovery in Mathematics

Before discussing how to *synthesize* a new mathematical theory, consider briefly how to *analyze* one, or how to construct a plausible chain of reasoning that stretches from a given discovery all the way back to well-known concepts.

### Analysis of a Discovery

One can rationalize a given discovery by working backwards, by reducing the creative act to simpler and simpler creative acts. For example, consider the concept of prime numbers. How might one be led to define such a notion, if one had never heard of it before? Consider the following plausible strategy:

If  $f$  is a function which transforms elements of  $A$  into elements of  $B$ , and  $B$  is ordered, then consider just those members of  $A$  which are transformed into *extremal* elements of  $B$ . This set is an interesting subset of  $A$ . Name it and study it.

When  $f(x)$  means "divisors of  $x$ ," and the ordering is "by length," this heuristic directs one to consider those numbers that have a minimal number of factors--that is, the primes. So this rule actually *reduces* our task from proposing the concept of prime numbers to two more elementary problems: (a) discovering ordering-by-length and (b) inventing divisors-of..

But suppose we know this general rule: "*If  $f$  is an interesting function, consider its inverse.*" It reduces the task of discovering divisors-of to the simpler task of discovering multiplication. Eventually, if followed far enough, this task reduces to the discovery of very basic notions like substitution, set-union, and equality. To explain how a given researcher might have made a given discovery, such an analysis must be continued until that inductive task is reduced to "discovering" the notions that the researcher started with, which were his *conceptual primitives*.

### Syntheses of Discoveries

Suppose a large collection of these heuristic strategies has been assembled (e.g., by analyzing a great many discoveries and writing down new heuristic rules whenever necessary). Instead of using them to *explain* how a given idea might have evolved, one can imagine starting from a basic core of knowledge and "running" the heuristics to *generate* new

concepts. It is simply the reversal of the process described in the last section: not *explanation*, but *generation*.

Notice that this *forward search* is much "bushier"--i.e., more branches or paths to follow--and much more explosive than the backwards analysis previously described. It is a much harder task to actually make a discovery than to rationalize--by hindsight--one already made. This is a common enough phenomenon, the "Why-didn't-I-think-of-that-sooner!" feeling.

Unconstrained forward search is too explosive (see Combinatorial Explosion in article *Search.Overview*); thus, we can hypothesize that the scientist employs some kind of informal rules-of-thumb or heuristics to constrain it. That is, he doesn't really follow rules like "*Look at the inverse of each known function  $f$* ", because that would take up too much time. Rather, his heuristic rules might be more naturally stated as productions (condition/action rules) like this: "*If  $f$  is 1-1 and  $\text{Range}(f) \ll \text{Domain}(f)$ , Then look at  $f$ -inverse.*" Henceforth, *heuristic rule* will mean a conditional rule-of-thumb. In any particular situation some subset of these rules will "trigger" and suggest a manageable space of plausible activities to perform. After exploring that space for a while, the situation will have changed and the cycle will begin anew. This layering of heuristic search is simply analogous to performing double induction instead of standard mathematical induction.

#### DESIGN OF THE AM PROGRAM

Mathematical inductive syntheses are precisely what AM does. The program consists of a large corpus of primitive mathematical concepts, each with a few associated heuristics. Each such heuristic is a situation-action rule that functions as a local *plausible move generator*. Some suggest tasks for the system to carry out, some suggest ways of satisfying a given task, etc. AM's activities all serve to expand AM itself, to enlarge upon a given body of mathematical knowledge. AM uses its heuristics as judgmental criteria to guide development in the most promising direction.

#### Representation

Each concept is represented as a frame-like data structure (unit, Being, schema, script,...) with 25 different facets (slots, parts, aspects,...). The types of facets include: EXAMPLES, DEFINITIONS, GENERALIZATIONS, DOMAIN/RANGE, ANALOGIES, INTERESTINGNESS, and many others. Modular representation of concepts provides a convenient scheme for organizing the heuristics; for example, the following strategy fits into the *EXAMPLES* facet of the *PREDICATE* concept:

If, empirically, 10 times as many elements FAIL some predicate P, as SATISFY it, then some *generalization* (weakened version) of P might be more interesting than P.

AM considers this suggestion after trying to fill in examples of each predicate. In fact, after AM attempts to find examples of *SET-EQUALITY*, so few are found that AM decides to generalize that predicate. The result is the creation of several new predicates, one of which happens to mean "Has-the-same-length-as"--that is, a rudimentary precursor to natural numbers.

Below is a typical concept, PRIMES, in a state long after AM defined and explored it.

NAME: Prime Numbers

DEFINITIONS:

ORIGIN: Number-of-divisors-of(x) = 2

PREDICATE-CALCULUS: Prime(x) =  $(\forall z)(z|x \supset z=1 \vee z=x)$

ITERATIVE: (for x>1): For i from 2 to Sqrt(x),  $\setminus(i|x)$

EXAMPLES: 2, 3, 5, 7, 11, 13, 17

BOUNDARY: 2, 3

BOUNDARY-FAILURES: 0, 1

FAILURES: 12

GENERALIZATIONS: Nos., Nos. with an even no. of divisors, Nos. with a prime no. of divisors

SPECIALIZATIONS: Odd Primes, Prime Pairs, Prime Uniquely-addables

CONJECTS: Unique factorization, Goldbach's conjecture, Extremes of Number-of-divisors-of

INTU'S: *A metaphor to the effect that Primes are the building blocks of all numbers*

ANALOGIES:

Maximally divisible numbers are converse extremes of Number-of-divisors-of

Factor a non-simple group into simple groups

INTEREST: Conjectures tying Primes to TIMES, to Divisors-of, to closely related operations

WORTH: 800

Creating a new concept is a well-defined activity: It involves setting up a new data structure like the one above and filling in entries for some of its facets or slots. Filling in a particular facet of a particular concept is also quite well defined and is accomplished by executing a collection of relevant heuristic rules.

### Control

AM is initially given a collection of 115 core concepts, with only a few facets filled in for each. Its sole activity is to choose some facet of some concept and fill in that particular slot. In so doing, new notions will often emerge. Uninteresting ones are forgotten, mildly interesting ones are kept as parts of one facet of one concept, and very interesting ones are granted full concept-module status. Each of these new modules has dozens of blank slots, hence the space of possible actions (blank facets to fill in) grows rapidly. The same heuristics are used both to suggest new directions for investigation and to limit attention, that is, both to sprout and to prune.

The fundamental kind of task that AM performs, its basic action, is filling in a given facet of a given concept. To decide which such task to work on next, AM maintains an *agenda* of tasks, a global job-list ordered by priority. A typical task is "*Fill-in examples of Primes*". The agenda may contain hundreds of entries such as this one. AM repeatedly selects the top task from the agenda and tries to carry it out. This is the whole control structure! Of course, AM creates plausible new tasks to place on the agenda and decides which task will be the best one to execute next and how to carry it out.

If the task is "*Fill in new Algorithms for Set-union*", then *satisfying* it would mean actually synthesizing some new procedures, some new LISP code capable of forming the union of any two sets. A heuristic rule is *relevant* to a task iff executing that rule brings AM closer to satisfying that task. Relevance is determined a priori by where the rule is stored. A rule tacked onto the Domain/range facet of the Compose concept would be presumed relevant to the task "*Check the Domain/range of Insert-o-Delete*".

Once a task is chosen from the agenda, AM gathers some heuristic rules which might be relevant to satisfying that task. They are executed, and then AM picks a new task. While a rule is executing, three kinds of actions or effects can occur:

1. Facets of some concepts can get filled in (e.g., examples of primes may actually be found and tacked onto the "Examples" facet of the "Primes" concept). A typical heuristic rule that might have this effect is:

To fill in examples of X, where X is a kind of Y (for some more general concept Y), Check the examples of Y; some of them may be examples of X as well.

For the task of filling in examples of Primes, this rule would have AM notice that Primes is a kind of Number and therefore look over all the known examples of Number. Some of those would be primes and would be transferred to the Examples facet of Primes.

2. New concepts may be created (e.g., the concept "primes which are uniquely representable as the sum of two other primes" may somehow be deemed worth studying). A typical heuristic rule that might result in this new concept is:

If some (but not most) examples of X are also examples of Y (for some concept Y), Create a new concept defined as the intersection of those 2 concepts (X and Y).

Suppose AM has already isolated the concept of being representable as the sum of two primes in only one way (AM actually calls such numbers "Uniquely-prime-addable numbers"). When AM notices that some primes are in this set, the above rule will create a brand new concept defined as the set of numbers that are both prime and uniquely prime addable.

3. New tasks may be added to the agenda (e.g., the current activity may suggest that



the following task is worth considering: "Generalize the concept of prime numbers"). A typical heuristic rule that might have this effect is:

If very few examples of X are found, Then add the following task to the agenda: "Generalize the concept X."

Of course, AM contains a precise meaning for the phrase "very few." When AM looks for primes among examples of already known kinds of numbers, it will find dozens of nonexamples for every example of a prime it uncovers. "Very few" is thus naturally implemented as a statistical confidence level.

The concept of an agenda is certainly not new. *Schedulers* have been around for a long time. But one important feature of AM's agenda scheme is a new idea: attaching, and using, a list of quasi-symbolic reasons to each task that explain why the task is worth considering, why it's plausible. *It is the responsibility of the heuristic rules to include reasons for any tasks they propose.* For example, reconsider the heuristic rule mentioned in (3) above. It actually looks more like the following:

If very few examples of X are found, then add the following task to the agenda: "Generalize the concept X," for the following reason: "X's are quite rare; a slightly less restrictive concept might be more interesting."

If the same task is proposed by several rules, then several different reasons for it may be present. In addition, one ephemeral reason also exists: Focus of attention. Any tasks that are similar to the one last executed get "Focus of attention" as a bonus reason. AM uses all these reasons to decide how to rank the tasks on the agenda. The "intelligence" AM exhibits is not so much "what it does" as the *order* in which it arranges its agenda. For example, alternating a randomly chosen task and the "best" task (the one AM chose to do) only slows the system down by a factor of 2, yet it totally destroys its credibility as a rational researcher (as judged by the human user of AM). This was one conclusion of an experiment performed upon AM.

AM uses the list of reasons in another way: Once a task has been selected, the quality of the reasons is used to decide how much time and space the task will be permitted to absorb, before AM quits and moves on to a new task.

A crucial *heritability property* holds: Any method for filling in facet *f* of concept *C* will also work for filling in facet *f* of any *specialization* of *C*. Thus, when the task "Fill in examples of *SET-EQUALITY*" is chosen, AM asks each *generalization* of *SET-EQUALITY* for help. It asks for ways to fill in examples of any Predicate, any Activity, any Concept, and finally of Anything. One such heuristic rule known to the Activity concept says: "Actually execute the activity on some random members of its domain." Hence, to fill in examples of *SET-EQUALITY*, its domain facet is inspected, and AM sees that it takes a pair of objects as its arguments. Then AM accesses the Examples facet of the concept *OBJECT*, where it finds a large list of objects. Obeying the heuristic rule, AM repeatedly picks a pair of them at random and sees if they satisfy *SET-EQUALITY* (by actually running the LISP function stored in the

Algorithms facet of *SET-EQUALITY*). While this will typically return False, it will occasionally locate--by random chance--a pair of equal sets.

Other heuristics, tacked onto other generalizations of *SET-EQUALITY*, provide additional methods for executing the task "Fill in examples of *SET-EQUALITY*." A heuristic stored on the concept *ANY-CONCEPT* says to symbolically instantiate the definition. A bag of tricks is provided for this purpose, one of which ("instantiate the base step of the recursion") works nicely on the recursive definition provided for *SET-EQUALITY*. Notice that, as one might expect, the more general the concept is, the weaker (more time-consuming) its heuristics are. For this reason, AM consults each concept's rules in order of increasing generalization.

Executing a task is achieved by locating relevant rules-of-thumb and evaluating them. The location is performed efficiently because all the concepts are related by generalization/specialization links, and because the above "heritability" property holds.

Notice the omnipresent reliance upon heuristic guidance. They propose the tasks (and associate reasons for them) for the agenda, they propose new concepts to be defined, they discover (by search, synthesis, or analysis) entries that can be put into specific facets of specific concepts. There are even heuristics for naming new concepts (based on how they were formed).

## RESULTS

### An Excerpt

To convey a bit of AM's flavor, we present a brief excerpt of it in action, a bit of a trace of one of its runs. After reading through it, the reader should be convinced that AM is *not* proving theorems, nor is it *randomly* manipulating entries in a knowledge base, nor is it *exhaustively* manipulating or searching. AM is carefully growing a network of data structures representing mathematical concepts, by repeatedly using heuristics both (a) to guide the choice of what task to work on next, and (b) to provide methods to satisfy the chosen task.

The following points are important but are not easily conveyed by any one example:

- (a) Although AM appears to have reasonable natural language abilities, very little effort was expended in this area.
- (b) It is important to ask how general the program is: Is the knowledge base "just right" (i.e., finely tuned to elicit this one chain of behaviors)? The answer is "No": The whole point of this project was to show that a relatively small set of general heuristics can guide a nontrivial discovery process. Keeping the program general and not finely tuned was a key objective. Each activity, each task, was proposed by some heuristic rule (like "look for extreme cases of X") that was used time and time again, in many situations. It was not considered fair to insert heuristic guidance that could only "guide" in a single situation. For example, the same heuristics that lead AM to decompose numbers (using *TIMES-inverse*) and thereby discover unique factorization, also lead to decomposing numbers (using *ADD-inverse*) and the discovery of Goldbach's conjecture.

Now here is the excerpt itself. A few of the lines have been anglicised (e.g., the addition of "e.g."), and some of the concepts had previously been renamed by the user (the human observor of the system). Otherwise, this was just as AM typed it:

**TASK 65: Fill in Examples of the concept Divisors-of.**

- 3 Reasons: (1) No known examples of Divisors-of so far.  
 (2) TIMES, which is related to Divisors-of, is now very interesting.  
 (3) Focus of attention: AM recently defined Divisors-of.

26 examples found, in 9.2 seconds. E.g.,  $\text{Divisors-of}(6) = \{1\ 2\ 3\ 6\}$ .

**TASK 66: Consider numbers having small sets of Divisors-of.**

- 2 Reasons: (1) Worthwhile to look for extreme cases.  
 (2) Focus of attention: AM recently worked on Divisors-of.

Filling in examples of numbers with 0 divisors.  
 8 examples found, in 4.8 seconds.  
 Conjecture: no numbers have precisely 0 divisors.

Filling in examples of numbers with 1 divisors.  
 1 examples found, in 4.8 seconds. e.g.,  $\text{Divisors-of}(1) = \{1\}$ .  
 Conjecture: 1 is the only number with precisely 1 divisor.

Filling in examples of numbers with 2 divisors.  
 24 examples found, in 4.8 seconds. e.g.,  $\text{Divisors-of}(13) = \{1\ 13\}$ .  
 No obvious conjecture. May merit more study.  
 Creating a new concept: "Numbers-with-2-divisors".

Filling in examples of numbers with 3 divisors.  
 11 examples found, in 4.8 seconds. e.g.,  $\text{Divisors-of}(49) = \{1\ 7\ 49\}$ .  
 All numbers with 3 divisors are also Squares.  
 Definitely merits more study.  
 Creating a new concept: "Numbers-with-3-divisors".

**TASK 67: Consider the square-roots of Numbers-with-3-divisors.**

- 2 Reasons:  
 (1) Numbers-with-3-divisors are unexpectedly also Perfect Squares.  
 (2) Focus of attention: AM recently worked on Nos-with-3-divisors.

All square-roots of Numbers-with-3-divisors seem to be Numbers-with-2-divisors.  
 E.g.,  $\text{Divisors-of}(\text{Square-root}(169)) = \text{Divisors-of}(13) = \{1\ 13\}$ .

Even the converse of this seems empirically to be true.  
 I.e., the square of each No-with-2-divisors seems to be a No-with-3-divisors.  
 The chance of coincidence is below acceptable limits.

Boosting the interestingness rating of each of the concepts involved.

*USER: Rename Numbers-with-2-divisors as Primes*

**TASK 68: Consider the squares of Numbers-with-3-divisors.**

- 3 Reasons: (1) Squares of Numbers-with-2-divisors were interesting.  
(2) Square-roots of Numbers-with-3-divisors were interesting.  
(3) Focus of attention: AM recently worked on Nos-with-3-divisors.

### AM as a Mathematician

Having seen how AM works and examples of "local" results, the final point here is to consider the totality of the mathematics that AM carried out. All of this was done by AM acting alone, with a human user watching it and occasionally renaming some concepts for his or her own benefit. Like a contemporary historian summarizing the work of the Babylonian mathematicians, current terms are used and criticism is by current standards.

AM began its investigations with scanty knowledge of a few set-theoretic concepts. Most of the obvious set-theory relations (e.g., de Morgan's laws) were eventually uncovered; since AM never fully understood abstract algebra, the statement and verification of each of these was quite obscure. AM never derived a formal notion of infinity, but it naively established conjectures like "a set can never be a member of itself," and procedures for making chains of new sets ("insert a set into itself"). No sophisticated set theory (e.g., diagonalization) was ever done.

After this initial period of exploration, AM decided that "equality" was worth generalizing and thereby discovered the relation "same-size-as." Natural numbers were based on this discovery, and soon most simple arithmetic operations were defined.

Since addition arose as an analog to union, and multiplication as a repeated substitution, it came as quite a surprise when AM noticed that they were related (namely,  $N + N = 2 \times N$ ). AM later rediscovered multiplication in three other ways: as repeated addition, as the numeric analog of the Cartesian product of sets, and using the cardinality of the power set of the union of two sets.

Raising to fourth-powers and fourth-rooting were discovered at this time. Perfect squares and perfect fourth-powers were isolated. Many other numeric operations and kinds of numbers were found to be of interest: Odds, Evens, Doubling, Halving, Integer-square-root, etc. Although it isolated the set of numbers that had no square root, AM was never close to discovering rationals, let alone irrationals. No notion of "closure" was provided to--or discovered by--AM.

The associativity and commutativity of multiplication indicated to AM that it could accept a BAG of numbers as its argument. When AM defined the inverse operation corresponding to Times, this property allowed the definition to be: "Any bag of numbers ( $>1$ ) whose product is  $x$ ." This was just the notion of factoring a number  $x$ . Minimally factorable numbers turned out to be what we call primes. Maximally factorable numbers were also thought to be interesting.

Prime pairs were discovered in a bizarre way: by restricting the domain and range of addition to primes (i.e., solutions of  $p + q = r$  in primes).

AM conjectured the fundamental theorem of arithmetic (unique factorization into primes) and Goldbach's conjecture (every even number  $>2$  is the sum of two primes) in a surprisingly symmetric way. The unary representation of numbers gave way to a representation as a bag of primes (based on unique factorization), but AM never thought of exponential notation. Since the key concepts of remainder, greater-than, gcd, and exponentiation were never mastered, progress in number theory was arrested.

When a new base of *geometric* concepts was added, AM began finding some more general associations. In place of the strict definitions for the equality of lines, angles, and triangles came new definitions of concepts comparable to Parallel, Equal-measure, Similar, Congruent, Translation, Rotation; plus many that have no common name (e.g., the relationship of two triangles sharing a common angle). A cute geometric interpretation of Goldbach's conjecture was found [Given all angles of a prime number of degrees, (0,1,2,3,5,7,11,...,179 degrees), then any angle between 0 and 180 degrees can be approximated (to within 1 degree) as the sum of two of those angles.] Lacking a geometry "model" (an analogic representation like the one Gelernter employed [Gelernter, 1963]), AM was doomed to propose many implausible geometric conjectures.

#### Limitations of AM

Although AM fared well according to several different measures of performance, users of this handbook may better utilize knowledge of its *limitations*.

As AM ran longer and longer, the concepts it defined were further and further from the primitives it began with, and the efficacy of its fixed set of 250 heuristics gradually declined. The key deficiency was that of adequate *meta-rules* (Davis, 1976a, Lenat, 1976, Laing, 1971): heuristics that cause the creation and modification of new heuristics. This lack would only be felt in a boot-strapping, open-ended task environment such as math research.

Many concepts that one might consider "primitive" are missing from AM: proof, tricks for finding counterexamples, numbers, etc. Very few of them are ever discovered by AM, and even those that are discovered will not have any powerful heuristics filled in for them. The limitations of a "too small" knowledge base can be overcome only by investing the additional time to enlarge it. With a learning system like AM, one can spend a couple man-hours wrestling with each new concept or let the program squander a probable greater amount of its time until it has discovered and mastered that concept to the same level of proficiency. It is a trade-off that almost always argues for the system-builder to spend more time enlarging the knowledge base by hand.

Analogies in general were under-utilized. Specifically, analogies between heuristics were never even considered. If one characterizes an analogy as a (partial) correspondence between two collections of objects and operators, then it is a small conceptual step to imagine heuristic rules that look for and develop such mappings: The image of partial matching comes immediately to mind. AM possessed a few such domain-independent rules, and they managed to produce some analogies (e.g., between multiplication and addition; between sets/union/same-size and numbers/add/equality), but the overall results were quite meager in this area.

## Conclusions

The AM project stands as a working demonstration that a few hundred general heuristic rules suffice to guide a searcher--an automated math researcher--as it explores and expands a large but incomplete base of math concepts. AM shows that creative theoretical research can be effectively modeled as heuristic search, just as Meta-Dendral (Buchanan, 1975) established a similar hypothesis for the more constrained, real-world field of mass spectroscopy.

AM's design combined four now popular themes: modular representation of knowledge, distributed control via production rules, reliance upon heuristics acting as plausible-move generators, and best-first searching via an agenda scheduler. AM introduced (1975) that control structure based upon an agenda of small plausible research tasks, each with an attached list of supporting reasons.

The main successes were the few novel ideas it came up with (including a new result in number theory, dealing with numbers having very many divisors), the ease with which new domains were discovered (e.g., number theory) or introduced by hand (plane geometry), and the apparently rational sequences of behavior that AM exhibited.

The continuation of this line of research by Lenat is the EURISKO program. The hypothesis that has been added is that the meta-level knowledge required to synthesize and reason about heuristics is a *subset* of the knowledge already in AM about synthesizing and reasoning about concepts. That is, EURISKO's meta-rules are merely some of the very general rules that AM already had. The only real change, then, from AM to EURISKO is to re-represent (recode) each heuristic from Lisp code as a full-fledged concept with facets. The heuristics, which deal with facets of concepts, will then be capable of dealing with each other. This work is currently in progress at Stanford University.

Future AM-like programs may serve as assistants to scientists and engineers, synergetically collaborating with them in the conception, planning, and execution of their research and development activities.

## References

See Bledsoe (1971), Brotz (1974), Buchanan (1975), Davis (1976a), Engelmores & Feigenbaum (1976), Evans (1968), Feigenbaum, Buchanan, & Lederberg (1971), Gelernter (1963), Hadamard (1946), Hempel (1962), Kling (1971), Knuth (1974), Koestler (1967), Laing (1971), Lakatos (1976), Lenat (1976), Minsky (1975), Papert (1972), Pivar & Finkelstein (1964), Poincare (1929), Polya (1954), Simon & Kotovsky (1963), Simon (1973), Wang (1960), and Winston (1970b).

## B. Miscellaneous Applications of Artificial Intelligence

### B1. The SRI Computer-based Consultant

#### Introduction

A computer-based consultant (CBC) is a computer system that contains a body of specialized knowledge about a particular task domain and that makes that knowledge conveniently available to users working in the domain. This article describes some research done at SRI on a computer-based consultant designed to help a novice mechanic work with electromechanical equipment. The goal of this research is to build a system that approximates a human consultant in its communication, perceptual, and reasoning skills.

The consultant was designed to answer spoken English questions from the user and to monitor the user's progress on the task, offering advice and reminders where necessary. To fit the needs of individual users, it is essential that the system be able to provide advice about the task at several levels of detail. In order to determine the appropriate level of detail, the CBC must form a model of the user, monitor his performance as he executes the task, and update internal models to reflect the current state of the *task environment*.

#### Design of the Computer-based Consultant.

The task of the SRI computer-based consultant is to help an inexperienced mechanic repair and modify complex electromechanical equipment. The mechanic works on a piece of equipment in a special "work station" where he is provided with a headset that enables him to talk to the system and to receive spoken replies, both in natural language. A commercially available phoneme synthesizer is used by the system to give "spoken" responses to the user, and a commercially available phrase recognizer is used to "understand" his speech. There is a television camera and a laser rangefinder that provide the visual component for the system. The laser rangefinder can also be used as a visual pointer so that the system can answer questions such as "Show me the pressure switch" by illuminating the pressure switch with the laser beam.

Requests for information by the user are translated into an internal representation or "model" by the natural language and visual components of the system. These models are used to structure communications with the user as he performs the task. For example, a question about the location of a part ("Where is the pump brace) is answered by reference to a stored geometric model that keeps track of the spatial relations between the parts. Other models are necessary for the natural language components of the system; for instance, a discourse model is needed to understand a spoken utterance.

#### Planning a sequence of constructions

The user of the CBC can ask it to plan a sequence of assembly steps and relate this sequence to him for execution. The CBC has a *planning component* for composing assembly and disassembly sequences. It has received much attention in recent research efforts. There are several types of knowledge that are important in the planning process. First,



there is the model of the air compressor itself, which is essentially a graph whose nodes correspond to the parts of the compressor and whose arcs correspond to the mechanical connection between the parts. Second, each type of connection has associated with it a set of procedures that tells how that connection is physically established. Third, each of these procedures may contain calls to other procedures that elaborate, in more detail, how a job is done. This *hierarchy of procedural knowledge* forms the basis for producing plans that can be given to the user at several levels of detail. This procedural model is used by the planning program to determine the order in which parts should be assembled. The planning program initially assumes that the parts can be connected in any order. By checking preconditions and the effects of performing each step, it reorders the steps in the plan to eliminate conflicts. For example, the pump can be installed only if there is no pulley on its shaft. The planner recognizes this fact and imposes an order on the plan so that the pump will be installed before its pulley is placed on the shaft. When all the conflicts have been resolved, the remaining steps of the plan can be solved in any order. This ability, to recognize alternative orderings in a plan, is important for a computer-based consultant: The user may take the initiative and proceed with certain steps of the assembly on his own, and the planner must recognize if the steps being taken are valid.

The plan is represented as a structure called a *procedural net*; a sample net is shown in Figure 1 (Hart, 1975). Each node corresponds to an assembly step at some level of detail. The procedural net is actually a hierarchy of plans, all of which accomplish the same task, but at varying levels of detail. The  $i$ th row in the net corresponds to a plan specified at the  $i$ th level of detail. Notice that the plan splits into two paths at level 2, indicating that the two subplans can be performed in either order. The dotted lines indicate the expansion of a step into a more detailed subplan.

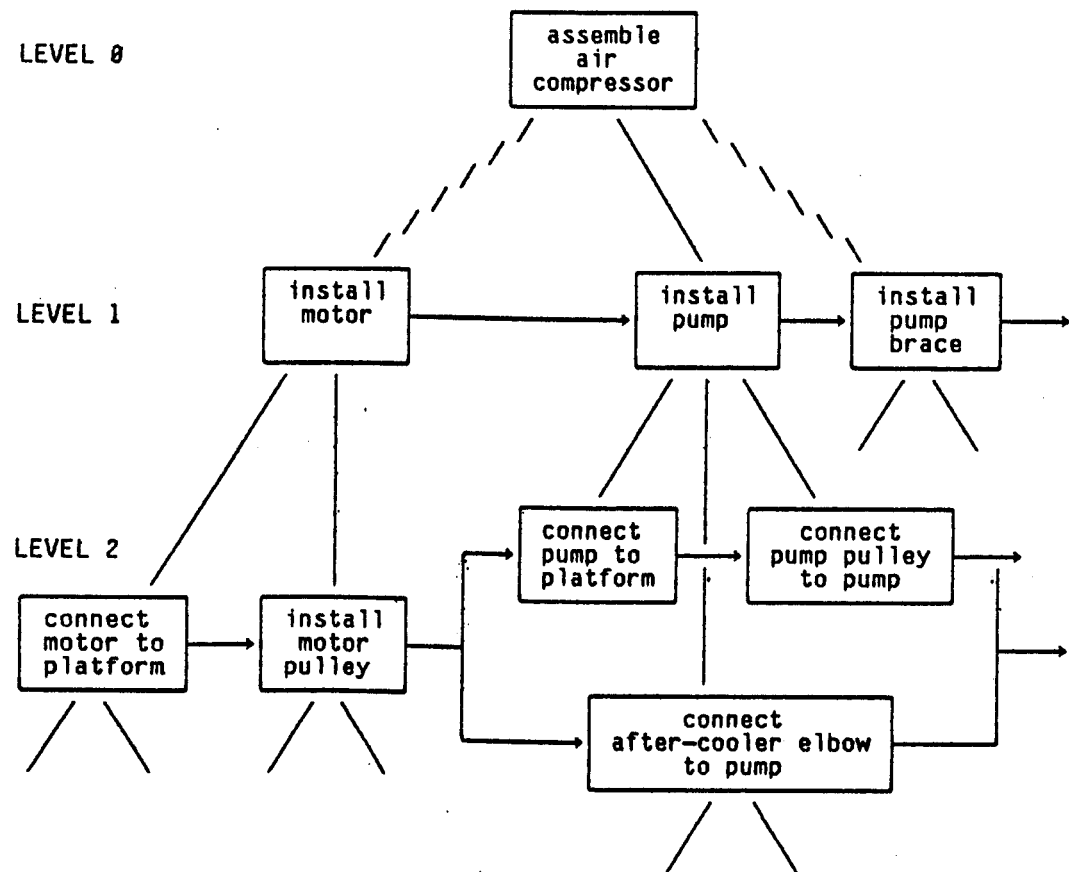


Figure 1. A fragment of a procedural net.

The procedural net is useful in specifying plans at the various levels of detail required by the user. The net is also used during planning to represent partially formed plans, so that the planner can be restarted during execution to modify an existing plan if new information requiring replanning is discovered as the assembly proceeds.

The system keeps an internal model of the "connectedness" of the different components of the air compressor as it is being assembled or disassembled. It also contains a graphical model of the locations of the different components of the air compressor. For example, the graphical model knows about the pump, motor, pressure switch, etc.; and the connectivity model knows about all the visible components.

### Interfacing to the real world: Vision and Dialogue

Vision is a difficult problem in the domain of electromechanical machinery because the equipment and its components usually have only a limited range of color and texture for a great variety of possible shapes. Shadows and highlights also depend on the stage of the assembly process. For these reasons, an approach has been taken that capitalizes on prior knowledge of visual appearances and geometric relations: There is a "tool recognizer" that accepts a limited semantic description of a tool, builds a model of the tool from this description, and uses this model to differentiate the tool from the set of probable alternatives.

There are two other visual modules, both of which rely on stored geometric models of the equipment. The first module allows the CBC to answer user questions of the form "Show me the X." It points at X with the laser rangefinder, which is accomplished by locating the outline of the visible surface of X. The second module allows the user to ask "What part is this?" while he is pointing at the unknown part with a wand equipped with a small light at the tip. A ray in space is defined by the wand tip and the camera lens. The ray intersects a geometric model of the equipment, providing the answer to the question.

The following is a portion of a dialogue between a novice mechanic and the CBC on the task "Assemble the air compressor." The example is taken from Hart, 1975.

CBC: Try to achieve assembled aircompressor.  
Novice: How?  
CBC: The top level plan is ... install pump ...  
install pumpbrace ... install pumppulley ...  
install belt ... install belthousingcover ...  
CBC: Ready for questions.  
Novice: Show me the pump.  
(Laser beam moves to point at pump.)  
CBC: Ready for questions.  
.  
.  
Novice: What is this? (Pointing with wand at tank  
platform)  
CBC: Tank platform.  
CBC: Ready for questions.  
Novice: None.  
CBC: Install pump.  
Novice: How?  
CBC: Connect pump to platform.  
Novice: How?  
CBC: Get 4 1&1/4-inch 5/16 hexagonal bolts and  
4 9/16 flat washers and 4 size 5/16 hexagonal  
nuts.  
Novice: OK.  
.  
.

### Summary

The main goal of the research on the SRI computer-based consultant was to create the fundamental technology needed to build such consultant systems. Much of this technology is independent of the particular consultant system being built or of the details of the task environment. Work on the electromechanical CBC described here was stopped in April, 1975. Since then a new consultant project has been begun at SRI extending many of the ideas from the 1975 system and adding new features.

The new consultant system is being used in two entirely different domains: mineral exploration (see article F2) and agricultural pest management. It appears that computer consultants are likely to remain both a fruitful area for Artificial Intelligence research and a practical means of deploying knowledge to people working in the specific task domains.

### References

See Hart (1975) and Nilsson (1975).

## B2. PROSPECTOR: An Expert System for Mineral Exploration

PROSPECTOR (Duda et al., 1978) is a computer-based consultation system that is being developed to aid geologists working on certain problems in hard-rock mineral exploration. Like other expert systems, such as INTERNIST (see article C2) and MYCIN (see article C1), PROSPECTOR attempts to represent a significant portion of the knowledge and the reasoning processes used by experts working in a specialized domain. The intended user of this program is an exploration geologist who is in the early stages of investigating an exploration site or prospect. We assume that he or she has a professional understanding of geology but, nevertheless, wants the assistance of a specialist in evaluating the findings.

In an attempt to keep the PROSPECTOR system fairly domain independent, the system maintains a clear separation between the geological knowledge base and the mechanisms that use this knowledge. Nevertheless, characteristics of the problem domain have certainly influenced the design of the system.

Expressed abstractly, the main function of PROSPECTOR is to match data from a particular situation against "models" that describe a moderately large number of disjoint classes of situations. In PROSPECTOR's domain, the models are formal descriptions of the most important types of ore deposits, and the data are primarily surface geological observations. The available data are assumed to be uncertain and incomplete, so that the conclusion is a probability or a degree of match. In addition, the program alerts the user to different possible interpretations of the data and identifies additional observations that would be most valuable for reaching a more definite conclusion.

A typical consultation session with PROSPECTOR begins with the user giving the system information about the most significant features of his prospect: the major rock types, minerals, and alteration products. The program matches these observations against its models and, when the user is finished volunteering information, proceeds to ask the user for additional information that will help confirm the best matching model. At any time during the consultation, the user can interrupt to volunteer new information, change previous statements, or request an evaluation. Thus, PROSPECTOR operates as a mixed-initiative system.

The following edited excerpt from a PROSPECTOR run illustrates many of these ideas. The data for the run describe a fictitious porphyry copper deposit that has some, but not all, of the desired characteristics. We begin at the point where the user starts volunteering information to the system. In this excerpt, the user's answers are always given in **emboldened upper case**, and comments are found in *italics*.

**1: THERE IS QUARTZ MONZONITE**

*QUARTZ MONZONITE (5)*

**2: THERE IS A CRETACEOUS DIKE**

*CRETACEOUS DIKE (5)*

**3: THERE IS SERICITE AND PYRITE**

*SERICITE (5)*

PYRITE (5)

5: THERE MAY BE BIOTITE

BIOTITE (2)

6: DONE

*These simple assertions describe the most prominent rocks, minerals, and alteration products in the target area. The system indicates its recognition of each statement by echoing the statement and appending an assigned certainty. Certainty factors range from -5 (positive that a situation does NOT hold) to 5 (positive that it does hold). All assertions made by PROSPECTOR have associated certainty factors. The program now enters the consequent mode of reasoning.*

Presently, the models in PROSPECTOR and their scores are:

- 1 Komatiitic nickel sulfide deposit -- KNSD (0.0)
- 2 Type-A porphyry copper deposit -- PCDA (0.0) **\*\*current hypothesis\*\***
- 3 Kuroko-type massive sulfide deposit -- MSD (.001999)
- 4 Western-states-type sandstone uranium deposit -- WSSU (0.0)
- 5 Mississippi-Valley-type lead/zinc deposit -- MVTD (-4.8)

Which of the above would you like to rule out? NONE

*The volunteered information was sufficient to rule out one of the models, but was not nearly specific enough to establish any of the others. The choice of the second model as the current hypothesis was based not on its certainty, but on the number and types of connections between it and the volunteered statements. To avoid a possibly fruitless investigation of unpromising models, the system gives the user the opportunity to exclude models of lesser interest. In this case, the user allows all of the models to be explored, and the system begins with the current hypothesis.*

I am considering the possibility of a  
( Type-A porphyry copper deposit -- PCDA )  
in the target area.

The following questions are intended to establish the nature of the petrotectonic setting.

1--To what degree do you believe that:  
( the target area is in a continental margin mobile belt ) ? -2

*This is a discouraging answer for the PCDA model, and it removes a context needed for the investigation of other properties of the continental margin mobile belt.*

The following questions are intended to determine the nature of the regional environment.

2--To what degree do you believe that:  
( there are granitic intrusives in the region ) ? 5

3--To what degree do you believe that:  
( the region contains an abundance of small stocks ) ? -3

8--To what degree do you believe that:  
( igneous rocks in the region have porphyritic texture ) ? WHY

I am trying to establish whether some of the intrusive rocks in your area have textures suggestive of a hypabyssal to subvolcanic environment. Porphyry textures resulting from the rapid cooling and crystallization of a silicate melt which contains large mineral grains are suggestive of relatively shallow levels of emplacement.

*Following MYCIN, PROSPECTOR provides a variety of commands for doing such things as requesting a summary, changing previous answers, or volunteering additional formation. The above response to a WHY command is not automatically generated from the rule but is merely stored text associated with the space.*

8--To what degree do you believe that:  
( igneous rocks in the region have porphyritic texture ) ? 5

My certainty in (the texture and morphology of the intrusive system suggest a subvolcanic (hypabyssal) regional environment ) is now: 4.924

Do you want to pursue this further? NO

*At this point the system noted that the certainty for the hypothesis about a hypabyssal environment was above 4.5, and it also noted that if the remaining rules were answered in the most unfavorable way possible, the certainty could not fall below 4.5. To save time, it let the user decide whether or not to continue working on this subgoal.*

*The questioning continues as above until it becomes necessary to consider properties that vary within the target area. Thus, for the first time it becomes necessary to be able to use*

*the rules more than once, and a communication convention must be established to distinguish different spatial locations.*

Porphyry copper deposits are often composed of characteristic zones of alteration and mineralization. (In some cases these can be looked upon as different stages superimposed upon one another.) I shall call the most prominent such zone Zone 1. Even though you may have already identified this zone as, say, the propylitic zone or the sericitic zone, I need to consider other possibilities as well. Thus, the following questions are intended to determine the possibilities for Zone 1.

25--To what degree do you believe that:  
( there are fresh-looking feldspars in Zone 1 ) ? -5

*Questioning continues in this manner. Eventually, PROSPECTOR establishes that two zones exist in the region of interest, and after examining each in detail, it concludes that their geologic makeup supports the hypothesis that Type-A porphyry copper exists in the region.*

On a scale from -5 to 5, my certainty that ( there are alteration zones that are favorable for a Type-A porphyry copper deposit ) is now 4.833

There are two favorable factors; in order of importance:

2.3.1: 1. there is a sericitic zone in Zone 1 (4.834)  
\*\* establishes 2.3.1 \*\*

2.3.1: 2. there is a potassic zone in Zone 2 (4.499)

There are many factors that would have been unfavorable, had no favorable factor existed to override them; in order of importance:

2.3.1: 3. there is a barren-core zone in Zone 1 (-5.0)

2.3.1: 4. there is a potassic zone in Zone 1 (-5.0)

2.3.1: 5. there is a propylitic zone in Zone 1 (-5.0)

2.3.1: 6. there is a sericitic zone in Zone 2 (-5.0)

2.3.1: 7. there is a propylitic zone in Zone 2 (-4.989)



2.3.1: 8. there is a barren-core zone in Zone 2 (-4.495)

For which of the above do you wish to see additional information?

### Knowledge representation

The network structure that is used to represent the the geological knowledge embodied in PROSPECTOR is called the *inference network*, which guides the plausible reasoning performed by the system. The nodes in this network correspond to various assertions, such as "There is pervasively biotized hornblende" or "There is alteration favorable for the potassic zone of a porphyry copper deposit." In a particular run, any assertion may be known to be true, known to be false, or suspected to be true with some probability.

Most of the arcs in the inference network define *inference rules* that specify how the probability of one assertion affects the probability of another assertion. For example, the presence of pervasively biotized hornblende suggests the potassic zone of a porphyry copper deposit, and the absence of any biotized hornblende is very discouraging for that conclusion. These inference rules correspond to the production rules used in MYCIN. The remaining arcs indicate that an assertion is the "context" for another assertion, preventing conclusions from being drawn until the right contexts are established. For example, one should establish that hornblende has been altered to biotite before asking about the degree of alteration.

The primary task confronting a geologist who wants to prepare a new model for PROSPECTOR is the representation of his or her model as an inference network. The current system contains models of five different types of deposits, developed in cooperation with five different consulting geologists. The following statistics give a rough indication of the size and complexity of these models.

Model	Number of Assertions	Number of Rules
Koroko-type massive sulfide	39	34
Mississippi-Valley-type lead/zinc	28	20
Type-A porphyry copper	187	91
Komatiitic nickel sulfide	75	49
Roll-front sandstone uranium	212	133
Total:	541	327

To allow certain kinds of logical reasoning by the system, we represent each assertion as a "space" in a partitioned semantic network (Hendrix, 1975a). A typical space asserts

the hypothetical existence of physical entities having specific properties (such as being composed of biotite) and participating in specific relations (such as an alteration relation). In addition, a large taxonomic network describes important element/subset relations among the terms mentioned, such as the fact that biotite is a mica, which in turn is a silicate, which in turn is a mineral.

The articulation of assertions as a set of relations allows the system to recognize subset/superset connections between pairs of assertions. For example, the assertion that "There is pervasively biotitized hornblende" is clearly related to the assertion that "There is mica"; assertion of the first also asserts the second, and denial of the second denies the first. This kind of recognition is used in two main ways. First, it provides important intermodel and intramodel connections beyond those given explicitly by the inference rules. Second, it allows the system to recognize connections between information volunteered by the user and the coded models.

#### Probabilistic reasoning

Some of the logical constraints that exist between spaces have probabilistic implications. In particular, if A is an instance of (subset of) B, then the probability of A can never exceed the probability of B. We maintain this constraint by automatically generating certain inference rules. For example, if evidence E could raise the probability of A above the probability of B, then we generate a rule from E to B that will increase the probability of B sufficiently to just satisfy the constraint. The exact procedure used here is described in Duda et al., 1977.

Since the various inference rules interconnect to form an inference network, when the user provides some evidence this information can change the probabilities of several hypotheses, which in turn can change the probabilities of hypotheses that depend upon them. The probability formulas determine exactly how these probability changes propagate through the inference net. (The reader might also refer to the handbook articles on IRIS and CASNET for other discussions of propagation.)

#### Control

As mentioned earlier, PROSPECTOR is a mixed-initiative system that begins by allowing the user to volunteer information about the prospect. This volunteered information is currently limited to simple statements in constrained English about the names, ages, and forms of the rocks and the types of minerals present. These statements are parsed by LIFER--a natural language interface facility developed by Hendrix (1977)--and represented as partitioned semantic networks. A network matching program compares each of these volunteered spaces against the spaces in the models, noting any subset, superset, or equality relations that occur.

If a volunteered space is exactly equal to a space in a model, the probability of the model space is updated, and that change is propagated through the inference network. If a volunteered space is a subset of a space in a model and if it has a higher probability than the model space, then once again the probability of the model space is updated, and that change is propagated through the inference network.

Unfortunately, if the volunteered space matches as a superset of a model space (which usually occurs), no probability change can be made unless the user expresses doubt about the situation. For example, if the user mentions biotite, the probability of the space that asserts that there is pervasively biotized hornblende is unchanged, unless the user has said that he or she doubts that there is any biotite. However, it is obvious that the system may want to follow up this observation, and the existence of the connection to the model is recorded.

When the user has finished the initial volunteering, PROSPECTOR scores the various models on the basis of the number and types of connections that have occurred and selects the best matching model for further investigation. Here the basic control strategy is MYCIN-like *backward chaining* or *consequent reasoning*. At any given time there is a current goal space whose existence is to be determined. The initial goal space is the one that corresponds to the best matching model. The various spaces in the models either represent evidence that can be sought from the user (are "askable") or internal hypotheses that are to be deduced from evidence (are "unaskable"). Naturally, the initial goal space is always unaskable. If the current goal space has any unestablished context spaces, they are pushed on the goal stack and one of them becomes the new current goal.

If the current goal is askable and has not been asked before, the user is asked about it; the effects of the answer are propagated through the inference network; and the process is repeated. If it is unaskable, it must be either the consequence of one or more inference rules or a logical combination of one or more other spaces. In the former case, the rules are scored to determine their potential effectiveness in influencing H, and the antecedent of the best scoring rule becomes the next goal. In the latter case a predetermined supporting space becomes the next goal. In either case the same procedure is repeated until either: (a) The top-level goal becomes so unlikely that another top-level goal is selected, (b) all of the askable spaces have been asked, or (c) the user interrupts with new volunteered information.

### Summary

This brief overview covers the basic knowledge representation and inference mechanisms used in PROSPECTOR. Many aspects of the system have not been mentioned, such as the treatment of quantitative evidence, the matching procedure, the use of graphical input, the inference network compiler, the explanation system, model acquisition aids, and the test and evaluation effort.

The five models in the current system are but a fraction of what is needed for comprehensive coverage, and even these models have only recently achieved the degree of completeness required for doing meaningful evaluations. Limited initial tests have shown very close agreement between the evaluations provided by the system and the evaluations of the model designers, using data from actual deposits of the types modeled. More information on the system, the extent of its geological knowledge, its performance on known deposits, and its possible applications can be found in Duda et al., 1978.

**References**

See Duda, Hart, & Nilsson (1976), Duda et al. (1977), Duda et al. (1978), Hendrix (1975a), Hendrix (1977), Pople, Myers, & Miller (1975), and ZADEH (1965).

### B3. RITA

Following a major trend in modern computing toward the wide distribution of small, inexpensive, powerful *personal computers* dedicated to single users, research at the RAND Corporation on systems and applications has produced RITA (RAND Intelligent Terminal Agent). Developed by Anderson and Gillogly at the Rand Corporation during the period 1976-1977 (Anderson, 1977), RITA is a *production system* patterned in some respects after the MYCIN medical diagnosis system (see article C1). RITA was designed to be a general-purpose system, capable of operating within a personal computer. A PDP-11 minicomputer is currently being used for the research.

RITA was designed to allow the user to develop powerful working programs incrementally, without having to master the intricacies of a procedure-oriented language.

Among the applications envisioned for RITA, two seemed especially promising: as *user agents* and *heuristic models*. *User agents* are programs aimed at performing routine, repetitive tasks, including automatic interacting with external information systems. The following are examples of the kinds of services that RITA-based user agents might provide:

- (a) automatically file copies of network mail, according to sender, receiver, or subject keywords;
- (b) periodically check for the receipt of new network mail or mail from a particular source and notify the user upon receipt;
- (c) transfer a file between two host computers on a network;
- (d) format a set of literature references to meet the requirements of a particular publisher or journal;
- (e) automatically query a remote database every 24 hours to see if new data has been filed that is specific to a given user's interest; or
- (f) check on-line appointment calendars of other personal computers to find a mutually convenient meeting time, schedule the meeting, and send announcements to attendees.

*Heuristic models* are programs aimed at making inferences about systems whose behavior can be described as a set of production rules and data objects. Heuristic modeling applications of RITA include TECA, which models naval tactical threats and is discussed in some detail below, and a model of terrorist confrontations and negotiations (Waterman & Jenkins, 1977).

#### The RITA System Architecture

Production system technology (see article Representation.B3) was used in RITA to provide features that its designers felt were crucial. In order to give users the ability to develop and use their own interface software, RITA was designed as a system sensitive to the following two requirements:

- (a) that user-developed software be capable of explaining its behavior upon request, and
- (b) that the user be able to incrementally modify the behavior of an agent.

The specifics of how a production system architecture can achieve these benefits are discussed in the articles on MYCIN and TEIRESIAS in this chapter (see C1 and C6).

All RITA applications are implemented as sets of user-specified production rules capable of operating upon data objects, files, and communications channels. These basic notions are illustrated and explained in the following paragraphs.

Data are stored within the RITA system as named "objects," which are collections of *attribute-value pairs*. The value of an attribute may be a scalar (a character string or number) or a list of values. Examples of RITA data objects are shown below with the RITA keywords capitalized.

OBJECT message: address-field IS "grm at rand-unix",  
status IS "unfiled";

OBJECT known-person: name IS "Gary Martins",  
address IS "grm",  
site IS "rand-unix",  
msg-file IS "m.accat";

RITA rules are stated in a formal, English-like language patterned after the output rule format used by MYCIN. The rules test or set attribute values within the database and may send or receive character strings from files or external processes. Rules may also test for patterns of general description within a character string; for example, the pattern specification

```
["date: " followed by 2 chars in "0123456789" 'month'
 followed by "/" followed by 2 chars in "0123456789"
 'day' followed by "/" followed by 2 chars in
 "0123456789" 'year']
```

would find the date field within the string

```
"time: 08:22:13 PST, date: 04/12/77, source: usc-lsib"
```

and set temporary pattern variables *month*, *day*, and *year* to the relevant substrings, for use later within the rule. An example of a RITA rule is shown below; several additional examples are contained in the discussion of applications of RITA at the end of this section.

```
RULE 5 IF: the state of the agent is "check response to 'show'"
 and the response of msg contains ["Message" followed
 by anything followed by " is discarded" followed by
 anything followed by the prompt of msg]
```

```
THEN: set the state of the agent to "get a message";
```

### Rule Application and Control

RITA's monitor is capable of applying rules in two modes: a cyclic and ordered mode, and a deductive, backward-chaining mode. The *cyclic scan* is the default; it tests rule condition-parts by cycling through the rules in a predetermined sequence. Whenever a true condition-part is found, the corresponding action-part is executed and the scan continues. The *ordered scan* operates similarly, with the exception that after a rule fires, the scan is restarted at the first rule in the set. In either case, the agent's operation is terminated if no applicable rules are found.

The backward-chaining deductive mode is similar to the reasoning method used in MYCIN (see article C1). It is initiated in RITA by execution of an action clause of the form "DEDUCE attribute OF object" during a cyclic or ordered scan. The action-part of each rule is scanned until a rule is found whose action sets the attribute value to that to be deduced. If its condition-part is true, the action is performed. If more data is required to determine the truth of the condition-part, then the missing attribute values are treated as subgoals and the search of the action-parts of rules is reinitiated for rules relevant to the achievement of these new subgoals. Upon completion of the deductive operation, the prior mode of operation (either the ordered or cyclic scan of rule condition-parts) continues.

### Performance of the System

The RITA system is currently being operated at a number of computer sites around the country. Diverse RITA-based user agents and heuristic models are under active development at these sites. Two such applications are discussed briefly below as illustrations of current RITA applications. Both are under development at the Naval Ocean Systems Center (NOSC) at Point Loma, California, as part of the Advanced Command and Control Architecture Testbed (ACCAT) program.

TECA (Threat Evaluation and Countermeasures Agent) is a heuristic model of the naval tactical threat situation. This model gathers updated information from several ARPANET data sources about the capabilities, positions, and movements of both friendly and hostile ships in order to produce a specification of the current tactical situation and to make simple recommendations regarding possible countermeasures.

TECA can interact with a user, guiding the analysis; or it can work in the background, freeing the user for other work. In either mode, TECA produces color video displays of current and projected tactical situation data, as well as textual output covering its own progress and recommendations.

The following is a sample TECA rule:

RULE compute-relative-threat-2:

```
IF   the state of the system is "compute relative threat"
    & there is a force(f) whose designation is
        "force of interest"
    & there is a force(tf) whose name is the
```

```

    threatening-force of force(f)
    & there is a platform(tp) whose name is in the
      platforms of force(tf)
    & whose name is not in the threat-plats-list
      of the system
    & whose raid-density is known

  THEN  set the attack-density of the system to
        the attack-density of the system + the raid-density
        of the platform(tp)
    & put the name of the platform(tp) into the
      threat-plats-list of the system;

```

The TECA model currently consists of about 250 rules. It is anticipated that the agent will have about 1,000 rules by the end of 1978, allowing the agent to handle situations such as: multidimensional threats, logistically constrained action radii, and projected courses of action.

NWSS (Navy WWMCCS Software Standardization) Man-machine Interface Agent, currently under development, has been designed to provide a flexible, modifiable interface to the NWSS database system (see article F4) on the WWMCCS (World-wide Military Command and Control System) computer network. RITA data objects are used to store information about the data available within the NWSS system. The user is queried by a set of RITA rules to determine his data needs, and these needs are then matched against the data available in NWSS. The RITA agent then formulates a valid query to the NWSS system and transmits it to the system. The response is presented to the user upon receipt.

The following is a sample NWSS Interface Agent rule:

**RULE determine-action-if-query-present:**

[This rule will ask the user if he wants to use an existing query if one is present. The answer expected is either Y or N.]

```

IF  the state of the agent is "build query"
    & the query-flag of the agent is "present"
    & there is a message whose category is a type of agent

```

```

THEN send query-present-portion of message to user
    & receive next (anything 'text' followed by
      1 char [carriage return])
    & set the answer of the user to uc('text')
    & set the type of the agent to the mode of the agent
    & send concat(" Value entered - ", answer of user)
      to user
    & set the state of the agent to "validate-reply";

```



The NWSS Interface Agent illustrates the primary use foreseen for RITA: the construction of interfaces tailored to external information systems that are too complex or broad in scope for the specific needs of a particular user. RITA is a prototype system meant to provide interfaces to the growing complex of systems available over interactive computer communication networks.

The research group at Rand has also used RITA as a basis for the exploration of *adaptive production systems*. These systems, in which RHS actions are used to modify the rule set itself (deleting existing rules, composing and instantiating new rules), offer significant advantages for the automated acquisition of knowledge on an incremental basis, through interaction with users and external systems. This work is described in Waterman (1977a, 1977b).

#### References

The purpose and design of the system is discussed in Anderson & Gillogly (1976a); a recent reference manual (Anderson, 1977) has also been published.

Also see Anderson et al. (1977), Anderson & Gillogly (1976b), Kernighan & Ritchie (1978), Ritchie & Thompson (1974), Waterman (1977a), Waterman (1977b), and Waterman & Jenkins (1977).

**B4. Artificial Intelligence in Information Retrieval****Motivation**

In dealing with complex databases such as those used by banks and airlines, the central problem faced by users is the formulation of queries comprehensible to the system. In order to make use of a database, a number of preliminary steps must be taken:

- (a) The user must decide what needs to be done, to satisfy his/her needs;
- (b) he must decide what information needs to be obtained from the database; and
- (c) he must translate these needs into queries that will be understood by the database management system.

Usually, business executives, government officials, and other decision makers have a good idea both of the kind of information they need to satisfy their operational needs and whether this information resides in their databases. Yet, to obtain the answer to a particular question, they generally need to employ the services of a technician who works with the database on a regular basis and who is thoroughly familiar with the details of the system: the file structure, the database management system (DBMS) on which it resides, the distribution among various computer systems, the coded field names for the data items, the kinds of values that different fields are expected to contain, and other idiosyncrasies.

As an example, let us consider a manager who would like to increment John Doe's salary by 10%. Clearly, this manager knows that the salary is stored in the company database and that, to satisfy his needs, he will have to find the salary of an employee named John Doe and update that salary. At this point the problem begins. Typically, all the salaries of employees--past and present--are kept in a file that does not include the name of the employee but rather his identification code, say, EMPID. The relationship between EMPID and his name, EMPNAM, would be found in another file. Then, the simple query "What is the salary of John Doe" needs to be decomposed into three steps:

- (a) FIND the EMPID of the employee with EMPNAM equal to 'JOHN DOE'.
- (b) FIND all salaries of the employee with that EMPID.
- (c) FIND the latest salary among those.

Once this sequence of steps has been defined, the last operation to be done is to express it in the query language of the particular database management system on hand. This procedure would produce something similar to the following:

```
FOR EMP IN EMP-FILE WITH EMPNAM EQ 'JOHN DOE'
BEGIN
  DECLARE X INTEGER X=0
  DECLARE Y INTEGER Y=0
  FOR SAL IN SAL-FILE WITH SALEMPID EQ EMP.EMPID
  IF X LT SAL.DATE THEN BEGIN
    X=SAL.DATE
    Y=SAL.SALARY
  END
```

END  
PRINT Y;

Translating the above back into English makes it clear that the steps above, currently performed by a technician, are not trivial.

A program designed to replace the technician would have to understand a decision maker's question, reformulate it in terms of the data that is actually stored, plan a sequence of requests for particular items from particular files on particular computers, open connections with remote sites, build programs to query the remote systems' DBMSs, monitor the execution of those programs, recover from any errors, and correlate the results.

With the goal of making databases directly available to decision makers, researchers in artificial intelligence have developed a number of prototype systems that, for many classes of questions, automate the procedures usually performed by technicians. In the following sections, we present several such systems concentrating on their distinguishing characteristics.

#### Current Systems

**LUNAR.** An early natural language retrieval system was Woods's LUNAR system (Woods, 1973; see article D). The LUNAR system introduced Augmented Transition Networks (ATNs) as a natural language parsing technique. The system was used to answer queries about the chemical analysis of lunar rocks.

**PLANES.** Woods's ATN formalism has been used in a variety of other systems. Waltz (Waltz, 1975) has devised a system called PLANES that answers questions about the maintenance and flight histories of airplanes. PLANES uses both an ATN and a *semantic grammar* (see article Natural Language.B3b). The latter is a grammar whose major components are not syntactic categories such as "subjects," "verbs," "noun-phrases," but semantic categories such as "planes," "pilots," "command-verbs," and so on. Using such a grammar, the sentence

Who pilots Air-Force-1?

would be analyzed as

<WHO> <PILOT-VERB> <PLANE>

and not as

<NOUN-PHRASE> <VERB> <NOUN-PHRASE> .

PLANES supports the processing of elliptical inputs; for example, if the user issues the question:

Who is the pilot of Air-Force-1?

and follows up with:

The navigator? ,

the system will correctly interpret the second question as:

Who is the navigator of Air-Force-1?

The PLANES system also supports clarification dialogues with users--i.e., it can ask for the user's help in disambiguating queries.



process without extensive use of syntactic data. Waltz is investigating ways of supplementing his case frames with nominal pieces of syntactic information.

LADDER. Another information retrieval system using a parser based on Woods's ATN technology is the LADDER system (Sacerdoti, 1977). It accesses a Naval database and was developed as a managerial aid to Navy decision makers. The parser used by LADDER is called LIFER Hendrix (1976a) and is based on a simplification of the ATN and case frame ideas.

In addition to using a semantic grammar, the LIFER system includes certain user-oriented features, such as spelling correction, processing elliptical inputs, and the ability to define and understand paraphrases during a query session.

Compared to PLANES, which has a parser based on *case frames* (see article Natural Language.B3b), LADDER uses a more syntactically oriented grammar. The LADDER parser could also support case frames, although this has not been attempted. The other main difference is that LADDER allows the user to increment the grammar with new patterns, which correspond to paraphrases of existing expressions. LADDER has also been extended to allow "macro paraphrases," so that a single sentence can now paraphrase a sequence of existing queries.

In LADDER, each semantic category corresponds to a portion of an eventual database query. After the sentence is parsed by LIFER, the query portions that correspond to the semantic categories found in the sentence are collected and given to a database interface component, IDA (Sagalowicz, 1977). As far as the user is concerned, the database is composed of a single file that contains all the fields appearing in the user question. IDA automatically builds the DBMS query program that takes into account the real file structure of the database. IDA uses heuristic techniques to automatically build this query program and, in particular, to provide the necessary "navigation" through the files and fields of the database.

Of all the projects presented here, LADDER is the only system to have strongly isolated the parser from the database. This segregation has required the development of heuristic techniques to build an efficient DBMS access request.

A number of systems have been developed without any specific domain application. These general-purpose DBMS access systems are discussed in the following paragraphs.

REL. Another applied natural language system with a design philosophy similar to LIFER is the REL (Rapidly Extendable Language) system of Thompson and Thompson (Thompson & Thompson, 1976). REL is a data retrieval system like LADDER and PLANES, but it requires that data be stored in a special REL database. The grammar rules of REL contain a context-free component and an argumentation component very much like those in LADDER. As its name implies, REL was intended to be easily extendable by interface builders. Much effort

has gone into making REL run rapidly. It is almost certainly faster than the other retrieval systems mentioned here. This speed was gained by an assembly language implementation, with the concomitant disadvantage that (new) DBMS access expressions cannot be written easily.

**ROBOT.** The Artificial Intelligence Corporation (Harris, 1977) introduced a commercial product for interfacing to databases called ROBOT. ROBOT is a system that maps English language queries into a language of database semantics that is independent of the contents of the database. The database itself is used as an extension of the dictionary and the structure of files within the database helps guide the parser when resolving ambiguities. In the example

WHICH NEW YORK EMPLOYEES LIVE IN BUFFALO? ,

the phrase "NEW YORK EMPLOYEES" is ambiguous. It could refer to employees, working in New York city, living in New York city, working for New York city, working in New York state, living in New York state, or working for New York state. The sentence itself seems to eliminate three of these six interpretations. By looking in the database, ROBOT finds that Buffalo is a city in the state of New York and makes the assumption that the only interpretation remaining is "employees working for the state of New York." It appears that it also needed to check that employees living in Buffalo could not work for the CITY of New York.

This approach is subject to several problems: The types of queries employed by users are heavily dependent on the content of the database, and any extensive recourse to a large database might greatly slow the parsing process. Furthermore, databases are coded largely in abbreviations that are unsuitable as lexical entries. Nevertheless, the notion of using the data itself to extend the capabilities of the language system is very attractive, although much work remains to be done to eliminate the obvious shortcomings.

**RENDEZVOUS.** The main function of RENDEZVOUS (Codd, 1974) is to reformulate the user query into a number of *canonical forms*. Using canonical words stored in a dictionary and numerous transformation rules, the system attempts to produce a query to issue to the relational DBMS. The transformation rules have direct access to the relational schema, and the structure of the database is used directly when parsing a user's request.

For example, consider the request,

List US carriers in the Mediterranean on 5/3/78.

First, RENDEZVOUS would use its first class of rules to rephrase the request as,

F US carriers in the Mediterranean on 5/3/78.

It will then translate "US carriers" to obtain:

F SHIP [NAM] WITH SHIP (NAT = US) AND SHIP (TYPE = CV)  
in the Mediterranean on 5/3/78

Next, it will detect that "Mediterranean" and "5/3/78" are ambiguous. Europeans typically write dates as day/month/year rather than month/day/year, and "Mediterranean" could be either a ship or a place name. To resolve the ambiguity, RENDEZVOUS engages in the following dialog:

By "Mediterranean", do you mean:

1. the ship named MEDITERRANEAN
2. the region of the world named Mediterranean
3. Other

Answer by 1,2 or 3... (Answer)2

By "5/3/78", do you mean:

1. the date May 3rd 1978
2. the date March 5th 1978
3. Other

Answer by 1,2 or 3... (Answer)1

Finally, the request becomes:

F SHIP [NAM] WITH SHIP (NAT = US) AND SHIP (TYPE = CV) AND  
POSIT ( REGION = 4 ) AND POSIT (POS-DATE = 050378)

where it is assumed that the database includes two files, a SHIP file containing static information about ships and a POSIT file containing the ship positions.

At this point, the request is in terms of the database. However, before handing it to the database for processing, the system rephrases the request in English as:

LIST THE NAMES OF SHIPS  
WHOSE NATIONALITY IS US  
AND WHO HAVE THE TYPE CV  
AND WHO WERE LOCATED IN REGION 4 ON MAY 03 1978.

and asks the user to confirm this interpretation. If the user doesn't agree with this rephrasing, the system would use a multiple choice selection system to let the user revise the query. Finally, RENDEZVOUS can fall back on this multiple choice selection process if the natural language processing fails completely. A first implementation currently exists but has only been tested on a very simple, small database.

In all of these systems, almost no artificial intelligence techniques are used except by the natural language parsers. Only the LADDER system appears to use any heuristic methods to plan the actual database query, and this feature was forced by the isolation of the parser from the database. In the projects presented in the next section, a great deal of emphasis has been placed on using artificial intelligence techniques in all aspects of database processing.

### Long-term Projects

A number of researchers are currently addressing the many long-range problems of accessing databases using natural language. These researchers include Mylopoulos and Roussopoulos (Mylopoulos & Roussopoulos, 1975); Sowa (Sowa, 1976); Walker, Erman, Newell, Nilsson, Paxton, Winograd, and Woods (Walker et al., 1977); and Sacerdoti et al. (Sacerdoti, 1977).

TORUS (Mylopoulos & Roussopoulos, 1975) is one such system. It provides access to

information about students and the educational process at the University of Toronto. This general information is contained in a semantic network. English queries are parsed into a semantic network, and functions are used to select the portion(s) of the semantic network that must be used as a response to the query.

The first implementation suffers from a number of limitations. Although TORUS was intended as a front end to an arbitrary DBMS, the first implementation encoded the database entirely in the semantic network and therefore never required access to the external database. The present scheme would require the entire database to be encoded into the semantic net, an immense and unnecessary task. Research currently in progress investigates this problem.

The intention is to distinguish two portions of the network: the upstairs portion, where only general semantic descriptions are contained, and the downstairs portion, where "examples" of the upstairs descriptions are stored. Supposedly, whenever a traversal of the upstairs/downstairs boundary was needed to answer a query, the database would be accessed.

Sacerdoti et al. (Sacerdoti, 1977) plan to extend their LADDER system. Among the many ideas they are exploring, one is to use a "partitioned semantic network" (see article Representation.B2) to represent the equivalent of the TORUS "upstairs," general knowledge about the information in the database. Partitions would be used to represent "theorems" that would help during the deductive question answering. One type of theorem might be used to indicate how to obtain relevant information from the database. An example of such a theorem to be used with the Naval database might be stated as follows:

- For every ship, there exists one record in the SHIP file where the field NAM is the name of the ship, the field NAT its nationality, the field CLASS its class, ...
- For every ship, there exists one record in the SHIPCLASS file where the field CLASNAM is the class of the ship, the field LGH its length, the field DFT its draft, ...
- For every ship, there exist records in the POSITION file where the field POSIT represents the position of the ship, POS-DATE the time where the ship was at that position, ...

A small demonstration of this capability has already been made using a very small semantic network.

### Conclusions

All the above systems attempt to provide the user with a relatively easy, direct access to databases. Most of the research has emphasized the natural language understanding abilities of the systems and the use of artificial intelligence techniques to understand requests. A few of these systems have also attempted to represent the knowledge



contained in the database using semantic nets. Practically no one has seriously studied the interface between such systems and the DBMS. Hopefully, in the future, some of the ideas being investigated currently by researchers in artificial intelligence will be integrated as part of a DBMS, rather than as later additions to these large systems. Much more investigation must be done to evaluate the special needs of the DBMSs and to find what artificial intelligence technologies are appropriate to satisfy these needs.

#### References

See Codd (1974), Harris (1977), Hendrix (1975a), Hendrix (1975b), Mylopoulos & Roussopoulos (1975), Sacerdoti (1977), Sagalowicz (1977), Sowa (1976), Thompson & Thompson (1975), Walker et al. (1977), Waitz (1975), and Woods (1973).

## References

- Anderson, R. H. The Use of production systems in RITA to construct personal computer *agents*. *Proceedings of the Workshop on Pattern-directed Inference Systems*, SIGART Newsletter (No. 63), 1977, pp. 23-28.
- Anderson, R. H., & Gillogly, J. J. *Rand Intelligent Terminal Agent (RITA): Design philosophy (R-1809-ARPA)*. Santa Monica, Ca.: The Rand Corp., February 1976. (a)
- Anderson, R. H., & Gillogly, J. J. The Rand Intelligent Terminal (RITA) as a network access aid. *AFIP Proceedings*, 1976, 45, 501-509. (b)
- Anderson, R. H., Gallegos, M., Gillogly, J. J., Greenberg, R., & Villanueva, R. *RITA Reference Manual (R-1808-ARPA)*. Santa Monica, Ca.: The Rand Corp., September 1977.
- Bennett, J. S., Creary, L. A., Engelmores, R. M., & Melosh, R. E. *SACON: A Knowledge-based consultant in Structural Analysis*, HPP 78-23, 1978.
- Bledsoe, W. W. Splitting and Reduction Heuristics in Automatic Theorem Proving. *Artificial Intelligence* 2, 1971, p. 73.
- Bledsoe, W. W. Splitting and Reduction Heuristics in Automatic Theorem Proving. *Artificial Intelligence* 2, 1971, p. 73.
- Brotz, D. *Embedding Heuristic Problem Solving Methods in a Mechanical Theorem Prover*, Stanford University, Computer Science Dept., Rep. No. STAN-CS-74-443, August 1974.
- Brotz, D. *Embedding Heuristic Problem Solving Methods in a Mechanical Theorem Prover*, Stanford University, Computer Science Dept., Rep. No. STAN-CS-74-443, August 1974.
- Brown, J. S., & Traub, J. F. On Euclid's Algorithm and the computation of polynomial greatest common divisors. *JACM*, 1971, 18(4), 505-514.
- Buchanan, B. G. *Applications of Artificial Intelligence to Scientific Reasoning*, Second USA-Japan Computer Conference. AFIPS and IPSJ, Tokyo, 1976, pp. 189-194.
- Codd, E. F. Seven Steps to Rendezvous with the Casual User. In J. W. Klimbie & K. I. Koffeman (Eds.), *Data Base Management*. New York: North Holland, 1974. Pp. 179-200.
- Davis, R. *Applications of Meta-Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases*, Stanford AI Lab Memo AIM-283, AI Lab, Stanford University, 1976. (a)
- Davis, R. Interactive transfer of expertise: Acquisition of new inference rules. *IJCAI* 5, 1977, 321-328.

- Davis, R., & Buchanan, B. Meta-level knowledge: Overview and Applications. *IJCAI 5*, 1977, 920-928.
- Davis, R., & Buchanan, B. Knowledge acquisition in rule-based systems: Knowledge about representations as a basis for system construction and maintenance. In D. Waterman & F. Hayes-Roth (Eds.), *Pattern-directed Inference Systems*. New York: Academic Press, 1978. Pp. 99-134.
- Duda, R. O., Gaschnig, J., Hart, P. E., Konolige, K., Reboh, R., Barrett, P., and Slocum, J. Development of the PROSPECTOR consultation system for mineral exploration. Final Report, SRI Projects 5821 and 6415. SRI International, Inc., Menlo Park, Calif., 1978.
- Duda, R. O., Hart, P. E., & Nilsson, N. J. Subjective Bayesian methods for rule-based inference systems. *AFIPS*, 1976, 45, 1075-1082.
- Duda, R. O., Hart, P. E., Nilsson, N. J., Reboh, R., Slocum, J., & Sutherland, G. L. Development of a computer-based consultant for mineral exploration. Annual Report, SRI Projects 5821 and 6415. SRI International, Inc., Menlo Park, Calif., 1977.
- Engelmore, R. S., & Feigenbaum, E. A. Inductive determination of the structure of proteins by automatic processing of electron density X-ray crystallographic data. Informal communications, Stanford University, 1976.
- Evans, M. A Program for the Solution of Geometric-Analogy Intelligence Test Questions. In M. Minsky (Ed.), *Semantic Information Processing*. Cambridge: MIT Press, 1968. Pp. 271-353.
- Evans, M. A Program for the Solution of Geometric-Analogy Intelligence Test Questions. In M. Minsky (Ed.), *Semantic Information Processing*. Cambridge: MIT Press, 1968. Pp. 271-353.
- Fahman, S. E. A System for representing and using real world knowledge. Doctoral dissertation, MIT AI Lab, September 1977.
- Fateman, R. J. *Essays in Algebraic Manipulation*, TR-95, MIT Computer Science Lab, April 1972.
- Fateman, R. J. An Approach to Automatic Asymptotic Expansions. *Proc. of an ACM Symposium on Symbolic and Algebraic Computation*, August 1976.
- Feigenbaum, E. A. The art of artificial intelligence: Themes and case studies in knowledge engineering. *IJCAI 5*, 1977, 1014-1029.
- Feigenbaum, E. A., Buchanan, B., & Lederberg, J. On Generality and Problem Solving: A Case Study Using the DENDRAL Program. In Meltzer & Michie (Eds.), *Machine Intelligence 6*. New York: American Elsevier, 1971. Pp. 165-190.
- Gelernter, H. Realization of a Geometry-Theorem Proving Machine. In E. A. Feigenbaum & Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963. Pp. 134-152.

- Gelernter, H. Realization of a Geometry-Theorem Proving Machine. In E. A. Feigenbaum & Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963. Pp. 134-152.
- Genesereth, M. R. DB: A high level data base system with inference, Memo 4, MACSYMA Group, MIT, December 1976.
- Genesereth, M. R. The difficulties of using MACSYMA and the function of user aids. *Proc. of the 1st MACSYMA Users' Conference*, NASA Report CP-2012, July 1977.
- Genesereth, M. R. *Automated Consultation for Complex Computer Systems*. Doctoral dissertation, Harvard University, September 1978.
- Gosper, R. W. Indefinite hypergeometric sums in MACSYMA. *Proc. of the 1st MACSYMA Users' Conference*, NASA Report CP-2012, July 1977.
- Grossman, R. *Some Data Base Applications of Constraint Expressions*, TR-158, MIT Computer Science Lab, February 1976.
- Hadamard, J. *The Psychology of Invention in the Mathematical Field*. New York: Dover, 1945.
- Hadamard, J. *The Psychology of Invention in the Mathematical Field*. New York: Dover, 1945.
- Harris, L. R. ROBOT: A high performance natural language processor for data base query. *SIGART Newsletter* (No. 61), Feb. 1977, pp. 39-40.
- Hart, P. E. Progress on a Computer Based Consultant. *IJCAI 4*, 1975, 831-841.
- Hearn, A. REDUCE 2 User's Manual, Stanford AI Project Memo AI-90, May 1969.
- Hearn, A. REDUCE 2: A system and language for algebraic manipulation. *Proceedings of the 2nd Symposium on Symbolic and Algebraic Manipulation*, March 1971.
- Hearn, A. C. REDUCE-2 Users' Manual, Univ. of Utah Computational Physics Group Report No. UCP-19, March 1973.
- Heiser, J. A computerized Psychopharmacology Advisor. HEAD-MED Report in the SUMEX Annual Report. Computer Science Dept., Stanford University, 1977-1978.
- Hempel, G. *Fundamentals of Concept Formation in Empirical Science*. Chicago: University of Chicago Press, 1952.
- Hendrix, G. G. Expanding the utility of semantic networks through partitioning. *IJCAI 4*, 1975, 115-121. (a)
- Hendrix, G. G. Expanding the utility of semantic networks through partitioning. *IJCAI 4*, 1975, 115-121. (a)
- Hendrix, G. G. Expanding the Utility of Semantic Networks through Partitioning, SRI Artificial Intelligence Group Technical Note 106, June 1975. (b)

- Hendrix, G. G. Human engineering for applied natural language processing. *IJCAI* 5, 1977, 183-191.
- Kernighan, B. W., & Ritchie, D. M. *The C Programming Language*. New Jersey: Prentice Hall, 1978.
- Kling, R. Reasoning by Analogy with Applications to Heuristic Problem Solving: A Case Study, Memo AIM-147, CS Rep. CS-216, Stanford University, August 1971.
- Kling, R. Reasoning by Analogy with Applications to Heuristic Problem Solving: A Case Study, Memo AIM-147, CS Rep. CS-216, Stanford University, August 1971.
- Knuth, D. *Surreal Numbers*. Reading, Mass.: Addison-Wesley, 1974.
- Knuth, D. *Surreal Numbers*. Reading, Mass.: Addison-Wesley, 1974.
- Koestler, A. *The Act of Creation*. New York: Dell, 1967.
- Koestler, A. *The Act of Creation*. New York: Dell, 1967.
- Laing, R. D. Rules and Metarules. In R. D. Laing (Ed.), *The Politics of the Family and Other Essays*. New York: Vintage Books, 1971. Pp. 103-116.
- Lakatos, I. *Proofs and Refutations: The Logic of Mathematical Discovery*. London: Cambridge Univ. Press, 1976.
- Lakatos, I. *Proofs and Refutations: The Logic of Mathematical Discovery*. London: Cambridge Univ. Press, 1976.
- Lenat, D. B. AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search, SAIL AIM-286, AI Lab, Stanford University, July, 1976. (Also issued as Comp. Sci. Rep. No. STAN-CS-76-570.)
- Lewis, V. E. User aids for MACSYMA. Proc. of the 1st MACSYMA Users' Conference, NASA Report CP-2012, July 1977.
- Mathlab Group *MACSYMA Reference Manual*, MIT Computer Science Lab, December 1977.
- Minsky, M. Frames. In P. Winston (Ed.), *The Psychology of Computer Vision*. New York: McGraw-Hill, 1975.
- Moses, J. Symbolic integration: The Stormy Decade. *CACM*, 1971, 14(8), 548-560.
- Moses, J. A MACSYMA Primer, Mathlab Memo No. 2, MIT Computer Science Lab, October 1975.
- Moses, J., & Yun, D. Y. The EZGCD Algorithm. Proc. of the ACM National Convention, August 1973.
- Musser, D. R. Multivariate polynomial factoring. *JACM*, 1975, 22(2), 291-307.

- Mylopoulos, J. & Roussopoulos, N. Using Semantic Networks for Data Base Management. Proc. International Conf. on Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 144-172.
- Nilsson, N. (Ed.) Artificial Intelligence--Research and Applications. Stanford Research Institute, Inc., Menlo Park, Calif., 1975.
- Norman, A. C. On Computing with Formal Power Series. Transactions on Mathematical Software (ACM), 1975, 1(4), 346-356.
- Papert, S. Teaching Children to be Mathematicians Versus Teaching About Mathematics. Intl. Jour. Math Ed. Sci. Tech. 3, July-Sept. 1972, No. 3, pp. 249-262.
- Papert, S. Teaching Children to be Mathematicians Versus Teaching About Mathematics. Intl. Jour. Math Ed. Sci. Tech. 3, July-Sept. 1972, No. 3, pp. 249-262.
- Pivar, M., & Finkelstein, M. Automation, using LISP, of Inductive Inference on Sequences. In E. C. Berkeley & D. G. Bobrow (Eds.), The Programming Language LISP: Its Operation and Applications. Cambridge: Information International, 1964.
- Poincare, H. The Foundations of Science: Science and Hypothesis, The Value of Science, Science and Method. New York: The Science Press, 1929.
- Poincare, H. The Foundations of Science: Science and Hypothesis, The Value of Science, Science and Method. New York: The Science Press, 1929.
- Polya, G. Mathematics and Plausible Reasoning (2 vols.). New York: John Wiley & Sons, 1954.
- Polya, G. Mathematics and Plausible Reasoning (2 vols.). New York: John Wiley & Sons, 1964.
- Pople, H. E., Jr., Myers, J. D., & Miller, R. A. DIALOG: A model of diagnostic logic for internal medicine. IJCAI 4, 1975, 848-855.
- Risch, R. The Problem of Integration in Finite Terms. Trans. of the AMS, May 1969, 139.
- Ritchie, D. M., & Thompson, K. The UNIX time-sharing system. Communications of the ACM, 1974, 17, 365-375.
- Rothstein, M. A New Algorithm for the Integration of Exponential and Logarithmic Functions. Proc. of the 1st MACSYMA Users' Conference, NASA Report CP-2012, July 1977.
- Sacerdoti, E. D. Language Access to Distributed Data With Error Recovery, SRI Artificial Intelligence Center Technical Note 140, February 1977.
- Sagalowicz, D. IDA: An intelligent data access program, SRI Artificial Intelligence Center Tech. Note 145, SRI International, Inc., Menlo Park, Calif., June 1977.
- Simon, H. Does Scientific Discovery Have a Logic? Philosophy of Science, 1973, 40(4), 471-480.

- Simon, H., & Kotovsky, K. Human Acquisition of Concepts for Sequential Patterns. *Psychology Review* 70, 1963, No. 6, 534-546.
- Sowa, J. F. Conceptual Graphs for a Data Base Interface. *IBM Journal of Research and Development*, 1976, 20(4), 336-357.
- Thompson, F. B., & Thompson, B. H. Practical Natural Language Processing: the REL System as Prototype. In M. Rubinfoff & M. C. Yovits (Eds.), *Advances in Computers* 13. New York: Academic Press, 1975. Pp. 109-168.
- Trager, B. M. *Integration of Algebraic Functions*. Doctoral dissertation, MIT Computer Science Lab, (in progress) 1978.
- Walker, D. E., Erman, L. D., Newell, A., Nilsson, N. J., Paxton, W. H., Winograd, T., & Woods, W. A. An Overview of Speech Understanding Research at SRI. *IJCAI* 5, 1977, 970-974.
- Waltz, D. L. Natural language access to a large database: An engineering approach. *IJCAI* 4, 1975, 868-872.
- Wang, H. Toward Mechanical Mathematics. *IBM Jour. Research and Development* 4, 1960, 1, 2-22.
- Wang, H. Toward Mechanical Mathematics. *IBM Jour. Research and Development* 4, 1960, 1, 2-22.
- Wang, P., & Rothschild, L. Factoring Multivariate Polynomials over the Integers. *Math. of Comp.*, July 1975, 29, 935-950.
- Waterman, D. A. Exemplary programming in RITA. In D. Waterman & F. Hayes-Roth (Eds.), *Pattern-directed Inference Systems*. New York: Academic Press, 1977. Pp. 261-279. (a)
- Waterman, D. A. *Rule-directed Interactive Transaction Agents: An Approach to Knowledge Acquisition (R-2171-ARPA)*. Santa Monica, Ca.: The Rand Corp., 1977. (b)
- Waterman, D. A., & Jenkins, B. *Heuristic Modeling Using Rule-based Computer Systems (P-5811)*. Santa Monica, Ca.: The Rand Corp., 1977.
- Winston, P. H. *Learning Structural Descriptions from Examples*, EE TR-76, Project MAC TR-231, MIT AI Lab, September, 1970. (b)
- Woods, W. A. Progress in natural language understanding, An application to lunar geology. *Proc. 1973 NCC, AFIPS*, 1973, 42, 441-450.
- Zadeh, L. A. Fuzzy sets. *Information and Control*, 1965, 8, 338-353.
- Zippel, R. Univariate Power Series Expansions in Algebraic Manipulation. *Proc. of an ACM Symposium on Symbolic and Algebraic Computation*, August 1976.

## Index

- Advisor, consultation system 465  
 agenda 468, 474  
 agricultural pest management system 486  
 algebraic manipulation research 462  
 algebraic problems 462  
 AND/OR tree 452, 455  
 Anderson, R. H. 495  
 askable hypotheses 493  
 ATNs 501, 503  
 ATNs, augmented transition networks 501  
 attribute-value pairs 496  
 augmented transition networks 501
- back-chaining 447  
 backward chaining 454, 493, 497  
 backward-chaining 449  
 backward-chaining reasoning 497  
 Barton, David 466  
 best-first search 468
- canonical forms 504  
 case frame 502  
 case frames 502, 503  
 clarification dialogues 501  
 cognitive abilities 445  
 combinatorial explosion 444, 472  
 combinatorics 464  
 computer assistants 1  
 concept, AM 468, 472  
 concepts, creation 473  
 conceptual primitives 471  
 conceptual primitives, AM 471, 473  
 consequent reasoning 493  
 constraint 465  
 constraint expressions 465  
 consultants 445  
 context-free grammar 503  
 CPM, limited inference algorithm 464  
 cyclic reasoning 497  
 cyclic scan 497
- database queries 500  
 Davis, Randall 449  
 DBMS, Database Management System 500  
 deductive mode 497  
 DENDRAL 444, 446  
 dialogue 1  
 discourse model 482  
 discovery, heuristics 468  
 discovery, mathematical concepts 468  
 domain independence 446  
 domain-independent constraint  
   propagation 465  
 domain-specific knowledge 1-449
- elliptical reference 502  
 empirical knowledge 1, 445, 447  
 EMYCIN 446  
 Engleman, Carl 462  
 EURISKO 481  
 exhaustive search 470  
 expert systems 1-449  
 expert systems, history of 444  
 explanation 1, 446-447, 450, 451-452,  
   456, 487  
 explanation procedures 1  
 explanation system 444
- Feigenbaum, E. 444  
 focus of attention 475  
 focus of attention, AM 475  
 formal deduction 469  
 formal domains 469  
 forward search 472  
 frame 451  
 frame-oriented interactive primer 465  
 frames 468  
 frames, AM 472
- generalization, concept 475  
 geological data models 487  
 geology mineral exploration system 487



Gillogly, J. J. 495  
goal tree 452, 455  
GPS 444  
grain size 445  
graph of model 482  
Grossman, R. 465

**HEADMED 1**

heritability property 475  
heuristic models 495  
Heuristic Programming Project, HPP 444  
heuristic rule 472  
heuristic search 444-445, 468, 470  
heuristics 468-482  
hierarchy of procedural knowledge 483  
hillclimbing 463  
hypotheses, askable 493  
human engineering 1  
hypotheses, unaskable 493

IDA, database interface component: 503  
impacts, economic and social 444  
incremental knowledge acquisition 499  
inductive inference 469  
inexact reasoning 1-449  
inference network 491  
inference rules 491  
Information Retrieval 500  
inheritance of properties 475  
interactive dialogue 487  
interactive program 497  
interactive transfer of expertise 445  
interactive tutoring systems 1  
INTERNIST 447  
IRIS 446

knowledge acquisition 1, 445, 447-449,  
452, 457  
knowledge base 445-449  
knowledge-based programming 462  
knowledge-based systems 1-449

LADDER 503  
laser vision component 482  
Lederberg, J. 444  
LIFER, NL interface facility 492, 503  
LUNAR 501

MACSYMA 444, 446, 462  
MACSYMA apprentice 466  
man-machine interactions 445  
man/machine interaction, MACSYMA 465  
Martin, Bill 462  
mathematical problems 462  
mathematical programs, advanced 1  
Mathlab '68 462  
medical consultant systems 445  
medical diagnosis systems 1  
Meta-DENDRAL 447  
meta-knowledge 448, 450, 452  
meta-level knowledge 448  
meta-rule 449, 453  
meta-rules, AM 480  
mineral exploration system 486  
minicomputer 495  
mixed-initiative knowledge acquisition 487  
mixed-initiative system 487, 492  
modeling inference networks 491  
models, evidence 493  
models, internal hypothesis 493  
modular inference mechanism 487  
modular inference mechanisms 503  
modular knowledge 487  
modular knowledge representation 447  
modular knowledge 503  
Moses, Joel 462  
multiple choice processing 505  
MYCIN 446-449, 452, 453, 496

natural language interface 482  
natural language, AM 476  
naval tactical threat system 497  
Non-algorithmic procedures 463  
NOSC, Naval Ocean Systems Center 497

- numerical problems 462
- NWSS, Navy Man-machine Interface Agent 498
- opacity of knowledge 446
- ordered scan 497
- parallel processing 465
- partitioned semantic net 491
- pattern matching 487, 496
- personal computer 495
- PIP 447
- PLANES, airplane maintenance and history system 501-503
- PLANNER 444
- planning capability 482
- planning component 482
- planning program 483
- plans, MACSYMA 467
- plausibility, of tasks in AM 475
- plausibility, task 475
- plausible move generator 472
- plausible reasoning, AM 469
- power/generality trade-off 476
- probabilistic reasoning 487, 492
- problem solving 1
- problem-solving systems 444
- procedural model 483
- procedural net 482, 483-484
- production rule 449
- production rules 447, 496
- production system 495
- production systems 468
- production systems, adaptive 499
- propagation 465
- propagation, of probabilistic hypotheses 492
- PROSPECTOR 444, 446, 487
- PROSPECTOR trace 487-491
- PUFF 1
- QA4 444
- real-world problem 444
- reasoning procedures 1
- REF-ARF 444
- REL, Rapidly Extendable Language 503
- relational database 464
- RENDEZVOUS, reformulation of user input 504
- RENDEZVOUS, reformulation of user input 505
- rephrasing user input 505
- representation of knowledge 1-449
- representation, algebraic expressions 466
- resolution theorem proving 469
- RITA data object 496
- RITA rules 496
- RITA, RAND Intelligent Terminal Agent 495
- ROBOT, parser interface 504
- rule model 452, 457
- rules 447
- SACON 1
- scaling 448
- schedulers 475
- schema 451, 452
- scoring models 487
- SCSIMP, hillclimbing algorithm 463
- search 444
- search skeleton 502
- semantic grammar 492, 501
- semantic network 464
- semantic pattern matcher 463
- simplification 463
- simplification of expressions 462, 463
- situation models 487
- slot, AM 468
- slots, AM 472
- spaces, in partitioned semantic nets 491
- specialization, concept 475
- speech interface 482

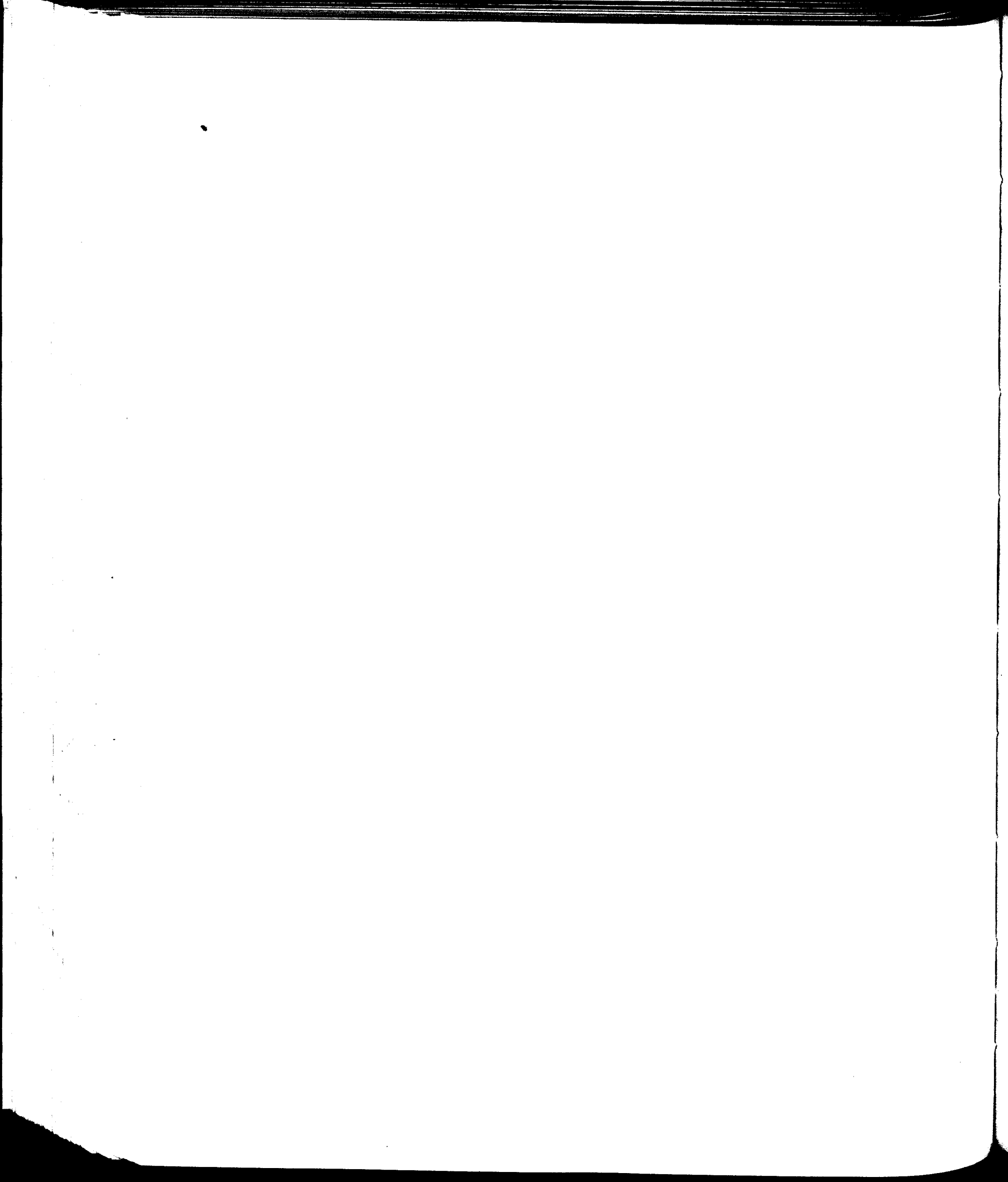
SRI Computer-based Consultant, SRI-  
CBC 482  
SRI-CBC sample dialogue 485  
SRI-CBC, air compressor assembly  
system 482  
strategy 449, 453  
symbolic algorithms 463  
symbolic reasoning 444, 445  
SYNCHEM 448  
SYNTHESIS 444

task environment 482  
taxonomic net 491  
TECA model 498  
TECA rule 497  
TECA, Threat Evaluation and  
Countermeasures Agent 497  
TEIRESIAS 447, 449-462, 496  
theorem representation 506  
TORUS, DBMS front end 505  
transfer of expertise 449  
tree of vectors 466  
type-testing 466

unaskable hypotheses 493  
user agents 495  
user model 482

vision 485  
vision, electromechanical domains 485  
vision, geometric models 485  
vision, semantic descriptions 485

WWMCCS 498



# Applications-oriented AI Research

## Part 2: Chemistry

# AI Applications in Chemistry

## Table of Contents

A. Applications of Artificial Intelligence to Chemistry . . . . .	521
1. Overview of Applications of Artificial Intelligence to Chemistry . . . . .	521
2. Chemical Analysis . . . . .	524
3. The DENDRAL Programs . . . . .	527
a. DENDRAL . . . . .	527
b. CONGEN and its Extensions . . . . .	531
c. Meta-DENDRAL . . . . .	536
4. CRYVALIS . . . . .	545
5. Artificial Intelligence and Organic Synthesis . . . . .	555
References . . . . .	565
Index . . . . .	570

## A. Applications of Artificial Intelligence to Chemistry

### A1. Overview of Applications of Artificial Intelligence to Chemistry

Computer programs have been developed to aid in almost every aspect of chemistry. As evidenced by recent articles in two journals devoted to uses of computers for chemical problems, *Computers and Chemistry* and *Journal of Chemical Information and Computer Science*, most of the computer programs have focused on numeric problems of data acquisition, data reduction, complex electronic energy calculations, and the like. By contrast, AI methods have found application in two major classes of nonnumeric chemical reasoning problems: (a) determining the molecular structure of an unknown organic compound, the "analysis" or "structure determination" problems; and (b) planning a sequence of reactions in order to synthesize organic chemical compounds, the "synthesis" problems. These problems are difficult for chemists, but not impossible. Solving them, however, requires more than a straightforward application of an algorithm to some data, since both the requisite knowledge about organic chemistry and the criteria for when to apply it are judgmental. Here, AI techniques are used to reduce combinatorially intractable hypothesis spaces (e.g., pruning an enormous set of candidate chemical structures or reactions to achieve a smaller set of plausible ones) and to represent knowledge in a way that makes its augmentation and revision simple.

### Representing Chemical Information Inside a Computer

Even before AI found an application in chemistry, computer scientists and chemists used a common tool: The graph. One insight that made chemical analysis and synthesis programs possible was that the "ball and stick" model used to represent molecules by some chemists was similar to the "nodes and arcs" model used by computer scientists in many applications (although the former is three-dimensional and the latter is two-dimensional). As can be seen in any textbook, many of the important structural features of a chemical compound can be represented by a planar graph: Using the terminology of graph theory, the molecular structure is represented as a bidirectional, planar graph where the edges represent chemical bonds and the nodes are labeled to represent the different types of chemical atoms. Thus, any convenient computer notation that captures these features will suffice for manipulating chemical structures in a computer--although some internal representations will be better for some purposes. Along with these representations, a number of graph manipulation methods have been developed for manual and machine use; but these are beyond the scope of the present discussion. Among other things, these include defining canonical forms for chemical graphs and methods for computing them efficiently, defining efficient graph isomorphism comparison methods, and defining graph enumeration methods.

The standard graphical representation of ethyl alcohol, or ethanol, is shown in Figure 1. The corresponding tabular representation of the same structure shown in Table 1 can be used to represent the same molecule inside a computer. (Double or triple bonds are easily represented by repeating the number of the connected node in the listing neighbors.) There are many internal representations of the information in this table: lists, alphanumeric arrays, record structures, or property lists. For example, the molecular structure can be

represented as a list of lists, where each sublist captures the information about one of the atoms--i.e., the sublist (2 C (1 3 7 8)) from row two of Table 1 captures the information about the second atom. However, this representation gives us only topological information about the molecule--the connections between atoms. It does not give us information about the lengths of edges (bond lengths), the angles between edges (bond angles), or the relative orientation of atoms in three-space (stereochemical information).

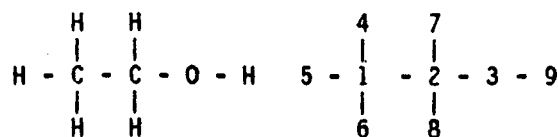


Figure 1. Graphical representation of ethyl alcohol with arbitrary numbering of nodes.

Table 1  
Connection Matrix for Ethyl Alcohol

Node Number	Atom Type	Neighbors
1	C	2,4,5,6
2	C	1,3,7,8
3	O	2,9
4	H	1
5	H	1
6	H	1
7	H	2
8	H	2
9	H	3

Storage requirements have been cut dramatically by following the simple convention of suppressing explicit mention of all hydrogen atoms, since they are so common in organic compounds. The number of hydrogen atoms attached to any other atom is always computed as the difference between the maximum number of allowed neighbors or atom valence and the number of nonhydrogen neighbors. Table 1 is thus simplified to the connection matrix shown in Table 2.

Table 2  
Connection Matrix for Ethyl Alcohol  
Without Hydrogen Atoms

Node	Type	Neighbors
1	C	2
2	C	1,3
3	O	2



### The Difficulty and size of Chemistry Problems

The combinatorics of chemical graphs is awesome. The number of possible ways of connecting 20 chemical atoms within valence constraints (say, 16 carbons, 2 oxygens, and 2 nitrogens) is in the millions or tens of millions. Pharmaceutical chemists, among others, often investigate much larger molecules.

Whether the problem is analyzing structures or combining structures with synthetic reactions, the combinatorial possibilities are greater than chemists can explore manually. Moreover, the knowledge base of chemistry is growing rapidly, as noted by Wipke [Ref Wipke-Ouchi-Krishnan-78]:

Over 250,000 chemical papers appear annually in the chemical literature reporting new facts and principles and according to Chemical Abstracts, the chemical literature is increasing at 8.5% per year compounded. Over 4,000,000 different chemical compounds have been reported in the literature. As in all fields of science it is virtually impossible for a single person to keep track of all the developments taking place, even in this seemingly narrow field of specialization.

New instruments are constantly under development for measuring properties of unknown compounds, and chemists are constantly seeking new methods of synthesizing compounds in the effort to produce them more economically (or, in some cases, to produce them at all).

The complexity of both synthesis and analysis problems varies from those whose solutions are immediately obvious, to a trained chemist, to those whose solutions require many years of innovative work by the world's best chemists. As mentioned, the problems are far from trivial because the requisite knowledge is both vast and incomplete and because the strategies for applying this knowledge are less than precise. In spite of these limitations, chemists have been highly successful in their problem solving, even without computers. One of the AI goals of the work described in this section is to understand the kinds of knowledge mechanisms by which chemists are able to arrive at plausible hypotheses without exhaustively considering complete hypothesis spaces.

We have noted that AI techniques have been useful in determining the structure of unknown chemical compounds (analysis), and specifying a set of steps to synthesize organic compounds (synthesis). The next \*N\* articles discuss analysis programs.

### References

## A2. Chemical Analysis

### Structure Elucidation

The elucidation of molecular structures is fundamental to the application of chemical knowledge to important problems in biology and medicine. Some of the areas in which chemists maintain active interest include: (a) identification of naturally occurring chemical compounds isolated from terrestrial or marine organisms; (b) verification of the identity of new synthetic materials; (c) identification of drugs and their metabolites in clinical studies; and (d) detection of metabolic disorders of genetic, developmental, toxic, or infectious origins through the identification of organic constituents excreted in abnormal quantities in human body fluids.

In most circumstances, especially in the areas of interest mentioned above, the powerful technique of xray crystallography is inapplicable (see article ), and chemists must resort to structure elucidation based on data obtained from a variety of other methods. Foremost among them is mass spectrometry (discussed in detail in the next section). If a chemist wants to determine the molecular structure of an unknown chemical compound, s/he first isolates a sample of the compound that is pure--i.e., contains no other compounds. Two questions must be answered:

1. What are the atoms in the molecules that make up the compound?
2. How are the atoms arranged (joined together) in the molecule?

The latter is addressed by structure elucidation programs. It is relatively simple to determine what the constituents of the molecule are (the first question), but the enormous number of possible combinations of these constituents makes the second question especially hard to answer. If the unknown substance is a crystal, or can be crystalized, then xray crystallography can be used to determine the exact locations and connections of atoms in a molecule in space. If this technique cannot be used, then the chemist must take a more complicated approach to structure elucidation. No other tests are available to tell the chemist the EXACT structure of his molecule; at best he can identify two groups of small, connected clusters of atoms, called molecular fragments. The first group of molecular fragments contains those that are guaranteed to occur in the unknown molecule. Thus, although the chemist does not know the structure of his molecule, he does know some of the subparts. The second group of molecular fragments is comprised of fragments that cannot occur in the unknown molecule. The fragments in both of these groups are called constraints, and they are derived from a variety of tests, including mass spectrometry. Said differently, a constraint is a piece of a graph that must or must not occur in the final graph of the molecule, which is how constraints are represented in the structure elucidation programs that we will discuss.

A group of atoms can be combined into literally millions of different molecular structures. An algorithm was developed by Lederberg (1964) to generate all possible acyclic molecules from a set of atoms; and Brown et al. developed an algorithm without this limitation. Thus it is possible to generate every possible molecular structure, but enormously expensive. However, given a set of constraints--molecular fragments guaranteed to occur in

or be absent from the unknown molecule--the exhaustive generation algorithm can be constrained to produce a relatively small set of molecular structures, one of which is very likely to be the unknown molecule. (There is no guarantee that the set will contain the unknown molecule, because of the heuristic nature of the reasoning that derives constraints from mass spectrometry tests.)

In the past, if the number of atoms in an unknown molecule was relatively small and the number of known constraints was relatively large, a chemist could figure out the molecular structure by hand. However, the manual approach has been successfully augmented by computer programs developed in the DENDRAL project at Stanford University. These programs either infer or are told of a set of constraints and the atomic constituents of the molecule. They use this information to constrain the set of molecular structures that is generated by Lederberg's (or more recently Brown's) algorithm. They do not generate all the possible molecular structures and then discard some according to the constraints; rather, they use the constraints to insure that only some structures get generated.

### Structure Elucidation with Constraints from Mass Spectrometry

Structure elucidation programs are designed to help organic chemists determine the molecular structure of unknown compounds. Experimental data from the unknown may be gathered from many different analytic techniques including mass spectrometry (MS), nuclear magnetic resonance spectroscopy (NMR), infrared spectroscopy (IR), ultraviolet spectroscopy (UV), and "wet chemistry" analysis. Mass spectrometry is still new and a developing technique. It is particularly useful when the quantity of the sample to be identified is very small, for mass spectrometry requires only micrograms of sample.

A mass spectrometer bombards the chemical sample with electrons, causing *fragmentations* and rearrangements of the molecules. Charged fragments are collected by mass. The data from the instrument, recorded in a histogram known as a mass spectrum, show the masses of charged fragments plotted against the relative abundance of the fragments at a given mass. Although the mass spectrum for each molecule may be nearly unique, it is still a difficult task to infer the molecular structure from the 100-300 data points in the mass spectrum, because not only does a spectrum contain "noise peaks" and overlapping peaks originating from many parts of the molecule, but the theory of mass spectrometry is not complete.

Throughout this section the following terms will be used to describe the actions of molecules in the mass spectrometer:

**Fragmentation**--the breaking of a connected graph (molecule) into fragments by breaking one or more edges (bonds) within the graph.

**Atom migration**--the detachment of nodes (atoms) from one fragment and their reattachment to other fragments. This process alters the masses of all of the fragments.

**Mass spectral process**--a fragmentation followed by zero or more atom migrations.

Other analytic techniques are commonly used in conjunction with, or instead of, mass spectrometry. Some rudimentary capabilities exist in structure elucidation programs to interpret proton NMR and Carbon 13 ( $^{13}\text{C}$ ) NMR spectra. For the most part, however, interpretation of other spectroscopic and chemical data has been left to the chemist. The programs still need the capability to integrate the chemist's partial knowledge into the generation of structural alternatives.

We will now consider two programs that utilize mass spectrometry constraints in the elucidation of organic compound structures: DENDRAL and Meta-DENDRAL.

### A3. The DENDRAL Programs

#### A3a. DENDRAL

##### Historical Perspective

In 1964 Joshua Lederberg developed the DENDRAL algorithm which would produce all possible acyclic (unringed) molecular structures, given a set of atoms. This enabled an exhaustive approach to structure elucidation. In 1965 the DENDRAL project started at Stanford. One intent of the project was to show that algorithmic programs which produced results exhaustively and at enormous expense could be augmented by some of the heuristic knowledge used by experts to produce much the same results with a fraction of the effort. The Heuristic DENDRAL Program achieved this end by augmenting the DENDRAL algorithm with a set of rules, those used by expert chemists to infer constraints on molecular structures from mass spectrographic information about the molecule. Unfortunately, pressing expert chemists to formulate rules about mass spectrometry was an arduous process. The theory of mass spectrometry was incomplete, and the rules about it were inexact and heuristic. In 1970, the Meta-DENDRAL project addressed the problem of inferring the rules of mass spectrometry from two sources of information: molecular structures, and their associated mass spectra. This is a continuing project.

In 1976, the CONGEN program became the center of attention in the DENDRAL project. This program replaced the original Heuristic DENDRAL structure generator (which could generate only unringed molecular structures) with a generator without this limitation. CONGEN is discussed in a separate article (D3b) because it has been used as a stand-alone system by research chemists.

#### DENDRAL

The Heuristic DENDRAL program was designed to find a relatively small set of possible molecular structures, given the atoms in the molecule and the mass spectrum of the molecule. The limitations of the DENDRAL algorithm were such that Heuristic DENDRAL could generate only acyclic (unringed) structures: Ketones, alcohols, ethers, thiols, thioethers, and amines. Fig. 1 shows an abbreviated taxonomy of these compounds.

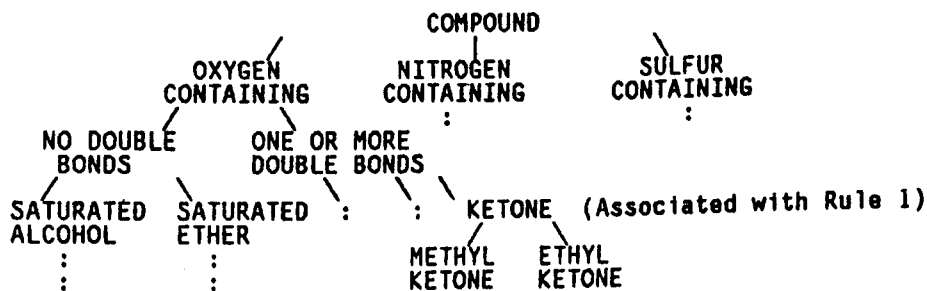


Figure 1. Fragment of a taxonomy of mono-functional acyclic organic compounds.

Heuristic DENDRAL has three functional parts:

1. **PLAN:** Planning in this context means redefining the problem in terms which will reduce the effort of the problem solver, e.g. redefine the problem of finding all possible combinations of a set of atoms to the problem of finding all such combinations which are consistent with constraints derived from mass-spectrometry. Automatic inference of these constraints is the planning part of Heuristic DENDRAL. The list of constraints has two parts: a list of molecular fragments (clusters of atoms) which must be in the final molecular structure, and a list of fragments which are forbidden to appear in the final structure.
2. **GENERATE:** Use these constraints to prevent the DENDRAL algorithm from generating structures which include forbidden subparts and which exclude mandatory subparts. The generator was originally derived from Lederberg's algorithm. When CONGEN was implemented as a stand-alone system, these constraints were provided by the chemists using the program, not by the planning part.
3. **TEST:** Rank the resulting list of candidate structures by simulating its behavior in a mass spectrometer. The structures which result in simulated spectra close to the empirical one are ranked high.

#### Planning: Inferring Constraints From The Mass Spectrum

Heuristic DENDRAL has available to it the mass spectrum and the atomic constituents of a molecule. From the latter it can infer the molecular weight,  $M$ , of the molecule. Many of the rules for interpreting mass spectra include  $M$ , for example, rule 1 (in Fig. 1):

If the spectrum for the molecule has 2 peaks at masses  $x_1$  and  $x_2$  such that

a.  $x_1 + x_2 = M + 28$

b.  $x_1 + 28$  is a high peak

c.  $x_2 - 28$  is a high peak

d. at least one of  $x_1$  or  $x_2$  is high

Then the molecule contains a ketone group.

In the context of Figure 1, we see that this piece of knowledge about mass spectrometry allows Heuristic DENDRAL to constrain its structure generating algorithm to produce molecules with a ketone group as a mandatory constituent. Rule 1, along with a number of other rules, significantly constrains the number of molecules generated by the structure generator. For example, given the spectrum for a molecule containing 8 carbons, 16 hydrogens and 1 oxygen, the constraint generating program eliminated from consideration (i.e., placed on a list of forbidden structures called BADLIST) all possible structures except those containing ethyl ketone 3, and so reduced the number of generated molecular structures from the topologically possible 790 to a constrained set of 3 (called the GOODLIST).

#### The Generator

The algorithm for generating molecular structures is complicated and has no AI content; we will discuss it only in general terms and refer the reader to Buchanan, Sutherland, Feigenbaum (1969) for a detailed discussion. The following article (A3b) and (Refs?) discuss the more recent CONGEN generator.

There are several design characteristics of the generator which are related to the enormous number of molecules that are combinatorially possible in an analysis problem. First, the generator must be *proved* to be complete--it must be able to generate all topologically possible molecular structures. It should also be non-redundant, that is, it should generate each structure only once. This was a problem for structures with rings, because Lederberg's algorithm treated symmetrical molecules as unique structures. A third characteristic is that the generator should be flexible enough to be focussed by constraints from the planning part. It should not blindly generate all possible structures, but only those which fulfill the constraints.

The structure generator is supplied with the composition of the molecule (its atoms and their numerosity), a spectrum, a list of constraints (GOODLIST and BADLIST) including likely substructures and impossible substructures. Its task is to generate all structures compatible with these data. Note that if GOODLIST and BADLIST are empty it will generate all *isomers* (structural variants) of the given composition.

Some simple checks are made by the generator. The composition should be compatible with the constraints inferred from the spectrum, and the structures generated should have only the types and amounts of atoms specified in the composition. Finally, the generator should not produce a structure which is known by DENDRAL to be unstable.

The structure generator essentially "grows" molecules, starting with a small fragment of the molecule and adding pieces of the composition to it. At any point in the growing process, there are numerous atoms or molecular fragments that can be added onto the growing structure, and there are many places where these parts can be attached. But generally the constraints offered by GOODLIST and BADLIST limit the number of possible structures that might be grown at any point in the growing process.

### The Testing and Ranking Programs

The programs MSPRUNE and MSRANK (Varkony, Carhart, and Smith, 1977) use a large amount of knowledge about mass spectrometry to make testable predictions from each plausible candidate molecule. Predicted data are compared to the data from the unknown compound, and some candidates are thrown out, while others are ranked.

MSPRUNE works with: (a) a list of candidate structures from the structure generator, and (b) the mass spectrum of the unknown molecule. It uses a fairly simple model of mass spectrometry to predict commonly expected fragmentations for each candidate structure. Predictions that deviate greatly from the observed spectrum are considered *prima facie* evidence of incorrectness, and the corresponding structures are pruned from the list. MSRANK then uses more subtle rules of mass spectrometry to rank the remaining structures according to the number of predicted peaks found (and not found) in the observed data, weighted by measures of importance of the processes producing those peaks.

### Research Results

The Heuristic DENDRAL project, from 1968 to the present, and including CONGEN, has produced a number of results of significance to chemists. The effort has shown that it is

possible to write a computer program that equals the performance of experts in some very specialized areas of science. Published papers on the program's analysis of aliphatic ketones, amines, ethers, alcohols, thiols, and thioethers (Duffield et al., 1969; Schroll et al., 1969; Buchanan (1970)) make the point that although the program does not know more than an expert (and in fact knows far less), it performs well because of its systematic search through the space of possibilities and its systematic use of what it does know. A paper on the program's analysis of estrogenic steroids notes that the program can solve structure elucidation problems for complex organic molecules (Smith et al., 1972). Another paper, on the analysis of mass spectra of mixtures of estrogenic steroids (without prior separation), establishes the program's ability to do better than experts on some problems (Smith et al., 1973). With mixtures, the program succeeds where people fail; the task of correlating data points with each possible fragmentation of each possible component of the mixture is too difficult for people to do. Several articles based on results from CONGEN demonstrate its power and utility for solving problems of medical and biochemical importance (Smith, 1975; Smith, 1976; Buchanan, 1976; Mitchell, 1978, and (Varkony, Carhart, & Smith, 1977)).

DENDRAL programs have been used to aid in structure determination problems of the following kinds:

- terpenoid natural products from plant and marine animal sources,
- marine sterols,
- organic acids in human urine and other body fluids,
- photochemical rearrangement products,
- impurities in manufactured chemicals,
- conjugates of pesticides with sugars and amino acids,
- antibiotics,
- metabolites of microorganisms, and
- insect hormones and pheromones.

CONGEN (discussed next) has also been applied to published structure elucidation problems by students in organic chemistry classes to check the accuracy and completeness of published solutions. In several cases, the program found structures that were plausible alternatives to the published structures (based on problem constraints that appeared in the article). This kind of information served as a valuable check on conclusions drawn from experimental data.

#### References

See Buchanan, Sutherland, & Feigenbaum (1969), Buchanan, Duffield, & Robertson (1971), Buchanan et al. (1976), Buchanan (1970), Buchanan (1970), Cheer et al. (1976), Churchman & Buchanan (1969), Duffield et al. (1969), Feigenbaum & Buchanan (1968), Lederberg (1964a), Lederberg (1964b), Lederberg (1965a), Lederberg (1965b), Lederberg (1965c), Lederberg (1969), Lederberg & Feigenbaum (1968), Mitchell (1978), Morrill, Smith, & Djerassi (1977), Schroll et al. (1969), Smith et al. (1972), Smith et al. (1973), Smith (1975), Smith & Carhart (1976), Smith, Konopelski, & Djerassi (1976), and Varkony, Carhart, & Smith (1977).



## A3b. CONGEN and its Extensions

## CONGEN: Interpretation of Constraints

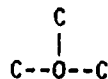
CONGEN (for CONstrained GENERator) is a program that was designed in 1976 to replace the old DENDRAL generator of acyclic structures. It has proved to be a powerful stand-alone program to assist the chemist in determining the molecular structure of unknown compounds. Its objective was twofold: (a) to allow the user to interactively specify certain types of structural information determined from any of several sources (e.g. spectroscopy, chemical degradation, method of isolation, etc.); and (b) to generate an exhaustive and nonredundant list of structures consistent with this information. Unlike the original Heuristic DENDRAL program, it does not infer *constraints* from mass spectra, but allows the chemist to specify them. Another difference between CONGEN and Heuristic DENDRAL is that the former can generate *cyclic* as well as *acyclic molecules*. The generation is a stepwise process, and the program allows interaction at every stage. Based upon partial results, the chemist may be reminded of additional information that he can specify, thus limiting further the number of structural possibilities.

CONGEN breaks the problem statement given by the chemist down in several different ways, for example: (a) hydrogen atoms are omitted until the final steps of processing; (b) parts of the graph containing no cycles are generated separately from cyclic parts (and combined at the end); (c) cycles containing only unlabelled nodes are generated before the nodes are labeled with the names of chemical atoms (e.g., carbon or nitrogen); and (d) cycles containing only three-connected nodes (e.g., nitrogen or tertiary carbon) are generated before two-connected nodes (e.g., oxygen or secondary carbon) are mapped onto the edges. At each step, several constraints may be applied to limit the number of emerging chemical graphs (Carhart et al., 1975).

There are two algorithms at the heart of CONGEN whose validity producing nonredundant structures has been mathematically proven (Brown and Masinter, 1974; Masinter, et al., 1974) and whose computer implementation has been well tested. Combined, they are designed to determine all topologically unique ways of assembling a given set of atoms, each with an associated valence, into molecular structures. The atoms may be chemical atoms with standard chemical valences, or they may be names representing *molecular fragments (superatoms)* of any desired complexity, where the valence corresponds to the total number of bonding sites available within the superatom. The algorithms can be thought of as performing *problem reduction*, and *reconstruction* or *subproblem recomposition* on molecular structures. The first, *partitioning*, algorithm breaks down the problem of finding a complete molecular structure into subproblems, for example, to find the structures of the ringed and non-ringed components of the molecule. The second, *embedding*, algorithm combines the substructures, found by partitioning, into complete molecular structures. Clearly, neither partitioning nor reconstruction can be unconstrained processes because of the combinatorics involved: There are simply too many possible subproblems to solve, and each of them may have many solutions. Consequently, combining subproblem solutions exhaustively is not feasible. In both algorithms, constraints are brought to bear to limit the size of the problem. Three types of constraints are:

1. Graph theoretic: Do not treat symmetric structures as unique.

2. Syntactic: Structures are constrained by the valences of the constituent atoms, e.g.



is impossible because oxygen is bivalent, i.e. only has two bonding sites.

3. Semantic: The chemist can provide additional information about the molecule which will help to determine its structure.

Substantial effort has been devoted to modifying these two basic procedures, particularly the structure generation algorithm--getting it to accept a variety of other structural information (constraints) and using it to prune the list of structural possibilities. Current capabilities include specification of good and bad substructural features, good and bad ring sizes, proton distributions and connectivities of isoprene units (Carhart & Smith, 1976). Usually the chemist has additional information (if only some general rules about chemical stability) of which the program has little knowledge, which he can use to limit the number of structural possibilities. For example, he may know that the chemical procedures used to isolate the compound would change organic acids to esters; thus the program would not need to consider structures with unchanged acid groups. In CONGEN, he is given the facilities to impart this knowledge interactively to the program.

To make CONGEN easy for research chemists to use, the program has been provided with an interactive "front end." This interface contains EDITSTRUC, an interactive structure editor; DRAW, a teletype-oriented structure display program; and the CONGEN "executive" program, which ties together the individual subprograms and aids the user with various tasks such as defining superatoms and substructures, creating and editing lists of constraints or superatoms, and saving and restoring superatoms, constraints, and structures from secondary storage (disc). The resulting system, is running on the SUMEX computing facility at Stanford and is available nationwide over the TYMNET network. It has recently been completely re-written in the BCPL language to run on a variety of other machines.

#### Limitations and Extensions

Although computer programs, including CONGEN, now exist to assist chemists in constructing structural isomers based on information about partial structures, the programs have one serious, common limitation. Each program must use non-overlapping structural fragments as building blocks. This limitation leads to at least two important problems. First, the chemist using such a program must select non-overlapping partial structures; otherwise an incomplete set of structures will result. This procedure, done manually, is time-consuming, and prone to error. Second, as a consequence of the first step, problems are solved less efficiently by the programs because a detailed environment of fewer atoms has been specified--to ensure the absence of overlaps.

The GOODLIST INTERPRETER is a first attempt to remove this limitation by simulating the manual procedure that the chemist uses to arrive at a set of nonoverlapping constraints. It is

designed to make more efficient use of information about required (GOODLIST plus superatoms) structural features of an unknown. Some early successes have demonstrated that new problems are brought within the realm of solution by use of the GOODLIST INTERPRETER that are impossible in CONGEN alone, due to the constraints on computational resources.

### Stereochemistry

One of the most important new additions to CONGEN deals with the problem of enumerating all the stereoisomers of a given compound.

The mathematical problem of enumerating stereoisomers was solved by Jim Nourse. Considerations of symmetry as embodied in the mathematical theory of groups played a decisive role in the solution. Coupled with the stereoisomer generator, and given an empirical formula and a number of constraints, CONGEN can generate all the stereoisomers that are possible solutions to the unknown target molecule to be elucidated.

While the solution to the enumeration of stereoisomers uses very little, if any, AI techniques, it solves a problem that human beings find very difficult to solve. Chemists usually learn to solve this problem by using visual intuition. The mathematics involved are deep enough so that the average chemist will not have the patience necessary to learn enough about the algorithm to use its insights in enumerating stereoisomers. One of central problems for AI work in chemistry now is how to use this new facility in structure elucidation.

### EXAMINE

Often in the course of a structure elucidation problem, a large number of candidate structures, perhaps a hundred or more, are generated; and additional constraints must be derived, either from further data analysis or from new experiments. The EXAMINE function written by Neil Gray is used from within CONGEN to survey, classify, display, or discard structures. This is very useful to the chemist who is searching for features common to a large number of the structures or for features that are unique to certain structures. The insights gained from using EXAMINE can be used in planning new experiments or in further data analysis. In pursuit of these objectives, the chemist can define functional groups and other structural features, or he can work with a predefined library of them. The EXAMINE function is then called, and it examines the list of candidate structures for the presence or absence of these features.

For example, the chemist can ask EXAMINE to look for all structures with exactly one labile proton. A labile proton is a hydrogen atom attached to a nitrogen atom or a hydrogen atom attached to an oxygen atom. The chemist can express this to EXAMINE as an exclusive OR: exactly one hydrogen attached to an oxygen atom in the structure OR (exclusive) exactly one hydrogen attached to a nitrogen atom in the structure.

The user can then request EXAMINE to draw those structures that have this characteristic and those that do not, in order to produce summary statistics on its frequency of occurrence or to discard those structures with or without it. While CONGEN is always able

to discard or prune away structures that do not satisfy certain constraints, EXAMINE provides the interactive ability to develop boolean combinations of constraints for pruning, substructure search, or subsequent classification.

## REACT

Before spectroscopy became a major tool of the structural chemist, all structure elucidation had to be done by means of reaction chemistry, and it is still a major tool in solving structures. REACT is an interactive program written by Ray Carhart, Tomas Varkony, and Dennis Smith. Although it is a close relative to the synthetic programs described below (see article D5), its purpose is to aid chemists in the structure elucidation task rather than to aid them in finding new synthetic routes.

To show how REACT can be used to reduce the number of candidate structures found by CONGEN, consider the following example. A dehydration reaction can be expressed as a production rule of the form: "If you see the pattern C-C-O, convert it to the pattern C=C." We now suppose that a dehydration reaction was applied to the unknown in question and yielded three distinct structures, which happened because the pattern C-C-O occurred in the molecule in three different places. This information can be used to eliminate structures from those under consideration: The structure list generated by CONGEN is passed to REACT; the dehydration reaction is defined by the user and then applied to all the candidate structures; those that do not yield exactly three products can be eliminated from consideration as candidate structures.

Although REACT does not contain stereochemical information, conformational information, or electronic information (the electro-negativities of its atoms and groups), it still can be used reliably in its structure elucidation function. Reactions used for structure determination tend to have high yield, to be reliable, and to involve simple separations. The reactions operate under a wide variety of conditions and usually involve rather simple changes to the unknown molecule. This being so, the perception routines do not need the sophisticated stereochemical, conformational, and electronic information of the organic synthesis programs discussed above.

## Summary

Research in the DENDRAL project has followed two themes: To build a performance program for analysis of molecular structures, and to explore with AI methods some problems of scientific inference. The performance of Heuristic DENDRAL has been evaluated in the same way as that of a research chemist: by publications. (See the conclusion of the article D3a on DENDRAL for references.) In addition, CONGEN is used daily by chemists to aid in solving structure elucidation problems.

Because of the combinatoric size of analysis problems, exhaustive problem-solving methods were not an option, and much thought was given to the knowledge which enabled chemists to solve these problems. DENDRAL was one of the first programs to demonstrate the power of encoding domain-specific, heuristic expertise, and was therefore one of the first projects to recognize knowledge acquisition as a major problem in AI (Buchanan, 197x; Davis, 1976a). The next article (D3c) discusses automatic inference of rules as one solution to the knowledge acquisition problem.

**References**

See Brown, Masinter, & Hjelmeland (1974), Brown & Masinter (1974), Carhart et al. (1975), Carhart & Smith (1976), Masinter et al. (1974), Sheikh et al. (1970), and Smith & Carhart (1978).

## A3c. Meta-DENDRAL

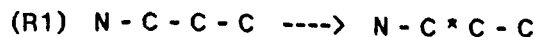
## Overview

The domain specific rules that constitute DENDRAL's knowledge about mass spectrometry were derived from consultation with experts in that field. Since the consultation process is time consuming, two alternatives to "handcrafting" knowledge bases have been explored. One is Interactive Transfer of Expertise (Davis ref. AIH ref). The other is automatic theory formation. Meta-DENDRAL is a program of the latter type. The rule formation task that Meta-DENDRAL performs is similar to the task of grammatical inference, sequence extrapolation, and concept formation (Hunt (1975), Hedrick (1974), Winston (1970a)). Programs that perform these tasks can all be thought of as "induction" programs because they formulate general rules (or concepts, or patterns) from examples.

Meta-DENDRAL is designed to infer theories for the Heuristic DENDRAL program, which represents knowledge about mass-spectrometry as production rules. Automatic rule formation was chosen as a paradigm for Meta-DENDRAL for two general reasons. First, this design poses interesting epistemological questions, and second, it is an arduous task to derive rules from human consultants, especially when the task-domain has only a small number of experts (as is the case in mass-spectrometry).

## Representation of Knowledge about Mass-spectrometry

In DENDRAL, knowledge is represented in *production rules*, each of which specifies a bond *fragmentation* in a particular context in a molecule. For example, one simple rule is:



Rules are interpreted for each molecule in the following way:

- (1) Find all places in the molecule that match the subgraph expressed by the left-hand side of the rule.
- (2) For each match, break the molecule at the bond marked with an asterisk in the right-hand side of the rule and save the fragment associated with the atoms to the left of the asterisk.
- (3) Record the mass of all saved fragments.

No migration of atoms between fragments is predicted by (R1).

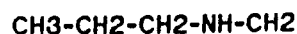
The language of processes (right-hand sides of rules) is relatively simple: One or more bonds from the left-hand side may break and zero, or one or more, atoms may migrate between fragments. The interpretation of rule R1 in the above example is straightforward: If a molecule contains a nitrogen atom and three carbon atoms bonded as N-C-C-C, then it will fragment in the mass spectrometer between the middle two carbon atoms, and the N-C fragment will be recorded in the spectrometer as a peak at the point in the spectrum corresponding to the molecular weight of this fragment.

### Formation of Mass Spectral Rules

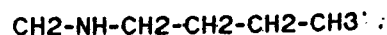
The task of Meta-DENDRAL is to infer rules like R1 above from empirical data. Meta-DENDRAL is provided with descriptions of the structures of a related set of molecules, and with the set of peaks produced by the fragmentation of each molecule in the mass spectrometer, and from these data it infers a small and fairly general set of mass-spectral rules to account for the fragmentations of the molecules and the corresponding spectral peaks.

**Training Instances.** In order to learn rules, the Meta-DENDRAL program is presented with many examples of actual I/O pairs from the mass spectrometer. Each I/O pair represents a molecular graph structure, together with a data point from the mass spectrum for that structure. The rules to be learned constitute a representation of the relevant fragmentations in the mass spectrometer. Typically, the program starts with a training set of six to ten related molecules and their associated spectra, each containing 50-150 data points--peaks marking the masses of recorded fragments (together with the relative abundance of fragments at those masses). (These are drawn from an infinitely large space of possible instances, of which only a few for each structural class of molecules are available from spectra libraries. Figure 1 shows part of a set of training data for a class of simple molecules containing the nitrogen atom and no rings. This structural class is known as *aliphatic amines*).

In a large molecule, rule (R1) may apply more than once. For example, the spectrum of CH<sub>3</sub>-CH<sub>2</sub>-CH<sub>2</sub>-NH-CH<sub>2</sub>-CH<sub>2</sub>-CH<sub>2</sub>-CH<sub>3</sub> will contain data points at masses 72 and 86 corresponding to the two fragments derived from the application of this rule:



and



For a number of reasons, data points are not uniquely associated with a single fragmentation and atom migration process (rule). For example, a single process may occur more than once in a molecule (as above), and more than one process may produce identical fragments, producing peaks at the same mass points in the spectra.

### Spectral Data Points and Mass-spectral Processes: Statistical and Semantically Constrained Associations

Purely statistical learning programs (Jurs, 1974) find associations indicated by the data without judging the meaningfulness of these associations. This feature can be advantageous; at times an investigator's bias inhibits his seeing associations, or an investigator may be looking for all possible associations. But it is a disadvantage when the number of associations is so large that the meaningful ones, unmarked, get lost in the crowd.

In contrast to statistical approaches, Meta-DENDRAL utilizes a *semantic model* of the domain. This model has been included for two important reasons. First, it provides guidance for the rule formation program in a space of rules that is much too large to search exhaustively and in a domain of input data that is often ambiguous. Second, it provides a

check for the meaningfulness of associations produced by the program, in a domain where the trivial or meaningless associations far outnumber the important ones.

**Semantic model of the domain.** The base-level, or zero-order, theory of mass spectrometry states that every subset of bonds within a molecule may break and that the resulting fragments, plus or minus migrating atoms, will all be recorded. This zero-order model of mass spectrometry is not specific enough to effectively constrain the rule search. Therefore, some general guidelines have been imposed on it, the so-called *half-order* theory.

The half-order theory asserts that bonds will break and atoms will migrate to produce data points. This theory orders the break-and-migrate process according to the following constraints:

**Constraints on fragmentations:**

Double bonds and triple bonds do not break.

No aromatic bonds break.

Only fragments larger than 2 carbon atoms show up in the data.

Two bonds to the same carbon atom cannot break together.

No more than 3 bonds break in any one fragmentation.

No more than 2 complete fragmentations occur in one process.

At most 2 rings fragment in a multiple-step process.

**Constraints on atom migration:**

At most 2 hydrogen atoms can migrate after a fragmentation.

At most 1 H<sub>2</sub>O unit is lost after any fragmentation.

At most 1 CO unit is lost after any fragmentation.

One of the most helpful features of this model is its *flexibility*: Any of the parameters can be easily changed by a chemist with other preconceptions; any of these assumptions can be removed and, as discussed in the following section, additional statements be substituted or added. This power to guide rule formation results in the program's discovering only rules within a well-known framework; on the other hand, it also results automatically in rules meaningful to the domain.

A chemist will often know more about the mass spectrometry of a class of molecules than is embodied in the half-order theory. It is important then to be able to augment the program's model by specifying class-specific knowledge to the program. This capability provides a way of forming new rules in the context of additional intuitions or biases about mass spectrometry. A chemist can thus see the "most interesting" rules (as defined by the augmentations) before the other rules. For example, one might be interested first in rules that mention at least one nitrogen atom before the numerous (and generally less interesting) rules that mention only carbon and hydrogen substructures.

**Learning strategy.** The Meta-DENDRAL program is based on a generator of production rules that uses predetermined syntax operating under the constraints of a semantic world model. The operation of Meta-DENDRAL can be summarized as follows:

1. Input.

- a. the structure of each of a set of related molecules (recall that Meta-DENDRAL



is not a structure elucidation program but infers rules of mass spectrometry, which associate molecular structures and their mass spectra),

- b. the spectral data points (peaks) for each of the molecules, and
- c. the half-order theory (or some semantic theory to constrain the generation of rules).

## 2. Step 1. (INTSUM)

For each molecule, explain each peak in its spectrum by finding one or more fragmentation processes that would account for the peak. The number of plausible fragmentation processes is limited by:

- a. considering only the fragmentations which are allowed by the half-order theory (e.g., no spectral peak can be explained by a fragmentation process that involves breaking a double bond), and

- b. considering only fragmentations which produce fragments with a molecular weight corresponding to the weight represented by the peak. (Recall that each peak in a mass spectrum represents a number of molecular fragments of a given mass.) For example, if the total weight of the molecule under inspection is M, and the spectrum has a large peak associated with a molecular weight of M-47 mass units, then the only fragmentation processes considered as explanations for this point would be those that produce a fragment with a molecular weight of M-47. The tens, or hundreds, of other processes which are consistent with the half-order theory, like cleaving off a hydrogen atom, are not even considered.

After each data point in the spectrum for each molecule has been explained by a plausible fragmentation process, the list of processes is summarized, since the same fragmentation processes will often be found to account for many spectral data points. The final product of INTSUM is a list of fragmentation processes with the total evidence for each such process.

## 3. Step 2. (RULEGEN)

The rules provided by INTSUM each account for a single fragmentation process in the context of a single molecule. As such, they are not general. The problem with general rules, on the other hand, is that a single one may subsume several of INTSUM's very specific fragmentations, BUT ALSO fragmentations not represented in the set produced by INTSUM. That is, a general rule may correctly explain many data points in mass spectra, (positive evidence), but may also predict points which do not occur in any of the spectra (negative evidence). The purpose of RULEGEN is to find a set of rules which are more general than those of INTSUM, using positive evidence as a criterion of success. Negative evidence which is introduced by these rules is handled by a later step, called RULEMOD.

RULEGEN works by "growing" a tree of fragmentation rules, starting with an overgeneral one and adding features to it so that it becomes more constrained. The rule that RULEGEN starts with is  $X * X$ , that is, the bond between any atoms

will break, and the mass of fragment X will be recorded in the mass spectrometer as a peak. Obviously, every fragmentation rule is a specialization of this one, and it is too general to be interesting. But by specifying values for four features--the identity of X, the number of non-hydrogen neighbours X has, the number of hydrogen neighbors X has, and the number of doubly-bonded neighbors X has--the general rule  $X \rightarrow X$  can be "grown" into something more interesting. This process will be described below.

#### 4. Step 3. (RULEMOD)

RULEGEN can generate rules which predict non-existent data points in the mass-spectral data. This negative evidence is the cost of the coarse method used by RULEGEN to find general rules. RULEMOD "tidies up" the rules produced by RULEGEN by merging rules, eliminating redundancies, and making rules more specific or general. In addition, if a rule has been used successfully for a time, but an instance is found in which it is inappropriate, RULEMOD can modify the rule accordingly.

#### 5. Output.

A set of mass spectral fragmentation rules which are specialized enough to be interesting, but general enough to be efficient and nonredundant.

We will now consider INTSUM, RULEGEN, and RULEMOD in more detail.

### The Meta-DENDRAL program

The program itself is organized as a series of *plan-generate-test* steps, as found in many AI systems (Feigenbaum, M16). After pre-scanning a set of several hundred molecular structure/spectral-data-point pairs, the program searches the space of fragmentation rules for plausible explanations and then modifies its rules on the basis of detailed testing. When rules generated from a training set are added to the model and another block of data is examined, the rule set is extended and modified further to explain the new data. The program iteratively modifies rules formed from the initial training set (adding to them); but it is currently unable to "undo" rules.

**Planning: Data Interpretation and Summary.** The planning step in the procedure is carried out by INTSUM. In this step, each spectral data point is reinterpreted as a list of fragmentation and atom migration processes (potential right-hand sides of rules) that are feasible explanations of the data point within the specified model (the half-order theory). This step must be done since the final rules should propose the underlying mass spectral processes that produce data points, not just the data points themselves. For each molecule in a given set, INTSUM produces the plausible mass spectral processes that might occur, that is, the breaks and combinations of breaks, with and without migration of atoms. INTSUM then examines the data points associated with each molecule, looking for evidence (spectral peaks) for each process. Finally, the planning step produces a summary showing the total evidence associated with each possible process. This summary is used during rule generation to avoid considering uninstantiated candidate rules.

**Generating Rules.** After the data have been interpreted in INTSUM, control passes to a heuristic search program known as RULEGEN, for rule generation. RULEGEN creates general rules by selecting "important" features of the molecular structure around the site of the fragmentations proposed by INTSUM. These important features are combined to form a subgraph description of the local environment surrounding the broken bonds. Each subgraph considered becomes the left-hand side of a candidate rule whose right-hand side is INTSUM's proposed process. Essentially, RULEGEN searches through a space of these subgraph descriptions, looking for successively more specific subgraphs that are supported by successively "better" sets of evidence. To constrain this search, it uses the constraints of the half-order theory, discussed above.

Conceptually, the program begins with the most general candidate rule,  $X^*X$  (where  $X$  is any unspecified atom and where the asterisk is used to indicate the broken bond, with the detected fragment written to the left of the asterisk). Since the most useful rules lie somewhere between the overly general candidate,  $X^*X$ , and the overly specific complete molecular structure (with specified bonds breaking), the program generates refined descriptions by successively specifying additional features. This is a coarse search; for efficiency reasons, RULEGEN sometimes adds features to several nodes at a time, without considering the intermediate subgraphs.

The program adds features (*attribute-value pairs*) to subgraphs, starting with the subgraph  $X^*X$ , always making each successor more specific than its parent. (Recall that each node can be described with any or all of the following attributes: atom type, number of nonhydrogen neighbors, number of hydrogen neighbors, and number of doubly bonded neighbors.) The program assigns one attribute at a time to all atoms that are the same number of atoms away from the breaking bond. Each of the four attributes is considered in turn, and each attribute-value for which there is supporting evidence generates a new successor. Although different values for the same attribute may be assigned to each atom at a given distance from the breaking bond, the coarseness of the search prevents the examination of subgraphs where this attribute is totally unimportant on *some* of these atoms.

A portion of the rule search tree leading to R1 is shown in Figure 1. Starting with the parent subgraph  $S_0$ , the "number-of-neighbors" attribute is specified for each atom adjacent to the break in subgraph  $S_1$ . "Atom type" is then specified for atoms adjacent to the break in  $S_2$ , and for atoms one bond removed from the break in  $S_3$ . At each step in the search there are also many other possible successors corresponding to assignments of other values to these same attributes, or assignment of values to other attributes.

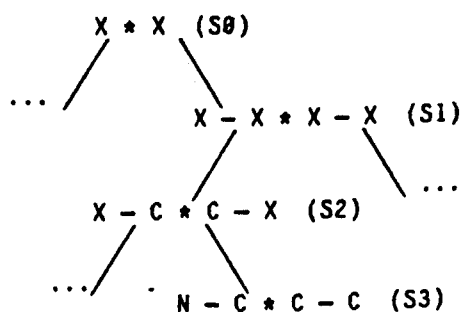


Figure 2. Portion of the RULEGEN search.

Each descendant is checked, to see if the supporting evidence is "better" (see below) than the evidence for the parent. Those that satisfy the test become new parents for a next level of descendants, which have one more feature specified. The program adds specifications to candidate rules until it finds a rule that is (a) specific enough to make correct predictions and (b) general enough to account for more than a few special cases. Subgraph S3 in Figure 2 meets these conditions and is output by the program as rule R1.



In (R1) the features that have been judged important are the atom types and the connections of four atoms; the other features and atoms have been generalized away. The point of generalizing is to abstract away unimportant attributes of atoms and unimportant atoms.

**Modifying Rules.** The last phase of the program (called RULEMOD) evaluates the plausible rules generated by RULEGEN and modifies them by making them more general or more specific. In order to extend the range of applicability of the rules, RULEMOD uses a less constrained model than RULEGEN. Rules generated by RULEGEN under an augmented half-order theory, for example, in which only fragments containing a nitrogen atom were considered, cannot immediately be applied by a performance program using a more general model. Therefore, RULEMOD refines the rule so that it can stand on its own in the context of a more general model. In contrast to RULEGEN, RULEMOD considers negative evidence (incorrect predictions) of rules in order to increase the accuracy of the rule's applications within the training set. RULEGEN performs a coarse search of the rule space, for reasons of efficiency, whereas RULEMOD performs a localized, fine search to refine the rules.

RULEMOD will typically output a set of 8 to 12 rules covering substantially the same training data points as the input RULEGEN set of approximately 25 to 100 rules, but with fewer incorrect predictions. This program is written as a set of five tasks (mentioned below) that are closely analogous to this aspect of human problem solving.

- (1) Remove redundant rules--give preference to simpler rules;
- (2) Merge similar rules;
- (3) Make rules more specific to avoid false predictions;

For example, rule (R1') below would be a specification of (R1) that RULEGEN would miss because it specifies different attributes (not just different values) for atoms that are the same distance from the broken bond (asterisk):



In this case, the number of hydrogen neighbors is specified for the first left-hand atom from the broken bond, but not for the first right-hand one.

- (4) Make rules more general to avoid unnecessary detail.

Rule (R1') for example, could be made more general by removing the atom type specification on one of the first atoms next to the asterisk, and by removing completely the rightmost atom in the subgraph, resulting in rule (R1'').



Again, because of the coarseness of its search, RULEGEN could not have considered this form of the rule. It is assumed that RULEGEN will produce good approximations to rules, and that RULEMOD will refine them.

**Integrating Subsequent Data.** A requirement for any practical learning program is the ability to integrate newly acquired data in an evolving knowledge base. New data may dictate that additional rules be added to the knowledge base or that existing rules be modified or eliminated. New rules may be *added* to the rule base by running RULEGEN on the new data and then running RULEMOD on the combined set of new and previously generated rules.

When an existing rule is *modified*, it is important to maintain the integrity of the modified rule over past training instances. Consider the following example: A new training instance is acquired and, after credit assignment questions are resolved, it is decided that rule R was incorrectly "triggered" by some situation S. The left-hand side of rule R must be modified so that it will no longer match S. In general, there would be many changes possible to R that would kill the match to S, but some are better than others. The correct changes to R are those which do not alter past correct applications of R. Of course there is no way of knowing which of the possible changes to R will turn out to be correct for future data; and once a change is selected, the possibility exists for the necessary backtracking at some future point.

A method has been developed for representing all versions of the left-hand side of a rule that are consistent with the observed data for all iterations thus far (Mitchell, 1977). This representation is referred to as the *version space* of the rule. By examining the version space of R, one can answer the question "Which of the recommended changes to R will preserve its performance on past instances?" The answer is simply "Any changes that yield a version of the rule contained in the version space." Using version spaces avoids the problem of selecting a single unretractable modification to R and therefore eliminates the need for backtracking. For example, all the elements of the version space that match some negative instance S are eliminated. Similarly, when new data are encountered in which a situation S' is found to correctly trigger R, only those elements of the version space that match S' are retained.

## Results

One measure of the proficiency of Meta-DENDRAL is the ability of a DENDRAL program using the learned rules to predict correct spectra of new molecules. One of the DENDRAL performance programs ranks a list of plausible hypotheses (candidate molecules) according to the similarity of their predictions (predicted spectra) to observed data. The rank of the correct hypothesis (i.e., the molecule actually associated with the observed spectrum) provides a quantitative measure of the "discriminatory power" of the rule set.

The Meta-DENDRAL program has successfully rediscovered known, published rules of mass spectrometry for two classes of molecules, including the aliphatic amines used as examples above. More importantly, it has discovered new rules for three closely related families of structures for which rules had not previously been reported. These are the mono-, di-, and tri-keto androstanes which share the common structural skeleton shown in Figure 2.

Meta-DENDRAL's rules for these classes have been published in the chemistry literature (Buchanan et al., 1976). Evaluations of all five sets of rules are discussed in that publication. This work demonstrates the utility of Meta-DENDRAL for rule formation in mass spectrometry for classes of structures.

The recent application of Meta-DENDRAL has been to a second spectroscopic technique: *13C-nuclear magnetic resonance* ( $^{13}\text{C-NMR}$ ) spectroscopy (Mitchell, 1978). This new version provides the opportunity to direct the induction machinery of Meta-DENDRAL under a model of  $^{13}\text{C-NMR}$  spectroscopy. It generates rules that associate the resonance frequency of a carbon atom in a magnetic field with the local structural environment of the atom.  $^{13}\text{C-NMR}$  rules have been generated and used in a candidate molecule-ranking program similar to the one described above.  $^{13}\text{C-NMR}$  rules formulated by the program for two classes of structures have been successfully used to identify the spectra of additional molecules (of the same classes, but outside the set of training data used in generating the rules). The rule-based molecule-ranking program performs at the level of a well-educated chemist in both the mass spectral and  $^{13}\text{C-NMR}$  domains.

#### References

See Buchanan (1976), Buchanan et al. (1976), Davis (1976b), Feigenbaum (MI6), Hedrick (1974), Hunt (1975), Jurs (1974), Minsky (1963), Mitchell (1977), Mitchell (1978), Samuel (1963), Simon & Lea (1973), Smith et al. (1977), Waterman (1970), Waterman (1974), and Winston (1970a).

#### A4. CRYNALIS

##### Motivation

The CRYNALIS system, which is still under development, is an attempt to apply Artificial Intelligence methodology to the task domain of protein crystallography. Although the computer has been an essential tool in x-ray crystallography research for many years, nearly all its applications have been in the areas of data collection, data reduction, Fourier analysis, graphics, and other essentially numerical tasks (Feigenbaum, Engelman, & Johnson, 1977). Those aspects of molecular structure inference that require symbolic reasoning or that use a significant amount of judgmental knowledge have traditionally been performed manually. A prime example is the task of *electron density map* interpretation.

In the course of deriving a protein structure, the crystallographer generates an electron density map, a three-dimensional description of the electron density distribution of a molecule. Due to the resolution imposed by the experimental conditions, the electron density map is an indistinct image of the structure that does not reveal the positions of individual atoms. The crystallographer must interpret the map in light of auxiliary data and general principles of protein chemistry in order to derive a complete description of the molecular structure. The goal of the CRYNALIS system is to integrate these diverse sources of knowledge and data to try and match the crystallographer's level of performance in electron density map interpretation. Automation of this task would shorten the time taken for protein structure determination by several weeks, to months, and would fill in a major gap in the construction of a fully automated system for protein crystallography.

##### Description of the problem

When crystallographers use the term "electron density map," they usually have in mind some pictorial representation of the electron density defined over a certain region of space. The most commonly used representation is a three-dimensional contour map, constructed by stacking layers of conventional two-dimensional contour maps drawn on transparent sheets. By carefully studying the map, the experienced protein crystallographer can find features that allow him to infer approximate atomic locations, molecular boundaries, groups of atoms, the backbone of the polymer, etc. After several weeks (or months), he has built a model of the molecular structure that conforms to the electron density map and is also consistent with his knowledge of protein chemistry, stereochemical constraints, and other available chemical and physical data (e.g., the amino acid sequence). Figure 1 shows a portion of a protein structure and the associated electron density map from which it was inferred.

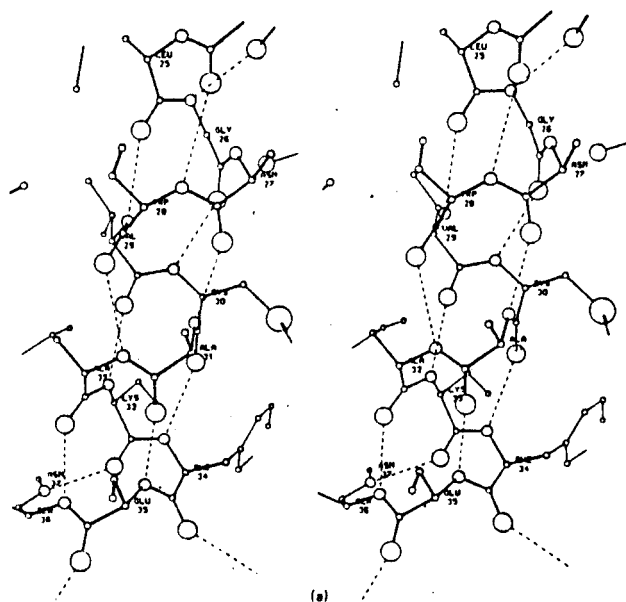


FIG. 1 A stereo-view of the electron density (b) at 2.8 Å of an  $\alpha$ -helix in lysozyme, and the molecular structure (a) corresponding to this density (Snape, 1974).

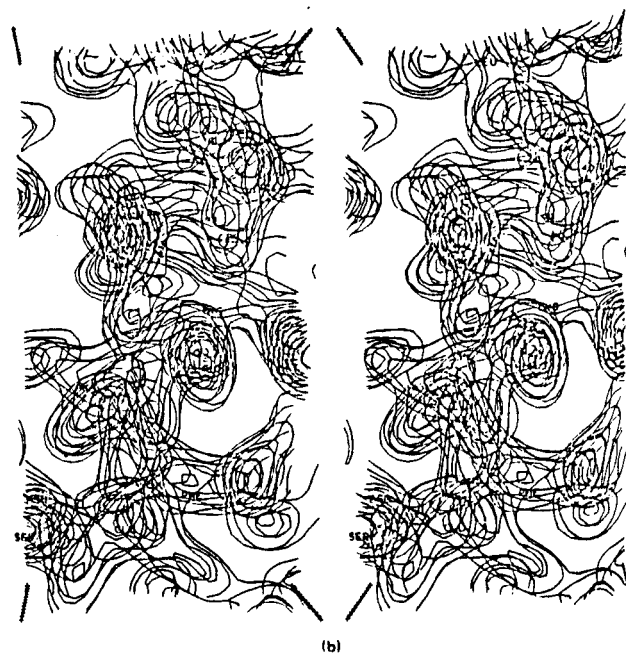


FIG. 1 (continued)

The automation of this task would require a computational system that could generate its own structural hypotheses, as well as display and verify them. This capability requires: (a) a representation of the electron density function suitable to machine interpretation, (b) a substantial chemical and stereochemical knowledge base, (c) a wide assortment of model-building algorithms and heuristics, (d) a collection of rules and associated procedures for using this knowledge to make inferences from the experimental data, and (e) a problem-solving strategy for applying the knowledge sources (KSs) effectively, so that the appropriate procedures are executed at the times that they are most productive.



Protein crystallographers who build models move continually across a large field of basic facts, special features of the data and implications of the partial model(s) already built, looking for any and all opportunities to add another piece to their structure. There are several desiderata to working in this "opportunistic" mode of hypothesis formation: (a) The inference-generating rules and the strategies for their deployment should be separate, (b) the rules should be separate from the mechanics of the program in which they are embedded, and (c) the representation of the hypothesis space should be compatible with the kinds of hypothesis-generating rules available. The modularity of such a system would allow users to add or change rules for manipulating the database, as well as to investigate different solution strategies without having to make major modifications to the system.

### The CRYNALIS Architecture: The Blackboard

A problem-solving paradigm that meets the above specifications, to a large degree, is that of HEARSAY-II (see article *Speech.02*)--specifically with respect to the issues of knowledge integration and focus of attention. In Hearsay-II, an "iterative guess-building" process takes place: A number of different *knowledge sources* (facts, algorithms, heuristics) cooperate when working on various descriptions of the hypothesis. In order to use the knowledge sources efficiently, a global database--the "blackboard"--is constructed that contains the currently active hypothesis elements at all levels of current description. The decision to activate a particular knowledge source is not preestablished but depends on the current state of the solution and what available knowledge source is most likely to make further progress. The control is, to a large extent, determined by what has just been learned: A small change in the state of the "blackboard" may provide the preconditions to instantiate further knowledge sources (an illustration of this process in the context of electron density map interpretation is given below).

Figure 2 shows the types of data and hypotheses that are used in CRYNALIS. As in Hearsay-II, the hypotheses are represented in a hierarchical data structure. In our case the different information levels can be partitioned into three, distinctly different "panels," but the concept of a globally accessible space of hypotheses is essentially the same for both systems. Figure 2 also illustrates how knowledge sources (only a small subset is shown) play the same role as in Hearsay-II--adding, changing, or testing hypothesis elements on the blackboard. Further explanation of these diagrams is given in Engelmore & Nii, 1977. The processes of generating or modifying hypotheses and of invoking knowledge sources is nearly identical to those described for the AGE system (Nii & Aiello, 1978).

### Representation of Knowledge in the System

As mentioned above, there are many diverse sources of information used in protein structure inference. The problem of representing all the knowledge in a form that allows its cooperative and efficient use in the search for plausible hypotheses is of central concern to the developers of CRYNALIS. The system currently under development draws upon many concepts that have emerged in the design of other large knowledge-based systems--e.g., the use of production rules and blackboards. We describe here how these concepts have been adapted to our particular task.

Knowledge consists of facts, algorithms, and heuristics (rules of good guessing). Facts

required for protein structure inference are general physical, chemical, stereochemical, and crystallographic *constraints*. Typical factual knowledge stored in the system includes physical properties of the elements commonly found in proteins, the molecular structure and chemical properties of the twenty amino acids, the bond lengths, and the symmetry properties of various crystal structures. These facts are encoded as tables or as property lists attached to specific structural entities.

Algorithms and heuristics comprise both the formal and informal knowledge that generates or verifies hypothesis elements. The representation of this type of knowledge in CRYVALIS follows two general principles:

- 1) Decompose identifiable areas of knowledge into elementary units, where each unit increments the hypothesis when specified preconditions are met.
- 2) Represent the elementary units as *situation-action rules*.

To illustrate:

IF: the name of the current-residue is GLU, and  
the shape of the subgraph is forked, and  
the length of the subgraph is between 40 and 75, and  
the number of associated peaks of the subgraph is  
greater than 1

THEN: conclude that the subgraph is matched, and  
generate a new superatom on the blackboard,  
with the following properties:  
Type is 'side-chain'  
Belongs to current-residue  
Data-link to subgraph with certainty factor 500

Note that several actions may be performed for a given situation. Not shown here, but present in the LISP implementation of these rules, is a position in the rule for variable bindings, to avoid repetitious calculation of parameters appearing in several situation-action clauses. Also note that at least one of the actions of each rule is to place a token on an *event list*. In the actual implementation, the syntax of the "action" clause is represented as a single function. An example follows:

syntax: (<inference type> <element being changed> <att-value pairs>)

example: (SUBGRAPH.MATCHED (GENSUPATOM)((TYPE 'SIDECHAIN)(BELONGSTO  
CURRENT.RESIDUE)(DATALINK (SUBGRAPH . 500))))

In this example, an event, SUBGRAPH.MATCHED, will be generated and queued on the event list. The event-list is used by the interpreter (discussed in the next section) to determine what to do next, that is, which set of knowledge sources to invoke after the current event has been processed.

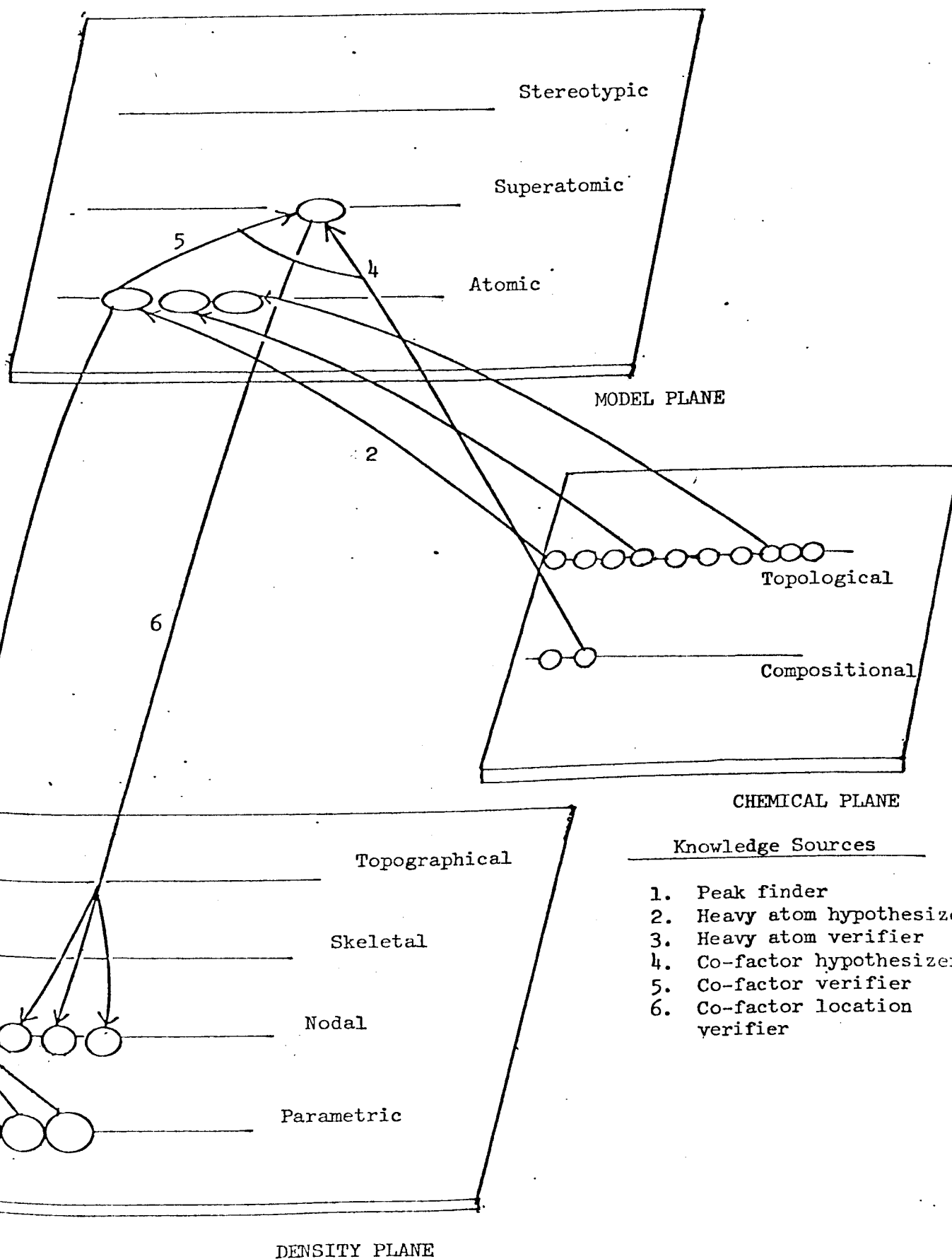
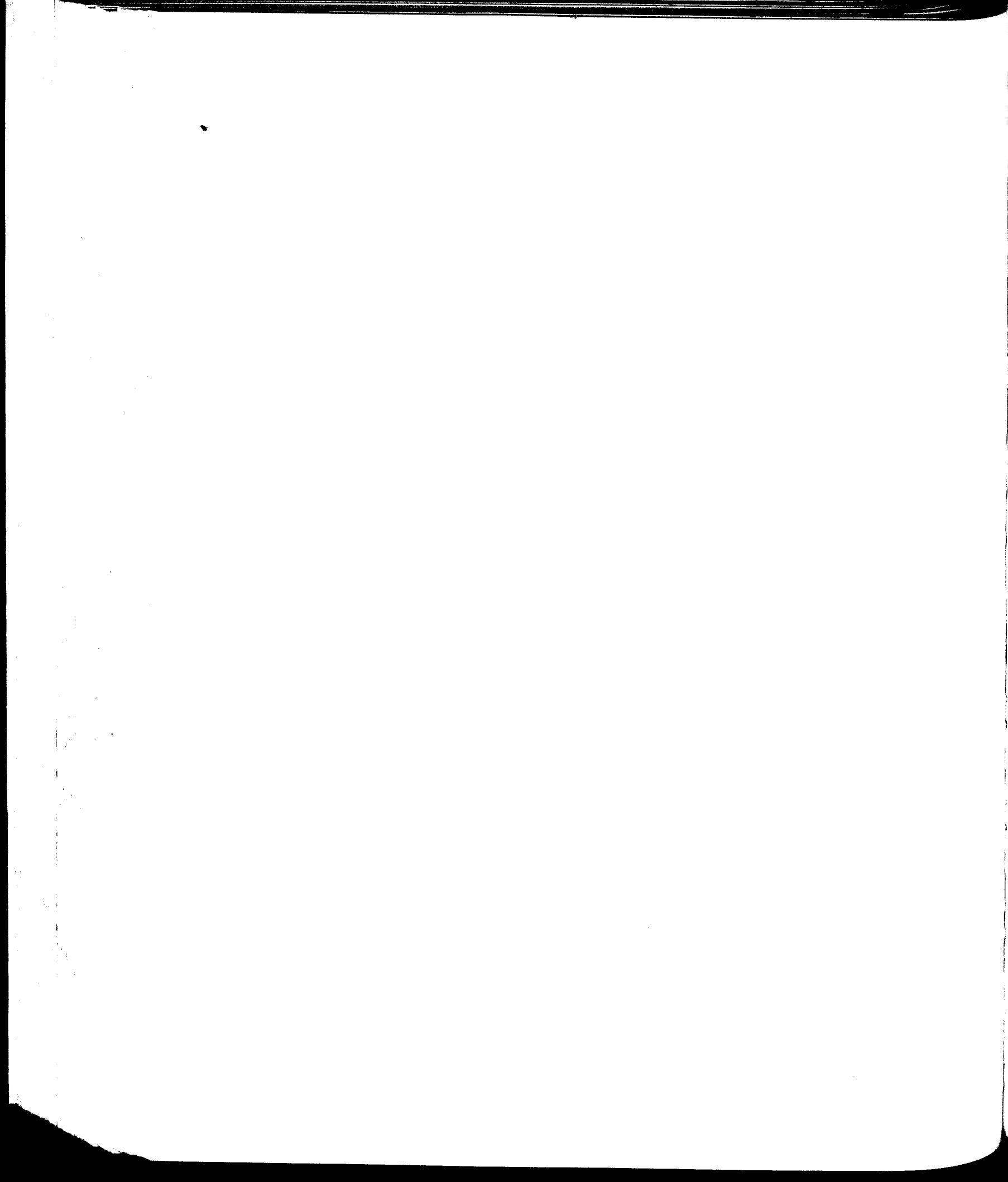


Figure 2. Panels of the CRYSLIS blackboard, and examples of the application of knowledge sources.



### Event-driven control

The CRYBALIS system uses an *event-driven control structure*. In this scheme, the current state of the hypothesis space determines what to do next. The monitor continually refers to a list of current events--the event-list--that is used to trigger those knowledge sources most likely to make further headway. As a knowledge source makes a change in the current hypothesis, it also places an item on the event-list to signify the type of change made. Thus, as events are drawn from the event-list for processing, new events are added, so that under normal conditions the monitor always has a means for choosing its next move.

The normal iterative cycle of problem solving uses the event-list to trigger knowledge sources, which create or change hypothesis elements and place new events on the event-lists. The system's behavior is *opportunistic*: it is guided primarily by what has been most recently discovered, rather than by the requirement to satisfy subgoals. An event-driven control structure was chosen partly to be efficient, in selecting appropriate knowledge sources, and partly to conform with the structure-modeling process normally employed by protein crystallographers.

### Rules

The formal and informal procedures that comprise our knowledge sources are expressed as rules, as discussed above. These rules are collected into sets, each set being judged appropriate to use when particular types of events occur. The events generally reflect the level at which the inference is being made, which in turn reflects the model's level of detail. The correspondence between *event classes* and rule sets is established by another set of rules, the *task rules*. The task rules are used to decide which KS or sequence of KSs to call in order to perform one of the typical tasks in building the structure--e.g., tracing the protein backbone between two anchor points. The decision is based on the state of the blackboard and the items on the event-list. The task rules thus form a second layer of rules, which directs the system's choice of knowledge sources for a given event, reflecting the system's knowledge of what it knows.

Once a task is either completed or fails, the system looks to a higher level of control to determine what to do next. At this higher level--the strategy level--the structure-building process can either try to solve the current subproblem by another method or shift attention to another region of the structure. Strategy level decisions are also expressed as rules and make use of the current state of the blackboard and event list. For example, one strategy rule is:

IF: the initialization task is complete, and  
the locations of two or more atoms are known  
(also called 'toeholds'), and  
these toeholds are separated by less than 6 residues  
in the amino acid sequence, and  
none of the intervening residues are identified from the data,

THEN: select the two-point chain-tracing task and focus on the  
subsequence bounded by the toeholds.

The part of the monitor that interprets and obeys the event rules may be likened to a middle-level project manager who knows which specialists to call in as new, partial solutions to a particular problem are discovered. Continuing the analogy, the middle-level manager occasionally gets stuck and needs help from higher level management. As mentioned earlier, some high-level decision (such as merging two or more events to produce a new event or shifting attention to another part of the blackboard), is required. This level of decision making is embodied in a set of strategy rules, which are used to direct the top-level flow of control. We thus have a completely rule-based control structure that employs three distinct levels of rules (or knowledge): the specialists, commonly called the knowledge sources; the task rules, representing knowledge about the capabilities of the specialists; and the strategy rules, which know when to use all available knowledge to solve the problem. Although this pyramidal structure of rules and meta-rules could continue indefinitely, the flexibility of knowledge deployment offered by our three-tiered system appears sufficient for this problem-solving system. Similar ideas in a simpler context have been explored by (Davis, 1976a) for the MYCIN system.

**System Performance -- an example**

To give some indication of the system's current level of performance, we present an annotated typescript in which a typical hypothesis formation task is completed. The example is the subproblem of extending the model from an "island of certainty," or anchor point, by using the crystallographic data to determine where to extend the model in space and by using the amino acid sequence to generate expectations of features that ought to be present in that region. The knowledge sources invoked in this example use an abstraction of the density map called *subgraphs*. A subgraph is a collection of segments obtained from a skeletonized density map, which hopefully matches an identifiable substructure in the protein--e.g., a side chain. The amino acid sequence assumed here is METHionine, LYSine, LYSine, TYRosine, etc. (the example uses data from the protein Rubredoxin). The example starts after passing control to a knowledge source called ANCHOR.TOEHOLD. The toehold of interest in this case is the sulphur atom in the methionine sidechain. This toehold is just a point in space and must be connected to the skeleton.

INFERENCE: EVENT-1 BY RULE 1 IN RULESET ANCHOR.TOEHOLD

EVENT NAME: TOEHOLD.ANCHORED  
CURRENT HYPOTHESIS ELEMENT: SA2  
NEW PROPERTIES: ((TYPE SIDECHAIN) (BELONGSTO (MET . 1))  
(SEGS (((1 SEG240) . 100) ((1 SEG238) . 100))) (MEMBERS (A3)))

The ANCHOR.TOEHOLD knowledge source has found subgraphs of the skeleton, but its limited knowledge cannot assign much certainty to the inference. The "real" matching of skeleton parts with expected residue is accomplished by MATCH.SDCHN. This knowledge source uses the shape of the subgraph, its length, the number of peaks associated with the candidate subgraph, and their heights. If a certainty factor (CF) of 500 or more is assigned, the sidechain is considered located (CF's have a range of -1000 to 1000; the CF combining function being the same as that used by MYCIN; see article C2).

INFERENCE: EVENT-2 BY RULE 3 IN RULESET MATCH.SDCHN

EVENT NAME: TOEHOLD  
CURRENT HYPOTHESIS ELEMENT: SA3  
NEW PROPERTIES: (SEGS (((1 SEG238) . 823) ((1 SEG240) . 555))))

If a sidechain is found, the trace tries to find the alpha carbon location by finding a peak of a certain type near the root of the sidechain. The KS used to propose an alpha carbon position is called POSSIBLE.CALPHA. The system assumes that the location of this peak is a more accurate guide than the skeleton for locating this class of atom.

INFERENCE: EVENT-3 BY RULE 5 IN RULESET POSSIBLE.CALPHA

EVENT NAME: C.ALPHA  
CURRENT HYPOTHESIS ELEMENT: A4  
NEW PROPERTIES: ((TYPE C) (NAME CA) (BELONGSTO (MET . 1))  
(D.PEAKS ((PK076 . 500))))

Once the toehold has been anchored, this trace becomes essentially a generate-and-test search, heavily constrained by the sequence. The basic control cycle for the trace is: Propose a sidechain, match it; propose a peptide, match that; and loop until a match fails. Sometimes the carbonyl group present in each peptide will appear as a small sidechain. If this happens, the proposed peptide will extend only from the last sidechain up to this pseudo-sidechain, and the peptide will fail to match. This failure prompts the system to try matching the "sidechain" as a carbonyl. Success of this match would mean that only half of the peptide has been found; the system can then propose a larger peptide which contains the old one, and proceed as before.

INFERENCE: EVENT-4 BY RULE 4 IN RULESET MATCH.PEPTIDE

EVENT NAME: PEPTIDE  
 CURRENT HYPOTHESIS ELEMENT: SA4  
 NEW PROPERTIES: ((TYPE PEPTIDE) (BELONGSTO (MET . 1))  
 (SEGS (((SEG6 SEG8) . 84))) (PEAKS (PK076 PK078)))

INFERENCE: EVENT-5 BY RULE 5 IN RULESET MATCH.CARBONYL.SC

EVENT NAME: CARBONYL.FOUND  
 CURRENT HYPOTHESIS ELEMENT: A5  
 NEW PROPERTIES: ((TYPE CO) (NAME CARBONYL) (BELONGSTO (MET . 1))  
 (SEGS (((1 SEG5) . 581))) (PEAKS (PK036)))

INFERENCE: EVENT-6 BY RULE 4 IN RULESET MATCH.PEPTIDE

EVENT NAME: PEPTIDE  
 CURRENT HYPOTHESIS ELEMENT: SA4  
 NEW PROPERTIES: ((SEGS (((SEG6 SEG8 SEG10) . 420)))  
 (PEAKS (PK076 PK078 PK036)))

INFERENCE: EVENT-7 BY RULE 7 IN RULESET MATCH.SDCHN

EVENT NAME: SIDECHAIN  
 CURRENT HYPOTHESIS ELEMENT: SA6  
 NEW PROPERTIES: ((TYPE SIDECHAIN) (BELONGSTO (LYS . 2))  
 (SEGS (((1 SEG242) . 527)))

INFERENCE: EVENT-8 BY RULE 5 IN RULESET POSSIBLE.CALPHA

EVENT NAME: C.ALPHA  
 CURRENT HYPOTHESIS ELEMENT: A6  
 NEW PROPERTIES: ((TYPE C) (NAME CA) (BELONGSTO (LYS . 2))  
 (D.PEAKS ((PK078 . 500)))

INFERENCE: EVENT-9 BY RULE 4 IN RULESET MATCH.PEPTIDE

EVENT NAME: PEPTIDE  
 CURRENT HYPOTHESIS ELEMENT: SA6  
 NEW PROPERTIES: ((TYPE PEPTIDE) (BELONGSTO (LYS . 2))  
 (SEGS (((SEG232 SEG16) . 600))) (PEAKS (PK017 PK125)))



*(Three more events, similar to those preceding, have been omitted.)*

INFERENCE: EVENT-13 BY RULE 6 IN RULESET MATCH.SDCHN

EVENT NAME: SIDECHAIN  
CURRENT HYPOTHESIS ELEMENT: SA9  
NEW PROPERTIES: ((TYPE SIDECHAIN) (BELONGSTO (TYR . 4))  
(SEGS (((6 SEG212 SEG40 SEG36 SEG35 SEG228) . 502))))

The matching cycle terminates in one of two ways. If the skeleton becomes so overconnected that the access function cannot propose the next subgraph (sidechain or peptide), the trace fails; or if the certainty of a match is too low and there are no rules to save the situation, the trace fails. Upon termination, one final knowledge source is called to link together hypothesis elements belonging to the same residue, creating an organizing "backbone."

INFERENCE: EVENT-14 BY RULE 3 IN RULESET TRACE.CLEANUP

EVENT NAME: LINK-CA-TO-PEPTIDE  
CURRENT HYPOTHESIS ELEMENT: SA4  
NEW PROPERTIES: ((MEMBERS (A4)))

*(Two more events, like the preceding one are omitted here.)*

INFERENCE: EVENT-17 BY RULE 7 IN RULESET TRACE.CLEANUP

EVENT NAME: BACKBONE  
CURRENT HYPOTHESIS ELEMENT: ST1  
NEW PROPERTIES: ((TYPE BACKBONE) (CF 511) (DIRECTION 1)  
(RANGE (1 . 4)) (MEMBERS (SA1 SA2 SA3 SA4 SA5 SA6 SA7 NIL)))

### Summary

At the present time, CRYSLIS is capable of performing only a small portion of the total task of electron density map interpretation. The development and implementation of all the knowledge sources required for the complete task is a long-term effort. CRYSLIS currently contains a relatively small knowledge base that permits the interpretation of portions of high-quality, high-resolution (2.0 Angstroms or better) electron density maps. The system is expected to evolve toward an extensive knowledge-based problem solver capable of complete interpretation of medium-quality, medium-resolution (2 to 2.5 Ang.) electron density maps. Although CRYSLIS is not yet worthy of serious attention by the protein-crystallographic community, its defects lie primarily in its relatively meager knowledge base and not in its design. As new knowledge sources are added to the system, its level of performance is expected to rise to the point where its use will be a significant aid in the determination of new protein structures.

**References**

See Davis, 1976a, Engelmores & Nii, 1977, Erman, 1976, Feigenbaum, Engelmores, & Johnson, 1977, Hayes-Roth & Lesser, 1976, and Nii & Aiello, 1978.

## A5. Artificial Intelligence and Organic Synthesis

### The Problem of Organic Synthesis

Organic chemists are interested in the logic and beauty of organic synthesis for its own sake, just as computer scientists are interested in the correctness and elegance of their programs. Organic synthesis is also central to the creation of new chemical products and more efficient processes for manufacturing old products. Therefore, there is great interest among both academic and industrial chemists in providing new tools to aid in finding new synthetic routes.

A synthesis problem begins with the structural description of a compound that someone wants synthesized, often because the compound has useful properties (e.g., a drug or a vitamin). Synthesis can also be a definitive confirmation of a postulated structure for an unknown compound in an analysis problem, since the synthesized compound and the unknown compound will, if identical, produce identical test results.

Chemists use the computer and AI techniques to systematically explore the *synthesis tree* and to help organize the immense body of available knowledge about chemical reactions. This approach, of exhaustively exploring the interesting branches of the synthesis tree, was called the logic-centered approach by Corey and Wipke, who first explored computer-aided organic synthesis. "Interesting" branches are those most likely to produce the desired result. "Interesting" is an extremely difficult concept to define and to cast into an algorithm, therefore, for now, the search must be guided interactively by the chemist. Some of the relevant considerations are: the efficiency of a reaction, the cost of materials, and the difficulty of meeting the experimental conditions that support a reaction.

The chemist represents the "target" structure graphically, as discussed in D, and relates it to simpler chemicals via known chemical reactions. He relates those to still simpler ones, until he reaches a set of compounds, comparable to starting materials readily available from chemical supply houses or which can be easily synthesized in a few steps in the laboratory. One plan for synthesizing the compound, called a "synthetic route," may involve dozens of separate reactions. If the molecule is at all complicated there are an immense number of distinct synthetic routes. For example, it has been calculated that for a simple steroid (about 20 atoms), over  $2.4 \times 10^{18}$  direct routes are possible Hendrickson (1975).

Synthetic routes can be visualized by means of an AND/OR tree (see section ....). The tree descends from the goal node, the target molecule, to the terminal nodes, equivalent to the starting materials. The branches connecting the nodes are chemical reactions. Since a synthesis plan involves combining compounds in reactions, the AND-links of the tree are present in any one synthesis route; alternative ways of making a compound anywhere within the plan are represented by OR-nodes.

### The Three Major Programs

There are three major programs in computer-aided organic synthesis. The earliest is LHASA (Logic and Heuristics Applied to Synthetic Analysis), which was written by Corey and Wipke at Harvard and is maintained at Harvard by Corey and his research group. SECS

(Simulation and Evaluation of Chemical Synthesis) is an outgrowth of LHASA, written by Wipke and maintained by Wipke and his research group at the University of California at Santa Cruz. It extended the LHASA paradigm by the inclusion of stereochemical and conformational information into all aspects of the computer program. The third major program is SYNCHEM (Synthetic Chemistry), written and maintained by H. L. Gelernter and his research group at the State University of New York at Stony Brook. The main features of these three programs are summarized in Table 3.

Table 3  
Chemical Synthesis Programs

Program	Principal Designer	Language \ Computers	Main Features
LHASA	E. J. Corey	Fortran IV PDP-10	Large procedural knowledge base of "transforms." Interactive, high-performance.
SECS	W. T. Wipke	Fortran IV many	Separate knowledge base of "transforms" with special interactive language for defining new ones (ALCHEM). Interactive graphics, and high-performance.
SYNCHEM	H. Gelernter	PL/1 IBM 370	Motivated by AI search problems. Evaluation during search done by the program not by a chemist.

Since SECS was designed to extend the methods in LHASA, much of the discussion of SECS is true of LHASA. However, SECS has additional features that are of interest to computer scientists. Of the three, only SECS is demonstrably machine independent.

#### Two Different Approaches

A major distinction between SECS (and LHASA) and SYNCHEM is that the former is oriented to high performance, while SYNCHEM is oriented more to AI issues. As a consequence of this fact and the fact that chemists' intuitions about "interesting" pathways are hard to define, SECS relies on a chemist's interacting with the program. SYNCHEM, on the other hand, searches the space without interactive guidance from a chemist. (This is not to say that SECS and LHASA lack interest or that SYNCHEM is incapable of high performance.)

In operational terms, the main difference is whether the evaluation function for the search procedure is explicitly given to the program and used without guidance from the chemist (SYNCHEM) or whether the evaluation function is not explicitly given to the program (SECS and LHASA). These are called the noninteractive and interactive approaches below. Although SECS can be reconfigured to run noninteractively, it is not the preferred way to run; a chemist's guidance gives better results.

### The Chemical Knowledge Base

The primary item of knowledge in chemical synthesis is the chemical reaction--a rule describing a situation in which a change can occur (to a molecular structure) plus a description of that change. For example, the reaction shown in Figure 2 describes a change to a molecule containing the substructure  $O=C-C-C=O$  in the presence of the reagent oxalyl chloride.



Figure 2. Graphical representation of a chemical reaction.

To design a synthesis route from starting materials to target molecule, knowledge of reactions can be used in either of two ways:

1. **Forward direction:** Apply known reactions to starting materials, then to the products of those reactions, the products of products, etc., until the target is reached. The combinatorics of this approach make it impossible in practice because there are thousands of possible starting compounds and only one target.
2. **Reverse direction:** Starting with the target molecule, determine which reactions might produce it. Then look for ways to make the precursors, and the precursors of precursors, etc., until starting materials are reached. Storing the reactions in the reverse direction makes it easier to search the tree of possible pathways.

All three programs have a large knowledge base of *reverse chemical reactions* called transforms, similar to production rules of the condition-action form, with the left-hand side being a substructure pattern to be matched in the target structure (or intermediate structure) and the right-hand side being a description of precursors that will produce the goal structure under specified reaction conditions. Figure 3 shows the reverse reaction plus all the associated conditions needed for the SECS program to work with the reaction in Figure 2. Each of the three projects have dealt with the problems of constructing a knowledge base in very different ways.

1. The LHASA knowledge base is a set of procedures. Although it contains very sophisticated chemistry knowledge, it is difficult to modify.

2. The SECS knowledge base contains about 400 separate transforms. New transforms can be defined by users and entered into the knowledge base without changes to the program. Because of its clarity, it is used for illustration and is discussed in detail below.

3. The SYNCHEM knowledge base is a library of reactions that can be updated by chemists without reprogramming. Each reaction is automatically compiled into a reverse reaction. In addition, the knowledge base contains a large library of starting compounds that are available commercially.

Each of the SECS transforms is stored on external storage independent of the SECS program; this feature enables the knowledge base to be tailored to a specific problem domain. Further, the number and complexity of transforms is not limited by the size of core memory. A simple, flexible language, called ALCHEM, is provided in which chemists can enter new transforms into the knowledge base.

ALCHEM embodies a model of what information is needed in order to adequately describe a reaction. According to this model, a transform consists of the following six sections:

- (1) Transform name.
- (2) Substructure key or pattern to be matched.
- (3) Character--used to help judge the relevance to strategic planning.
- (4) Scope and limitations.
- (5) Reaction conditions--which must not be violated by the remainder of the molecule containing the substructure key.
- (6) Manipulation statements--describing the graph transformations to be performed.

This will be clarified below with an example.

In the reaction shown in Figure 2, one of the Oxygens double bonded to Carbon is replaced by a single bond to a Chlorine. To go from a graphical representation of a synthetic reaction to the graphical representation of a SECS transform, we reverse the left- and right-hand sides and specify additional important conditions. Using the ALCHEM language, the Chemist could interactively enter the following representation of this transform.

```

comment      ; Chloroenones
comment      ; O=C-C=C-CL GOES TO*
comment      ; Reagent: Oxaly1 Chloride
comment      ; Ref: C.H. Heathcock, R.D.Clark
comment      ; Journal of Organic Chemistry,
comment      ; 41,636-643 (1976).

transform name      CHLOR-ENONE
substructure key    O=C-C=C-CL  <1 = 2 - 3 = 4 - 5>
Priority            100

Character          CHARACTER ALTERS GROUP

Scope and Limitations
                  IF ACID IS OFF PATH THEN KILL
                  IF ESTER IS OFF PATH THEN KILL
                  IF HYDROGEN IS ALPHA TO ATOM 4 THEN
                    BEGIN
                    IF HYDROGEN IS ALPHA TO ATOM2
                    THEN SUBTRACT 75 FROM PRIORITY
                    DONE

Manipulation Statements
                  BREAK BOND 3
                  DELETE ATOM 5
                  ADD 0 OF ORDER 2 to ATOM 4.

```

\*In the actual reaction, of course, the chlorinated compound COMES FROM the precursor.

Referring to the manipulation statements, "BREAK BOND 3" refers to the third bond from the left in the substructure key; the double bond between the two carbons is reduced to a single bond. Similarly, "DELETE ATOM 5" refers to the Chlorine atom CL, the fifth atom from the left. When the program is actually run, a compiler called SYNCOM translates the ALCHEM statements into machine readable form before the program is run.

### A Brief Description of SECS

SECS and LHASA have been designed to divide the work between the chemist and the computer in the most optimal way. In a recent paper Wipke et al. (1977),

explain their philosophy.

Our performance goal for the program was that the program should be able to help a chemist find many more good and innovative syntheses than the chemist could working alone. Because of the complexity of the problem domain, we felt the chemist and computer working together with each assigned tasks for which they are best suited, and with efficient interaction between the two, would be more effective than either working alone. Our goal was not to replace the chemist, but to augment the chemist's problem solving capabilities.

### Graphics

The chemist communicates with the SECS program using a graphics terminal with a CRT, a mini-computer, a keyboard, and a light pen. Using the pen, the chemist draws on the screen the graphical structure of the "target" molecule to be synthesized. Much effort has gone into human engineering. The SECS graphics routines are designed to be as near as possible to the chemists' normal modes of thought, which is the structure diagram or the molecular model. There are similar facilities in LHASA. By convention, hydrogen atoms are suppressed, as discussed above. Another convention is that only noncarbon atoms (called "heteroatoms") are labeled. This is useful since the majority of nonhydrogen atoms in organic molecules are carbon.

### Application of a Transform

Applying a transform is not simply a matter of matching the substructure key to a molecule and, if the subgraph fits, executing the graph manipulation statements. The scope and limitations determine much of the context in which the transform applies. Also, it is necessary to check three-dimensional information and electronic environment information (that is, the tendency of the atoms in the molecule to be positively or negatively charged) in order to make an accurate assessment about whether a transform applies. A common situation in synthetic chemistry is that we have a functional group to modify and a reagent to change it, but the functional group is hindered (spatially) by another functional group or another portion of the molecule. In such cases, the reagent molecules cannot react with the group and change it; although they might in other spatial contexts.

Without the three-dimensional information given by the so-called "model-building" routines, the program has no way of knowing that the transform cannot apply. After the spatial modelling has been done, the program can perceive that even though the required functional group is present, the transform cannot be applied directly because it is inaccessible to the reagent molecules. If the transform is very high priority, a means-end analysis can be done to find ways of altering the molecule, so that the given functional group is accessible.

### Strategies for Applying Transforms

Like the transforms in the knowledge base, the strategies are stored separately from the program. Strategies is one of the major areas of exploration and research for all three of the programs. Examples of strategies used to choose which transforms to apply in a given situation are shown below (Wipke et al., yr.?):

- 1) Try to cleave the target into two nearly equal fragments.
- 2) Simplify rings of eight or more members by joining the opposite sides to form two smaller rings.
- 3) Try to break bonds along elements of symmetry to generate identical fragments.



Given a molecule to synthesize, the strategies are used to create goals. Operationally, the goals are the instantiation of a strategy in a particular instance. The chemist also has access to these goals and can add, modify, or delete them interactively before letting the program attempt to satisfy them.

The problem of how to organize the interaction between a rule base and strategies is an active area of AI research. It has been discussed, for example, by Davis (Davis, 1976a) in the context of using meta-rules to guide the invocation of domain rules by MYCIN. There are several issues of interest to AI in the implementation of chemical synthesis strategies. First is how much strategy to use. If the strategies capture too many of the chemists' strong biases about fruitful and unfruitful reaction pathways, then the program will be unlikely to discover, or help the chemist discover, any novel synthetic routes. The whole point of the logic-centered approach is to explore pathways outside the chemist's normal intuitive beam. On the other hand, if the strategies are too weak then the chemist is swamped with too much information; the program then explores pathways that a chemist regards as absurd.

Second is the problem of how to organize the rules so that the strategy component determines which to apply. In SECS the strategy component uses the "character" slot of the transform (see Figure 3 above). If the strategy is to cleave a large ring, then the strategy component will look at the statement of the character of each transform to see which ones break rings. Those that do are candidates to be applied to the molecule. However, this is merely a first filter; the transform must be examined in detail to see if it applies to the molecule in question.

Third are the issues of what knowledge should go into the strategy rules and what should go into the rule base of transforms. Strategy rules can save time, at the possible expense of missing pathways that are unlikely but that would still work. For example, if the chemist has a year to work on a particular molecule, to explore as many of its possible synthetic pathways as he can, then he will employ quite a different set of strategies than if he has only a few months for the problem.

#### A Brief Description of SYNCHEM

The aims of Gelernter's group on SYNCHEM are stated very clearly in Gelernter et al., 1977:

Extraordinarily rapid progress during the early stages of an attack on a new problem area is a rather common occurrence in AI research; it merely signifies that the test cases with which the system has been challenged are below the level of difficulty where combinatorial explosion of the number of pathways in the problem space sets in....It is the goal of AI research to move that threshold higher and higher on the scale of problem complexity through the introduction of heuristics--heuristics to reduce the rate of growth of the solution tree, heuristics to guide the development of the tree so that it will be rich in pathways leading to satisfactory problem solutions, and heuristics to direct the search to the "best" of these pathways.

SYNCHEM is noninteractive. The molecule to be synthesized is input, and the program uses heuristic search to look for the best synthetic route. The program decides which node of the tree to develop further, by estimating the "cost" of reaching the goal from that node plus the estimated "cost" of reaching that node from starting materials. One of the interesting AI issues is that the program's definition of "cost" depends on the context of the problem as well as on static features such as efficiency of reactions, the monetary cost of materials, etc. For example, costs are measured differently in an exploratory research context than in an industrial production context.

The long-range hope of the SYNCHEM group is that the study of AI in this domain will lead to new insights in AI and also eventually to a noninteractive system that will be of use to chemists.

The first version of SYNCHEM, written largely by N. S. Sridharan, was operational between 1971 and 1974. SYNCHEM2 supersedes it and contains many improvements, some of which are discussed below. One improvement is a switch from Wiswesser Linear Notation (WLN) for molecular structures to a more manageable linear notation. As a result, however, the catalog of starting materials is no longer available to the evaluation function, so that weaker criteria are now used for detecting a starting material.

SYNCHEM2 has been designed so that further changes in the representation can be made easily, with minimum reprogramming. One of the main drawbacks of the original SYNCHEM was that it entirely neglected stereochemistry. SYNCHEM2 now incorporates stereochemistry into its representation of molecules and into its transform evaluation rules. The representation is flexible enough to include electronic and conformational (roughly bond lengths, bond angles, and other three-dimensional information) information. The format for specifying a transform, which had been a simple fixed-field input form, was redesigned to be similar to the ALCHEM facility in SECS.

Transforms were always applied serially in SYNCHEM, that is, to one functional group at a time in a molecule. A new feature in SYNCHEM2, called "multiple match," allows the program to apply transforms more intelligently to all the appropriate functional groups in a molecule. More specifically, the new program now recognizes that multiple occurrences of a functional group, under certain circumstances, can all be transformed by a reaction.

#### SYNCHEM Solution Evaluation

The following quotation (Gelernter et al., 1977)

illustrates the difference between organic synthesis and a more familiar domain such as theorem proving.

In particular, unlike much of the earlier work in problem-solving....where any valid sequence of transformations from premises to goal provided an acceptable solution, we were not to be satisfied by an indicated synthesis route of very low yield, or one requiring difficult or inefficient separations of goal molecules from by-products along the way, at least not before the machine had tried and failed to find a more efficient procedure of higher yield....It is the question of

relative merit of proposed solutions under the constraints of the problem that represents a substantial departure from most of the work reported in the literature of artificial intelligence.

The complexities of the domain are highlighted by the fate of one of the most significant results produced by the program. SYNCHEM proposed a synthetic route for a naturally occurring antibiotic that was at that time under development by A. R. Rinehart's group at the University of Illinois. The route was considered interesting enough to merit a laboratory investigation. However, the laboratory attempt failed. One of the crucial steps in the synthesis route could not be accomplished in the laboratory and the proposed route had to be reluctantly abandoned. No successful routes to the molecule have yet been found. All synthetic routes, whether proposed by a computer program like SYNCHEM or by a person, are provisional until they can be verified by experiment.

### SYNCHEM Search Strategy

SYNCHEM'S search strategy algorithm first expands the goal node to find all its precursors. Next it computes the cost of reaching the target molecule from the precursors, taking into account the efficiency and difficulty of the reactions. It also estimates the difficulty of synthesizing the precursor nodes from the available starting materials. Subgoal selection criteria are a function of both the accumulated heuristic estimates of reaction merit and yield along the path from subgoal to goal, and of a prediction of the probable reaction merit and yield along the best path from starting materials to the subgoal. SYNCHEM updates the merit ratings with information associated with each intermediate structure. Merit, as mentioned above, is based on most recent estimates of compound complexity (i.e., difficulty in synthesizing it) and reaction path merit (yield, cost, etc.) after each cycle of subgoal generation. The selection of a new subgoal always begins with a new scan of the tree from the top. It is a best-first procedure: If newly acquired information changes the ratings for subgoals, the next subgoal selected can lie on a completely different branch of the tree. In this way, the program will never develop an unfortunate choice (pathway down to starting materials) before backtracking and exploring more fruitful branches.

### Summary

Computer-aided chemical synthesis is a potentially powerful new tool for both research and industrial chemists. The utility of any of the programs discussed here critically depends on the size and accuracy of their knowledge base of organic chemical reactions. Although far from complete, the knowledge bases now contain highly detailed descriptions of numerous synthetic reactions. All of the programs have convincingly demonstrated their ability to find plausible synthetic routes for important organic materials, often in less time than chemists working alone. The SECS program is known to have a user community of chemists in Europe and North America, who add new transforms as well as use the program for synthesis planning. The effort spent on human engineering for chemists has made it possible for chemists to use the program effectively (and want to use it) and independently of the program's designers. One of the long-range hopes of chemists and computer scientists working in computer-aided organic synthesis is that this work on knowledge bases will lead to an improved classification of chemical reactions.

Because the heuristic search paradigm fits the synthesis planning problem well, AI research has had much to offer. In addition, current AI work on knowledge-based expert systems provides concepts and tools for representation and management of these large, ever changing sets of chemical facts and relations. In return, the complexity of this problem area has challenged current AI methods and has led to the exploration of new ideas for search and planning.

#### References

See Banks (1976), Cory & Wipke (1969), Gelernter et al. (1977), Gund, Andose, & Rhodes (1977), Kernighan & Ritchie (1978), Streitwieser & Heathcock (1976), and Wipke et al. (1977).

## References

- Banks, J. E. **Naming Organic Compounds**. Philadelphia: W. B. Saunders Co., 1976.
- Brown, H., & Masinter, L. An Algorithm for the Construction of the Graphs of Organic Molecules. **Discrete Mathematics**, 1974, 8, 227.
- Brown, H., Masinter, L., & Hjelmeland, L. Constructive Graph Labeling Using Double Cosets. **Discrete mathematics**, 1974, 7, 1.
- Buchanan, B. G. Scientific theory formation by computer. **Proceedings of NATO Advanced Study Institute on Computer Oriented Learning Processes**, Noordhoff, Leyden, 1976.
- Buchanan, B. G., Smith, D. H., White, W. C., Gritter, R. J., Feigenbaum, E. A., Lederberg, J., & Djerassi, C. Application of AI for Chemical Inference XXII. Automatic rule formation in mass spectrometry by means of the Meta-DENDRAL program. **Journal of the American Chemical Society**, 1976, 98(20), 6168-6178.
- Buchanan, B. G., Sutherland, G. L., & Feigenbaum, E. A. Heuristic DENDRAL: A Program for Generating Explanatory Hypotheses in Organic Chemistry. In B. Meltzer & D. Michie (Eds.), **Machine Intelligence 4**. Edinburgh: University Press, 1969.
- Buchanan, B. G. MI 5
- Buchanan, B. G., 1976
- Buchanan, B. G., DENDRAL
- Buchanan, B. G., DENDRAL
- Buchanan, B. G., Duffield, A. M., Robertson, A. V., An Application of Artificial Intelligence to the Interpretation of Mass Spectra. In G. W. A. Milne (Ed.), **Mass Spectrometry Techniques and Appliances**. New York: John Wiley & Sons, 1971. P. 121.
- Buchs, A., Delfino, A. B., Duffield, A. M., Djerassi, C., Buchanan, B., Feigenbaum E. A., & Lederberg, J. Applications of Artificial Intelligence for Chemical Inference VI. Approach to a General Method of Interpreting Low Resolution Mass Spectra with a Computer. **Helvetica Chimica Acta**, 1970, 53, 1394.
- Buchs, A., Duffield, A. M., Schroll, G., Djerassi, C., Delfino, A. B., Buchanan, B. G., Sutherland, G. L., Feigenbaum, E. A., & Lederberg, J. Applications of Artificial Intelligence For Chemical Inference IV. Saturated Amines Diagnosed by Their Low Resolution Mass Spectra and Nuclear Magnetic Resonance Spectra. **Journal of the American Chemical Society**, 1970, 92, 6831.
- Carhart, R. E., & Smith, D. H. Applications of Artificial Intelligence for Chemical Inference XX. *Intelligent Use of Constraints in Computer-Assisted Structure Elucidation*. **Computers and Chemistry**, 1976, 1, 79.

- Carhart, R. E., Smith, D. H., Brown, H., & Djerassi, C. Applications of Artificial Intelligence for Chemical Inference XVII. An Approach to Computer-Assisted Elucidation of Molecular Structure. *Journal of the American Chemical Society*, 1975, 97, 5755.
- Cheer, C., Smith, D. H., Djerassi, C., Tursch, B., Braekman, J. C., & Daloz, D. Applications of Artificial Intelligence for Chemical Inference XXI. Chemical Studies of Marine Invertebrates XVII. The Computer-Assisted Identification of [+-]Palustrol in the Marine Organism *Cespitularia* sp., aff. *Subviridis*. *Tetrahedron*, 1976, 32, 1807.
- Churchman, C. W., & Buchanan, B. G. On the Design of Inductive Systems: Some Philosophical Problems. *British Journal for the Philosophy of Science*, 1969, 20, 311-323.
- Cory, E. J., & Wipke, W. T. Computer assisted design of complex organic synthesis. *Science*, 1969, 166, 178-192.
- Davis, R. Applications of Meta-Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases, Stanford AI Lab Memo AIM-283, AI Lab, Stanford University, 1976. (a)
- Davis, R. Applications of meta-level knowledge to the construction, maintenance, and use of large knowledge bases. Doctoral dissertation (STAN-CS-76-552), Stanford University, July 1976. (b)
- Duffield, A. M., Robertson, A. V., Djerassi, C., Buchanan, B. G., Sutherland, G. L., Feigenbaum, E. A., & Lederberg, J. Application of Artificial Intelligence for Chemical Inference II. Interpretation of Low Resolution Mass Spectra of Ketones. *Journal of the American Chemical Society*, 91(11), 1969.
- Engelmore, R. S., & Nil, H. P. A Knowledge-based System for the Interpretation of Protein X-ray Crystallographic Data, Stanford Heuristic Programming Project Rep. HPP-77-2, Computer Science Dept., Stanford University, 1977.
- Erman, L. D. Overview of the Hearsay Speech Understanding Research. Working Papers in Speech Recognition-IV--The HEARSAY II System, Carnegie-Mellon University, Computer Science Speech Group, 1976.
- Feigenbaum, E. A., Engelmore, R. S., & Johnson, C. K. A Correlation Between Crystallographic Computing and Artificial Intelligence Research. *Acta Crystallographica*, 1977, A33, 13.
- Feigenbaum, E. A., & Buchanan, B. Heuristic DENDRAL: A Program for Generating Explanatory Hypotheses in Organic Chemistry. In B. J. Kinariwala & F. F. Kuo (Eds.), *Proc. Hawaii Int. Conf. on System Sciences*, University of Hawaii Press, 1968.
- Feigenbaum, E. A., *Machine Intelligence* 6
- Gelernter, H. L., Sanders, A. F., Larsen, D. L., Agarwal, K. K., Boivie, R. H., Spritzer, G. A., & Searleman, J. E. Empirical explorations of SYNCHEM. *Science*, 1977, 197(4308), 1041-1049.

- Gund, P., Andose, J. D., & Rhodes, J. B. Computer assisted analysis in drug research. In W. T. Wipke & W. J. Howe (Eds.), *Computer-assisted Organic Synthesis*. Washington, D. C.: American Chemical Society, 1977. Pp. 179-187.
- Hayes-Roth, F., & Lesser, V. R. Focus of attention in a distributed-logic speech understanding system. *Proc. of IEEE, Int. Conf. on ASSP*, Philadelphia, Pa., 1976.
- Hedrick, C. A computer program to learn production systems using a semantic net. Doctoral dissertation, Graduate School of Industrial Admin., Carnegie-Mellon, July 1974.
- Hendrickson ??
- Hunt, E. B. *Artificial Intelligence*. New York: Academic Press, 1975.
- Jurs, P. C. Chemical data interpretation using pattern recognition techniques. In W. T. Wipke, S. R. Heller, R. J. Feldman, & E. Hyde, (Eds.), *Computer Representation and Manipulation of Chemical Information*. New York: Wiley-Interscience, 1974. Pp. 265-285.
- Kernighan, B. W., & Ritchie, D. M. *The C Programming Language*. New Jersey: Prentice Hall, 1978.
- Lederberg, J. *Computation of Molecular Formulas for Mass Spectrometry*. San Francisco: Holden-Day, 1964. (a)
- Lederberg, J. DENDRAL-64: A System for Computer Construction, Enumeration and Notation of Organic Molecules as Tree Structures and Cyclic Graphs. Part I. Notational algorithm for tree structures. NASA CR.57029, 1964. (b)
- Lederberg, J. DENDRAL-64: Part II. Topology of cyclic graphs. NASA CR.68898, 1965. (a)
- Lederberg, J. Systematics of organic molecules, graph topology and Hamilton circuits. A general outline of the DENDRAL system. NASA CR-48899, 1965. (b)
- Lederberg, J. Topological Mapping of Organic Molecules. *Proc. Nat. Acad. Sci.*, 1965, 53(1), 134-139. (c)
- Lederberg, J. DENDRAL-64: Part III. Complete chemical graphs; embedding rings in trees. TR, NASA, 1969.
- Lederberg, J., & Feigenbaum, E. A. Mechanization of Inductive Inference in Organic Chemistry. In B. Kleinmuntz (Ed.), *Formal Representations for Human Judgment*. New York: John Wiley, 1968.
- Masinter, L., Sridharan, N. S., Carhart, R., & Smith, D. H. Application of Artificial Intelligence for Chemical Inference XII: Exhaustive Generation of Cyclic and Acyclic Isomers. *Journal of the American Chemical Society*, 1974, 96, 7702.
- Minsky, M. Steps toward artificial intelligence. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963. Pp. 406-450.

- Mitchell, T. M. Version spaces: An approach to rule revision during rule induction. *IJCAI* 5, 1977, 305-310. (Also Heuristic Programming Project Memo HPP-77-13, Computer Science Dept., Stanford University, 1977.)
- Mitchell, T. M., & Schwenzler, G. M. Applications of AI for chemical inference, XXV: A computer program for automated empirical  $^{13}\text{C}$  NMR rule formation. *Organic Magnetic Resonance*, 1978, 11(8), 378.
- Morrill, K., Smith, D. H., & Djerassi, C. Computer-assisted Analysis of the High Resolution Mass Spectra of Macrolide Antibiotics. *Organic Mass Spectrometry*, 1977, XXX,
- Nii, H. P., & Aiello, N. AGE (Attempt to Generalize): Profile of the AGE-0 System, Heuristic Programming Project Working Paper HPP-78-5, Computer Science Dept., Stanford University, June 1978.
- Samuel, A. L. Some studies of machine learning using the game of checkers. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill, 1963. Pp. 71-105.
- Schroll, G., Duffield, A. M., Djerassi, C., Buchanan, B. G., Sutherland, G. L., Feigenbaum, E. A., & Lederberg, J. Application of Artificial Intelligence for Chemical Inference III. Aliphatic Ethers Diagnosed by Their Low Resolution Mass Spectra and NMR Data. *Journal of the American Chemical Society*, 1969, 91, 7440.
- Sheikh, Y. M., Buchs, A., Delfino, A. B., Schroll, G., Duffield, A. M., Djerassi, C., Buchanan, B., Sutherland, G. L., Feigenbaum, E. A., & Lederberg, J. Applications of Artificial Intelligence for Chemical Inference V. An Approach to the Computer Generation of Cyclic Structures. Differentiation Between All the Possible Isomeric Ketones of Composition,  $\text{C}_6\text{H}_{10}\text{O}$ . *Organic Mass Spectrometry*, 1970, 4, 493.
- Simon, H., & Lea, G. Problem solving and rule induction: A unified view (rev.). CMU Complex Information Processing Working Paper 227, Carnegie-Mellon, June 1973.
- Smith, D. H. Applications of Artificial Intelligence for Chemical Inference. XV. Constructive Graph Labelling Applied to Chemical Problems. Chlorinated Hydrocarbons. *Analytical Chemistry*, 1975, 47, 1176.
- Smith, D. H., Buchanan, B. G., Engelmores, R. S., Duffield, A. M., Yeo, A., Feigenbaum, E. A., Lederberg, J., & Djerassi, C. Applications of Artificial Intelligence for Chemical Inference VIII. An Approach to the Computer Interpretation of the High Resolution Mass Spectra of Complex Molecules. Structure Elucidation of Estrogenic Steroids. *Journal of the American Chemical Society*, 1972, 94, 5962.
- Smith, D. H., Buchanan, B. G., Engelmores, R. S., Adlercreutz, H. & Djerassi, C. Applications of Artificial Intelligence for Chemical Inference IX. Analysis of Mixtures Without Prior Separation as Illustrated for Estrogens. *Journal of the American Chemical Society* 1973, 95, 6078.
- Smith, D. H., & Carhart, R. E. Applications of Artificial Intelligence for Chemical Inference XXIV. Structural Isomerism of Mono- and Sesquiterpenoid Skeleton 1,2-. *Tetrahedron*, 1976, 32, 2513.



- Smith, D. H., & Carhart, R. E. Structure Elucidation Based on Computer Analysis of High and Low Resolution Mass Spectral Data. In M. L. Gross (Ed.), **High Performance Mass Spectrometry: Chemical Applications**. Wash., D. C.: American Chemical Society, 1978. P. 325.
- Smith, D. H., Konopelski, J. P., & Djerassi, C. Applications of Artificial Intelligence for Chemical Inference. XIX. Computer Generation of Ion Structures. **Organic Mass Spectrometry**, 1976, 11, 86.
- Smith, R. G., Mitchell, T. M., Chestek, R. A., Buchanan, B. G. A model for learning systems. **IJCAI 5**, 1977, 338-343. (Also Heuristic Programming Project Memo HPP-77-14, Computer Science Dept., Stanford University, 1977.)
- Smith, R. L. **Artificial Intelligence in CAI**. Unpublished working paper, IMSSS, Stanford University, 1976.
- Streitwieser, A., & Heathcock, C. H. **Introduction to Organic Chemistry**. New York: MacMillan, 1976.
- Varkony, T. H., Carhart, R. E., & Smith, D. H. **Computer Assisted Structure Elucidation, Ranking of Candidate Structures, Based on Comparison Between Predicted and Observed Mass Spectra**. Paper presented at the ASMS meeting, Washington, D.C., 1977.
- Waterman, D. A. Generalization learning techniques for automating the learning of heuristics. **Artificial Intelligence**, 1, 1970, 121-170.
- Waterman, D. A. **Adaptive production systems**. Complex Information Processing Working Paper 285, Dept. of Psychology, Carnegie-Mellon, December 1974.
- Winston, P. H. **Learning structural descriptions from examples**. Doctoral dissertation (MIT AI-TR-231), MIT, September 1970. (a)
- Wipke, W. T., Braun, H., Smith, G., Choplin, F., & Sieber, W. SECS--Simulation and evaluation of chemical synthesis: Strategy and planning. In W. T. Wipke & W. J. House (Eds.), **Computer-assisted Organic Synthesis**. Washington D. C.: American Chemical Society, 1977. Pp. 97-127.

## Index

13C-NMR 544

acyclic molecular structures 527  
acyclic molecules 524, 531  
AGE 547  
ALCHEM 558  
aliphatic amines 537  
analysis, molecular structures 1-524-527  
analytic chemistry 553  
AND/OR tree 555  
atom migration 536  
attribute-value pairs 541  
automatic rule formation 536  
automatic theory formation 536

backtracking 543  
BADLIST 528  
best-first procedure 563  
blackboard 547-553  
Brown, H. 524

canonical forms 1  
Carhart, Ray 534  
CF, certainty factors 551  
chemical applications 1-565  
chemical reactions 555  
chemical synthesis 1  
combinatorial explosion 523, 555, 557,  
562  
combinatorics 1-524  
completeness 561  
CONGEN 527, 531-536  
constraint generator 527-531  
constraint satisfaction 524, 545-553  
constraints 524-527-531-536, 547  
constraints, bond fragmentations 538  
constraints, combinations 533  
constraints, semantic 537  
Corey, E. J. 555  
cost 562  
CRYALIS 545

cyclic molecules 524, 531

Davis, R. 561  
DENDRAL 525-527-530-531, 536, 543  
di-keto androstanes 544  
domain rules 561

EDITSTRUC, interactive structure  
editor 532  
electron density map 545  
electron trees 534  
evaluation function 563  
event classes 549  
event list 548  
event-driven control structure 549  
EXAMINE 533  
explanation 540  
explanation, mass spectral data 540

flexibility 1, 538, 550  
fragmentation 525, 536-545  
fragmentations 525

Gelernter 561  
Gelernter, H. 555  
generality, of rules for molecular  
processes 539  
generate and test 552  
GOODLIST 528  
GOODLIST INTERPRETER 532  
graphical representations 1  
graphics 560  
Gray, Neil 533

half-order theory 538  
HEARSAY-II 547  
heuristic reasoning, structural  
elucidation 527

- heuristic reasoning, structure
  - elucidation 525
- heuristic search 541, 562
- heuristics 562
- hypothesis formation 545, 546
- hypothesis spaces 523
  
- I/O pairs 537
- imbedding algorithm 531
- induction programs 536
- induction/inference, in mass spectral processes 537
- inference 549
- instances, training 537
- interactive constraint satisfaction 533
- interactive goal definition 561
- interactive knowledge acquisition 536, 558
- interactive transfer of expertise 536
- interestingness 538, 556
- interestingness, synthesis 555
- INTSUM, interpretation and data summary program 540, 542
- isomer enumeration 533
- isomers 529
  
- knowledge base, chemical synthesis 557
- knowledge source 553
- knowledge sources 546, 547
- KS sequence 549
  
- learning strategies 538
- Lederberg, J. 524
- LHASA, Logic & Heuristics Applied to Synthetic Analysis 555
- library of reactions 558
- LISP 548
  
- man/machine interactions, chemical synthesis 556
- manipulation statements 558
  
- mass spectral process 525
- mass spectrometry 524-527-531-536
- matching cycle, example of 552
- means-end analysis 560
- META-DENDRAL 527-531, 536
- meta-rules 561
- meta-rules, CHRYSALIS 550
- migration 525
- model building 560
- modular system 547
- molecular fragments 524-527, 531
- molecular structures 1-524
- molecular weight 539
- mono-keto androstanes 544
- multiple match 562
- MYCIN 550, 561
  
- negative evidence 539
- Nourse, Jim 533
  
- opportunistic problem solving 549
- organic synthesis program 555
  
- pattern matching 558
- plan-generate-test 540
- plausible hypotheses 523
- positive evidence 539
- power/generalizability trade-off 561
- problem space 562
- production rules 536, 547, 557
- protein x-ray crystallography 545
- pruning 533
  
- REACT, interactive structure elucidation program 534
- reaction representations 558
- representation of formal knowledge 548
- representation of informal knowledge 548
- representation of knowledge, mass spectrometry 536

- representation of synthetic reaction routes 555
- representation, molecular structures 1
- reverse chemical reactions 557
- rule-based control 550
- RULEGEN 539
- RULEGEN, rule generation 540-542
- RULEMOD 540
- RULEMOD, rule modification 542
- rules, mass spectrometry 527-531, 536-545
- search strategy 563
- SECS, Simulation and Evaluation of Chemical Synthesis 555
- semantic model 537
- simulation of laboratory reactions 534
- situation-action rules 548
- slots, chemical transforms 561
- specialization 539
- specialization, of fragmentation rules 539
- statistical learning 537
- stereochemistry 522, 533, 562
- stereoisomer generation 533
- strategies, applying transforms 560
- strategy rules 561
- strategy, CHRYSALIS 549
- structure determination 1
- structure elucidation 1, 524-527, 531-536-553
- structure generation algorithm 531
- structure generator 527-531
- subgoal selection function 563
- subgraphs 551
- substructure keys 558
- substructure search 533
- substructure specifications 532
- superatoms 531
- symbolic reasoning, chemistry 1
- SYNCHEM 561
- SYNCHEM, Synthetic Chemistry 555
- SYNCHEM2 562
- SYNCOM 559
- synthesis tree 555
- synthesis, molecular structures 1-524
- synthetic chemistry 555-565
- target structure 555, 557
- task rules 549
- topology of molecules 522
- transforms 557, 561
- transforms, application 560
- tri-keto androstanes 544
- triggering 543, 549
- version space 543
- Wipke, W. T. 555
- Wisswesser Linear Notation 562
- working backwards, chemical synthesis 557
- working forward, chemical synthesis 557
- xray crystallography 524
- zero-order theory, mass spectrometry 538

# Applications-oriented AI Research

## Part 3: Medicine

# AI Applications in Medicine

## Table of Contents

A. Overview of Artificial Intelligence Applications in Medicine . . . . .	575
B. MYCIN . . . . .	581
C. CASNET . . . . .	589
D. INTERNIST . . . . .	593
E. Present Illness Program - PIP . . . . .	598
F. Digitalis Advisors . . . . .	682
G. IRIS . . . . .	685
References . . . . .	618
Index . . . . .	612

## A. Overview of Artificial Intelligence Applications in Medicine

There are two main areas where AI techniques are being used in medicine. One area is the application of pattern recognition and scene analysis techniques to the interpretation of x-rays; the other is medical decision making where the task is the construction of consultation programs for various medical domains. It is the second area that is the subject of this section.

The motivation for the development of expert computer-based medical consultation systems is twofold. First, there are obvious benefits to society from providing reliable and thorough diagnostic services--perhaps even at a reduced cost. It has been observed (Ledley & Lusted, 1959) that most of the errors made by clinicians are errors of omission, that is, they did not consider enough diseases. A computer program could be designed to exhaustively consider all of the diseases in its domain. Furthermore, there are some tasks that computers can perform more rapidly and accurately, such as calculating doses of medicines, particularly in cases where dosage is critical and many factors must be taken into account in the calculation (as with digitalis). There are also some tasks that physicians are notoriously poor at performing and that are routine enough for the computer to do, such as the prescription of anti-microbial therapy.

The second motivation for development of these systems stems from current interests in computer science. Clinical medicine has been a very fertile area for the study of cognitive processes ever since the diagnostic process has been studied extensively (Jacquez, 1963). There is a highly developed medical taxonomy; a large, relatively well-organized knowledge base; and a large number of experts in the domain. Furthermore, the type of problem solving that occurs in the domain is repetitive. These attributes reflect some of the prerequisites of the domain for a developing field of AI known as *knowledge engineering*. This field takes AI beyond the stage of "toy" problems to confront large, real-world problems.

Computer-based consultation brings with it many formidable social, psychological, and ethical problems that must be addressed by the system builders. These problems include: validating the systems, exporting them to hospitals and clinics, getting physicians and patients to accept them, and deciding the responsibility for decisions made by these systems.

In the following sections, aspects of the diagnostic process and medical decision making will be discussed, as well as a number of AI issues related to the representation and manipulation of medical knowledge.

### Medical Decision Making

There are three main aspects of medical decision making: data gathering, diagnosis, and treatment recommendation. Data gathering is concerned with obtaining the patient history and clinical and laboratory data. The clinical data consist of symptoms, which are the subjective sensations reported by the patient--such as headache, chest pain, etc.--and signs, which are objective and observable by the physician (Feinstein, 1967). Manifestation refers to any sign, symptom, or finding. Laboratory results generally are referred to as findings. Diagnosis is the process of using this data to determine the illness. The three aspects are not independent; disease hypotheses are used to direct further information

gathering, while treatment recommendation depends on the diagnosis and generally requires more information gathering. Often, the decision to do a test includes a physician's estimate of the cost, both in terms of money and danger to the patient, which are weighed against the value of the information gained. Gathering information, diagnosing the disease, and deciding on a treatment regimen constitute a consultation. Figure 1 illustrates this process in relation to the course of the disease.

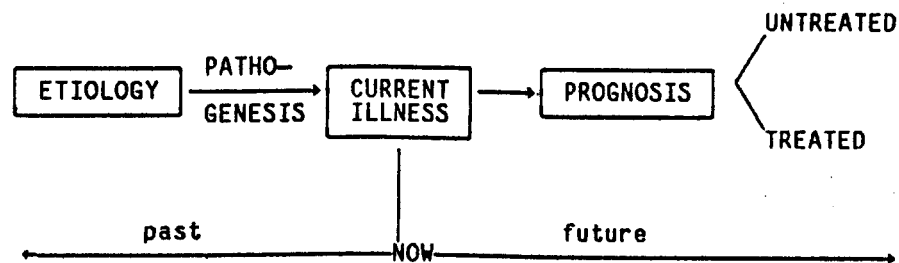


Figure 1. Consultation process depicting current state of medical knowledge.

This characterization of a consultation highlights the current state of medical knowledge. Etiology refers to the ultimate causes of the disease; pathogenesis refers to the way in which the disease developed from its causes. A consultation proceeds by determining the etiology. A treatment is then formulated for the identified diseases and their causes. Often, however, the medical knowledge is incomplete and it is not possible to determine the causes of a disease. In these cases, treatments must be based only on the knowledge of the symptoms or characteristics of the diseases. Some diseases are very well understood and knowledge about them is based on various kinds of models and knowledge about specific mechanisms. Other diseases are not very well understood and knowledge about them is only associational; for example, treatment is prescribed on the basis of symptoms of closely associated diseases for which treatments are known.

During a consultation the physician performs at least two mental processes: reasoning and judgment (Ledley & Lusted, 1959). Reasoning involves making clinical decisions using various formal and logical techniques. This process is evident primarily in the diagnosis phase. Judgment has come to mean the use of various "intangibles" such as general feelings about the case and past clinical experience, which help the physician to make clinical decisions. These are evident during prognosis and therapy recommendations. Artificial intelligence has attempted to model both of these processes.

There are, however, some aspects of consultations that computers cannot do, such as the physical examination. The physician gains much firsthand information from general appearance, facial expressions, etc., that is inaccessible to the computer. The design of computer consultation systems must, therefore, take this factor into account and offer mechanisms for the representation of these types of information secondhand.



### History of Computers in Medicine

The use of computers in medical decision making began in the early 1960s with the implementation of programs that performed well-known types of statistical analyses. These programs focused on the diagnosis aspect of the consultation: They accepted a set of findings and selected one disease from a fixed set, using methods such as pattern recognition through discriminant functions, Bayesian decision theory, and decision tree techniques (Croft, 1972; Nordyke, Kulikowski, & Kulikowski, 1971). Slightly more complex programs performed "sequential diagnosis." Here, when there is not enough information to make a reliable diagnosis, the next patient test (to get more information) is determined by a strategy that selects the "best" test based on three factors: the cost of the test, the danger to the patient, and the amounts of discriminating information needed and made available by the test.

The appeal of using statistical methods is that the resulting decisions are "optimal" according to specified criteria. Unfortunately, these statistical systems proved unsatisfactory. The mathematics that these systems have been based upon have assumed that the patient has only one disease and that the data is not erroneous. More fundamentally, certain assumptions and simplifications concerning the independence and mutual exclusivity of various disease states that were made in order to make the statistical techniques practical were found to be unjustified. Furthermore, many prior and conditional probabilities required for complete analysis were simply not available.

Since the early 1970s there has been an increasing application of AI techniques to performing medical decision making. Some of the formalisms, techniques, and languages developed in AI were directly applicable to medicine before, but the new understanding of the nature of the task called for new ways of representing knowledge and reasoning. For example, the classical AI problem-solving techniques of state-space search and theorem proving (see chapter Search, Search) were not directly applicable. Consider a simple application of state-space search to the planning of a treatment. If one assumes that the "initial state" is the diseased patient, that the final state is the "healthy patient," and that the "operators" are various drugs, physical therapies, surgical procedures, etc., it would appear that simple search would find a path between the initial and final states. But there are two fundamental problems. First, the initial state, the disease of the patient, is rarely known with certainty. Second, the application of an operator--i.e., a treatment--is not guaranteed to result in an expected state. In order to deal with these problems, methods for representing inexact knowledge and for performing plausible reasoning have been developed in each of the consultation systems described below.

From the standpoint of AI, medical diagnosis is a hypothesis formation (see article C4) problem. The diagnosis task is to use the clinical findings to form a consistent set of disease hypotheses (not to use findings to select one disease from a fixed set of possible diseases). These hypotheses are typically related to one another in various ways. Each existing system exhibits a different approach to this hypothesis formation problem.

### The State of the Art

The state of the art in computer-based medical decision making is represented by the programs described in the following articles. These programs are MYCIN (Shortliffe, 1976),

INTERNIST (Pople, 1975), CASNET/GLAUCOMA (Weiss, Kulikowski, & Safir, 1978), PIP (Szolovits & Pauker, 1978), IRIS (Trigoboff & Kulikowski, 1977), and the digitalis advisors (Silverman, 1974; Swartout, 1977b). There are now several other programs under development that use the techniques and ideas developed in the above systems. These include PUFF (Feigenbaum, 1977), a pulmonary function program, HODGKINS (Safrans, Desforges, & Tschilis, 1976), a system for performing diagnostic planning for Hodgkins disease, and HEAD-MED (Heiser, 1977, 1978), a psychopharmacology advisor. During the development of all these programs, certain issues arose concerning the construction of the programs and their acceptance by the medical community. The major issues and the ways in which these were addressed by the individual systems are also described below.

**Representation of knowledge.** Two distinct types of medical knowledge must be represented: (a) general knowledge of diseases, manifestations, causal mechanisms, etc., and (b) specific knowledge about the patient, the current medical history, the current therapies, etc. The usual representation formalisms of AI--semantic nets (see article Representation.B2), production rules (see article Representation.B3), frames (see article Representation.B7), and predicate calculus (see article Representation.B4)--are not directly applicable because of the inexact nature of medical knowledge. In all of the consultation systems that have been developed, these representations have been augmented, for example, using a numerical way of expressing strength of belief or strength of association. For example, in MYCIN, the medical knowledge is represented as a set of production rules augmented by "certainty factors." These certainty factors express the strength of belief in the conclusion of a rule, given that all of the premises are true. CASNET uses a causal network representation (basically a semantic network with the one relation, CAUSES) where each CAUSES link is qualified by a number that represents the strength of causality. In INTERNIST, a taxonomy of diseases is stored as a huge tree with each node representing a disease. Associated with each disease node is a list of manifestations, with numerical weights reflecting the strength of association between the disease and the manifestation. In PIP, the frame formalism is augmented by numbers that reflect both the strength of belief in a slot filler and the degree to which the frame itself applies to this patient. In IRIS, where the semantic net and production rule formalisms have been combined, a facility for incorporating an arbitrary representation of strength of belief has been included. Finally, the procedural representation is used in the Digitalis Advisors; it contains a mathematical model of the action of digitalis.

**Clinical reasoning.** Clinical reasoning is based on the ways different pieces of evidence for particular hypotheses are combined. Each system has a different approach to this problem but most employ the technique of *thresholding*; if the numerical score of a hypothesis exceeds a certain pre-set threshold (defined by the expert physician), then the hypothesis is believed to be true. The clinical reasoning of MYCIN involves determining parameters (e.g., the infections and causative organisms of a patient) using production rules. The premises of a rule are considered true if the combined value of the associated certainty factors exceeds a predefined threshold. If several rules contribute to a conclusion about a parameter, then their certainty factors are functionally combined to form a composite certainty factor for this conclusion. These confidence-factor combining functions are based on probability theory. In CASNET, a status measure is associated with each state in the causal network. Weights are propagated both in the forward and backward direction depending on disease causality. A state is considered "confirmed" if its status exceeds a specified threshold. In INTERNIST, disease hypotheses are scored by a procedure that

takes account of the strength of association among: (a) the manifestations exhibited by the patient and the disease, (b) the manifestations associated with the disease that are not present in the patient, (c) and the confirmed diseases causally related to this one. Disease hypotheses are ranked, and the top-ranked diseases are investigated further. When the difference between the scores of the top two disease hypotheses reaches a predefined criterion, the top ranking disease is confirmed. PIP combines two different methods of reasoning: categorical and probabilistic. Categorical decisions are based on logical criteria rather than numerical values. The probabilistic reasoning involves scoring the disease frames.

A frame can be confirmed either on logical or probabilistic criteria. In IRIS, an attempt is made to confirm nodes of a semantic net as being true for the patient. Information is passed between the nodes of the semantic net via sets of production rules associated with the links. These production rules can encode both logical and probabilistic decisions.

**Explanation and justification.** The explanation and the justification of a system's line of reasoning are important factors for the acceptance of consultation systems by physicians. Explanation involves showing the user the line of reasoning used in making a particular diagnosis; justification is concerned with the medical accuracy and reliability of the knowledge and the reasoning strategies used.

Only two systems currently address the issue of explanation. MYCIN explains a diagnosis by printing out an English version of the chain of rules used. More complex explanation facilities are provided by TEIRESIAS (Davis, 1977), an explanation and knowledge acquisition system developed in the context of MYCIN. The OWL Digitalis Advisor provides English explanations of its reasoning that are generated directly from the OWL code. The detail of the explanation can be controlled by the program.

Both INTERNIST and CASNET are able to summarize the consultation by displaying scores of the hypotheses and statuses of states; however, they are unable to explain the methods they used to arrive at these scores.

The issue of justification is a complex one. Both CASNET and MYCIN can cite references to the research literature in support of diagnoses and treatment recommendations. CASNET is able to provide alternative recommendations based on differing expert opinions. At the heart of the justification issue is the accuracy and reliability of the expert's knowledge and whether this knowledge has been accurately captured in the representation formalism. Often medical experts have differing opinions, and it is not clear whether a consensus should be sought or whether the different opinions should all be represented. CASNET and MYCIN have been developed with the collaboration of groups of experts, and the rules typically represent a general consensus of opinion. The other systems were developed with one main expert; so consensus was not an issue.

**Validation.** Just as the various instruments and drugs used by physicians must be validated, so must consultation programs. So far, CASNET and MYCIN have undergone relatively extensive clinical trials and have been rated as "expert" in their respective domains by human experts. INTERNIST has yet to undergo formal clinical trials, but it is informally rated as an expert in internal medicine. The Digitalis Therapy Advisors have performed well in limited trials.

**Acquisition of knowledge.** Knowledge acquisition is the transfer of the experts'

knowledge and expertise to the program. Currently the only successful way of doing this is through a knowledgeable intermediary. TEIRESIAS was an attempt to let the expert communicate directly with the consultation program; it took the computer scientist out of the chain. But it appears that the current state of the art of knowledge acquisition does not yet permit this facility.

### Summary

Despite the extensive work that has been done, none of these systems is in routine clinical use. Physicians have not for the most part accepted them. The main reason is that they have yet to satisfy the "indispensability" criterion: They are not indispensable to the practice of medicine and physicians perform adequately without them. The only AI program that is in routine medical use is PUFF, a pulmonary function program, which is used because it saves the physician a lot of time. Constructed using EMYCIN (the MYCIN system with the knowledge of infectious diseases removed), PUFF uses a set of about 55 rules about pulmonary dysfunction. The program suggests treatment recommendations that can be overridden by the physician.

In order for AI programs to make a significant impact on health care, at least in the short term, it appears that PUFF's example should be followed. The ingredients for a successful application in medicine seem to be (a) a careful choice of the medical problem and (b) the cooperation of interested experts. The domain must be narrow and relatively self-contained; the use of the computer should aid, not replace, the physician; and the task should be one that the physician either cannot do or is willing to let a computer do.

To summarize, the main focuses of activity in the area of medical decision making today are: knowledge engineering, the acquisition of knowledge from experts; knowledge representation, for building and maintaining the large medical knowledge bases; strategy design, for reasoning with the medical knowledge; and program designs that feature explanation capabilities, of their reasoning to users.

### References

See AIM 1975-1978, Davis, Buchanan, & Shortliffe 1977, Croft 1972, Feigenbaum 1977, Feinstein 1967, Gorry & Barnett 1968, Heiser, 1977 1978, Jacques 1963, Ledley & Lusted 1959, Nordyke, Kulikowski, & Kulikowski 1971, Safrans, Desforges, & Tsihliis 1976, Weiss, Kulikowski, & Safir 1977, and Weiss, Kulikowski, & Safir 1978.

## B. MYCIN

The MYCIN system Shortliffe, 1976; Davis, 1977 was designed to provide consultative advice on diagnosis and therapy for infectious diseases. Such advice is often required in hospital care because the attending physician may not be an expert on infectious diseases that show themselves during a period of hospitalization--as, for example, when a cardiology patient develops an infection after heart surgery. Time considerations compound the problem. A specimen (blood, urine, etc.) from a patient can show some early evidence of bacterial growth within 12 hours, but generally it takes 24 to 48 hours (or more) for positive identification. The physician must therefore often decide in the absence of complete information whether or not to start treatment and what drugs to use if treatment is required. Both of these may be difficult questions.

The trace that follows shows the initial and final parts of a sample interaction between a physician and MYCIN (*italicized* comments provide additional commentary but are not part of the actual dialogue). MYCIN, in effect, "interviews" the doctor about his patient, collecting information from which it infers the diagnosis and selects an appropriate therapy.

-----PATIENT-1-----

1) Patient's name: (first-last)

\*\* FRED SMITH

*MYCIN collects some background data about the patient.*

2) Sex:

\*\* MALE

3) Age:

\*\* 55

4) Have you been able to obtain positive cultures from a site at which  
Fred Smith has an infection?

\*\* YES

-----INFECTION-1-----

5) What is the infection?

\*\* PRIMARY-BACTEREMIA

6) Please give the date and approximate time when signs or symptoms of  
the primary-bacteremia (INFECTION-1) first appeared. (mo/da/yr)

\*\* MAY 5, 1975

The most recent positive culture associated with the primary-bacteremia  
(INFECTION-1) will be referred to as:

-----CULTURE-1-----

7) From what site was the specimen for CULTURE-1 taken?

\*\* BLOOD

8) Please give the date and time when this blood culture (CULTURE-1)  
was obtained. (mo/da/yr)

\*\* MAY 9, 1975

The first significant organism from this blood culture (CULTURE-1)  
will be referred to as:

-----ORGANISM-1-----

9) Enter the identity of ORGANISM-1:

\*\* UNKNOWN

*Typically the identity of the organism is not known and hence must be inferred by MYCIN using various other pieces of laboratory and clinical information. MYCIN proceeds to ask questions that could help to determine the identity of the organism.*

- 10) Is ORGANISM-1 a rod or coccus (etc.):  
 \*\* ROD
- 11) The gram stain of ORGANISM-1:  
 \*\* GRAMNEG
- 12) Have there been POSITIVE cultures from sites which are NOT associated with infections about which you are seeking advice?  
 \*\* NO
- 13) Did ORGANISM-1 grow aerobically?  
 \*\* YES (.8)
- 14) Did ORGANISM-1 grow anaerobically?  
 \*\* YES
- 15) What is the suspected portal of entry of ORGANISM-1 into this sterile site?  
 \*\* GI (.6)
- ...

*More questions follow, to gather sufficient information to infer possible identities of the infecting organisms.*

My therapy recommendation will be based on the following probable infection(s) and potential causative organism(s):

INFECTION-1 is PRIMARY-BACTEREMIA

- <Item 1> The identity of ORGANISM-1 may be PSEUDOMONAS-AERUGINOSA  
 <Item 2> The identity of ORGANISM-1 may be KLEBSIELLA-PNEUMONIAE  
 <Item 3> The identity of ORGANISM-1 may be E.COLI  
 <Item 4> The identity of ORGANISM-1 may be BACTEROIDES-FRAGILIS  
 <Item 5> The identity of ORGANISM-1 may be ENTEROBACTER  
 <Item 6> The identity of ORGANISM-1 may be PROTEUS-NON-MIRABILIS

- 46) On a scale of 0 to 4, where higher numbers indicate increasing severity how would you rate Fred Smith 's degree of sickness?  
 \*\* 3
- 47) Does Fred Smith have a clinically significant allergic reaction to any antimicrobial agent?  
 \*\* NO
- 48) Fred Smith 's weight in kilograms (or <number> POUNDS):  
 \*\* 70

*Note that 2 drugs cover for 6 potential organisms.*

[Rec 1] My preferred therapy recommendation is as follows:  
 In order to cover for Items <1 2 3 5 6>:

Give: GENTAMICIN

Dose: 119 mg (1.7 mg/kg) q8h IV [or IM] for 10 days

Comments: Modify dose in renal failure

In order to cover for Item <4>:

Give: CLINDAMYCIN

Dose: 595 mg (8.5 mg/kg) q6h IV [or IM] for 14 days

Comments: If diarrhea or other GI symptoms develop, patient should be evaluated for possible pseudomembranous colitis.

Do you wish to see the next choice therapy?

*Alternative choices are available.*

\*\* YES

Which, if any, of the above drugs do you wish eliminated from consideration?

\*\* CLINDAMYCIN

[Rec 2] Next best therapy recommendation:

In order to cover for Items <2 3 4 5 6>:

Give: CHLORAMPHENICOL

Dose: 1 gm (15 mg/kg) q6h IV for 14 days

Comments: Monitor patient 's white count

In order to cover for Item <1>:

Give: GENTAMICIN

Dose: 119 mg (1.7 mg/kg) q8h IV [or IM] for 10 days

Comments: Modify dose in renal failure

### Knowledge Base Organization

The medical knowledge used by MYCIN is encoded as production rules (see article Representation.B3) of the sort shown in Figure 1 below. The rules are stored internally in the INTERLISP form shown, from which the English version is generated. Each rule is a single "chunk" of domain-specific information indicating an action (in this case a conclusion) that is justified if the conditions specified in the premise are fulfilled. Since the rule uses a vocabulary of concepts common to the domain, it forms, by itself, a comprehensible statement of some piece of domain knowledge. As will become clear, this characteristic is useful in many ways.

Each rule is highly stylized--with the if/then format and the specified set of available primitives. While the LISP form of each is executable code (the premise, in fact, is simply EVALuated by LISP to test its truth; and the action, EVALuated to make its conclusions), this tightly structured form makes it possible to examine the rules as well as execute them. For example, the rules can be translated into a readable English format as in Figure 1. This translation capability has been used in MYCIN to explain the program's inferences to the expert. The ability to explain a line of reasoning leading to a conclusion and to justify why the program is asking a particular question in a given case is important. Physicians are more likely to accept the recommendations of a system that can explain its rationale for making them. This ability is expanded in the TEIRESIAS article C8.





or, in INTERLISP:

(MEMBF CNTXT SITE STERILESITES) |||| Predicate Object Attribute Value

MEMBF is a predicate, and the triple says that the site of the current object (an organism, in this case) is a member of the class of sterile sites. There is a standardized set of some 24 domain-independent predicate functions (e.g., SAME, KNOWN, DEFINITE) and a range of domain-specific attributes (e.g., IDENTITY, SITE), objects (e.g., ORGANISM, CULTURE), and associated values (e.g., E.COLI, BLOOD). These form the "vocabulary" of conceptual primitives available for use when constructing rules.

A rule premise is always a conjunction of clauses, but it may contain arbitrarily complex conjunctions or disjunctions nested within each clause. (Instead of writing rules whose premise would be a disjunction of clauses, a separate rule is written for each clause.) The action part indicates one or more conclusions that can be drawn if the premises are satisfied, making the rules purely inferential.

Medical facts about the patient (see Figure 3 below) are represented as 4-tuples made up of an associative triple and its current CF. Positive CFs indicate a predominance of evidence confirming a hypothesis; negative CFs indicate predominance of disconfirming evidence.

(IDENT ORGANISM-2 KLEBSIELLA .25)  
 (IDENT ORGANISM-2 E.COLI .73)  
 (SENSITIVS ORGANISM-1 PENICILLIN -1.0)  
 (IMMUNOSUPPRESSED PATIENT-1 YES 1.0)

Figure 3. MYCIN 4-tuple.

MYCIN's model of inexact reasoning permits the coexistence of several plausible values for a single attribute, if this is suggested by the evidence. For example, after attempting to deduce the identity (IDENT) of an organism, MYCIN may have concluded (correctly) that there is evidence of both E.coli and Klebsiella.

To summarize, there are two major forms of knowledge representation in use in the performance program: (a) the attributes, objects, and values--which form a vocabulary of domain-specific conceptual primitives, and (b) the inference rules expressed in terms of these primitives.

### The Inference Engine

In MYCIN, rules are invoked in a simple *backward-chaining* fashion that produces an exhaustive *depth-first* search of an AND/OR goal tree (see article *Search.Problem.Reduction*). Assume that the program is attempting to determine the identity of an infecting organism. It retrieves all the rules that make a *conclusion* about the topic (i.e., that mention the identity of bacteria in their action) and invokes each one in turn, evaluating each *premise* to see if the conditions specified have been met. For the sample rule above, this process would begin with determining the type of infection. Since the type of the infection is unknown, it is set up as a subgoal and the process recurs.

The search is thus depth-first (because each premise condition is thoroughly explored in turn); the tree that is sprouted is an and/or goal tree (because rules may have OR conditions in their premise); and the search is exhaustive (because the rules are inexact; so that even if one succeeds, it was deemed a wisely conservative strategy to continue to collect all evidence about the subgoal.

The subgoal that is set up is a generalized form of the original goal. Thus, for the first clause in the rule ("the infection is primary-bacteremia"), the subgoal set up is "determine the type of infection." The subgoal is therefore always of the form "determine the value of *attribute*" rather than "determine whether the *attribute* is equal to <value>." By setting up the generalized goal of collecting all evidence about an attribute, the performance program effectively exhausts each subject as it is encountered, and thus tends to group together all questions about a given topic. This feature results in a system that displays a much more focused, methodical approach to the task, which is a distinct advantage where human engineering considerations are important. The cost is the effort of deducing or collecting information that is not strictly necessary. However, since this unnecessary effort occurs rarely--only when the <attribute> can be deduced with certainty to be the <value> named in the original goal--it has not proven to be a problem in practice.

If after trying all relevant rules to resolve a subgoal, the total weight of the evidence about a hypothesis falls between  $-.2$  and  $.2$  (an empirical threshold), the answer is regarded as still unknown. This result would occur if no rules were applicable, if the applicable rules were too weak, if the effects of several rules offset each other, or if there were no rules for this subgoal at all. In any of these cases, when the system is unable to deduce the answer, it asks the user for the value of the subgoal (using a phrase that is stored along with the attribute itself).

This strategy, of always attempting to deduce the value of a subgoal and asking the user only when deduction fails, insures a minimum number of questions. It could also mean, however, that work might be expended searching for a subgoal, arriving perhaps at a less than definite answer when the user might already know the answer with certainty. To prevent this inefficiency, some of the attributes have been labeled "laboratory data," to indicate that they represent information available to the program as results of quantitative tests. In these cases the deduce-then-ask procedure is reversed, and the system will attempt to deduce the answer only if the user cannot supply it. Given the desire to minimize both tree search and the number of questions asked, there is no guaranteed optimal solution to the problem of deciding when to ask for information and when to try to deduce it. But the distinction described has performed quite well and seems to embody an appropriate criterion.

Two other additions to straightforward tree search increase the inference engine's efficiency. First, before the entire list of rules for a subgoal is retrieved, the program attempts to find a sequence of rules that would establish the goal with certainty, based only on what is currently known. Since this is a search for a sequence of rules with  $CF=1$ , the result is termed a *unity path*. Besides efficiency considerations, this process offers the advantage of allowing the program to make "commonsense" deductions with a minimum of effort (rules with  $CF=1$  are largely definitional). Because there are few such rules in the system, the search is typically very brief.

Second, the inference engine performs a partial evaluation of rule premises. Since many attributes are found in several rules, the value of one clause (perhaps the last) in a

premise may already have been established while the rest are still unknown. If this clause alone would make the premise false, there is clearly no reason to do all the search necessary to establish the others. Each premise is thus "previewed" by evaluating it on the basis of currently available information. The result is a Boolean combination of TRUEs, FALSEs, and UNKNOWNs; and straightforward simplification (e.g.,  $F \times U = F$ ) indicates whether the rule is guaranteed to fail.

### Therapy Selection

After MYCIN determines the significant infections and the organisms that cause them, it proceeds to recommend an antimicrobial regimen if this is appropriate. The MYCIN therapy selector (Clancey, 1979) uses a description of the patient's infections, causal organisms, a ranking of drugs by sensitivity, and a set of drug *preference categories* (such as "propose 2 drugs: one second choice drug and one third choice drug") to recommend a drug regimen. The algorithm will also modify dosages in the case of renal failure in the patient. The program can provide detailed explanations about how it made a regimen choice and can accept and critique a regimen proposed by the physician.

### Acquisition and Use of New Knowledge

The representation of knowledge as production rules and the ability to explain specific rules allow MYCIN to interact with an expert clinician in a manner that permits the system to acquire and use new knowledge. The TEIRESIAS system (see article C6; see also Davis, 1976a) works in conjunction with MYCIN and allows the expert to inspect faulty reasoning chains and then add and modify any rules or clinical parameters required to augment and repair the medical knowledge of MYCIN.

When the expert is dissatisfied with the system's performance on a particular case, MYCIN is able to explain how it made the erroneous conclusions and guide the expert while he is determining the source of the reasoning "bug." To correct the reasoning, the expert may elect to enter new rules or alter existing ones. The user enters his requests through what is nearly a natural language interface. These requests are parsed and used by the system to create a new internal rule that is then presented to the user for inspection. This interaction helps minimize any misunderstanding between the clinician and MYCIN.

Once this new rule is accepted and understood by the system, the next consultation will make use of it and alter its recommendations accordingly. This ability permits the system to interact directly with the domain experts without intervention of a programmer.

### Concluding Remarks

Formal evaluations of the MYCIN system have been done that indicate that MYCIN compares favorably with infectious disease experts in diagnosing and selecting therapy for patients with bacteremia and meningitis. At present, however, the system is not used on the wards primarily due to its incomplete knowledge of the complete spectrum of infectious diseases.

MYCIN is one of the first of a new breed of computer systems: systems that step out of the toy worlds of AI and into the real world. These systems must deal with many of the social and psychological problems of *man/machine interactions*. Issues such as modularity and representation of knowledge, reasoning in specific domains, explanation of a system's logic, and the ability to accumulate and use new information must be considered with equal attention to programming and interfacing problems. MYCIN has been designed with these issues in mind and has consequently shown promise as a real-world aid to the clinician.

#### References

See Shortliffe (1976) and Davis (1976a).

C. CASNET

The Causal Associational Network (CASNET) program (Weiss, Kulikowski, & Safir, 1977) is a computer system for performing medical diagnosis developed at Rutgers University. The major application of CASNET has been in the domain of glaucoma. The system represents a disease not as a static "state," but as a *dynamic process* that can be modeled as a network of causally linked pathophysiological states. The system diagnoses a patient by determining the pattern of pathophysiological causal pathways present in the patient and identifying this pattern with a *disease category*. Once the disease category is explicitly identified, the most appropriate treatments can be prescribed. The causal model also makes possible a prediction of the likely future course of a disease both if treated and if untreated.

Representation of Medical Knowledge

A CASNET model consists of three "planes of knowledge," as shown in Figure 1.

CLASSIFICATION TABLES

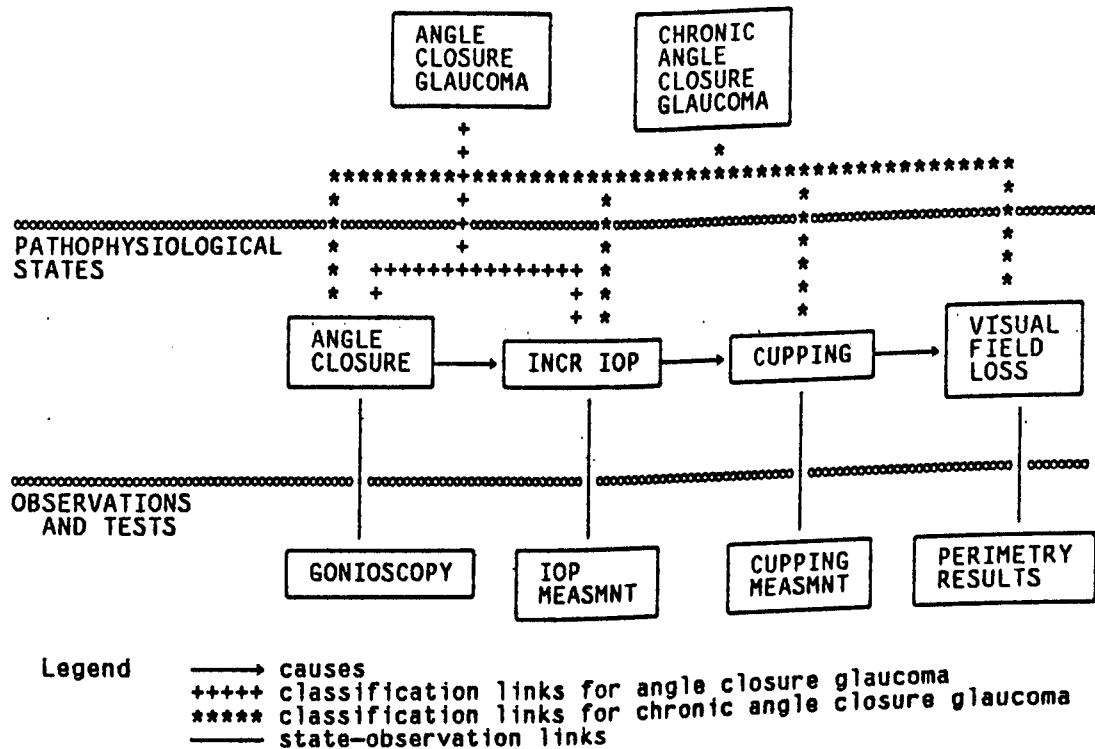


Figure 1. Part of the causal network model for glaucoma.

The plane of pathophysiological states is the heart of the model. The nodes in this plane represent elementary hypotheses about the disease process, and arcs here represent

a causal connection between two elementary hypotheses; for example, INCREASED INTRAOCULAR PRESSURE....CAUSES....CUPPING OF THE OPTIC DISK. Associated with each link is a forward weight or *confidence factor*, a number on a 1-5 scale, where 5 corresponds to "(almost) always causes" and 1 to "rarely causes." The determination of these weights and their utility in confirming or disconfirming the presence of a pathophysiological state are discussed in the section in this article on *Reasoning*.

The plane of observations contains nodes representing evidence gathered from the patient. These include signs, symptoms, and laboratory tests. During a consultation some or all of these nodes will be instantiated. Nodes in this plane are linked to nodes in the pathophysiological plane. The links have associated confidences, again on a 1-5 scale, reflecting the degree to which the particular test, symptom, or sign *supports* the associated state. For example, a scotoma (a perimetry measurement) strongly indicates VISUAL FIELD LOSS so it has a confidence value of 5. The same test, however, could have a different confidence value depending on the results; for example, 15 mm of Hg could be considered evidence for INCREASED INTRAOCULAR PRESSURE, but a result of 30 would be definite evidence and would carry a greater confidence value. The confidence values with which observations are linked to pathophysiological states are predetermined by the designers of CASNET.

In general, there is usually more than one test for a particular state, and the same test indicates more than one state. Each test also has an associated cost that reflects both monetary cost and danger to the patient. Some states might not have a corresponding test since such a test might not exist or might be judged too difficult or costly to use for a particular pathology.

The third plane contains the disease classification tables. A classification table defines a "disease" as a set of confirmed and denied pathophysiological states. It also contains a set of treatment statements for that disease.

STATES	DISEASES	TREATMENTS
ANGLE CLOSURE		
INCR IOP	ANGLE-CLOSURE-GLAUCOMA	TREATMNT1...
CUPPING		
VFL	CHRONIC-ANGLE-CLOSURE GLAUCOMA	TR1, TR2...

Figure 2. A classification table.

For example, the classification table indicates that if a patient is found to have ANGLE CLOSURE and INCREASED INTRAOCULAR PRESSURE but neither CUPPING nor VISUAL FIELD LOSS, then he has ANGLE CLOSURE GLAUCOMA; if he has ANGLE CLOSURE and INCREASED INTRAOCULAR PRESSURE and CUPPING and VISUAL FIELD LOSS, then he has CHRONIC ANGLE CLOSURE GLAUCOMA. The concept represented in the classification tables is that a disease is dynamic with respect to time and that confirmed states further down a *pathway*

represent more advanced stages of the disease. The states in a classification table will generally be on the same pathway. A "starting state" is a state with no causes in the network (also called a basic disease mechanism). Inadequate understanding of disease mechanisms

or incomplete models sometimes lead to classification tables containing states from more than one pathway.

### Reasoning

Figure 2 illustrates how CASNET defines a disease as a conjunction of causally related pathophysiological states. Diagnosis in CASNET is a matter of finding one or more causal pathways between these states. Reasoning in CASNET is designed to maximize the likelihood of finding these pathways, given a set of signs, symptoms, and test results.

A diagnostic session begins with the program's asking the user (physician) a series of questions about the patient. The physician answers with values for any tests, signs, and symptoms, or he answers UNKNOWN. These values, together with the confidences associated with the tests and the weights associated with the causal arcs, are used to compute a *status*, or confidence factor, for each node in the causal net.

The STATUS of a state is affected both by the results of its associated tests and by the STATUSs of the states around it. For example, if A causes B and B is confirmed by observation, then there is strong evidence for A. A general algorithm is used to propagate these weights on a state, both in the forward direction (i.e., along the direction of the causal link) and in the backward direction. A state is marked *confirmed* if its STATUS is greater than a preset threshold, it is marked *denied* if its STATUS is less than a second threshold, otherwise it is *undetermined*. The program uses a strategy for selecting the next question, based on the cost of the test and on the likelihood that it will lead to the confirmation or denial of a state.

After all available symptoms and findings have been entered and after the STATUS's have been computed, the classification tables are used to determine diagnoses and treatments. The tables are selected to cover all confirmed nodes. The strategy for selecting the tables is to find the starting states for which causal pathways can be generated that reach the largest number of confirmed states without traversing a denied state. This procedure is repeated until all of the confirmed states are covered.

The treatment statements of the selected classification tables are then used to select a therapy for the indicated diseases. Like a state, a treatment has an associated STATUS that is interpreted as its confidence in its success as a treatment. The treatment with the highest STATUS is selected. This assessment is repeated for all selected classification tables. A final algorithm decides whether some treatments are subsumed by others, and then the final treatment recommendations are printed. If desired, a short summary of research justifying the diagnosis and treatment can also be printed. The current glaucoma model contains about 150 states, 350 tests, and 50 classification tables.

**Concluding Remarks**

CASNET adopts a strictly "bottom-up" approach to the problem of diagnosis, working from the tests, through the causal pathways, to a diagnosis. The separation of medical knowledge (encoded in the causal network) from reasoning strategies (embodied in the program) will make the expansion of the disease model, when new research discoveries are made, a simple matter. The program is continually being tested and updated by a computer-based network of collaborators. The model also provides a convenient way of following the progress of a patient's disease over multiple visits--the causal net can be used to view the disease progression, both forwards and backwards, along the pathways. Although CASNET has been used primarily in the area of glaucoma, the representational scheme and decision-making procedures are applicable to other disease areas that are understood well enough to make the process of disease known. The program's performance has been evaluated by ophthalmologists and is considered close to expert level.

**References**

See Weiss, Kulikowski, & Safir (1978).



## D. INTERNIST

INTERNIST (Pople, 1975; Pople, 1977) is a medical consultation program in the domain of internal medicine developed jointly by H. Pople, a computer scientist, and J. Myers, a specialist in internal medicine, both at the University of Pittsburgh. The program is presented with a list of *manifestations*, of disease in a patient (e.g., symptoms, physical signs, laboratory data, and history), and it attempts to form a diagnosis. The diagnosis consists of a list of diseases that would account for the manifestations. Using information presented during the course of the consultation, the program is able to discriminate between competing disease hypotheses. The current version of the program only formulates diagnoses and does not recommend treatments.

One of the major goals of the INTERNIST project has been to model the way clinicians do diagnostic reasoning. The program has been used to explore the way that certain symptoms evoke particular diseases in the mind of the clinician: how hypothesized diseases generate expectations of other symptoms, how a clinician focuses on a particular disease area and temporarily ignores certain other symptoms that he judges irrelevant, and how he decides between competing disease hypotheses.

From the standpoint of computer science, INTERNIST is solving a *theory formation* or *hypothesis formation* problem. Determining a satisfactory diagnosis involves inferring a set of hypotheses to explain the patient data. In INTERNIST, the data are manifestations and the hypotheses are diseases.

Internal medicine is complicated because a patient may suffer from a number of diseases simultaneously. Although some diseases are more likely to be associated than others, the possible combinations are too numerous to encode a priori. Pople (1977[IJCAI]) suggests that a conservative estimate of this number is 10 to the 40th. Clearly, diagnosis of a set of diseases present in a patient is nontrivial. INTERNIST-I accomplishes this diagnosis by sequentially establishing the diseases that best fit the data. INTERNIST-II is an improvement over its predecessor because it establishes the set of diseases in parallel and therefore avoids some of the annoying artifacts of sequential processing, such as considering a number of incorrect diagnoses before "focusing in" on the correct one.

### Overview of INTERNIST-I

For INTERNIST-I a *problem* is defined as a set of mutually exclusive disease hypotheses. If a patient has a number of diseases, INTERNIST-I must solve that number of problems. In brief, INTERNIST-I finds a set of diseases that account for some or all of a set of symptoms, then it picks one disease from the set on the basis of a scoring schema, which is the solution for one of the problems. Then it finds another set of diseases that account for some or all of any remaining symptoms and again picks the most likely of these alternatives. It continues in this manner until all symptoms have been accounted for.

### Representation of Medical Knowledge

INTERNIST's knowledge of diseases is organized into a *disease tree*, or taxonomy, using the "form-of" relation (see Fig. 1). For example, Hepatocellular disease is a form of liver disease.

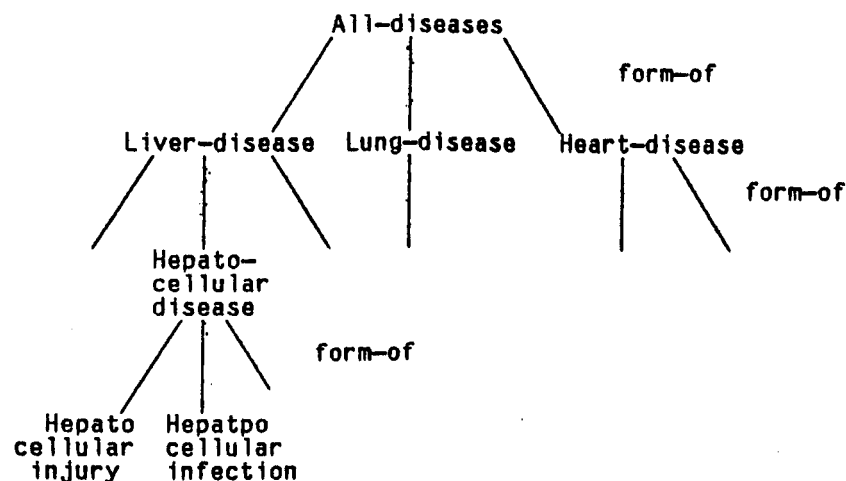


Figure 1. The structure of the disease tree.

The top-level classification in this tree is by organs--heart disease, lung disease, liver disease, etc. A *disease node's* offspring are refinements of that disease, terminal nodes being individual diseases. A nonterminal node and its subtree are referred to as a *disease area*, while a terminal node is referred to as a *disease entity*. The disease hierarchy is predetermined and fixed in the system.

Diseases and their manifestations are related in two major ways: (a) a manifestation can evoke a disease and (b) a disease can manifest certain signs and symptoms. These relations can loosely be thought of as probabilities:  $p(D|M)$  (the conditional probability of  $D$  given  $M$ ) and  $p(M\D)$ , respectively. The strength of these relations is given by a number on a 0-5 scale where 5 means that the manifestation is always associated with the disease and 0 means that no conclusions can be drawn about the disease and the manifestation. Each disease in the tree is associated with its relevant manifestations. Several other kinds of relationships are superimposed on the disease tree to capture causal, temporal, and other association patterns among diseases.

The disease tree and its associated manifestations are constructed and maintained separately from the normal diagnosis program. All known evokes and manifest relations are entered for the terminal nodes (diseases) of the tree. A list of manifestations is then computed for each nonterminal of the tree by taking the intersection of the manifestation lists of that node's offspring. In this way, the manifestations "percolate" up through the tree to the most general disease with which they are associated and are stored only with this node. This means that manifestations associated with a nonterminal disease node are, by implication, also associated with every node (terminal or non terminal) beneath it in the tree. As well as providing storage economy, this information is used during the consultation for selecting disease areas on which to focus. For example, jaundice (yellowing of the skin) will be associated with some nonterminal disease (e.g., hepatitis) under liver diseases, and its presence in a patient will cause the consultation program to investigate diseases in that disease area.

Various properties are associated with each manifestation. The most important ones are TYPE and IMPORT. TYPE is a measure of how expensive it is to test for a manifestation, both in terms of financial cost and physical risk to the patient. TYPE is used to order the questions asked by the consultation program: the questions about less expensive manifestations are asked first. The IMPORT of a manifestation is a measure of how easily it can be ignored in a diagnosis. The manifestation "Shellfish ingestion" can easily be ignored, but a liver biopsy showing caseating granulomas must be explained.

### Reasoning

At the beginning of a consultation, a list of manifestations is entered. As each manifestation is entered, it evokes one or more nodes of the disease tree. A model is created for each evoked disease node and consists of 4 lists.

- Observed manifestations that this disease cannot explain. This list is called the *shelf*.
- Observed manifestations that are consistent with the disease.
- Manifestations that should be present if this disease is the correct diagnosis but that have not been observed in the patient.
- Manifestations consistent with this disease but that have not yet been observed in the patient.

After the initial entry of manifestations, the disease tree consists of nodes that have been "lit up" (evoked) and those that have not. A diagnosis corresponds to a set of lit terminal nodes that account for all of the symptoms. Generally at this stage, very few of the terminal nodes will be lit up, so the program must ask for further information. To get this further information, the program will focus on a disease area and formulate a problem.

Each disease model is scored, receiving a positive score

for each manifestation it explains and a negative score for each manifestation that it cannot explain. Both are weighted by IMPORT. It receives a bonus if it is linked causally to a disease that has already been confirmed. The disease models are partitioned into two sets: (a) the top-ranked model and the diseases that are mutually exclusive to it (alternatives), and (b) the diseases that are complementary to the top-ranked model. For example, if the top-ranked node is hepatocellular injury, then other evoked liver diseases will be alternatives to it, while lung or heart diseases will be complementary.

Having formulated a problem by partitioning the disease models, the system follows one of several strategies, depending on the number of candidate diseases in the problem set. If there are many (>4) alternative hypotheses, it attempts to rule out as many as possible. Questions about manifestations that strongly indicate a disease (high  $p(M|D)$ ) are selected first. If these manifestations are not present, then this disease can be ruled out. If there are between 2 and 4 possibilities, the program attempts to discriminate between them. Then questions about manifestations that strongly indicate one  $D_1$  (high  $p(M|D_1)$ ) and weakly

indicate another disease  $D_2$  (low  $(p(M \setminus D_2))$ ) are selected. These questions are able to discriminate between the two diseases. If there is only one candidate, then questions that have a good chance of confirming this disease are asked. Sometimes, if there is not enough data, it will not be possible to confirm one of the terminal nodes, and a more general diagnosis is given (e.g., "liver disease").

After a disease is confirmed, its manifestations are marked "accounted for"; bonus scores are given to (previously manifested) diseases that are causally linked to this one; and focus shifts to the new top-ranked disease and the formulation of a new problem.

### INTERNIST-II

There was a major problem with INTERNIST-I. In complex cases the program had a tendency to begin the analysis by focusing first on totally inappropriate areas. While the final diagnosis was usually correct, the initial meandering was annoying to clinicians. The cause of the problem was traced to the sequential method of problem formulation. The simultaneous formulation of several problems is being investigated in INTERNIST-II.

### Representation of Medical Knowledge

INTERNIST-II uses the same database as INTERNIST-I, but it is augmented by a set of *constrictor relations*. These are manifestations that do not evoke a particular disease but, rather, a general area of infirmity. For example, jaundice alerts clinicians to the presence of liver disease. It does not discriminate between liver diseases, but it does delimit this disease area. Formally, a disease area constrained by a constrictor manifestation is a subtree of the disease tree, in this case the subtree of liver diseases.

### Reasoning

A problem for INTERNIST-I is to find a set of terminal nodes on the disease tree that account for a set of manifestations. It then chooses one node from the set and formulates another problem. INTERNIST-II does not start a diagnosis by formulating a set of terminal nodes, because the number of combinations of terminal disease nodes that may account for a set of manifestations is enormous. Instead, INTERNIST-II partitions the disease tree into disease areas, which collectively account for all the manifestations. Constrictor manifestations are used to make the partitions. If a patient manifests more than one constrictor, then the disease tree will be partitioned into more than one disease area. The conjunction of all the disease areas is called the *root structure* and is formally a set of subtrees of the disease tree. A root structure accounts for all the patient's manifestations. The problem for INTERNIST-II is to decide which terminal nodes (actual diseases) within the root structure best account for the manifestations. This objective is accomplished by partitioning the root structure into smaller subtrees in exactly the same way that the disease tree was partitioned into the root structure, namely, by using manifestations that strongly suggest a disease area, (only this time, the disease area is smaller). The process of partitioning the root structure into smaller areas continues just as long as all the manifestations are accounted for.

This is a summary account of the operation of INTERNIST-II. In actuality it is more complicated. See Pople (1977) for a complete explication. The main point of INTERNIST-II, however, is that it diagnoses a patient's diseases by dividing the disease tree into smaller and smaller subtrees, until such time as it achieves a set of terminal nodes that account for all the manifestations.

#### Concluding Remarks

INTERNIST I and II have successfully combined a bottom-up and top-down approach to medical diagnosis. The patient data evoke certain disease hypotheses (bottom-up) that are then used to predict (top-down) other manifestations that should be present if the hypothesis is to be confirmed. The system is purely associational. It does not attempt to model any disease processes but considers a disease as a static category and diagnosis as the task of assigning a patient to one or more categories. INTERNIST-I has a large database, currently containing over 500 of the diseases of internal medicine (about 75% complete). It has displayed expert performance in complex cases involving multiple diseases. Pople and Myers expect that the system will be in clinical use in the next few years.

#### References

See Pople (1975) and Pople (1977).

### E. Present Illness Program - PIP

The Present Illness Program (PIP) is being developed at MIT (Pauker et al., 1976; Szolovits & Pauker, 1976; Szolovits & Pauker, 1978). It has been used for taking present illnesses of patients with edemas (accumulation of excess fluids in the body) and patients with renal (kidney) disease. Taking a present illness is different from performing a complete diagnosis. It is the typical consultation a patient has with a general practitioner; the patient usually presents a *chief complaint* that becomes the initial focus of the consultation and only very low-cost sources of information--such as patient history, physical examination, and routine lab tests--are used to make a diagnosis. High-cost or high-risk procedures that may be necessary for a complete diagnosis are not used.

The medical knowledge in PIP is represented as a network of *frames* (see article Representation.B7). The frames are centered around diseases, clinical states, and physiological states (hereafter called the "patient situation") and contain data such as typical findings, relationships to other patient situations, and rules for judging how well a set of *findings* exhibited by a patient "match" the situation described by the frame. *Matching* is the key strategy in the diagnosis. Diagnosis involves matching findings to disease frames and then selecting a set of frames that cover all of the findings. There are, at present, 36 frames for dealing with renal disease.

Currently, the program does not prescribe treatment recommendations. Originally the system was written in CONNIVER [AIH ref], but this version was too slow and it has been recoded to run in MACLISP.

### Representation of Medical Knowledge

The general medical knowledge in PIP is knowledge about diseases, the patient situation; findings, results of the physical examination and reported symptoms; and the relationships between these entities. This medical knowledge is organized as a *frame system*. Part of a typical frame is shown in Figure 1.

#### ACUTE-GLOMERULONEPHRITIS

##### Typical Findings

TRIGGERS	(EDEMA with LOCATION=FACIAL.....)
FINDINGS	(ANOREXIA.....)

##### Logical Decision Criteria

IS-SUFFICIENT	(None)
MUST-HAVE	(None)
MUST-NOT-HAVE	(None)

##### Complementary Relations to Other Frames

CAUSED-BY	(STREPTOCOCCAL-INFECTION, ...)
CAUSE-OF	(SODIUM-RETENTION, ...)
COMPLICATED-BY	(ACUTE-RENAL-FAILURE, ...)
COMPLICATION-OF	(CELLULITIS)

##### Differential Diagnosis

CHRONIC-HYPERTENSION implies CHRONIC-GLOMERULONEPHRITIS  
 RECURRING-EDEMA implies NEPHROTIC-SYNDROME

#### Scoring

```
(((PATIENT WITH AGE=CHILD) -> 0.8)
 ((PATIENT WITH AGE=MIDDLE-AGED) -> -.5)
 )
(((EDEMA with SEVERITY = not MASSIVE) -> 0.1)
 ((EDEMA with SEVERITY = MASSIVE) -> -1.0)
 )
```

Figure 1. Part of the frame for acute glomerulonephritis (kidney stones).

The slots in the frame are grouped into categories as shown. The *typical findings* are those that are expected in a patient having the disorder that is described by the frame. However, patients with the disorder need not exhibit all of the typical findings. It is the job of the matching algorithm to compute a "goodness of fit" of findings about a patient to a frame. Some of the typical findings have the special status TRIGGER. TRIGGERS are key elements of the clinical decision-making strategy. A TRIGGER is a finding that is sufficiently strongly related to a disorder that presence of the disorder in the patient makes the PIP system attend to the disorder frame as an *active hypothesis*. For example, FACIAL EDEMA is listed above as a TRIGGER for ACUTE GLOMERULONEPHRITIS, meaning that PIP will consider this disease as an active hypothesis if a patient displays facial edema.

The *logical decision criteria* are rules that permit the confirmation or rejection of a hypothesis on the basis of a small number of key findings. Findings strongly correlated with a disease will be listed in the slot IS-SUFFICIENT. If any of these findings are reported, they will be sufficient to confirm the presence of the disease.

The relations between frames reflect the ways in which disorders are related in medicine. Sometimes disease mechanisms are well understood and it is possible to say that one disorder CAUSES another or is a COMPLICATION-OF another. If mechanisms are poorly understood, the disorders may simply be ASSOCIATED. The latter frames are complementary, that is, they represent other disorders that the patient might have in addition to the initial disorder. In contrast, the *differential diagnosis* slots indicate mutually exclusive disorders--the patient may have one of them and not the disorder represented by the current frame.

The final slot indicates how the findings are scored for the disorder represented by the frame. This score indicates the "goodness of fit" of this disorder to the findings. The statements comprising this slot are sets of clauses that are evaluated in turn. Within a clause, evaluation terminates when one of the conditions in the clause is true; its score will be used. The local score for a frame is the sum of the values of the clauses, normalized by the maximum total score possible. Thus, 1 denotes complete agreement, while arbitrarily large negative numbers denote complete disagreement.

### Reasoning

The clinical strategy used by PIP is based on the manipulations of hypotheses and findings. Knowledge about findings is stored separately from the frame system since a finding can be applicable to many frames. A hypothesis is an instantiation of a disorder frame. There are 3 kinds of hypotheses: (a) confirmed, (b) active, and (c) semi-active. Hypotheses with ratings (as computed by the scoring process) that are higher than a preset threshold

are considered *confirmed hypotheses*. *Active hypotheses* are those with at least one confirmed trigger finding; and these contend for the focus of attention. *Semi-active hypotheses* are the immediate neighbors of the active hypotheses in the frame system. They correspond to hypotheses that, although not strong enough to be investigated, are "at the back of the consultant's mind."

The consultation begins with the physician telling the system the main symptoms and signs of a patient. The program then takes the initiative and tries to determine the validity of any active hypotheses by selecting and asking appropriate questions.

The program works through the following cycle:

1. Acquire a new finding. This task is accomplished by asking a sequence of questions that characterizes the finding according to its possible descriptions.
2. Process the finding. All of the frames where this finding is relevant are located.
3. Update the list of active hypotheses. Several kinds of actions can be taken at this point: Remove an active hypothesis if the finding matches a MUST-NOT-HAVE rule; confirm a hypothesis if the premise of an IS-SUFFICIENT rule is now true; activate a hypothesis if the new finding is one of the hypothesis triggers or if the finding allows the premise of a differential diagnosis link to succeed; or revise the score of the hypothesis if the finding matches a scoring rule. If a new hypothesis is activated, then all of its immediate relatives are made into semi-active hypotheses.
4. Select the next finding to ask about. The highest rated hypothesis becomes the focus of attention, and a question is generated for the next unexplored finding. If there are no hypotheses, a question about a finding for the highest rated causally related frame is asked. Questioning terminates when there are no more active hypotheses or causal relatives with findings to be determined.

If the logical decision criteria are insufficient to confirm or deny a hypothesis, the score of the hypothesis is computed by combining (a) the value of a function that measures the fit of observed findings and typical (expected) findings for the frame (called the *matching score*), and (b) the value of a function that is the ratio of the number of findings accounted for by the hypothesis to the total number of findings (the *binding score*). The



matching score in turn consists of two parts, a local score for the frame (described above) and a score propagated from causally related frames.

### Concluding Remarks

Like INTERNIST (see article C2) and unlike MYCIN (see article C1), PIP is intended to simulate the clinical reasoning of physicians. The way in which the general medical knowledge has been represented as a system of hypothesized disorder frames and clinical findings reflects this intent, as do the strategies used to select questions for confirming a hypothesis.

The system uses two types of reasoning, *categorical* and *probabilistic*. Decisions about the applicability of a hypothesis are determined using logical decision criteria (the IS-SUFFICIENT, MUST-HAVE and MUST-NOT-HAVE rules) that a physician uses. When these are insufficient, the probabilistic methods (the computation of matching scores and binding scores) are used. Both kinds of reasoning feature a combination of local and global decision strategies. Local strategies decide how well the findings fit a particular frame, while global strategies determine how well a set of frames fits the findings.

There are a number of difficulties with the program. The questioning can be erratic, since the top-ranked hypotheses tend to alternate rapidly. This oscillation is unlike a physician's line of reasoning, which tends to concentrate on questions that resolve one hypothesis at a time. There is also the problem of when to stop the questioning. The current approach is to stop questioning only when all questions about all possibly relevant hypotheses have been exhausted. This strategy seems too conservative; many irrelevant questions tend to get asked.

### References

See Pauker et al. (1976), Szolovits & Pauker (1976), and Szolovits & Pauker (1978).

### F. Digitalis Advisors

There has been considerable work at MIT to develop programs that advise physicians on the administration of the drug digitalis (Silverman, 1974; Swartout, 1977a; and Swartout, 1977b). These programs are not concerned with diagnosing the need for the drug in a patient; rather they determine an appropriate treatment regimen and its subsequent management for patients known to require digitalis.

Digitalis is administered to patients with erratic heart-beat to stabilize the heart rhythm. The therapeutic effect of digitalis is achieved by maintaining the proper amount of the drug in the bloodstream. The body, however, excretes the drug through the kidneys and liver. Furthermore, overdoses of digitalis are toxic and can cause the very symptoms the drug is prescribed to cure. A typical digitalis regimen consists of an initial dose that is then modified in response to the effects of the drug on the patient and to the amount of drug being passed by the kidneys.

A mathematical model of the effect of Digitalis in the body has existed since 1967

which accounts for the relation between the level of body drug stores (as effected by body weight, renal function, etc.) and the incidence of Digitalis toxicity. However, application of this model requires that a physician adjust the dosages of Digitalis recommended by the model to allow for special sensitivity a patient might have (or might develop) to the drug. A skilled physician is still required to monitor a patient's progress after the initial dose of Digitalis is recommended by the mathematical model.

More recently at MIT, pauk75-??? have developed a program which makes a model of the effect of Digitalis in a *specific* patient, and modifies the model in response to feedback about the patient over time. Previously, serum (blood) levels of Digitalis had been used to provide feedback, but they proved unsatisfactory alone, and now clinical signs (e.g. nausea, or increased heartbeat irregularity) are used to assess whether the patient is responding well to Digitalis, or suffering from the toxic effects of the drug, or having no reaction at all.

The combination of a general, mathematical model of the effects of Digitalis, and a patient-specific model which incorporates clinical data in a continuous feedback cycle has resulted in a Digitalis advisor program which performs comparably to expert cardiologists in the prescription and maintenance of digitalis therapy.

The Digitalis advisory programs operate as follows:

1. Interact with a physician to build a model of a *specific* patient. This involves asking about the heart condition that Digitalis will regulate, about the condition of the kidneys, etc.
2. Prescribe a daily dosage level on the basis of the patient's height and weight, and the information gathered in (1).
3. Follow-up sessions assess the patient's response to the dosage recommendations in (2), and modify the dosage if necessary.

In 1977 this program was augmented with the ability to make explanations, but for the sake of brevity (and because article CB is devoted to this issue) we will not discuss it here. See Swartout, 1977a for a complete discussion of the explanation facility.

**THE PATIENT-SPECIFIC MODEL**

What follows is part of an annotated transcript with the Digitalis advisor.

What is the age of the patient?  
Enter numerical value \*\* 56

What is the weight of the patient?  
Units: Kg  
Enter numerical value \*\* 72

What is the sex of the patient?  
The possibilities are:  
1. Male  
2. Female  
Enter single value \*\* MALE

What is the cardiac rhythm?  
The possibilities are:  
1. Normal sinus rhythm  
2. Atrial fibrillation  
3. Atrial flutter  
4. Pat  
5. Pat with block  
6. Non-paroxysmal junctional tachycardia  
Enter single value \*\* 2

*So far the system is determining what it needs to know to make a patient-specific model. Later, it will ask why the patient is being put on Digitalis, whether the patient is currently using the drug, etc.*

Is there regularization of ventricular rhythm?  
The possibilities are:  
1. Present  
2. Absent  
Enter single value \*\* WHY?

*The user of the system wants an explanation of what motivates a question. The system responds by converting the goals on its goal stack to English to produce the first two sentences of the explanation. The system then finds and displays a canned explanation which is associated with the concept that the question is being asked about. The canned explanation usually provides information which could not be deduced from the code of the Advisor:*

My top level goal is to begin therapy. I am now trying to check the cardiac rhythm. If the cardiac rhythm is atrial fibrillation and there is ventricular-rhythm regularization this suggests that the pacemaking impulses from the ventricles are no longer coming from the atria. If the status of ventricular-rhythm regularization is present it is possible there may be non-paroxysmal junctional tachycardia which is a sign of digitalis toxicity.

*The system continues to probe the user so that it can build a patient-specific model. Eventually it arrives at the following point:*

How long do you wish to take to reach full digitalization?

The possibilities are:

1. Instantaneous rate (1-2 hours)
2. Rapid rate (1 day)
3. Moderate rate (2-3 days)
4. Slow rate (4-7 days)

Enter single value \*\* 3

Parameters used to make recommendations:

Body store goal: 0.544Mg

Projected amount on board: 0.000Mg

Body stores shift: 0.544Mg

Remaining time to reach goal: 48.0 hours

Half-life: 43.7 hours 1.8 days

Daily loss at equilibrium: 0.175 Mg

Absorption factor: 0.75

*The system then asks how many times a day the user wishes to administer Digitalis, and on the basis of this makes dosage recommendations.*

*The next interaction with the Advisor comes in a follow-up session. The point of this session is for the Advisor to determine whether the patient shows any therapeutic effect of the drug, or whether s/he displays any signs of toxicity, and adjust the dosage accordingly. There are three levels of therapeutic effect: None, Partial, and Complete. Similarly, there are three levels of toxicity: None, Possible, and Definite. There are therefore nine therapeutic/toxic states, and each has an associated set of recommendations associated with it. Digitalis toxicity is identified by five different types of signs and symptoms, including non-cardiac signs (nausea, etc.), and direct cardiac signs of toxicity (e.g. and increase of over 20% in the number of premature ventricular contractions). If any cardiac manifestations are present, the patient is considered definitely toxic; the category of Possibly toxic is indicated by various combinations of signs and symptoms from classes other than the cardiac signs.*

*We will not consider the follow-up session in detail here. See Swartout, 1977a for a complete transcript.*

### Summary

The performance of the Digitalis advisor reported in gorr78-??? suggest that the advisor can perform at least as well as physicians in the prescription and monitoring of Digitalis therapy. In particular, the advisor was used to make recommendations about therapy for a group of patients who were under the care of house staff in a hospital, with the advice of an attending physician.)

## G. IRIS

The design goals for IRIS (Trigoboff & Kulikowski, 1977; Trigoboff, 1978) are different from those of the other consultation systems constructed to be expert clinical decision-making systems in a particular medical domain. IRIS was designed to be a tool for building and experimenting with such systems.

Developed at Rutgers University and written in INTERLISP, it was designed to permit easy experimentation with alternative representations of general medical knowledge, clinical strategies, and modes of interaction. It was designed to be used by a computer specialist in collaboration with a domain expert. A consultation system for glaucoma has been developed using IRIS.

IRIS uses a combination of two well-established representation formalisms for representing knowledge, *semantic nets* and *production rules* (see articles Representation.B2 and Representation.B3). The semantic net consists of nodes representing patient information and uses a large and extendable set of *link types* for associating this medical knowledge. A set of production rules is associated with each link of the network. The transmission of information between nodes of the semantic network is controlled by the production rules. This process is called *propagation* and is the basis of any clinical strategy implemented in IRIS.

### Representation of Medical Knowledge

As with the other medical consultation systems, IRIS makes a (very sharp) distinction between the general medical knowledge and any patient-specific knowledge. The general medical knowledge is represented partly as a semantic net and partly as production rules. The nodes of the net represent clinical concepts such as pathophysiological states, diseases, symptoms, findings, treatments, etc. Examples of nodes in the glaucoma application are OPEN ANGLE GLAUCOMA, SCOTOMA, PILOCARPINE THERAPY. The links represent relations between the nodes--e.g., CAUSES, TREATMENT-FOR, SYMPTOM-OF, ASSOCIATED-WITH.

The patient-specific knowledge gathered during a consultation is represented as a set of knowledge structures called "Information SPECifications" (ISPECs). ISPECs are associated with nodes of the semantic net and are created, deleted, and modified during the course of the consultation. An ISPEC is an assertion about the patient and is essentially a frame (see article Representation.B7) with the following slots:

**NODE** - The name of the associated node in the semantic net. The node represents the concept being asserted about this patient.

**SIDE** - This slot indicates the half of the body to which this ISPEC refers. Its possible values are LEFT, RIGHT, or NIL. Some nodes in the net will be applicable to a left organ and a right organ (e.g., eye) while others are not (e.g., headache, diabetes). The use of SIDE provides an economical representation, since many nodes might otherwise be duplicated in the net.

**MB** - This slot is a measure of belief that reflects the degree of system belief in the assertion represented by the ISPEC. Any numerical method of

representing degrees of certainty can be implemented here. In the glaucoma application, the confidence factor mechanism of MYCIN (see article C1) has been implemented. The MB is a pair of numbers: SB (strength of belief) and SD (strength of disbelief). The actual MB is the difference between these two numbers and ranges from total belief to total disbelief.

**TIME** - The time slot is a list of two dates, the date the ISPEC became true of the patient and the date the ISPEC ceased to be true. The system can also work with a "coarser" view of time: PAST, PAST-OR-PRESENT, and FUTURE. This time representation is part of the mechanism for dealing with multiple visits and for following a patient through a given course of therapy.

**MODIFIERS** - These are further specifications and qualifications of the basic ISPEC. Examples of modifiers are VALUE, DEGREE, COLOR, and WIDTH. These modifiers do not appear in all ISPECs, but only in those to which they are applicable. These modifiers allow further patient-specific specifications of the concept in the semantic net. For example, "severely increased intraocular pressure" is represented as an ISPEC for INCREASED INTRAOCULAR PRESSURE with modifier DEGREE: SEVERE.

**TYPE** - The type slot of an ISPEC determines the way in which it is interpreted. An arbitrary number of types is possible. Currently implemented TYPES are NIL (the standard and default), FAMILYHISTORY, PATIENTHISTORY, and a number of others that are used by the diagnosis strategy--CHOSEN, COVERED-BY, SUBSUMED-BY, and TREATED-BY.

The statement "The pressure is 10 in the right eye" is equivalent to the ISPEC:

NODE = INTRAOCULAR PRESSURE  
 SIDE = RIGHT  
 MB = (1,0)  
 TIME = PRESENT  
 MODS = VALUE: 10  
 TYPE = NIL

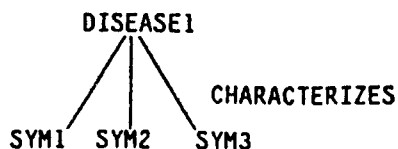
### Reasoning

IRIS makes no commitment to any particular strategy of question selection. Currently a "questionnaire" strategy has been implemented. At the beginning of a consultation the program runs through a set of questions and the user answers them.

In the applications of IRIS where consultation and diagnosis are the goal, ISPECs are associated first with the set of symptoms displayed by the patient. In IRIS's knowledge base, symptom nodes are linked to, among other things, disease nodes. Thus, a set of disease nodes can be activated by the symptoms; a disease node is said to explain the symptom nodes that characterize it. Disease nodes are also linked to treatment nodes, and when IRIS has determined which disease(s) holds for a patient, it will activate the appropriate (linked) treatment nodes.

The process of nodes evoking each other in IRIS is called *propagation of ISPECs*, because an ISPEC is associated with a symptom, or disease, or treatment node relevant to a patient. When symptoms evoke a disease or when a disease evokes a treatment, an ISPEC is created. This propagation of information and generation of inferences between any linked nodes in the semantic net is controlled by a set of production rules associated with the link. If the ISPECs associated with the node at the tail of the link satisfy the precondition pattern of a rule, then the actions specified by the rule will be performed at the node at the head of the link. Typical actions include the creation or deletion of ISPECs and the modification of MBs. Thus, IRIS uses a *forward-chaining* reasoning process.

An important propagation pattern is that of the "propagation cone." Consider the rule:  
 if SYMPTOM1 and SYMPTOM2 and SYMPTOM3 then DISEASE1  
 In the semantic net, the nodes in this rule would be represented as follows:



Clearly, an ISPEC should only propagate to DISEASE1 if all three symptoms are present. In the case depicted above, propagation should be from the base of the "cone" to the "apex." This propagation pattern is achieved by (Essentially AND-ing SYM1, SYM2, and SYM3 into one production to insure that ALL symptoms are present before a disease node is evoked, associating the same decision table with all three CHARACTERIZES links.) In some cases the direction of propagation will be from apex to base; for example, when propagating "COVERED-BY" ISPECs from a treatment node to each of the diseases it treats.

The production rules are encoded as *decision tables* to make their execution more efficient. Consider the following set of production rules:

- R1: If A and B then D
- R2: If B and (not C) then (not E)
- R3: If A and B and (not C) then F

In evaluating these rules, A and C are evaluated twice and B three times. A decision table encoding these three rules is:

	R1	R2	R3
A	+		+
B	+	+	+
C		-	-
	*	*	*
D	+		
E		-	
F			+

A column of the decision table corresponds to a rule. A condition is evaluated only once, and the result is used in each applicable column.

The IRIS claim is that any clinical strategy can be implemented using the available medical primitives. In fact, the propagation of weights in CASNET, therapy selection in MYCIN, and the formation of composite hypotheses in INTERNIST II were implemented with very little effort (Trigoboff, 1978).

#### Clinical strategy of IRIS for glaucoma diagnosis

The clinical strategy for the glaucoma application is implemented via a set of 6 special nodes in the semantic net: CHOSEN-DIAGNOSIS, CHOSEN-TREATMENT, POSSIBLE-DIAGNOSIS, POSSIBLE-TREATMENT, UNEXPLAINED-SYMPTOM, and UNTREATED-PATHOLOGY. The goal of the consultation is (a) to have one or more ISPECs associated with the nodes CHOSEN-DIAGNOSIS and CHOSEN-TREATMENT, and (b) to have all ISPECs associated with UNEXPLAINED-SYMPTOMS and UNTREATED-PATHOLOGY be TYPE=COVERED-BY. As findings are entered, they propagate ISPECs to the node UNEXPLAINED-SYMPTOMS. Propagation across SYMPTOM-OF links will result in ISPECs with varying CFs (confidence factors), associated with a number of disease nodes. Any disease with a high enough CF will propagate an ISPEC to the node POSSIBLE-DIAGNOSIS. After all data has been entered, the diseases associated with POSSIBLE-DIAGNOSIS are then investigated in turn. Each diagnosis temporarily receives TYPE=CHOSEN, and TYPE=COVERED-BY propagates to each symptom explained by this disease. The number of explained symptoms is used as a measure of the explanatory power of a disease. This process, of temporary assignment, is repeated for each possible diagnosis; and the disease that explains the most symptoms is given a permanent TYPE=CHOSEN. If there are any unexplained symptoms, the process is repeated.

A similar strategy using the nodes POSSIBLE-TREATMENT, CHOSEN-TREATMENT, and UNTREATED-PATHOLOGY is used to select treatments.



### Concluding Remarks

IRIS has been explained in the context of its Glaucoma application, but it was designed to represent medical knowledge from ANY domain, and to implement a variety of clinical strategies. (Recall that aspects of CASNET, MYCIN, and INTERNIST-II have all been implemented in IRIS.)

This generality is feasible because the representation of knowledge is itself very general (augmented semantic nets.) In principle, knowledge from any (medical or non-medical) domain can be represented. A second characteristic of IRIS that makes it very general is the separation of clinical strategy, both conceptually and operationally, from medical knowledge. Note that to implement the "consultation" strategy, IRIS needed to "know about" only six nodes in the knowledge base: chosen diagnosis, chosen treatment, possible diagnosis, possible treatment, unexplained symptom, and untreated pathology. These six concepts are inherent to the clinical strategy of consultation; every other node in the knowledge base is conceptually and operationally independent of the implementation of the clinical strategy.

### References

See Trigoboff & Kulkowski (1977) and Trigoboff (1978).

## References

- AIM Workshop Proceedings, Proceedings of the 1st Fourth Annual Aim Workshops 1975-8, Dept. of Computer Science, Rutgers University, 1975-1978.
- Clancey, W. Tutoring rules for guiding a case method dialogue. *International Journal of Man-Machine Studies*, 1979, 11, 25-49.
- Croft, J. Is Computerized Diagnosis Possible? *Computers and Biomedical Research*, 1972, 5(4), 361-367.
- Davis, R. Applications of Meta-Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases, Stanford AI Lab Memo AIM-283, AI Lab, Stanford University, 1976. (a)
- Davis, R. Interactive transfer of expertise: Acquisition of new inference rules. *IJCAI* 5, 1977, 321-328.
- Davis, R., & Buchanan, B. Meta-level knowledge: Overview and Applications. *IJCAI* 5, 1977, 920-928.
- Davis, R., & Buchanan, B. Knowledge acquisition in rule-based systems: Knowledge about representations as a basis for system construction and maintenance. In D. Waterman & F. Hayes-Roth (Eds.), *Pattern-directed Inference Systems*. New York: Academic Press, 1978. Pp. 99-134.
- Davis, R., Buchanan, B., & Shortliffe, E. H. Production Rules as a Representation for a Knowledge-base Consultation Program. *Journal of Artificial Intelligence*, 1977, 8(1), 15-45.
- Felgenbaum, E. A. The art of artificial intelligence: Themes and case studies in knowledge engineering. *IJCAI* 5, 1977, 1014-1029.
- Feinstein, A. *Clinical Judgment*. Baltimore: William & Wilkins, 1967.
- Gorry, A., & Barnett, O. Sequential diagnosis by computer. *Journal of the American Medical Association*, 1968, 205, 849-854.
- Heiser, J. A computerized Psychopharmacology Advisor. HEAD-MED Report in the SUMEX Annual Report. Computer Science Dept., Stanford University, 1977-1978.
- Jacquez, J. A. *The Diagnostic Process*. Ann Arbor, Mich.: Mallory Lithography, 1964.
- Ledley, R., & Lusted, L. Reasoning foundations of medical diagnosis. *Science*, 1959, 130(3366), 9-21.
- Nordyke, R., Kulikowski, C. A., & Kulikowski, C. W. A Comparison of Methods for the Automated Diagnosis of Thyroid Dysfunction. *Computers and Biomedical Research*, 1971, 4(4), 374-389.

- Pauker, S., Gorry, A., Kassirer, J., & Schwartz, W. Towards the Simulation of Clinical Cognition--Taking a Present Illness by Computer. *American Journal of Medicine*, June 1976, 60, 981-996.
- Pople, H. The formation of Composite Hypotheses in Diagnostic Problem Solving--An Exercise in Synthetic Reasoning. *IJCAI 5*, 1977, 1030-1037.
- Pople, H. E. The DIALOG Model of Diagnostic Logic and Its Use in Internal Medicine. *IJCAI4*, Tbilisi, USSR, 1975.
- Safrans, C., Desforges, J., & Tsihliis, P. *Diagnostic Planning and Cancer Management*, MIT/LCS/TR-169, MIT, 1976.
- Shortliffe, E. H. *Computer-Based Medical Consultations: MYCIN*. New York: Elsevier, 1976.
- Silverman, H. A Digitalis Therapy Advisor, MAC TR-143, Computer Science Dept., MIT, 1974.
- Swartout, W. A Digitalis Therapy Advisor with Explanations, MAC TR-176, Computer Science Dept., MIT, 1977. (a)
- Swartout, W. A Digitalis Therapy Advisor with Explanations. *IJCAI 5*, 1977, 819-825. (b)
- Szolovits, P., & Pauker, S. Research on a Medical Consultation Program for Taking the Present Illness. *Proc. 3rd Illinois Conf. on Medical Information Systems*, November 1976.
- Szolovits, P., & Pauker, S. Categorical and Probabilistic Reasoning in Medical Diagnosis. *Journal of Artificial Intelligence*, 1978, 10. In press.
- Trigoboff, M. *IRIS: A Framework for the construction of Clinical Consultation Systems*. Doctoral dissertation, Dept. of Computer Science, Rutgers University, 1978.
- Trigoboff, M., & Kulikowski, C. IRIS: A System for the Propagation of Inferences in a Semantic Net. *IJCAI 5*, 1977, 274-280.
- Weiss, S., Kulikowski, C., & Safir, A. A Model-Based Consultation System for the Long-Term Management of Glaucoma. *IJCAI 5*, 1977, 826-832.
- Weiss, S., Kulikowski, C., & Safir, A. A Model-Based Method for Computer-Aided Medical Decision-Making. *AI Journal*, August 1978. In press.

## Index

- action clause 583
- active hypotheses 600
- active hypothesis 599
- AND/OR tree 586
- ANNA 602
- antimicrobial therapy 581-589
- associations, INTERNIST 594
- associative triple 584
- attribute 584, 586
- attribute-value 586
- augmented links 605
  
- backward chaining 585
- binding score 600
- bottom-up approach 592, 597
  
- CASNET 577, 589-593, 608
- CASNET/GLAUCOMA 589-593
- categorical reasoning 601
- causal disease pathway 590
- causal model 589-593
- causal network 589-593
- certainty factor 586
- certainty factor, MYCIN 584
- CF, certainty factors 584, 585
- classification tables 590, 591
- clause 584
- clinical reasoning 578
- clinical strategy 1, 595-596
- clinical strategy, IRIS 606-610
- complementary frames 599
- conclusion 585
- confidence factor 584, 590
- confidence factors 608
- confirmed hypotheses 600
- confirmed states 591
- CONNIVER 598
- constrictor relation 596
- consultation systems 1
- cost 590, 594
  
- decision criteria 599
- decision tables 607
- denied states 591
- depth-first search 585
- diagnostic reasoning 1, 595-596
- diagnostic strategy 1
- differential diagnosis 599
- Digitalis Advisors 602
- disease area 594, 596
- disease category 589-593
- disease entity 594
- disease hypotheses 593
- disease model, INTERNIST 595
- disease node 594
- disease tree 593
- disorder frame 599
- dynamic disease process 589, 590
  
- EMYCIN 580
- EVOKE relation 594
- explanation 602
- explanatory diagnosis power 608
  
- findings 598
- findings, PIP 598
- forward-chaining 607
- frame relations, PIP 599
- frame system 598
- frames 598-601, 605
  
- glaucoma 589, 606
- glaucoma consultation system 605
- goodness of fit, PIP 599
  
- HEADMED 577
- HODGKINS 577
- hypotheses, active 599
- hypotheses, semi-active 600
- hypothesis confirmation 599
- hypothesis formation 577, 593

hypothesis rejection 599  
hypothesis status, CASNET 591  
hypothesis status, PIP 600

IMPORT property 594  
inexact inferences 584  
inexact knowledge 577  
inexact reasoning 1, 577, 585, 593  
infectious disease consultant system 581-589  
inference 605  
inference rules 585  
inferential rules 585  
INTERLISP 605  
Internal medicine consultation program 593  
INTERNIST 593, 601  
INTERNIST II 608  
INTERNIST-II 596  
IRIS 577, 605  
ISPEC 605-610

Judgemental reasoning 576  
justification 579

knowledge acquisition 587  
knowledge engineering 1

link types 605  
LISP 583

MACLISP 598, 602  
man/machine interactions 588  
MANIFEST relation 594  
manifestations 593  
matching 598  
matching score 600  
medical applications 1  
medical consultant systems 1  
medical decision making 1

medical diagnosis systems 1  
model, diagnostic reasoning  
(INTERNIST) 593  
MYCIN 577, 581-589, 601, 608  
MYCIN, Sample dialogue 581-583

natural language, MYCIN 587

object 584  
OWL 579

pathway 590  
patient-specific model 602  
PIP 598-601  
planes of knowledge 589  
plausible reasoning 1, 577, 595, 596  
predicate function 584  
preference categories 587  
premise 585  
premise clause 583  
Present Illness Program 598-601  
probabilistic reasoning 601  
problem 593  
production rules 583, 605  
production system 581-589  
productions, MYCIN 587  
propagation 605  
propagation of ISPECs 607  
propagation, IRIS 605  
PUFF 577, 580

representation of medical knowledge 1,  
589, 593-595, 596  
representation, clinical strategies 605  
representation, medical knowledge 605  
root structure 596

scoring 595  
semantic grammar 587

semantic net 605  
sequential diagnosis programs 577  
sequential processing, INTERNIST 596  
shelf 595  
simulation of clinical reasoning 601  
status 591  
status, of hypothesis in CASNET 591  
status, of state 591  
strategies 608  
strategy, reasoning 591  
supports 590

TEIRESIAS 587  
theory formation 593  
therapy selection 587  
thresholding 578  
top-down approach 597  
treatment regimen system 602  
TRIGGER key elements 599  
triggering hypotheses 600  
TYPE property 594

undetermined states 591  
unity path 586

validation 579  
value 584, 586

# Applications-oriented AI Research

## Part 4: Education

# AI Applications in Education

## Table of Contents

A. Historical Overview of Education Applications . . . . .	617
B. Issues and Components of ICAI Systems . . . . .	620
C. ICAI Systems . . . . .	626
1. SCHOLAR . . . . .	626
2. WHY . . . . .	631
3. SOPHIE . . . . .	636
4. WEST . . . . .	642
5. WUMPUS . . . . .	649
6. BUGGY . . . . .	655
7. EXCHECK . . . . .	659
References . . . . .	666
Index . . . . .	671



## A. Historical Overview of Education Applications

Educational applications of computer technology have been under development since the early 1960s. These applications have included scheduling courses, managing teaching aids, and grading tests. The predominant application, however, has involved using the computer as a device that interacts with the student directly, rather than serving as an assistant to the human teacher. For this kind of application, there have been three general approaches.

The "ad lib" or "environmental approach" is typified by Papert's LOGO laboratory (Papert, 1970), that allowed students more or less free-style use of the machine. Students are involved in programming; it is conjectured that learning problem-solving methods takes place as a side effect of using tools that are designed to suggest good problem-solving strategies to the student. The second approach uses games and simulations as instructional tools; once again the student is involved in an activity--for example, doing simulated genetics experiments--for which learning is an expected side effect. The third computer application in education is computer-assisted instruction (CAI). Unlike the first two approaches, CAI makes an explicit attempt to instigate and control learning (Howe, 1973). This third use of computer technology in education is the focus of the following discussion.

The goal of CAI research is to construct instructional programs that incorporate well-prepared course material in lessons that are optimized for each student. Early programs were electronic "page-turners" that printed prepared text and simple, rote drills; and practice monitors, which printed problems and responded to the student's solutions using prestored answers and remedial comments. In the intelligent CAI (ICAI) programs of the 1970s, course material is represented independently of teaching procedures so that problems and remedial comments can be generated differently for each student. Research today focuses on the design of programs that can offer instruction in a manner that is sensitive to the student's strengths, weaknesses, and preferred style of learning. The role of AI in computer-based instructional applications is seen as making possible a new kind of learning environment.

This overview surveys how AI techniques have been used in research attempting to create intelligent computer-based tutors. In the next article, some design issues are discussed and typical components of ICAI systems are described. Subsequent articles describe some important applications of artificial intelligence techniques in instructional programs.

### Frame-oriented CAI Systems

The first instructional programs took many forms, but all adhered to essentially the same pedagogical philosophy. The student was usually given some instructional text (sometimes without using the computer) and asked a question that required a brief answer. After the student responded, he was told whether his answer was right or wrong. The student's response was sometimes used to determine his "path" through the curriculum the sequence of problems he is given (see Atkinson & Wilson, 1969). When the student made an error, the program branched to remedial material.

The *courseware author* attempts to anticipate every wrong response, prespecifying branches to appropriate remedial material based on his ideas about what might be the underlying misconceptions that would cause each wrong response. Branching on the basis of response was the first step toward *individualization of instruction* (Crowder, 1962). This style of CAI has been dubbed ad-hoc, frame-oriented (AFO) CAI by Carbonell (1970b), to stress its dependence on author specified units of information. (The term "frame" as it is used in this context predates the more recent usage in AI--see Article Representation.B7-- and refers to a block or page or unit of information or text.) Design of ad-hoc frames was originally based on Skinnerian stimulus/response principles. The branching strategies of some AFO programs have become quite involved, incorporating the best learning theory that mathematical psychology has produced (Atkinson, 1972; Fletcher, 1975; Kimball, 1973). Many of these systems have been used successfully and are available commercially.

### Intelligent CAI

In spite of the widespread application of AFO CAI to many problem areas, many researchers believe that most AFO courses are not the best use of computer technology:

In most CAI systems of the AFO type, the computer does little more than what a programmed textbook can do, and one may wonder why the machine is used at all....When teaching sequences are extremely simple, perhaps trivial, one should consider doing away with the computer, and using other devices or techniques more related to the task. (Carbonell, 1970b, pp. 32, 193)

In this pioneering paper, Carbonell goes on to define a second type of CAI that is known today as "knowledge-based" or "intelligent" CAI (ICAI). Knowledge-based systems and the previous CAI systems both have representations of the subject matter they teach, but ICAI systems also carry on a natural language dialogue with the student, and use the student's mistakes to diagnose his misunderstandings.

Early uses of AI techniques in CAI were called "generative" CAI (Wexler, 1970), since they stressed the ability to generate problems using a large database representing the subject they teach. (See Koffman & Blount, 1975., for a review of some early generative CAI programs and an example of the possibilities and limitations of this style of courseware.) However, the kind of courseware that Carbonell was describing in his paper was to be more than just a problem generator--it was to be a computer *tutor* that had the inductive powers of its human counterparts. ICAI programs offer what Brown (1977) calls a *reactive learning environment*, in which the student is actively engaged with the instructional system and his interests and misunderstandings drive the tutorial dialogue. This goal was expressed by other researchers trying to write CAI programs that extended the medium beyond the limits of frame selection:

Often it is not sufficient to tell a student he is wrong and indicate the correct solution method. An intelligent CAI system should be able to make hypotheses based on a student's error history as to where the real source of his difficulty lies. (Koffman & Blount, 1975.)

### The Use of AI Techniques in ICAI

The realization of the computer-based tutor has involved increasingly complicated computer programs and has prompted CAI researchers to use artificial intelligence techniques. Artificial intelligence work in natural language understanding, representation of knowledge, and methods of inference, as well as specific AI applications like algebraic simplification, calculus, and theorem proving, have been applied by various researchers toward making CAI programs that are more intelligent and more effective. Early research on ICAI systems focused on *representation* of the subject matter. Benchmark efforts include SCHOLAR, the geography tutor of Carbonell and Collins (see article C1), EXCHECK, the logic and set theory tutors by Suppes et al. (article F7), and SOPHIE, the electronics troubleshooting tutor of Brown and Burton (article C3). The high level of domain expertise in these programs permits them to be responsive in a wide range of problem-solving interactions.

These ICAI programs are quite different from even the most complicated frame-oriented, branching program.

Traditional approaches to this problem using decision theory and stochastic models have reached a dead end due to their oversimplified representation of learning.... It appears within reach of AI methodology to develop CAI systems that act more like human teachers. (Laubsch, 1975)

However, an AI system that is expert in a particular domain is no necessarily an expert *teacher* of the material--"ICAI systems cannot be AI systems warmed over" (Brown, 1977). A teacher needs to understand what the student is doing, not just what he is supposed to do. AI programs often use very powerful problem-solving methods that do not resemble those used by humans; in many cases, CAI researchers borrowed AI techniques for representing subject domain expertise but had to modify them, often making the inference routines *less powerful*, in order to force them to follow human reasoning patterns, so as to better explain their methods to the student, as well as to understand his methods (Smith, 1976; Goldberg, 1973).

In the mid-1970s, a second phase in the development of ICAI tutors has been characterized by the inclusion of expertise in the tutor regarding 1) the student's learning behavior and 2) tutoring strategies (Brown & Goldstein, 1977). AI techniques are used to construct models of the learner that represent his knowledge in terms of "issues" (see article C4) or "skills" (Barr & Atkinson, 1975) that should be learned. This model then controls tutoring strategies for presenting the material. Finally, some ICAI programs are now using AI techniques to explicitly represent these *tutoring strategies*, gaining the advantages of flexibility and modularity of representation and control (Burton & Brown, 1979; Goldstein, 1977; Clancey, 1979a).

### References

The best general review of research in ICAI is Brown & Goldstein (1977). Several papers on recent work are collected in a special issue of the *International Journal of Man-Machine Studies*, Volume 11, 1979.

## B. Issues and Components of ICAI Systems

The main components of ICAI systems are 1) its problem-solving expertise, the knowledge that the system tries to impart to the student, 2) the student model, indicating what the student does and does not know, and 3) tutoring strategies, which specify how the system presents material to the student). (See Self, 1974, for an excellent discussion of the differences and interrelations of the types of knowledge needed in an intelligent CAI program.) Not all of these components are fully developed in every system. Because of the size and complexity of intelligent CAI programs, most researchers tend to concentrate their efforts on the development of a single part of what would constitute a fully usable system. Each component is described briefly below.

### The Expertise Module--Representing Domain Knowledge

The "expert" component of an ICAI system is charged with the task of generating problems and evaluating the correctness of student solutions. The CAI system's knowledge of the subject matter was originally envisioned as a huge static database that incorporated all the facts to be taught. This idea was implicit in the early drill-and-practice programs and was made explicit in *generative CAI* (see Article A). Representation of subject matter expertise in this way, using *semantic nets* (Article Representation.B2), has been useful for generating and answering questions involving causal or relational reasoning (Carbonell & Collins, 1973; Laubsch, 1975; and see Articles C1 and C2 on the SCHOLAR and WHY systems).

Recent systems have used *procedural representation* of domain knowledge, e.g., how to take measurements and make deductions (see Article Representation.B9). This knowledge is represented as *procedural experts* that correspond to subskills that a student must learn in order to acquire the complete skill being taught (Brown, Burton, & Bell, 1975). *Production rules* (Article Representation.B3) have been used to construct modular representations of skills and problem-solving methods (Goldstein, 1977; Clancey, 1979a). In addition, Brown & Burton (1975) have pointed out that *multiple representations* are sometimes useful for answering student questions and for evaluating partial solutions to a problem (e.g., a semantic net of facts about an electronic circuit and procedures simulating the functional behavior of the circuit). Stevens & Collins (1978) considered an evolving series of "simulation" models that can be used to metaphorically reason about the behavior of causal systems.

It should be noted that not all ICAI systems can actually solve the problems they pose to a student. For example, BIP, the BASIC Instructional Program (Barr, Beard, & Atkinson, 1975), can't write or analyze computer programs: BIP uses sample input/output pairs (supplied by the course authors) to test students' programs. Similarly, the procedural experts in SOPHIE-I could not debug an electronic circuit. In contrast, the production rule representation of domain knowledge used in WUMPUS and GUIDON enables these programs to solve problems independently, as well as to criticize student solutions (Goldstein, 1977, and Clancey, 1979a). Being able to solve the problems, preferably in all possible ways, correctly and incorrectly, is necessary if the ICAI program is to make fine-grained suggestions about the completion of partial solutions.

An important idea in this connection is that of an *articulate expert* (Goldstein, 1977).

Whereas typical expert AI programs have data structures and processing algorithms that do not necessarily mimic the reasoning steps used by humans and are, therefore, considered "opaque" to the user, an articulate expert for an ICAI system must be designed to enable the explanation of each problem-solving decision it makes in terms that correspond (at some level of abstraction) to those of a human problem solver. For example, the electronic circuit simulator underlying SOPHIE-I (see Article C3), which is used to check the consistency of a student's hypotheses and to answer some of his questions, is an opaque expert on the functioning of the circuit. It is a complete, accurate and efficient model of the circuit, but its mechanisms are never revealed to the student since they are certainly not the mechanisms that he is expected to acquire. In WEST, on the other hand, while a (competent and efficient) opaque expert is used to determine the range of possible moves that the student could have made with a given roll of the dice, an *articulate* expert, which only models pieces of the game-playing expertise, is used to determine possible causes for less-than-optimal student moves.

ICAI systems are distinguished from earlier CAI approaches by the separation of teaching strategies from the subject expertise to be taught. However, the separation of subject-area knowledge from instructional planning requires a structure for organizing the expertise that captures the difficulty of various problems and the interrelationships of course material. Modeling a student's understanding of a subject is closely related conceptually to figuring out a representation for the subject itself or for the language used to discuss it.

Trees and lattices showing prerequisite interactions have been used to organize the introduction of new knowledge or topics (Koffman & Blount, 1975.). In BIP this lattice took the form of a *curriculum net* that related the skills to be taught to example programming tasks that exercised each skill (Barr, Beard, & Atkinson, 1976). Goldstein (1979) called the lattice a *syllabus* in the WUMPUS program and emphasized the developmental path that a learner takes in acquiring new skills. For arithmetic skills used in WEST, Burton & Brown (1976a) use levels of *issues*. Issues proceed from the use of arithmetic operators to strategies for winning the game, to meta-level considerations for improving performance. Burton and Brown believe that when the skills are "structurally independent," the order of their presentation is not particularly crucial. This representation is useful for modeling the student's knowledge and coaching him on different levels of abstraction. Stevens, Collins, & Goldin (1978) have argued further that a good human tutor does not merely traverse a predetermined network of knowledge in selecting material to present. Rather, it is the process of ferreting out student misconceptions that drives the dialogue.

### The Student Model

The modeling module is used to represent the student's understanding of the material to be taught. Much recent ICAI research has focused on this component. The purpose of modeling the student is to make hypotheses about his misconceptions and suboptimal performance strategies so that the tutoring module can point them out, indicate why they are wrong, and suggest corrections. It is advantageous for the system to be able to recognize alternate ways of solving problems, including the incorrect methods that the student might use resulting from systematic misconceptions about the problem or from inefficient strategies.

Some early frame-oriented CAI systems used mathematical *stochastic learning models*, but this approach failed because it only modeled the probability that a student would give a specific response to a stimulus. In general, knowing the probability of a response is not the same as knowing what a student understands--the former has little diagnostic power (Laubsch, 1975).

Typical uses of AI techniques for modeling student knowledge include 1) simple *pattern recognition* applied to the student's response history and 2) flags in the subject matter semantic net or in the rule base representing areas that the student has mastered. In these ICAI systems, a student model is formed by comparing the student's behavior to that of the computer-based "expert" in the same environment. The modeling component marks each skill according to whether evidence indicates that the student knows the material or not. Carr & Goldstein (1977) have termed this component an *overlay model*--the student's understanding is represented completely in terms of the expertise component of the program (see Article C5).

In contrast, another approach is to model the student's knowledge not as a subset of the expert's, but rather as a perturbation or deviation from the expert's knowledge--a "bug". (See, for example, the SOPHIE and BUGGY systems--Articles C3 and C6.) There is a major difference between the overlay and "buggy" approaches to modelling: In the latter approach it is not assumed that the student reasons as the expert does, but simply knows less. Rather the student's reasoning can be substantially different from expert reasoning.

Other information that might be accumulated in the student model includes the student's preferred modes for interacting with the program, a rough characterization of his level of ability, a consideration of what he seems to forget over time, and an indication of what his goals and plans seem to be for learning the subject matter.

Major sources of evidence used to maintain the student model can be characterized as: (a) implicit, from student problem-solving behavior; (b) explicit, from direct questions asked of the student; (c) historical, from assumptions based on the student's experience; and (d) structural, from assumptions based on some measure of the difficulty of the subject material (Goldstein, 1977). Historical evidence is usually determined by asking the student to rate his level of expertise on a scale from "beginner" to "expert." Early programs like SCHOLAR used only explicit evidence. Recent programs have concentrated on inferring "implicit" evidence from the student's problem-solving behavior. This approach is complicated because it is limited by the program's ability to recognize and describe the strategies being used by the student. Specifically, when the expert program indicates that an inference chain is required for a correct result, and the student's observable behavior is wrong, how is the modelling program to know which of the intermediate steps are unknown or wrong applied by the student? This is the *apportionment of credit/blame problem*; it has been an important focus of WEST research.

Because of inherent limitations in the modelling process, it is useful for a "critic" in the modeling component to measure how closely the student model actually predicts the student's behavior. Extreme inconsistency or an unexpected demonstration of expertise in solving problems might indicate that the representation being used by the program does not capture the student's approach. Finally, Goldstein (1977) has suggested that the modeling process should attempt both to measure whether or not the student is actually learning and to discern what teaching methods are most effective. Much work remains to be done in this area.

### The Tutoring Module

The tutoring module of ICAI systems must integrate knowledge about natural language dialogues, teaching methods, and the subject area to be taught. This is the module that communicates with the student: selecting problems for him to solve, monitoring and criticizing his performance, providing assistance upon request, and selecting remedial material. The design of this module involves issues like "When is it appropriate to offer a hint?" or "How far should the student be allowed to go down the wrong track?"

These are just some of the problems which stem from the basic fact that teaching is a skill which requires knowledge additional to the knowledge comprising mastery of the subject domain. (Brown, 1977)

This additional knowledge, beyond the representation of the subject domain and of the student's knowledge, is about how to teach.

Most ICAI research has explored teaching methods based on *diagnostic modelling* in which the program debugs the student's understanding by posing tasks and evaluating his response (Collins, 1976; Brown & Burton, 1975; Koffman & Blount, 1975.). The student is expected to learn from the program's feedback of which skills he uses wrongly, which he does not use (but could use to good advantage), etc. Recently, there has been more concern with the possibility of saying the just right thing to the student so that he will realize his own inadequacy and switch to a better method (Carr & Goldstein, 1977; Burton & Brown, 1979; Norman, Gentner, and Stevens, 1976). This new direction is based on attempts to make a bug "constructive" by establishing for the student that there is something inadequate in his approach, and giving enough information so that the student can use what he already knows to focus on the bug and characterize it so that he avoids this failing in the future.

However, it is by no means clear how "just the right thing" is to be said to the student. We do know that it depends on having a very good model of his understanding process (the methods and strategies he used to construct a solution). Current research is focussing on means for representing and isolating the bugs themselves (Stevens, Collins, & Goldin, 1978; Brown & Burton, 1978a).

Another approach is to provide an environment that encourages the student to think in terms of debugging his own knowledge. In one BIP experiment (Wescourt and Hemphill, 1978), explicit debugging strategies (for computer programming) were conveyed in a written document and then a controlled experiment was undertaken to see whether this fostered a more rational approach for detecting faulty use of (programming) skills.

Brown, Collins, and Harris (1978) suggest that one might foster the ability to construct hypotheses and test them (the basis of understanding in their model) by setting up problems in which the student's first guess is likely to be wrong, thus "requiring him to focus on how he detects that his guess is wrong and how he then intelligently goes about revising it."

The Socratic method used in WHY (Stevens & Collins, 1977) involves questioning the student in a way that will encourage him to reason about what he knows, and so modify his conceptions. The tutor's strategies are constructed by analyzing protocols of real-world student/teacher interactions.

Another teaching strategy that has been successfully implemented on several systems is called *coaching* (Goldstein, 1977). Coaching programs are not concerned with covering a predetermined lesson plan within a fixed time (in contrast with SCHOLAR). Rather, the goal of coaching is to develop the acquisition of skill and general problem solving abilities, and it works by engaging the student in a computer game (see Article A). In a coaching situation, the immediate aim of the student is to have fun, and skill acquisition is an indirect consequence. Tutoring comes about when the computer coach, which is "observing" the student's play of the game, interrupts him and offers new information or suggests new strategies. A successful computer coach must be able to discern what skills or knowledge the student might acquire, based on his playing style, and to judge effective ways to intercede in the game and offer advice. WEST and WUMPUS (Articles C4 and C5) are both coaching programs.

Socratic tutoring and coaching represent different styles for communicating with the student. All mixed-initiative tutoring involves following some dialogue strategy. This will include decisions about when and how often to question the student, and methods for presentation of new material and review. For example, by design, a coaching program is not intrusive, and only rarely lectures. On the other hand, a Socratic tutor questions repetitively, requiring the student to pursue certain lines of reasoning. Recently ICAI research has turned to making explicit these alternative *dialogue management* principles. Collins (1976) has pioneered the careful investigation and articulation of teaching strategies. Recent work has explored the representation of these strategies as *production rules* (see Clancey, 1979a and Article C2 on Collins and Stevens' WHY system).

For example, the tutoring module in the GUIDON program, which discusses MYCIN-like "case diagnosis" tasks with a student (see Clancey, 1979a, and Article C1 on MYCIN), has an explicit representation of discourse knowledge. Tutoring rules select alternative dialogue formats on the basis of economy, domain logic, and tutoring or student modeling goals. Arranged into procedures, these rules cope with various recurrent situations in the tutorial dialogue: for example, introducing a new topic, examining a student's understanding after he asks a question that indicates unexpected expertise, relating an inference to one just discussed, giving advice to the student after he makes a hypothesis about a subproblem, and wrapping up the discussion of a topic.

### Conclusion

In general, ICAI programs have only begun to deal with the problems of representing and acquiring teaching expertise and of determining how this knowledge should be integrated with general principles of discourse. The programs described in the articles to follow have all investigated some aspect of this problem, and none offer an "answer" to the question of how to build a computer-tutor. Nevertheless, these programs have demonstrated potential tutorial skill, sometimes often showing striking insight into students' misconceptions. Research continues toward making viable AI contributions to computer-based education.



### References

Goldstein (1977) gives a clear discussion of the distinctions between the modules discussed here, concentrating on the broader, theoretical issues. Burton & Brown (1976a) also discuss the components of ICAI systems and their interactions and present a good example. Self (1974) is a classic discussion of the kinds of knowledge needed in for a computer-based tutor.

### C. ICAI Systems

#### C1. SCHOLAR

An important aspect of tutoring is the ability to generate appropriate questions for the student. These questions can be used by the tutor to indicate the relevant material to be learned, to determine the extent of a student's knowledge of the problem domain, and to identify any misconceptions that he might have. Given that the knowledge base of a tutoring program can't contain all of the "facts" that are true about the domain, the tutor should be able to reason about what it knows and make *plausible inferences* about facts in the domain. In addition to responding to the student's questions, the tutor should be able to take the initiative during a tutoring dialogue by generating good tutorial questions.

SCHOLAR is one such *mixed-initiative* computer-based tutorial system; both the system and the student can initiate conversation by asking questions. SCHOLAR was the pioneering effort in the development of computer tutors capable of coping with unanticipated student questions and of generating subject matter in varying levels of detail, according to the context of the dialogue. Both the student's input and the program's output are in English sentences.

The original system, created by Jaime Carbonell, Allan Collins, and their colleagues at Bolt, Beranek and Newman, Inc., tutored students about simple facts in South American geography (Carbonell, 1970b). SCHOLAR uses a number of *tutoring strategies* for composing relevant questions, determining whether or not the student's answers are correct, and answering questions from the student. Both the knowledge representation scheme (see below) and the tutorial capabilities are applicable to other domains besides geography. For example, NLS-SCHOLAR was developed to tutor computer-naive people in the use of a complex text-editing program (Grignetti, Hausman, & Gould, 1975).

In addition to investigating the nature of tutorial dialogues and human plausible reasoning, the SCHOLAR research project explored a number of AI issues, including:

1. How can real-world knowledge be stored effectively for the fast, easy retrieval of relevant facts needed in tutoring?
2. What general reasoning strategies are needed to make appropriate inferences from the typically incomplete database of the tutor program?
3. To what extent can these strategies be made independent of the domain being discussed (i.e., be dependent only on the *form* of the representation)?

#### The Knowledge Base--Semantic Nets

In SCHOLAR, knowledge about the domain being tutored is represented in a *semantic net* (see Article Representation.B2). Each node or "unit" in the net, corresponding to some geographical object or concept, is composed of the name associated with that node and a set of properties. These properties are lists of attribute-value pairs. For example, Figure 1 shows a representation of the unit for Peru:

PERU:

((EXAMPLE-NOUN PERU))

(1 0)

*"importance" of unit is high*

(SUPERC (1 0) COUNTRY)

(SUPERP (1 6) SOUTH/AMERICA)

*link to superordinate units*

(LOCATION (1 0)

*values of LOCATION attribute follow:*

(IN (1 0) (SOUTH/AMERICA (1 0) WESTERN))

(ON (1 0) (COAST (1 0) (OF (1 0) PACIFIC))

(LATITUDE (1 4) (RANGE (1 0) -18 0))

(LONGITUDE (1 5) (RANGE (1 0) -82 -68))

(BORDERING/COUNTRIES (1 1)

(NORTHERN (1 1) (LIST COLUMBIA ECUADOR))

(EASTERN (1 1) BRAZIL)

.

.

.

Figure 1. The unit for PERU.

Attributes can be English words (other units) that are defined elsewhere in the net or one of several special attributes such as "SUPRA" for superattribute, "SUPERC" for superconcept or superordinate, "SUPERP" for superpart, or case structure attributes used in parsing (see below). An example of SUPRA might be the fact that "fertile" refers to "soil" and "soil" refers to "topography"; of SUPERP, that Peru is part of South America; of SUPERC, that Peru is a country. Values can also be *importance tags*, like the expressions "(1 0)" after LOCATION in Figure 1 and "(1 1)" after EASTERN: the lower the number, the more important the property. SCHOLAR uses these tags to measure the relevance of a node with respect to the topic under discussion (see below).

### Tutorial Dialogue

An important part of the SCHOLAR research has been an emphasis on the nature of tutorial dialogue, in particular the Socratic style of tutoring, where the tutor attempts to diagnose the student's misconceptions and then present material that will force the student to see his own errors (Collins, Warnock, & Passafiume, 1974). Many human tutorial protocols have been analysed to see what strategies human students follow, in order to try to incorporate these strategies in the system (see also Article C2 on the groups continuing research in this direction). The following example is an excerpt from a session with the South American geography tutor. The student's questions and answers are preceded by "\*\*\*", and annotations are shown in *italics*:

What is the language of Argentina?

**\*\* PROBABLY SPANISH**

Very good.  
Approx what is the area of Argentina?

*Student changes topic...*

**\*\* TELL ME SOMETHING ABOUT PERU**

Peru is a country.  
It is located in South America  
The capital of Peru is Lima.  
Now answer the question you didn't answer before.

*...and finally answers the question.*

**\*\* THE AREA OF ARGENTINA IS 1,100,000 SQUARE MILES**

Approximately true, but a more exact value is 1,200,000.  
The eastern bordering country of Peru is ---.

*Student makes an error and is corrected...*

**\*\* CATHOLICISM**

Wrong. Catholicism is not a country but a religion.  
The correct answer is: Brazil  
Where is Peru?

*...and once again changes topic.*

**\*\* Is Asuncion hot?**

Since Asuncion is part of Paraguay, yes.

**\*\* WHAT IS THE MOST IMPORTANT DIFFERENCE BETWEEN PERU AND PARAGUAY?**

The primary difference is: The population of Peru is larger by a factor of 7.8.

The natural language processing used by SCHOLAR is based on a *case grammar* system (Article Natural Language.B3b), and the semantic interpretation of a student's response is heavily guided by the geography knowledge in the semantic net. Natural language processing is important in SCHOLAR because it allows the student to take the initiative and ask questions that were not anticipated by the program. However, comprehension of English input is facilitated by a taxonomy of expected "question types." The case grammar is also used by the system to construct questions and presentations of new material from the semantic network. English output is composed of short, simple sentences, with no embedded clauses and a limited repertoire of verbs--generally some form of the verb "to be."

A simple *agenda* is used to keep track of topics that are being discussed. Timing considerations and relevance (importance tags) affect the generation and pruning of topics

on this agenda. Continuity between questions is weak, however, since SCHOLAR does not plan a series of questions to make a point. SCHOLAR is capable of diagnosing a student's confusion only by following up one question with a related question.

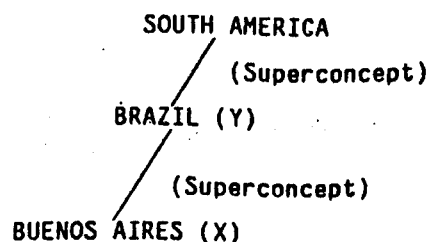
### Making Inferences

SCHOLAR's inference strategies, for answering student questions and evaluating student answers to its questions, are designed to cope with the incompleteness of the information stored in the semantic net database. Some of the important strategies used to reason with incomplete knowledge are given below. These abilities have been explored further in current research dealing with default reasoning (Reither, 1978) and plausible reasoning (Collins, 1978).

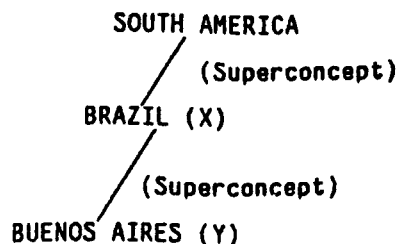
**Intersection search.** Answering questions of the form "Can X be a Y?" (e.g., "Is Buenos Aires a city in Brazil?") is done by an intersection search: The superconcept (SUPERC) arcs of both nodes for X and Y are traced until an intersection is found (i.e., a common superconcept node is found). If there is no intersection, the answer is "NO." If there is an intersection node Q, SCHOLAR answers as follows:

If  $Q=Y$ , then "YES";  
If  $Q=X$ , then "NO, Y IS AN X."

For example, the question "Is Buenos Aires in Brazil?" is answered YES because Brazil is a SUPERC of Buenos Aires in the net ( $Q=Y$ ):



But, the question "Is Brazil in Buenos Aires?" gets the response "NO, BRAZIL is a country."



**Common superordinate.** Otherwise, if Q is not X or Y, the program focuses on the two elements that have Q as a common superordinate. If they are contradictory (contain suitable CONTRA properties) or have distinguishing, mutually exclusive properties (e.g., different

LOCATIONS), the answer is "NO"; otherwise the system answers "I DON'T KNOW." Answering "Is X a part of Y?" questions is similar, except SUPERP (superpart) arcs are used for the intersection process.

**Open and closed sets.** In order to look for all objects in the system that satisfy some condition (e.g., "How many cities in Columbia are on the Atlantic?"), a distinction must be made about whether the resulting set of these objects is *closed* (explicitly contains all such objects) or *open* (contains some and need not contain all such objects). In SCHOLAR's net, sets are tagged by the course author as either open or closed, thus giving an indication of the *incompleteness* of the system's knowledge.

While it is easy for SCHOLAR to answer questions (such as, "Is France a country in South America?") about closed sets, similar questions about open sets require the use of special techniques--SCHOLAR must be able to reason about the *extent* of its knowledge. For example, if we ask SCHOLAR whether or not rubber is a product of, say, Guyana and if there is no explicit information about Guyanese rubber production in the semantic net, SCHOLAR starts to reason with what it knows: It compares the importance tags of the agricultural products of Guyana with the importance tags for rubber in countries that it knows produce rubber. It happens that Peru is a known rubber producer. SCHOLAR looks at the I-tag of rubber for Peru and compares this tag to the largest (least important) I-tags associated with Guyana. If it finds that these are comparable, SCHOLAR infers that it knows as much about Guyana (up to some large I-tag value) as it knows about Peru. Therefore SCHOLAR infers that it would know about rubber production in Guyana if it were important. Since rubber is not listed for Guyana, SCHOLAR makes the plausible (though uncertain) conclusion that rubber is not produced there. At this point SCHOLAR answers that it doesn't know for certain and gives the information about rubber and Peru.

SCHOLAR's use of knowledge about the extent of its knowledge in this kind of *plausible reasoning* is unique in AI research and represents an application of *meta-level knowledge* (see Representation.Overview).

### Summary

The inferencing strategies used by SCHOLAR are independent of the *content* of the semantic net, and are applicable in different domains. The inferences produced are fairly natural; that is, they cope with the incomplete knowledge by employing reasoning processes similar to those that people use. The SCHOLAR project as a whole provides an ongoing environment for research on discourse, teaching strategies and human plausible reasoning (see Article C2 on recent research, including the WHY system).

### References

Carbonell (1970a) is a classic paper, defining the field of ICAI and introducing the SCHOLAR system. Collins (1976) is an illuminating study of human tutorial dialogues. Collins et al. (1975) discusses inference mechanisms and Collins (1978) reports extended research on human plausible reasoning. Grignetti, Hausman, & Gould (1975) describes NLS-SCHOLAR.

## C2. WHY

Recent research by Allan Collins, Al Stevens, and their ICAI research group at Bolt, Beranek and Newman, Inc., has focused on developing computer-based tutors that can discuss complex systems. Their previous research on SCHOLAR (Article C1), a system that tutors facts about South American geography, led them to investigate the nature of tutorial dialogues about subject matter that was not just factual--where the causal and temporal interrelations between the concepts in the domain were of interest and where student's errors could involve not only forgotten facts, but also misconceptions about why processes work the way they do. Stevens & Collins (1977) are building a new system, called WHY, that tutors students in the causes of rainfall, a complex geophysical process that is a function of many interrelated factors; no single factor can be isolated that is both necessary and sufficient to account for rainfall.

In their research on tutorial dialogue of this type, the BBN group has focused on three questions that are central themes throughout ICAI research (Stevens, Collins, & Goldin, 1978):

1. How can a good tutor's use of questions, statements, and examples be characterized? What is the "goal structure" of a Socratic tutor? (See below.)
2. What types of misconceptions do students have? How do tutors diagnose these misconceptions from the errors students make?
3. What are the abstractions and viewpoints that tutors use to explain physical processes?

By analyzing tutorial dialogues between human experts and students, Collins and Stevens identify elements of a theory of tutoring. These are incorporated into a tutorial program, which is then used to find the weak points of the theory for further investigation. The current version of the WHY system is the first of a series of iterations of this sort. The work so far has concentrated on the first topic above, the nature of Socratic tutoring.

### Socratic Tutoring Heuristics

Collins (1976) argues that learning to reason about and understand complex processes is best accomplished by dealing with specific problems and cases and trying to generalize from them. Socratic dialogue is especially appropriate for tutoring complex subjects where factors interact and where their interaction accounts for the phenomenon under consideration. In an effort to explicitly model the nature of the Socratic dialogue the current version of the WHY system incorporates 24 *heuristics* which control the student/system interaction, like:

- If* the student gives as an explanation of causal dependence one or more factors that are not necessary,
- then* pick a counterexample with the wrong value of the factor and ask the student why his causal dependence doesn't hold in that case.

This rule forces the student to consider the necessity of a particular factor. For example, if the student gives rainfall as a reason for growing rice, then the computer-generated counterexample "Why do they grow rice in Egypt, where there isn't much rainfall?" challenges the student's explanation of rice growing. These heuristic rules are designed to facilitate tutorial dialogues in which students must consider combinations of factors that are necessary for rainfall, eliminate irrelevant factors, and attempt to generalize from specific cases to general conditions. (See Collins, 1976, for a complete discussion of the tutoring rules.)

WHY's knowledge base about rainfall is represented in *script*-like data structures, which encode temporal relations in the rainfall process; for example, "First water evaporates from an ocean or sea, then the water-laden air mass is carried over land by winds, which in turn causes the air mass to cool, which causes precipitation." (See Schank & Abelson, 1977, on scripts, as well as Article Representation.87.) This knowledge representation is adequate for capturing many of the characteristics of a tutorial dialogue, but there are other kinds of knowledge about rainfall that aren't represented here, discussed below.

The following excerpt is from a dialogue with the current version of the WHY system. The system prompts the student to suggest causes of rainfall, to look for prior or intermediate causes, and finally to suggest a general rule. When such a rule is proposed, the system finds a counterexample and forces revision of the rule to account for it. The system proceeds in this way until a rule of sufficient generality is found. The protocol is annotated with the heuristics employed by the system at each juncture in the conversation.

*Heuristic 1: Ask for a prediction about a particular case.*

Do you think the Amazon jungle has heavy rainfall or light rainfall?

\*\* HEAVY RAINFALL

*Heuristic 2: Ask for prior causes.*

Why does the Amazon jungle have heavy rainfall?

\*\* BECAUSE ITS NEAR THE ATLANTIC

Yes, the Amazon jungle is near the equatorial Atlantic.

*Heuristic 3: Ask for intermediate causes.*

How does that affect rainfall in the Amazon jungle?

\*\* THE WATER IN THE CURRENT IS WARM

*(Section Omitted)*

*Heuristic 4: Ask for subsequent causes.*

What happens after the moisture-filled air reaches the land?

\*\* I DON'T KNOW



*Heuristic 5: Suggest a factor and ask if it is a cause.*

Do you think the Andes mountain range has any effect on the rainfall in the Amazon jungle?

**\*\* YES**

*Heuristic 3: Ask for intermediate causes.*

How?

**\*\* THE PREVAILING WIND HITS THE ANDES AND THE AIR COOLS**

*Heuristic 3: Ask for intermediate causes.*

How does the fact that prevailing wind hits the Andes, lead to air cooling?

*(Section Omitted)*

*Heuristic 6: Form a general rule for an insufficient cause*

Do you think that any place with mountains has heavy rainfall?

**\*\* YES**

*Heuristic 7: Pick a counterexample for an insufficient cause, and Heuristic 2: Ask for prior causes.*

Southern California has mountains. Why doesn't Southern California have heavy rainfall?

### Current Research

**Tutorial goals.** One of the shortcomings of the existing system is that it doesn't have long-term "goals" for the tutorial dialogue. Implicit in the tutorial rules are some idea about local management of the interaction, but global strategies about the tutoring session are absent. Human tutors, however, admit to having goals like "Concentrate on one particular part of the causal structure of rainfall at a time," or "Clear up one misconception before discussing another." Stevens & Collins (1977) set about codifying these goals and strategies for incorporation into the WHY system. They analyzed tutoring protocols in which human tutors commented on what they thought the students did and didn't know, and on why they responded to the students as they did. From this analysis, two top-level goals became apparent:

1. Refine the student's causal structure, starting with the most important factors in a particular process and gradually incorporating more subtle factors.
2. Refine the student's procedures for applying his causal model to novel situations.

**Student misconceptions.** The top-level goals involve subgoals of identifying and correcting the student's misconceptions. Stevens & Collins (1977) classified these subgoals into five categories corresponding to types of bugs and how to correct them:

**Factual Bugs.** Dealt with by correcting the student. Teaching facts is not the goal of Socratic tutoring; interrelationships of facts are more important.

**Outside-domain bugs.** Misconceptions about causal structure, which the tutor chooses not to explain in detail. For example, the "correct" relationship between the temperature of air and its moisture-holding capacity is often stated by the tutor as a fact, without further explanation.

**Overgeneralization.** When a student makes a general rule from an insufficient set of factors (e.g., any place with mountains has heavy rainfall), the tutor will find counterexamples to probe for more factors.

**Overdifferentiation.** When a student counts factors as necessary when they are not, the tutor will generate counterexamples to show that they are not.

**Reasoning bugs.** Tutors will attempt to teach students skills such as forming and testing hypotheses and collecting enough information before drawing a conclusion.

If a student displays more than one bug, human tutors will employ a set of heuristics to decide which one to correct first:

1. Correct errors before omissions.
2. Correct causally prior factors before later ones.
3. Make short corrections before longer ones.
4. Correct low-level bugs (in the causal network) before correcting higher level ones.

**Functional relationships.** The bugs just discussed are all domain independent, that is, they would occur in tutorial dialogues about other complex processes besides rainfall. But some bugs are the results of specific misconceptions about the functional interrelationships of the concepts of the specific domain. For example, one common misconception about rainfall is that "cooling causes air to rise" (Stevens, Collins, & Goldin, 1978). This is not a simple factual misconception, nor is it domain independent. It is best characterized as an error in the student's functional model of rainfall.

In fact, the script representation used in the WHY system for capturing the temporal and causal relations of land, air, and water masses in rainfall proved inadequate to get at all of the types of student misconceptions. Recent work has investigated a more flexible representation of *functional relationships*, which allows the description of the processes that collectively determine rainfall from multiple viewpoints--e.g., *temporal-causal-subprocess* view

captured in the scripts, *functional* viewpoint which emphasizes the roles that different objects play in the various processes (Stevens, Collins, & Goldin, 1978). Misconceptions about rainfall are represented as errors in the student's model of these relationships. A functional relationship has four components: (a) a set of actors, each with a role in the process; (b) a set of factors that affect the process--the factors are all attributes of the actors (e.g., water is an actor in the Evaporation relationship and its temperature is a factor); (c) the result of the process--this is always a change in an attribute of one of the actors; and (d) the relationship that holds between the actors and the result, or how an attribute gets changed. These functional relationships may be the result of models from other domains that are applied metaphorically to the domain under discussion (Stevens & Collins, 1978).

### Summary

The WHY system started as an extension of SCHOLAR by the implementation of rules that characterize Socratic tutoring heuristics. Subsequently, an effort was made to describe the global strategies used by human tutors to guide the dialogue. Since these were directed towards dispelling students' misconceptions, five classes of misconceptions were established, as well as means for correcting them. Many misconceptions are not domain independent and the key to more versatile tutoring lies in continuing research on knowledge representation.

### References

The most recent reference on the research reported here is Stevens, Collins, & Goldin (1978). The tutorial rules are discussed fully in an excellent article by Collins (1976). The later work on the goal structure of a tutor is reported in Stevens & Collins, 1977. Finally, recent work on conceptual models and multiple viewpoints of complex systems is discussed in Stevens & Collins (1978).

## C3. SOPHIE

SOPHIE (a SOPHisticated Instructional Environment) is an ICAI system developed by John Seely Brown, Richard Burton, and their colleagues at Bolt, Beranek and Newman, Inc., to explore the objective of a wider range of student initiatives during the tutorial interaction (Brown, Burton, & Bell, 1975). The SOPHIE system provides the student with a learning environment in which he learns problem-solving skills by trying out his ideas, rather than by instruction. The system has a model of the problem-solving knowledge in its domain as well as numerous heuristic strategies for answering the student's questions, criticizing his hypotheses, and suggesting alternative theories for his current hypotheses. SOPHIE enables the student to have a one-to-one relationship with an "expert" who helps him create his own ideas, experiment with these ideas and, when necessary, debug them.

Figure 1 illustrates the component modules of the original SOPHIE-I system (Brown, Rubinstein, & Burton, 1976) and the additional capabilities added for the SOPHIE-II system, discussed later in this article.

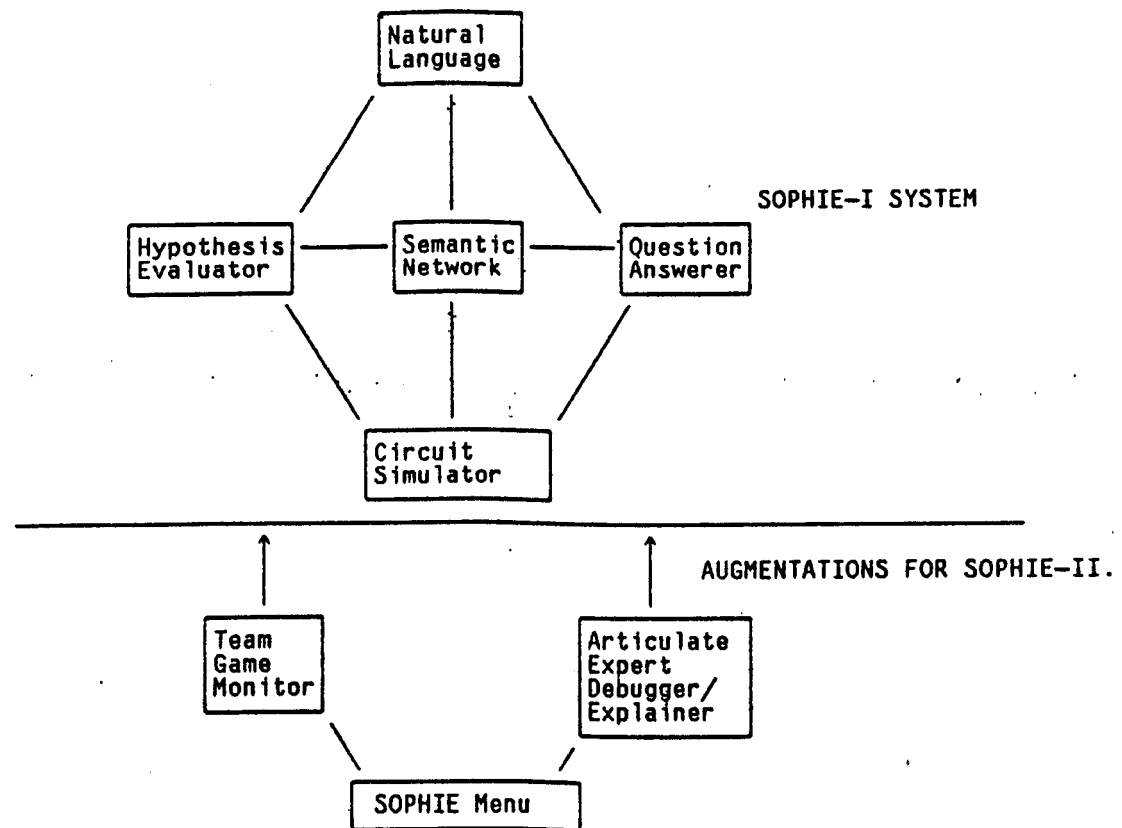


Figure 1. SOPHIE-I and SOPHIE-II.

## SOPHIE-I

SOPHIE teaches problem-solving skills in the context of a simulated, electronics laboratory. The problem facing the student is to find the "faults" in a malfunctioning piece of equipment whose characteristics have been given to him by taking "measurements." The student obtains a sequence of measurements (such as voltages, currents, resistances, etc.) to determine what is wrong with the equipment and asks questions about the measurements and about what fault hypotheses he should consider, given the information he has discovered so far. SOPHIE evaluates the hypotheses by considering all of the information that the student should have been able to derive from his current set of measurements. The system also judges the merits of a student's suggestion for a new measurement in light of the prior sequence of measurements. SOPHIE can decide if a measurement is valuable, depending on whether it eliminates or isolates a fault, and will inform the student when a proposed measurement provides no new information.

The following portion of a sample protocol illustrates some of the linguistic and inferential capabilities of the SOPHIE system (from Brown & Burton, 1975). SOPHIE randomly selects a fault (in the performance of a circuit component), inserts it into a model or simulation of the circuit, and tells the student how the controls are set. The student is given a schematic diagram of the circuit, and the dialogue refers to components in the schematic diagram (e.g., resistor 8 (R8), node connection 1 (N1), capacitor 5 (C5)). The student begins his troubleshooting by performing measurements in the faulted circuit.

Welcome to Sophie, a reactive learning environment. The piece of equipment you are studying is a regulated power supply. Would you like to start with a faulted instrument that needs fixing?

\*\* YES

The instrument has a fault and its present control settings are:

CC	1.0	CURRENT CONTROL
CR	HIGH	CURRENT RANGE SWITCH
LOAD	20	LOAD RESISTANCE IN OHMS
VC	1.0	VOLTAGE CONTROL
VR	HIGH	VOLTAGE RANGE SWITCH

\*\* WHAT IS THE OUTPUT?

The output voltage is 11.7 volts.

*Notice that "output" was disambiguated as "output voltage."*

\*\* WHAT IS IT IN A WORKING INSTRUMENT?

In a working circuit the output voltage is 19.9 volts.

*In case the student does not know what the required value of measurement should have been, he can ask for any measurement in a properly functioning instrument. The student then requests an internal measurement.*

\*\* WHAT IS THE VOLTAGE BETWEEN R8 AND Q1?

The voltage at N1 is 47.6 volts.

**\*\* REPLACE Q6**

*The student believes that he has discovered the problem (in transistor Q6) and tries to fix it. Before a component is replaced, however, the system queries him as to how he thinks it is faulted.*

I am going to ask you some questions about how Q6 is faulted.  
Are any junctions shorted?

**\*\* I DON'T KNOW**

Are any terminals open?

**\*\* YES**

Which terminal(s) are open?

**\*\* THE BASE**

Q6 is not faulted in that manner. Q6 not replaced.

*After learning that Q6 does not have an open base, the student asks if this was at least a reasonable hypothesis.*

**\*\* IS IT POSSIBLE THAT Q6 HAS AN OPEN BASE?**

*Before the system's evaluation procedures can be applied to the student's idea or hypothesis, the hypothesis must be completely specified. To help the student out, the system suggests a plausible complete specification of it in an abbreviated form familiar to the student.*

Is the complete fault mode EB/EC/BC = Open/Ok/Open what you intended?

**\*\* SURE**

That fault is consistent with all of the measurements you have made.

*The hypothesis is reasonable, but other measurements will indicate that another component is faulty.*

### Natural Language Processing--Semantic Grammar

In an instructional environment, it is important that the student be provided with a convenient way in which to communicate his ideas to the system. The student will become quickly frustrated if he has to try several ways of expressing an idea before the system can understand it. SOPHIE's natural language understander copes with various linguistic problems such as *anaphoric references* and context-dependent deletions and *ellipsis*, which occur frequently in natural dialogues.

SOPHIE's natural language capabilities are based on the concept of a *semantic grammar* in which the usual syntactic categories such as noun, verb, and adjective are replaced by semantically meaningful categories (Burton, 1976b, and Burton and Brown, 1979). These categories represent concepts known to the system--such as "measurements," "circuit elements," "transistors" and "hypotheses." For each concept there is a grammar rule that gives the alternate ways of expressing that concept in terms of its constituent concepts. Each rule is encoded as a LISP procedure that specifies the order of application of the various alternatives in each rule.

A grammar centered around semantic categories allows the parser to deal with a certain amount of "fuzziness" or uncertainty in its understanding of the words in a given statement; that is, if the parser is searching for a particular instantiation of a semantic category, and the current word in the sentence fails to satisfy this instantiation, it skips over that word and continues searching. Thus, if the student uses certain words or concepts that the system doesn't know, the parser can ignore these words and try to make sense of what remains. In order to limit the negative consequences that may result from a misunderstood question, SOPHIE responds to the student's question with a full sentence that tells him what question is being answered.

### Inferencing Strategies

In order to interact with the student, SOPHIE performs several different logical and tutorial tasks. First, there is the task of answering *hypothetical questions*. For example, the student might ask, "If the base-emitter junction of the voltage limiting transistor opens, then what happens to the output voltage?"

A second task SOPHIE must perform is that of *hypothesis evaluation*, where the student asks, "Given the measurements I have made so far, could the base of transistor Q3 be open?" The problem here is not to determine if the assertion "the base of Q3 is open" is true, but whether this assertion is logically consistent with the data that have already been collected by the student. If it is not consistent, the program explains why it is not. When it is consistent, SOPHIE identifies which information supports the assertion and which information is independent of it.

A third task that SOPHIE must perform is *hypothesis generation*. In its simplest form this involves constructing all possible hypotheses that are consistent with the known information. This procedure enables SOPHIE to answer questions like, "What could be wrong with the circuit (given the measurements that I have taken)?" The task is solved using the *generate-and-test* paradigm with the hypothesis evaluation task described above performing the "test" function.

Finally, SOPHIE can determine whether a given measurement is *redundant*, that is, if the results of the measurement could have been predicted from a complete theory of the circuit, given the previous measurements.

SOPHIE accomplishes all of these reasoning tasks using an *inference mechanism* that relies principally on a general-purpose *simulator* of the circuit under discussion. For example, to answer a question about a changed voltage resulting from a hypothetical modification to a circuit, SOPHIE first interprets the question with its parser and then, using this interpretation, simulates the desired modification. The result is a Voltage Table that represents the voltages at each terminal in the modified circuit. The original question is then answered in terms of these voltages.

The tasks of hypothesis evaluation and hypothesis generation are handled in a similar manner, using the simulator. When evaluating hypotheses, SOPHIE attempts to determine the logical consistency of a given hypothesis. To accomplish this task, a simulation of the hypothesis is performed on the circuit model and measurements are taken of the result. If the values of any of these measurements are not equivalent to the measurements taken by the student, then a counterexample has been established and it is used to critique the student's hypothesis.

When generating hypotheses, SOPHIE attempts to determine the set of possible faults or hypotheses that are consistent with the observed behavior of the faulted instrument. This task is performed by a set of specialist procedures that propose a possible set of hypotheses to explain a measurement and then simulate them to make sure that they explain the output voltage and all of the measurements that the student has taken. Hypothesis generation can be used to suggest possible paths to explore when the student has run out of ideas for what could be wrong with the circuit or when he wishes to understand the full implications of his last measurement. It is also used by SOPHIE to determine when a measurement is redundant.

#### SOPHIE-II: The Augmented SOPHIE Lab

Extensions to SOPHIE include: (a) a *troubleshooting game* involving two teams of students and (b) the development of an *articulate expert debugger/explainer*. The simple reactive learning environment has also been augmented by the development of frame-oriented CAI lesson material, used to prepare the student for the laboratory interaction (Brown, Rubinstein, & Burton, 1976). The articulate expert not only locates student-inserted faults in a given instrument, but can articulate exactly the deductions that lead to its discovery, as well as the more global strategies that guide the trouble-shooting scenario.

Experience with SOPHIE indicates that its major weakness is an inability to follow up on student errors. Since SOPHIE is to be reactive to the student, it will not take the initiative to explore a student's understanding or suggest approaches that he does not consider. However, the competitive environment of the troubleshooting game, in which partners share a problem and work it out together, was found to be an effective means of exercising the student's knowledge of the operation of the instrument being debugged. Finally, an experiment involving a minicourse--and exposure to the frame-based texts, the expert, and the original SOPHIE Lab--indicated that long-term use of the system is more effective than a single, concentrated exposure to the material (Brown, Rubinstein, & Burton, 1976).



### Summary

The goal of the SOPHIE project was to create a learning environment in which the student would be challenged to explore ideas on his own and to create conjectures or hypotheses about a problem-solving situation. The student receives detailed feedback as to the logical validity of his proposed solutions. In cases where the student's ideas have logical flaws, SOPHIE can create relevant counterexamples and critiques. The SOPHIE system combines domain-specific knowledge and powerful domain-independent inferencing mechanisms to answer questions that even human tutors might find it extremely difficult to answer.

### References

Brown, Burton, & Bell (1975) give a complete description of the early work on SOPHIE, and Brown, Rubinstein, & Burton (1976) report on the later work. Also see Brown & Burton (1975).

#### C4. WEST

Development of the first *computer coach* was undertaken by Richard Burton and John Seely Brown at Bolt, Beranek and Newman, Inc., for the children's board game of "How the West Was Won." The term "coach" describes a computer-based learning environment where the student is involved in an activity, like playing a computer game, and the instructional program operates by "looking over his shoulder" during the game and occasionally offering criticisms or suggestions for improvement (Goldstein, 1977). This research focused on identifying: (a) *diagnostic strategies* required to *infer* a student's misunderstandings from his observed behavior and (b) various explicit *tutoring strategies* for directing the tutor to say the right thing at the right time (Burton & Brown, 1976a, and Burton & Brown, 1979). The intention of this work was to use these strategies to control the interaction so that the instructional program took every possible opportunity to offer help to the student without interrupting so often as to become a nuisance and destroy the students fun at the game. By guiding a student's learning through discovery, computer-based coaching systems hold the promise of enhancing the educational value of the increasingly popular computer gaming environments.

#### Philosophy of the Instructional Coach

The pedagogical ideas underlying much of computer coaching research in WEST can be characterized as *guided discovery learning*. It assumes that the student *constructs* his understanding of a situation or a task based on his prior knowledge. According to this theory, the notion of misconception or *bug* plays a central role in the construction process. Ideally, a bug in the student's knowledge will cause an erroneous result in his behavior, which the student will notice. If the student has enough information to determine what caused the error and can then correct it, the bug is referred to as *constructive*. The role of a tutor in an informal environment is to give the student extra information in situations that would otherwise be confusing to him, so that he can determine what caused his error and can transform nonconstructive bugs into constructive ones (see Fischer, Brown, & Burton, 1978 for further discussion).

However, an important constraint on the coach is that it should not interrupt the student too often. If the coach immediately points out the student's errors, there is a danger that the student will never develop the necessary skills for examining his own behavior and looking for the causes of his mistakes himself. The tutor must be perceptive enough to make relevant comments, but not be too intrusive, destroying the fun of the game. The research on the WEST system examined a wide variety of tutorial strategies that must be included to create a successful coaching system.

#### How the West Was Won

"How the West Was Won" was originally a computer board game designed by Bonnie Anderson of the Elementary Mathematics Project at the PLATO computer-based education system at the University of Illinois (Dugdale & Kibbey, 1977). The purpose of this original (nontutorial) program was to give elementary-school students drill and practice in arithmetic. The game resembles the popular Chutes and Ladders board game, and briefly goes something

like this: At each turn a player receives three numbers (from spinners) with which he constructs an arithmetic expression using the operations of addition, subtraction, multiplication, and division. The numeric value of the completed expression is the number of spaces the player can move, the object of the game being to get to the end first.

However, just trying to combine the three numbers to make the biggest valued expression is not always the best strategy, because there are several special features on the game board. Towns occur every ten spaces and if a player lands on one, he skips ahead to the next town. There are also shortcuts, and if he lands on the beginning of one a player advances to the other end of the shortcut. Finally, if the player lands on the space that his opponent is occupying, the opponent is bumped back two towns. The spinner values in WEST are small, so these special moves are encouraged (i.e., landing on towns or shortcuts or on your opponent).

### Diagnostic Modeling

There are two major related problems that must be solved by the computer coach. They are (1) when to interrupt the student's problem-solving activity, and (2) what to say once it has been interrupted. In general, solutions to these problems require both techniques for determining what the student knows (procedures for constructing a *diagnostic model*) and explicit tutoring principles about interrupting and advising. These, in turn, require theories about how a student forms abstractions, how he learns, and when he is apt to be most receptive to advice. Unfortunately, few, if any, existing psychological theories are precise enough to suggest anything more than caution.

Since the student is primarily engaged in a gaming or problem-solving activity, diagnosis of his strengths and weaknesses must be unobtrusive to his main activity. This objective means that the diagnostic component cannot use pre-stored tests or pose a lot of diagnostic questions to the student. Instead, the computer coach must restrict itself, mainly to inferring a student's shortcomings from what he does in the context of playing the game or solving the problem. This objective can create a difficult problem--just because a student does not use a certain skill while playing a game does not mean that he does not *know* that skill. Although this point seems quite obvious, it poses a serious diagnostic problem: The absence of a potential skill carries diagnostic value if and only if an expert in an equivalent situation would have used that skill. Hence, apart from his outright errors, the main window a computer-based coach has on a student's misconceptions is through a *differential modeling* technique that compares what the student is doing with what the expert would be doing in his place. This difference provides hypotheses about what the student does not know or has not yet mastered. (See the related discussion of *overlay models* in Article C5.)

Constructing the differential model requires that two tasks be performed by the coach, using the computer *Expert* (the subprogram that is expert at playing the game WEST). The first task of the coach is to *evaluate* the student's current move with respect to the set of possible alternative moves that an Expert might have made in the exact same circumstances. The second task is to determine what *underlying skills* were used to select and compose the student's move and each of the "better" moves of the Expert. To accomplish the evaluative task, the Expert need only use the results of its knowledge and

reasoning strategies, available as better moves. However, for the second task, the coach has to consider the "pieces" of knowledge involved in move selection and in the generation of better moves, since the absence of one of these pieces of knowledge might explain why the student failed to make a better move.

#### Tutoring by Issue and Example -- A General Paradigm

One of the top-level goals driving the coach is the objective that its comments be both relevant to the situation and memorable to the student. The *Issues and Examples* tutoring strategy provides a framework for meeting these two constraints. Issues are concepts used in the diagnostic process to identify, at any particular moment, what is relevant. Examples provide concrete instances of these abstract concepts. Providing both the description of a generic issue (a concept used to select a strategy) as well as a concrete example of its use increases the chance that the student will integrate this piece of tutorial commentary into his knowledge. In the Issues and Examples paradigm, the issues embody the important concepts underlying a student's behavior. They define the space of concepts that the Coach can address--the facets of the student's behavior that are monitored by the Coach.

In WEST, there are three levels of Issues on which a Coach can focus: At the lowest level are the basic mathematical skills that the student is practicing (the use of parentheses, the use of the various arithmetic operations, and the form or pattern of the student's move as an arithmetic expression). The second level of Issues concerns the skills needed to play WEST (like the special moves: bump, town, and shortcut) and the development of a *strategy* for choosing moves. And at the third level are the general skills of game playing (like watching your opponent to learn from his moves), which are not addressed by the WEST program.

Each of the Issues is represented in two parts, a *recognizer* and an *evaluator*. The issue recognizer is data-directed; it watches the student's behavior for evidence that he does or does not use a particular concept or skill. The recognizers are used to construct a *model* of the student's knowledge. The issue evaluators are goal-directed; they interpret this model to determine the student's weaknesses. The issue recognizers of WEST are fairly straightforward but are, nevertheless, more complex than simple pattern matchers. For example, the recognizer for the PARENTHESIS Issue must determine not only whether or not parentheses are present in the student's expression, but also whether they were necessary for his move, or for an optimal move.

Figure 1 is a diagram of the modeling/tutorial process underlying the Issues and Examples paradigm. Figure 1a presents the process of constructing a model of the student's behavior. It is important to observe that without the Expert it is impossible to determine whether the student is weak in some skill or whether the skill has not been used because the need for it has arisen infrequently in the student's experience.

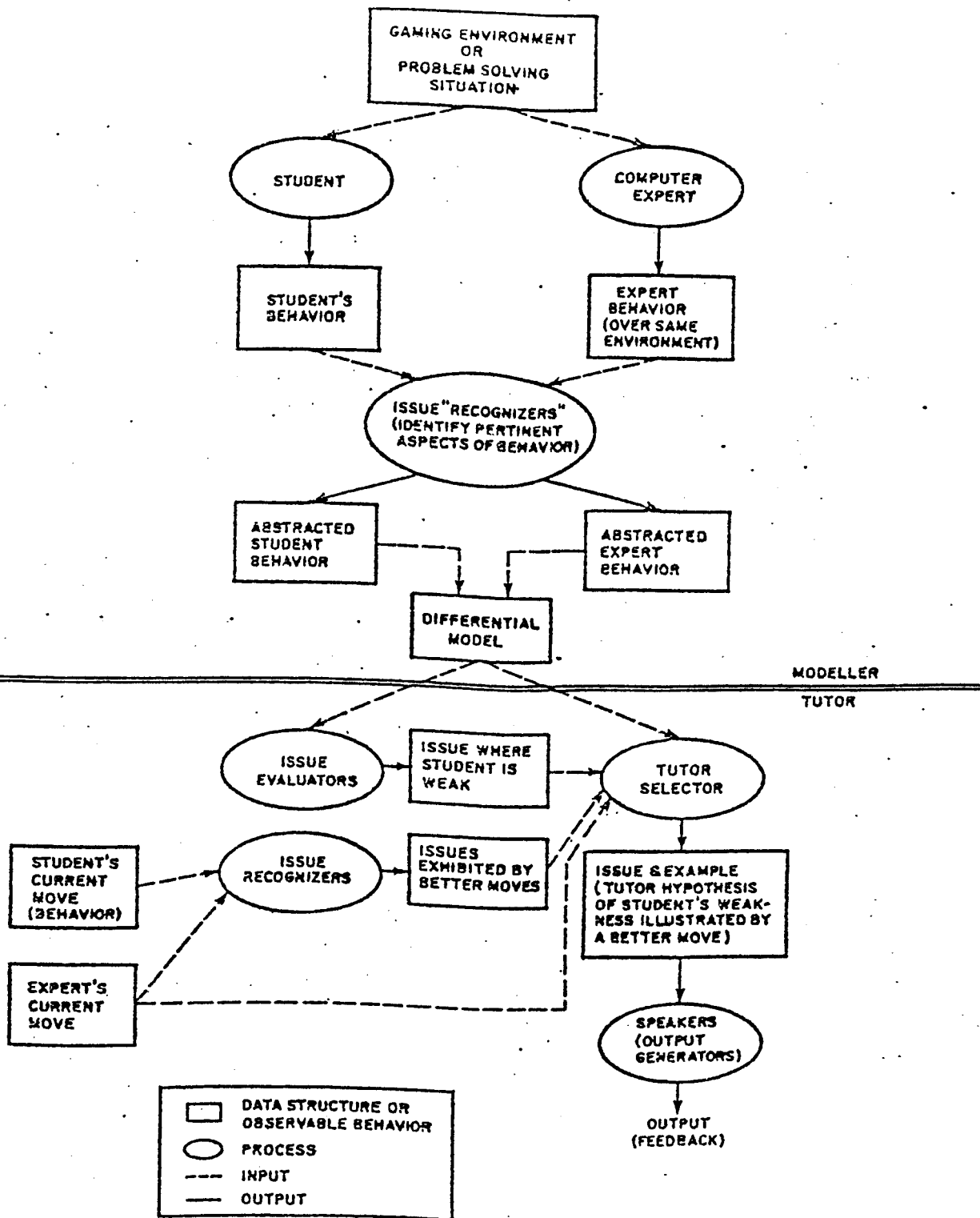


Figure 1. Diagram of the Modeling/Coaching Process

### The Coaching Process

Figure 1b presents the top level of the coaching process. When the student makes a less than optimal move (as determined by comparing his move with that of the Expert), the Coach uses the evaluation component of each Issue to create a list of Issues on which it has assessed that the student is weak. From the Expert's list of better moves, the Coach invokes the Issue recognizers, to determine which issues are illustrated by these better moves. From these two lists of Issues, the Coach selects an Issue and the move that illustrates it (i.e., creates an example of it) and decides, on the basis of *tutoring principles*, whether or not to interrupt. If the two lists have no Issues in common, the reason for the student's problem lies outside the collection of Issues, and the Coach says nothing.

If the Coach decides to interrupt, the selected Issue and Example are then passed to the *explanation* generators, which produce the feedback to the student. Currently, the explanations are stored in a procedures, called *Speakers*, attached to each Issue. Each Speaker is responsible for presenting a few lines of text explaining its Issue. (See also the related discussion of computer coaching in Article C5 on WUMPUS).

### Tutoring Principles

General tutoring principles dictate that, at times, even when relevant Issues and Examples have been identified, it may be inappropriate to interrupt. For example, what if there are two competing Issues, both applicable to a certain situation? Which one should be picked? The Issues in WEST are sufficiently independent that there is little need to consider their prerequisite structure, e.g., whether the use of parentheses should be tutored before division (but see the description of the *syllabus* in WUMPUS, Article C5). Instead, additional tutoring principles must be invoked to decide which one of the set of applicable Issues should be used.

In WEST, experiments have been conducted using two alternate principles to guide this decision. The first is the Focus Strategy, which ensures that, everything else being equal, the Issue most recently discussed is chosen--the Coach will tend to concentrate on a particular Issue until evidence is present to indicate that it is mastered. The alternative principle is the Breadth Strategy, where Issues that have not recently been discussed tend to be selected. This strategy minimizes a student's boredom and insures breadth of concept coverage.

The rest of WEST's strategies for deciding whether to raise an issue and what to say can be placed in the four categories listed below, with example rules of each:

1. **Coaching Philosophy.** Tutoring principles can enhance a student's likelihood to remember what is said. For example, "When illustrating an issue, use an Example (an alternative move) only when the result or outcome of that move is dramatically superior to the move made by the student."
2. **Maintaining Interest in the Game.** The Coach should not destroy the student's inherent interest in the game by interrupting too often. For example, "Never tutor on two consecutive moves," or "If the student makes an exceptional move, identify why it is good and congratulate him."

3. **Increasing Chances of Learning.** Four levels of hints are provided by the WEST tutor, at the student's request: (a) isolate a weakness and directly address that weakness, (b) delineate the space of possible moves at this point in the game, (c) select the optimal move and tell why it is optimal, and (d) describe how to make the optimal move.
4. **Environmental Considerations.** The Coach should consider the game-playing environment. For example, "If the student makes a possibly careless error, one for which there is evidence he knows better, be forgiving."

#### Noise in the Model

When the student does not make an optimal move, the program knows only that at least one of the issues required for that move was not employed by the student. Which of these issues blocked the student from making the move is not known. In practice, blame is apportioned more or less equally among all of the issues required for a missed better move. One effect of this apportionment is the introduction of *noise* into the model, that is, blame will almost certainly be apportioned to issues that are, in fact, understood. Also, since the system does not account for the entire process that a person uses to derive a move, the set of issues is, by definition, incomplete. This is the second source of noise in the differential model. A third source of noise in the model is the difficulty of modeling certain human factors such as boredom or fatigue that cause inconsistent behaviors. For example, students are seldom completely consistent. They often forget to use techniques that they know, or get tired and accept a move that is easy to generate but which does not reflect their knowledge.

Another source of noise is inherent in the process of learning. As the student plays the game, he acquires new skills. The student model, which has been accumulating during the course of his play, will not be up to date, that is, it will still show the newly learned issues as "weaknesses." Ideally, the "old pieces" of the model should decay with time. Unfortunately, the costs involved in this computation are prohibitive. To avoid this particular failing of the model, the WEST Coach removes from consideration any issues that the student has used recently (in the last three moves), assuming that they are now part of his knowledge.

To combat the noise that arises in the model, the Evaluator for each issue tends to assume that the student has mastery of the issue. Some coaching opportunities may be missed but eventually, if the student has a problem addressed by an issue, a pattern will emerge.

#### Experiences with West

WEST has been used in elementary school classrooms. In a controlled experiment, the coached version of WEST was compared to an uncoached version. The coached students showed a considerably greater variety of patterns, indicating that they had acquired many of the more subtle patterns and had not fallen permanently into "ruts" that prevented them from seeing when such moves were important. Moreover, and perhaps most important of all, the students in the coached group enjoyed playing the game considerably more than the uncoached group (Goldstein, 1979).

**References**

The most recent and most complete discussion of the WEST coach is Burton & Brown (1979).



## C5. WUMPUS

This article describes a *computer coach* for WUMPUS, a computer game in which the player must track down and slay the vicious Wumpus while avoiding pitfalls that result in certain, if fictional, death (Yob, 1975). The coach described here is WUSOR-II, one of three "generations" of computer coaches for WUMPUS developed by Ira Goldstein and Brian Carr at MIT (Carr & Goldstein, 1977). (For discussions of WUSOR-I and -III, see Stansfield, Carr, & Goldstein, 1976, and Goldstein, 1979, respectively.) To be a skilled Wumpus-hunter one must know about logic, probability, decision theory, and geometry. A deficit in one's knowledge may result in being eaten by the Wumpus or falling through the center of the earth. In keeping with the philosophy of computer coaching, students are highly motivated to learn these fundamental skills.

The design of the WUSOR-II system involves the interactions of the specialist programs shown in Figure 1. There are four modules: the Expert, the Psychologist, the Student Model, and the Tutor. The Expert informs the Psychologist of two facts: (a) if the player's move is nonoptimal and (b) which skills are needed for him to discover better alternatives. The Psychologist employs this comparison to formulate hypotheses about which domain-specific skills are known to the student. These hypotheses are recorded in the Student Model, which represents the student's knowledge as a subset of the Expert's skills--an *overlay model* (see Article B and Carr & Goldstein, 1977). The Tutor uses the student model to guide its interactions with the player. Basically, it chooses to discuss skills not yet exhibited by the player in situations where their use would result in better moves. Goldstein (1977) provides a more detailed discussion of the structure and function of these coaching modules. (Also see the discussion of the WEST computer coach in Article C4.)

The central box of Figure 1 contains a representation for the problem-solving skills of the domain being tutored. It is, in essence, a formal representation of the syllabus. The Expert is derived from the skills represented therein, as is the structure of the student model. The Psychologist derives expectations from this knowledge regarding which skills the student can be expected to acquire next, based on a model of the relative difficulty of items in the syllabus. The Tutor derives relationships *between* skills such as analogies and refinements, which can be employed to improve its explanations of new skills (see Goldstein, 1979).

### Theoretical Goals: Toward a Theory of Coaching

The approach to the design of computer coaches in WUSOR-II is to construct *rule-based representation* (see Article Representation.B3) for (a) the skills needed by the Expert to play the game, (b) the modeling criteria used by the Psychologist, and (c) the alternative tutoring strategies used by the Tutor. Each is expanded below:

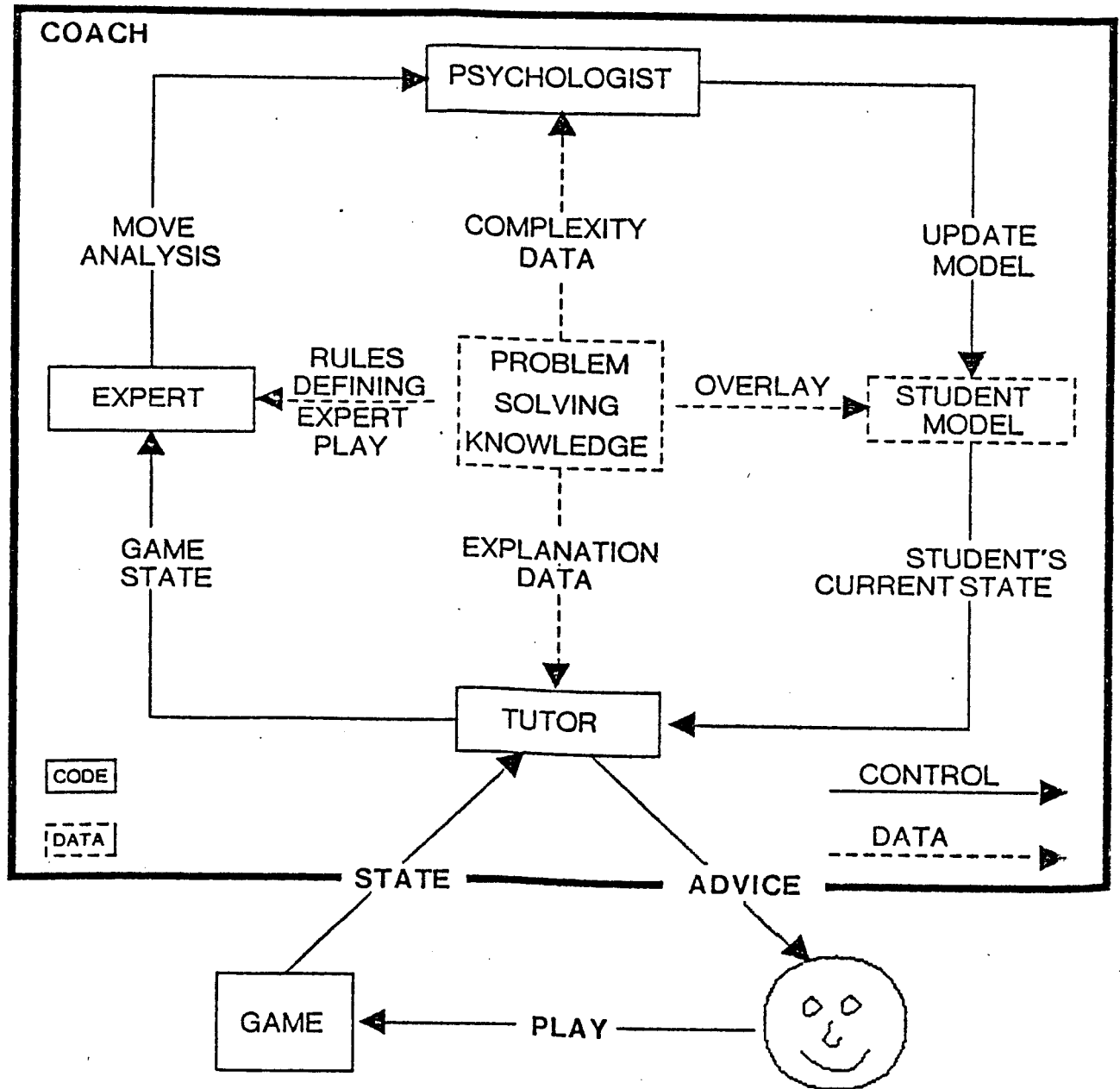


Fig. 1. Simplified block diagram of a computer coach.

The Expert uses rules that embody the knowledge or skills required to play the game to analyze the player's behavior. The virtue of a rule-based representation of expertise is that its modularity both allows tutoring to focus concisely on the discussion of specific skills and permits modeling to take the form of hypotheses regarding which rules are known by the player.

The Psychologist uses *rules of evidence* to make reasonable hypotheses about which of the Expert's skills the player possesses. Typical rules of evidence are:

Increase the estimate that a player possesses a skill if the player explicitly claims acquaintance with the skill, and decrease the reliability if the player expresses unfamiliarity.

Increase the estimate that a player possesses a skill if the skill is manifest in the player's behavior, and decrease the estimate if the skill is not manifest in a situation where the Expert believes it to be appropriate; hence, implicit as well as overt evidence plays a role.

Decrease the estimate that a player possesses a skill if there is a long interval since the last confirmation was obtained (thereby modeling the tendency for a skill to decay with little use).

The Tutor uses *explanation rules* to select the appropriate topic to discuss with the player and to choose the form of the explanation. These rules include:

*Rules of simplification* that take a complex statement and reduce it to a simpler assertion. Simplification rules are essential if the player is not to be overwhelmed by the Tutor's explanations.

*Rules of rhetoric* that codify alternative explanation strategies. The two extremes are explanation in terms of a general rule and explanation in terms of a concrete instance.

### The WUMPUS Expert

In WUMPUS, the player is initially placed somewhere in a randomly connected warren of caves and told the neighbors of his current location. His goal is to locate the horrid Wumpus and slay it with an arrow. Each move to a neighboring cave yields information regarding that cave's neighbors. The difficulty in choosing a move arises from the existence of dangers in the warren--bats, pits, and the Wumpus itself. If the player moves into the Wumpus's lair, he is eaten. If he walks into a pit, he falls to his death. Bats pick the player up and randomly drop him elsewhere in the warren.

The player can minimize risk and locate the Wumpus by making the proper logistic and probabilistic inferences from warnings that he is given. These warnings are provided whenever the player is in the vicinity of a danger. The Wumpus can be smelled within one or two caves. The squeak of bats can be heard one cave away and the breeze of a pit felt one cave away. The game is won by shooting an arrow into the Wumpus's lair. If the player exhausts his set of five arrows without hitting the creature, the game is lost.

The Wumpus Expert uses a rule-based representation, consisting of approximately 20 rules, to infer the risk of visiting new caves. Five of these rules are shown below:

- L1 *positive evidence rule.* A warning in a cave implies that a danger exists in a neighbor.
- L2 *negative evidence rule.* The absence of a warning implies that no danger exists in any neighbors.
- L3 *elimination rule.* If a cave has a warning and all but one of its neighbors are known to be safe, then the danger is in the remaining neighbor.
- P1 *equal likelihood rule.* In the absence of other knowledge, all of the neighbors of a cave with a warning are equally likely to contain a danger.
- P2 *double evidence rule.* Multiple warnings increase the likelihood that a given cave contains a danger.

#### A Sample Protocol with the WUSOR-II Computer Coach

A transcript of an interaction with the WUSOR-II coach is illustrated below. The player's responses are preceded by a \*\*.

Hello, Timmy. There are 15 caves, 2 bats, 2 pits, and 1 Wumpus. You are now at cave 15 with neighbors 4, 14 and 0. Brrr! There is a draft. You are near a pit. What a stench! The Wumpus is near. What now?

\*\* 4

You are now at cave 4 with neighbors 15, 14 and 2. Brrr! Squeak! A bat is near. What now?

*The goal of the Coach is to tutor a beginner in the relevant logical, probabilistic, and strategic knowledge needed to play the game. For example, the Expert informs the Tutor that cave 14 should be treated as more dangerous than 0 or 2 since there is multiple evidence (from the drafts in 15 and 4) that 14 contains a pit. If the player now moved to cave 14, a coaching situation might occur as follows:*

\*\* 14

Timmy, it isn't necessary to take such large risks with pits. One of cave 2 and 14 contains a pit. Likewise one of cave 0 and 14 contains a pit. This is multiple evidence of a pit in cave 14 which makes it quite likely that cave 14 contains a pit. It is less likely that cave 0 contains a pit. Hence, we might want to explore cave 0 instead. Do you want to take back your move?

Although it is not apparent from these simple remarks, every module of the Coach contributed to the dialogue. These contributions are summarized below:

The Expert analyzes all possible moves, using its set of skills. The outcome of its analysis is a ranking of possible moves with an attached list that associates the skills that would be needed to make each move. For example, using the five skills listed earlier, the Expert recognizes that cave 14 is the most dangerous move and cave 0 is the safest move.

Essentially, the Expert provides the following proof for use by the Psychologist and Tutor modules. (The proof is given here in English for readability: the Expert's actual analyses are in the programming language LISP.)

Lemma 1: The Wumpus cannot be in 0, 2, or 14 since there is no smell in 4. (Application of the negative evidence rule L2 for 2-cave warning of Wumpus.)

Lemma 2: Caves 0 and 2 were better than 14 because there was single evidence that caves 0 and 2 contained a pit, but double evidence for cave 14. (Application of the double evidence rule P2.)

Lemma 3: Cave 2 is more dangerous than cave 0, since 2 contains a bat, and the bat could drop you in a fatal cave. (We know this fact because the squeak in 4 implied a bat in 14 or 2; but the absence of a squeak in 15 implies no bat in 14. Hence, by elimination rule L3, there is a bat in 2.)

The Psychologist, after seeing Timmy move to cave 14, decreases the Student Model weight indicating familiarity with the Double Evidence rule P2, since the Expert's proof indicates that this heuristic was not applied. Table 1 is the Psychologist's hypotheses regarding which skills of the Expert the student possesses.

Table 1.

A Typical Student Model Maintained by the Coach

RULES	APPROPRIATE	USED	PER CENT	KNOWN
L1	5	5	100	Yes
L2	4	3	75	Yes
L3	4	2	50	?
L4	5	5	100	Yes
L5	4	1	25	No

Modeling raises many issues. One subtlety is that the move to 14 above may be evidence of a more elementary limitation--a failure to understand the logical implications of the draft warning--i.e., that a pit is in a neighboring cave. The current state of the Student Model is used by the Psychologist to determine, in the event of a nonoptimal move, which skill is in fact missing. The Student Model indicates the level of play that can be expected from this player--the player might be a beginner with incomplete knowledge of the basic rules of

the game, a novice with understanding of the logical skills, an amateur with knowledge of the logical and the more elementary probability skills, etc. The Psychologist would attribute the student's error in the current situation to unfamiliarity with a skill at his current level of play--in this case, Timmy is a player who has mastered the logical skills and is learning the basic probability heuristics. Hence, the coach's explanation focused on explaining the double evidence heuristic.

The Tutor is responsible for abridging the Coach's response to the player's move to cave 14. (The complete explanation generated by the Expert were the three lemmas shown above.) Such pruning is imperative if the Coach is to generate comprehensible advice. Hence, the Tutor prunes the complete analysis on the basis of simplification rules that delete those parts of the argument that are already known to the player on the basis of the Student Model and those portions that are too complex. Here, the coach deleted Lemma 1, the discussion of the Wumpus danger, because it is based on the negative evidence skill that the Student Model attributes to the player. Lemma 2, the elimination argument for bats, is potentially appropriate to discuss; but a simplification strategy directs the Coach to focus on a single skill. Additional information will be given by the Coach if requested by the player.

#### Conclusions

The novelty of this research is that in a *single system* there is significant domain expertise, a broad range of possible interaction strategies available to the tutor, and a modeling capability for the student's current knowledge state. Informal experience with over 20 players of various ages has shown WUSOR-II to be a helpful learning aid, as judged by interviews with the players. The short-term payoff from this research is an improved understanding of the learning and teaching processes. The long-term payoff is the development of a practical educational technology, given the expected decrease in hardware costs.

#### References

Carr & Goldstein (1977) describe WUSOR, the overlay model, and related theory. Also see Goldstein (1977), Goldstein (1979), and Stansfield, Carr, & Goldstein (1976).

## C6. BUGGY

BUGGY is a program that can accurately determine a student's misconceptions (bugs) about basic arithmetic skills. The system, developed by John Seely Brown, Richard Burton and Kathy Larkin at Bolt, Beranek and Newman, Inc., provides a mechanism for explaining why a student is making a mistake, as opposed to simply identifying the mistake. Having a detailed model of a student's knowledge that indicates his misconceptions is important for successful tutoring.

A common assumption among teachers is that students do not follow procedures very well and that *erratic* behavior is the primary cause of a student's inability to perform each step correctly. Brown & Burton (1978a) argue that students are remarkably competent procedure followers, but they often follow the wrong procedures. By presenting examples of systematic incorrect behavior, BUGGY allows teachers to practice diagnosing the underlying causes of a student's errors. Using BUGGY, teachers gain experience at forming hypotheses about the relationship between the symptoms of a bug that a student manifests and the underlying misconception. This experience helps teachers become more aware of methods or strategies available for diagnosing their student's problems properly.

## Manifesting Bugs

Experience with BUGGY indicates that forming a model of what is wrong with a student's method of performing a task is often more difficult than performing the task itself. Consider, for example, the following addition problems and their (erroneous) solutions. They were provided by a student with a "bug" in his addition procedure:

41	328	989	66	216
+ 9	+917	+ 52	+887	+ 13
----	----	----	----	----
50	1345	1141	1053	229

Once you have discovered the bug, try testing your hypothesis by simulating the buggy student--predict his results on the following two test problems:

446	201
+815	+399
----	----

The bug is simple. In procedural terms, after determining the carry, the student forgets to reset the "carry register" to zero; he accumulates the amount carried, across the columns. For example, in the student's second problem ( $328 + 917 = 1345$ ), he proceeds as follows:  $8 + 7 = 15$ , so he writes 5 and carries 1;  $2 + 1 = 3$  plus the 1 carried is 4; finally,  $3 + 9 = 12$ , but the 1 carried from the first column is still there--it has not been reset--so adding it to the final column gives 13. If this is the correct bug, then the answers to the test problems will be 1361 and 700. (This bug is really not so unusual; a child often uses his fingers to remember the carry and might forget to bend them back after each column.)

The model built by BUGGY incorporates both correct and incorrect *subprocedures* that simulate the student's behavior on particular problems and capture what parts of a student's skill are correct and what parts are incorrect. BUGGY represents a skill, such as addition, as

a collection of subskills, for example, one of which is knowing how to "carry" a digit into the next column. The subprocedures in BUGGY that correspond to human subskills are linked into a *procedural net* (Sacerdoti, 1974), which is BUGGY's representation of the entire human skill. If all the subprocedures in BUGGY's procedural net for addition work correctly, then BUGGY will do addition problems correctly. On the other hand, replacing correct subprocedures with ones that are faulty will result in systematic errors of the kind shown above. Brown and Burton call a procedural network with one or more faulty subprocedures a *diagnostic model* because it is a way of representing systematic errors. The model has been used in two ways. First, it can diagnose a student's errors and pinpoint the bug(s) in the student's skill. Second, it can help to train a teacher to diagnose student errors by "playing the part" of a student with one or more buggy subskills.

When BUGGY is to diagnose a student's errors, its task is to modify the correct procedural network of, say, subtraction until it accounts for all of the student's answers, both right and wrong. This modification is done by systematically replacing correct subprocedures with incorrect variants until a consistent diagnostic model is found. There are currently 70 primitive faulty subprocedures for subtraction. These are explored exhaustively while attempting to determine a consistent diagnostic model. If a single variant or bug is insufficient to explain a student's behavior, then combinations of two bugs are tried. (Because of the overwhelming number of combinations of three or more bugs, these are not used to form diagnostic models.) In this manner, 330 "bugs" have been identified, each with a bug description. Interactions among bugs and the ramifications of a buggy subprocedure's being called by several high-order procedures constitute major challenges for designing efficient simulations of multiple bugs. Note also that this technique requires a large amount of compute time and is amenable only to domains where bugs can be explicated in a more or less complete way.

The procedural network for subtraction (and its bugs) was tested in an experiment using the results of a test administered to 1,325 students in the fourth, fifth, and sixth grades. Using the BUGGY model, it was found that about 40% of the students demonstrated consistently buggy behavior. That is, BUGGY was able to construct a diagnostic model for student's errors demonstrating that they were using nonrandom, though incorrect procedures.

#### Using Buggy to Train Teachers

BUGGY can also be used to train teachers to diagnose student errors. This facility is illustrated by a protocol excerpted from Brown, 1977 in which BUGGY presents a team of teachers with incorrectly solved math problems and supervises their diagnosis of the bug.

#### Protocol of a Team Using BUGGY

Welcome to Bugland. I have chosen a bug.  
Here is an example of the bug.

17  
+5  
--



13

*The system presents an example in which the student didn't get the right answer.*

Now you can give me problems to determine the bug.

```

** 18   43
** +6   +79
   --   ---
    15   23

```

*The teachers generate problems, and BUGGY solves them incorrectly, following a buggy procedure. The team is trying to diagnose the bug in BUGGY'S addition procedures by presenting BUGGY with problems to solve. However, they are using problems similar to the original one, which is a poor strategy because it may leave other symptoms of the bug undiscovered.*

#### \*\* GOT THE BUG!

*The team thinks that they know what the problem is. Omitted here for space, but shown below, BUGGY asks them to formulate the bug in English. This step has the effect of making the team verbalize the bug clearly. It is a difficult task, and the results are often ambiguous. BUGGY does not "read" the description of the bug, because it does not have the natural language abilities to do so. Instead, it generates a series of problems for the team to solve. The team must solve them incorrectly, using the bug that they think they have identified, to convince BUGGY that they have in fact discovered the bug.*

Here are some problems to test your theories about the bug.

What is:

```

  21
+39
---
** 51

```

That's not the bug I have. Using my bug:

```

  21
+39
---
  15

```

Try giving me some more examples...

*The team fails BUGGY's test by being unable to solve the given problem. That is, their bug is not the same one that BUGGY is using. If they had solved 5 problems "correctly" (that is, incorrectly because of a bug), then BUGGY would credit them with a correct diagnosis. Now, however they must set more problems for BUGGY to solve, in an attempt to deduce the bug.*

```

**   51   99   68
** +1707 +99   +9

```

```

-----  ---  --
  21   36   23

```

**\*\* GOT THE BUG!**

Please describe the bug you think I have:

**\*\* THE STUDENT IS ADDING ALL THE DIGITS TOGETHER.**

Here are some problems to test your theory about the bug.  
What are:

```

  33  1091   8   28   90
+99  +60  +34  +70  +6
---  ---  ---  ---  --
** 24   17   15   17   15

```

Very good. My description of the bug is:

The student always sums up all the digits, with no regard to columns.

**Summary**

The central idea of this research is that *procedural networks* can be used to build diagnostic models. This representation scheme facilitates an appropriate decomposition of a skill into subskills, to make explicit the control structures underlying a collection of skills, and to execute directly the resulting *diagnostic model*. Rather than being a subset or simplification of expert rules, as in *overlay modeling* (Carr & Goldstein, 1977; see Articles B, C5 and C4), the procedural network is a model of a student's behavior that is structured in terms of semantically meaningful deviations from the correct procedure. Each subskill, whether correct or incorrect, is explicitly represented as a subprocedure in the network.

**References**

Brown & Burton (1978a) is the most recent and complete report on BUGGY. Also see Brown, Burton, Hausmann, Goldstein, Huggins and Miller (1977) and Brown, Burton, and Larkin (1977).

## C7. EXCHECK

EXCHECK is an Intelligent Computer-aided Instruction system designed and implemented by Patrick Suppes and his colleagues at the Institute of Mathematical Studies in the Social Sciences (IMSSS) at Stanford University. It is a general-purpose instructional system used principally to present complete, university-level courses in logic, set theory, and proof theory. In the courses taught using the EXCHECK system, lesson material is presented to the student at his computer terminal, followed by exercises consisting of theorems that he is to prove using the program's theorem prover. The courses are taught on IMSSS's CAI system, which uses computer-generated speech and split-screen displays. Several hundred Stanford students take these courses each year.

From an AI point of view, the most interesting aspects of the EXCHECK system are the procedures and the underlying theories of mathematical reasoning that permit this interaction to take place in a natural style closely approximating standard mathematical practice. These include natural language facilities, natural-deduction based proof procedures, theorem provers, decision procedures for some simple mathematical theories, procedures for analyzing and summarizing proofs, and procedures for conducting dialogues about some elementary mathematical structures.

Examples of the kind of natural language accepted and generated are given in the proofs and dialogues presented below. The basic logic is a variant of Suppes's (1957) formulation of natural deduction augmented by high-level inference procedures that are the analogs of proof procedures used in standard mathematical practice.

### Understanding Informal Mathematical Reasoning

The mathematical reasoning involved in the set theory and proof theory courses is complex and subtle. The fundamental AI problem of EXCHECK is making the program capable of understanding informal mathematical reasoning: The program must be able to follow mathematical proofs presented in a "natural" manner. That is, just as the intent of natural language processing is to handle languages that are actually spoken, the intent of natural proof processing is to handle proofs as they are actually done by practicing mathematicians. In general, such proofs are presented by giving a sketch of the main line of argument along with any other mathematically significant information that might be needed to completely reconstruct the proof. This style should be contrasted with the derivations familiar from elementary logic, where each detail is presented and the focus of attention is on syntactic manipulations rather than on the underlying semantics.

A major aspect of the problem of machine understanding of natural proofs is finding languages that permit users to express their proofs in the fashion described above. Such languages, in turn, must find their basis in an analysis or model of informal mathematical reasoning. Finding these natural proof languages should be compared to the problem of finding high-level "natural" or "English-like" programming languages. For more detailed discussions of these issues, see Blaine & Smith (1977), Smith (1976), and Smith et al. (1975). A simple example of understanding informal mathematical reasoning and fuller discussion of the techniques involved follows.

## Student Proof

We present two proofs of the elementary theorem,

Thm: If  $A \subset B$  then  $\neg(B \subseteq A)$

where " $\subset$ " is used for proper subset and " $\subseteq$ " is used for subset.

First, the proof is given in the informal style of standard mathematical practice.

We want to show that if  $A \subset B$ , then  $\neg(B \subseteq A)$ .

Assume  $A \subset B$ . We show  $\neg(B \subseteq A)$  by indirect proof. Assume that  $B \subseteq A$ . Since  $A \subset B$  then, by definition,  $A \subseteq B$  and  $A \neq B$ . Since  $A \subseteq B$  and  $B \subseteq A$  then  $A = B$ . But this is a contradiction and, hence, the assumption that  $B \subseteq A$  is false. Therefore,  $\neg(B \subseteq A)$ .

The following typescript shows how one student did the proof in the EXCHECK system. Input from the student is in boldface type and comments are in *italics*; everything else is generated by the program. The program keeps track of the goal that the student is currently trying to establish; the initial goal is the theorem to be proven. EXCHECK indicates acceptance of an inference by returning the top-level prompt **\*\***; if a suggested inference is not acceptable, EXCHECK returns an error message.

Derive: If  $A \subset B$  then  $\neg(B \subseteq A)$

**\*\* hyp (1) \*\***  $A \subset B$

*The hypothesis of the theorem is assumed. The goal is automatically reset to the consequent of the theorem.*

**\*\* raa**

**assume (2) \*\***  $B \subseteq A$

*The student begins an indirect proof (the command 'raa' is a mnemonic for reductio ad absurdum). The program assumes the negation of current goal. The goal is now any contradiction.*

**\*\* 1definition Number or Name? \*\*** proper subset

**1 Df. proper subset**

**(3)  $A \subseteq B$  and  $A \neq B$**

*The definition of proper subset is applied to line 1.*

**\*\* 2,3establish \*B = A**

**2,3 Establish**

**(4)  $B = A$**

*The student asks the theorem prover to check the simple set theoretic inference.*

**\*\* 3,4contradiction**

Using **\*\* ok**

3,4.Contradiction

(5)  $B \not\subseteq A$

The student indicates that lines 3 and 4 lead to a contradiction. EXCHECK returns the negation of assumption (2).

**\*\* qed**

Correct

*EXCHECK accepts the derivation.*

The following informal *review* printout was generated by the program from the proof given in the above typescript.

Derive: If  $A \subset B$  then  $\neg(B \subseteq A)$

Assume (1)  $A \subset B$

By *raa* show:  $\neg(B \subseteq A)$

Assume (2)  $B \subseteq A$

From 1, by definition of proper subset,

(3)  $A \subseteq B$  and  $A \not\subseteq B$

From 2,3 it follows that,

(4)  $A = B$

3,4 lead to a contradiction; hence, assumption 2 is false:

(5)  $\neg(B \subseteq A)$

### Natural Inference Procedures

There are no significant structural differences between the detailed informal proof and the student's proof as presented to EXCHECK. The same steps occur in the same relations to each other. Such global or structural fidelity to natural proofs is a major research goal of the EXCHECK project and depends upon the development of *natural inference procedures*. Some of these, such as the HYPOTHESIS and INDIRECT PROOF procedures used in the above proof, are familiar from standard logical systems. The procedure used in the application of the definition of proper subset to line (1) is called IMPLIES. It is used to derive results that, intuitively speaking, follow by applying a previous result or definition. It is considerably more complex than the inference procedures usually found in standard logical systems. An even more complex natural inference procedure used in the above proof is the ESTABLISH procedure. In general, ESTABLISH is used to derive results that are consequences of prior results in the theory under consideration, in this case in the theory of sets. Eliminating the need to cite specific results in the theory, which would disrupt the main line or argument, is important and is discussed further in the section on ESTABLISH, below.

The inference procedures in EXCHECK are intended not only to match natural inferences in strength but also to match them in degree and kind. However, there are differences. EXCHECK inference procedures must always be invoked explicitly--in standard practice, particular inference procedures or rules are usually not cited explicitly. For example, compare how the student expresses the inferences that result in lines (3) and (4) with their counterparts in the informal proof. The explicit invocation of inference procedures basically requires that two pieces of information be given: first, the inference procedure to be used; and, second, the previous results to be used--in particular, explicit line numbers must be used.

Explicitness is not disruptive of mathematical reasoning--neither is the reduction of complex inferences to smaller inferences nor the use of explicit line numbers disruptive, in the sense of distracting the student from the main line of the mathematical argument. They are both simple elaborations of the main structure. However, having to think about what inference rule to use can interrupt the main line of argument. The success of a system for interactively doing mathematics depends crucially upon having a few powerful and natural inference procedures with clear criteria of use, which are sufficient to handle all the inferences.

#### IMPLIES

IMPLIES is used to derive results by applying a previous result or definition as a rule of inference in a given context. This form of inference is probably the most frequent naturally occurring inference. While the basic pattern is simple, the refinements that must be added to the basic form to get a procedure that handles most of the naturally occurring cases result in a computationally complex procedure. The following is a simple example of the basic pattern:

(i) A is a subset of B

! definition (Name or number) \*subset

(i)  $(\forall x)(x \in A \rightarrow x \in B)$

In this example, the student directed the program to apply the definition of subset to line (i) and IMPLIES generated the result:  $(\forall x)(x \in A \rightarrow x \in B)$ . While the student thinks he is applying the definition of subset to line (i), the procedure actually invoked is the IMPLIES procedure. It is important to note that in a use of the IMPLIES procedure the student indicates what axiom, definition, theorem, or line to apply to which lines, and the IMPLIES procedure generates the formula that is the result of the inference.

The IMPLIES procedure seems to correspond closely to naive notions of inference, in that logically unsophisticated but mathematically sophisticated users can use it very well after seeing the basic explanation and a few simple examples. However, the IMPLIES rule does have a fault: It is a purely logical inference procedure and that can occasionally cause problems for users, because mathematicians tend to think in terms of set theoretic rather than logical consequence. (See the discussion of the ESTABLISH rule for more on this distinction.)

## ESTABLISH

The following example of a simple use of ESTABLISH is taken from the typescript above.

- (2)  $B \subseteq A$
- (3)  $A \subseteq B$  and  $A \supseteq B$

- \*2,3 establish \*B = A
- 2,3 Establish
- (4)  $B = A$

The ESTABLISH rule allows users to simply assert that some formula is an elementary set-theoretic truth or is an elementary set-theoretic consequence of prior results. In the above example, ESTABLISH is used to infer from  $A \subseteq B$  and  $B \subseteq A$  that  $A = B$ .  $A = B$  is a set theoretic consequence but not a logical consequence of  $A \subseteq B$  and  $B \subseteq A$ . If ESTABLISH handled only logical consequence, the student would have had to explicitly cite the relevant set theoretic theorems or definitions needed to reduce the inference to a purely logical inference. This is not only disruptive of the line of argument but also difficult to do. Even the most experienced logicians and mathematicians have difficulty ferreting out all the axioms, definitions, and theorems needed to reduce even simple inferences to purely logical inferences.

All of the examples so far are extremely simple if considered in terms of the full capabilities of the ESTABLISH procedure. ESTABLISH uses a theorem prover that can prove about 85% of the first 200 theorems in the set theory course.

## Proof Analysis and Summarization

EXCHECK contains procedures that generate informal summaries and sketches of proofs. Such analyses and summaries are useful not only as a semantic basis for the program, to better understand proofs and to better present proofs, but also to give guidance to the student (see the proof summary below for an example of the kind of guidance that can be generated). The summarization procedures analyze the proof by breaking it into parts (or "subproofs") and isolating the mathematically important steps. They also permit a goal-oriented interpretation of the proof where the program keeps track of what is to be established at that point (i.e., the current goal); which lines, terms, etc., are relevant; and how the current line or part fits into the whole structure. MYCIN's consultation explanation system (see article C1) uses a similar approach. Goldstein (1977) also uses summarization techniques in the rhetorical modules of the WUMPUS coach (article C5).

The summaries presented below were generated by EXCHECK from a student proof of the Hausdorff maximal principle. The original line numbers have been retained in to give a sense of how much of the proof has been omitted in the summary. In the first summary only the top-level part of the proof is presented; the proofs of its subparts are omitted. Also, all mathematically or logically insignificant information is omitted. In these proofs and summaries " $D$  contains  $E$ " is synonymous with " $E \subseteq D$ ". Also,  $C$  is a chain iff both  $C$  is a set of sets, and given any two elements of  $C$ , at least one is a subset of the other.

Derive: If  $A$  is a family of sets then  
every chain contained in  $A$  is contained in some maximal chain in  $A$

Proof:

Assume (1)  $A$  is a family of sets

Assume (2)  $C$  is a chain and  $C \subseteq A$

Abbreviate:  $\{B: B \text{ is a chain and } C \subseteq B \text{ and } B \subseteq A\}$

by:  $C$ chains

By Zorn's lemma,

(23)  $C$ chains has a maximal element

Let  $B$  be such that

(24)  $B$  is a maximal-element of  $C$ chains

Hence,

(25)  $B$  is a chain and  $C \subseteq B$  and  $B \subseteq A$

It follows that,

(31)  $B$  is a maximal chain in  $A$

Therefore,

(32)  $C$  is contained in some maximal chain in  $A$

Figure 1. Informal summary of a proof of the Hausdorff maximal principle.

The summary above is not the only one that could be generated; it essentially presents only the main part of the proof. Subparts of the main part could have been included or even handled independently if so desired.

The proof analysis and summarization procedures will also generate the following kind of summary, which is an attempt to sketch the basic idea of the proof.

Derive: If  $A$  is a family of sets then  
every chain contained in  $A$  is contained in some maximal chain in  $A$

Proof:

Use Zorn's lemma to show that

$\{B: B \text{ is a chain and } C \subseteq B \text{ and } B \subseteq A\}$

contains a maximal element  $B$ . Then show that  $B$  is a maximal chain in  $A$  which contains  $C$ .

Figure 2. An example summarization.

The summarization in Figure 2 was obtained from that in Figure 1 by tracing backwards the history of the maximal chain in  $A$  that contains  $C$ . That is, the general form of the theorem to be proven is  $(\exists x)FM(x)$ , which is proven by showing  $FM(t)$  for some term  $t$ . Usually, in proofs of this form, the most important piece of information is the term  $t$ . Tracing backwards in this particular proof yields that there are two terms involved. The first is the set of all chains in  $A$  containing  $C$ , and the second is any maximal element of the set of all chains in  $A$  containing  $C$ .



### Elementary Exercises and Dialogs

Another form of reasoning done by students is the solution of problems. A great many problems in elementary mathematics take the form of asking the student to give finite objects satisfying certain conditions. For example, given the finite sets A and B the student might be asked to give a function F that is a bijection (i.e., 1-1 and onto) from A to B. For a large class of such problems there are programs that will generate a tree of formulas and other information from the original statement of the problem. We call such trees *verification trees* for the problem. Essentially, the verification tree for a problem constitutes a reduction of the original (usually not directly verifiable) condition to a collection of directly verifiable conditions (the formulas at the leaves). These trees have the property that the failure of the formula at a node in the tree explains the failure of formulas at any of its ancestors. Similarly, the failure of a formula at a node is explained by the failure of formulas at any of its descendants.

For example, in the above problem of supplying a bijection F from A onto B, suppose that the student forgets to specify a value for some element of A, say, 3. The first response to the student might be: "The domain of F isn't A." The student might then ask: "Why?" The program would then answer (going towards the leaves), "Because there is an element of A that has not been assigned a value in B." The student might then ask, "Which one?" Since the routines that evaluate the formulas at the leaves provide counterexamples if those formulas fail, the program could then respond, "3." Or going back to the first response by the program ("The domain of F isn't A") the student might say, "So?" The program could then move a step towards the root (the original statement of the conditions) and say, "Then F is not a map from A into B." The student might then again say: "So?", to which the program could respond: "F is not a bijection from A onto B."

The highly structured information in the verification tree provides the semantic base for a dialogue with the student in which the program can explain to the student what is wrong with the answer. It should be noted that more complex forms of explanation are available. In particular, the program could have said at the beginning that, "Because 3 is not given a value by F, the domain of F is not A and hence F is not a bijection from A onto B."

### Summary

A primary activity in mathematics is finding and presenting proofs. In the EXCHECK system an attempt is made to handle natural proofs--proofs as they are actually done by practicing mathematicians--instead of requiring that these proofs be expressed as derivations in an elementary system of first order logic. This objective requires the analysis of inferences actually made and the design and implementation of languages and procedures that permit such inferences to be easily stated and mechanically verified. Some progress has been made in handling natural proofs in elementary mathematics, but there is a considerable amount of work yet to be done.

### References

See Blaine & Smith (1977), Smith et al. (1975), Smith & Blaine (1976), Suppes (1957), and Suppes (1960).

## References

- Atkinson, R. C. Ingredients for a theory of instruction. *American Psychologist*, 1972, 27, 921-931.
- Atkinson, R. C., & Wilson, H. A. (Eds.) *Computer-assisted instruction*. New York: Academic Press, 1969.
- Barr, A., & Atkinson, R. C. Adaptive instructional strategies. Paper presented at the IPN Symposium 7: Formalized Theories of Thinking and Learning and their Implications for Science Instruction, Kiel, September 1975.
- Barr, A., Beard, M., & Atkinson, R. C. A rationale and description of a CAI program to teach the BASIC programming language. *Instructional Science*, 1975, 4, 1-31.
- Barr, A., Beard, M., & Atkinson, R. C. The computer as a tutorial laboratory: The Stanford BIP project. *International Journal of Man-Machine Studies*, 1976, 8, 567-596.
- Blaine, L. H., & Smith, R. L. Intelligent CAI: The role of curriculum in suggesting computational models of reasoning. *Proceedings: 1977 Annual Conference, ACM, Seattle, 1977*.
- Brown, J. S. Uses of artificial intelligence and advanced computer technology in education. In R. J. Seidel & M. Rubin (Eds.), *Computers and Communications: Implications for Education*. New York: Academic Press, 1977.
- Brown, J. S., Burton, R. R., and Larkin, K. M. Representing and Using Procedural Bugs for Educational Purposes. *Proceedings 1977 Annual Conference of Association for Computing Machinery, Seattle, Oct. 1977, 247-255*.
- Brown, J. S., Burton, R. R., Hausmann, C., Goldstein, I., Huggins, B., and Miller, M. Aspects of a theory for automated student modelling. (BBN Report No. 3549). Cambridge, Mass.: Bolt, Beranek and Newman, 1977
- Brown, J. S., Burton, R. R., Miller, M., deKleer, J., Purcell, S., Hausmann, C., and Bobrow, R. Steps toward a theoretical foundation for complex, knowledge-based CAI. (BBN Report No. 3135). Cambridge, Mass.: Bolt, Beranek and Newman, 1975.
- Brown, J. S., Collins, A., and Harris, G. Artificial Intelligence and Learning Strategies. In H. O'Neil (Ed.), *Learning Strategies*. New York: Academic Press, 1978.
- Brown, J. S., Rubinstein, R., Burton, R. Reactive Learning Environment for Computer Assisted Electronics Instruction (BBN Report No. 3314). Cambridge: Bolt, Beranek, & Newman, 1976.
- Brown, J. S., & Burton, R. Multiple Representations of Knowledge for Tutorial Reasoning. In D. G. Bobrow & A. Collins (Eds.), *Representation and Understanding: Studies in Cognitive Science*. New York: Academic Press, 1976. Pp. 311-349.

- Brown, J. S., & Burton, R. R. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 1978, 2(2), 155-192. (a)
- Brown, J. S., Burton, R. R., & Bell, A. G. Sophie: A Sophisticated Instructional Environment for Teaching Electronic Troubleshooting (An Example of AI in CAI). *International Journal of Man-Machine Studies* (1975), 7.
- Brown, J. S., & Goldstein, I. P. *Computers in a Learning Society*, Testimony for the House Science & Technology Subcommittee on Domestic and International Planning, Analysis, & Cooperation, October 1977.
- Burton, R. R. Semantic grammar: An engineering technique for constructing natural language understanding systems, BBN Report 3453, December 1976 (b).
- Burton, R. R., and Brown, J. S. A Tutoring and Student Modelling Paradigm for Gaming Environments. *Proc. for the Symposium on Computer Science and Education*, Irvine, CA, February 1976. (Also, *SIGCSE Bulletin*, 1976, 8, 236-246.) (a).
- Burton, R. R., and Brown, J. S. An investigation of computer coaching for informal learning activities. *International Journal of Man-Machine Studies* (1979) 11, 5-24.
- Burton, R. R., and Brown, J. S. Toward a natural-language capability for computer-assisted instruction. In H. O'Neil (Ed.), *Procedures for Instructional Systems Development*. New York: Academic Press, 1979, 273-313.
- Carbonell, J. R. AI in CAI: An artificial intelligence approach to computer-aided instruction. *IEEE Transactions on Man-machine Systems*, 1970, MMS-11(4), 190-202. (a)
- Carbonell, J. R. *Mixed-initiative Man-computer Instructional Dialogues* (BBN Rep. No. 1971). Cambridge, Mass.: Bolt, Beranek, & Newman, 1970. (b)
- Carbonell, J. R., & Collins, A. Natural Semantics in Artificial Intelligence. *IJCAI* 3, 1973, 344-351.
- Carr, B. & Goldstein, I. *Overlays: A theory of modeling for computer aided instruction*, AI Memo 406, Massachusetts Institute of Technology, Cambridge, Mass., 1977.
- Clancey, W. Tutoring rules for guiding a case method dialogue. *International Journal of Man-Machine Studies*, 1979, 11, 25-49.
- Clancey, W. Dialogue management for rule-based tutorials. *IJCAI* 6, 1979, in press.
- Collins, A. Processes in Acquiring Knowledge. In R. C. Anderson, R. J. Spiro, & W. E. Montague (Eds.) *Schooling and the Acquisition of Knowledge*. Hillsdale, N.J.: Erlbaum Assoc., 1976. Pp. 339-363.
- Collins, A. Fragments of a Theory of Human Plausible Reasoning. *Proceedings of TINLAP-2*, 1978, 194-201.

- Collins, A., Warnock, E. H., Aiello, N., & Miller, M. L. Reasoning from Incomplete Knowledge. In D. G. Bobrow & A. Collins, *Representation and Understanding*. New York: Academic Press, 1975. Pp. 383-415.
- Collins, A., Warnock, E. H., & Passafiume, J. J. *Analysis and synthesis of tutorial dialogues* (BBN Report 2789). Cambridge, Mass.: Bolt, Beranek, & Newman, 1974.
- Crowder, N. A. Intrinsic and extrinsic programming. In J. E. Coulson (Ed.), *Proceedings of the conference on application of digital computers to automated instruction*, New York: Wiley, 1962, 58-55.
- Dugdale, S. & Kibbey, D. *Elementary mathematics with PLATO*. Urbana, IL: University of Illinois (Computer-based Education Research Laboratory), July 1977.
- Fischer, G., Brown, J. S., & Burton, R. R. Aspects of a theory of simplification, debugging, and coaching. *Proceedings of the 2nd Annual Conf. of Canadian Society for Computational Studies of Intelligence*, July 1978.
- Fletcher, J. D. Modeling the learner in computer-assisted instruction. *Journal of Computer-Based Instruction*, 1975, 1, 118-126.
- Goldberg, A. *Computer-assisted instruction: The application of theorem-proving to adaptive response analysis* (Tech. Rep. 203). Stanford, CA: Stanford University, Institute for Mathematical Studies in the Social Sciences, 1973.
- Goldstein, I. *The Computer as Coach: An athletic paradigm for intellectual education*, AI Memo 389, Massachusetts Institute of Technology, Cambridge, Mass., 1977.
- Goldstein, I. The genetic epistemology of rule systems. *International Journal of Man-Machine Studies*, 1979, 11, 51-77.
- Goldstein, I., & Papert, S. Artificial Intelligence, language, and the study of knowledge. *Cognitive Science*, 1977, 1(1), 84-123.
- Grignetti, M. C., Hausmann, C., & Gould, L. An *intelligent* on-line assistant and tutor--NLS-SCHOLAR. *Proceedings of the National Computer Conference*, San Diego, Calif., 1975, pp. 775-781.
- Groen, G. J. The theoretical ideas of Piaget and educational practice. In P. Suppes (Ed.), *Impact of research on education: Some case studies*. Washington, D.C.: National Academy of Education, 1978. Pp. 267-317.
- Hart, R. O., & Koffman, E. B. A Student Oriented Natural Language Environment for Learning LISP. *IJCAI 4*, 1976, 391-396.
- Howe, J. A. M. Individualizing computer-assisted instruction. In A. Ellithorn & D. Jones (Eds.), *Artificial and human thinking*. Amsterdam: Elsevier, 1973. Pp. 94-101.

- Kimball, R. B. **Self-optimizing computer-assisted tutoring: Theory and practice** (Tech. Rep. 206). Stanford, Calif.: Stanford University, Institute for Mathematical Studies in the Social Sciences, 1973.
- Koffman, E. B., & Blount, S. E. **Artificial Intelligence and automatic programming in CAI. Artificial Intelligence**, 1975, 6, 215-234.
- Laubsch, J. H. **Some Thoughts about Representing Knowledge in Instructional Systems. IJCAI 4**, 1975, 122-125.
- Miller, M. L., & Goldstein, I. **Problem solving grammars as formal tools for intelligent CAI. Proc. of the Fall Conference of the Assoc. for Computing Machinery**, Seattle, October 1977.
- Miller, M. L. **A structured planning and debugging environment for elementary programming. International Journal of Man-Machine Studies**, 1979. In press.
- Norman, D. A., Gentner, D. R., and Stevens, A. L. **Comments on learning schemata and memory representation. In D. Klahr (Ed.), Cognition and Instruction. Hillsdale: Erlbaum Associates**, 1976.
- Papert, S. **Teaching children programming. IFIP Conference on Computer Education. Amsterdam: North Holland**, 1970.
- Reither, R. **On Reasoning by Default. TINLAP-2**, 1978, 210-218.
- Ruth, G. **Analysis of algorithm implementations (MAC TR-130). Cambridge, Mass.: Massachusetts Institute of Technology**, 1974.
- Sacerdoti, E. D. **Planning in a hierarchy of abstraction spaces. Artificial Intelligence**, 1974, 5, 115-135.
- Schank, R. C., & Abelson, R. P. **Scripts, Plans, Goals, and Understanding. Hillsdale, N.J.: Lawrence Erlbaum**, 1977.
- Self, J. A. **Student models in computer-aided instruction. International Journal of Man-Machine Studies**, 1974, 6, 261-276.
- Smith, R. L. **Artificial Intelligence in CAI. Unpublished working paper, IMSSS, Stanford University**, 1976.
- Smith, R. L., & Blaine, L. H. **A generalized system for university mathematics instruction. SIGCUE Bulletin**, 1976, 8(1), 280-288.
- Smith, R. L., Graves W. H., Blaine, L. H., & Marinov, V. G. **Computer-assisted axiomatic mathematics: Informal rigor. In O. Lacarme & R. Lewis (Eds.), Computers in education, IFIP (Part 2). Amsterdam: North-Holland**, 1975. Pp. 803-809.

- Stansfield, J. L., Carr, B. P., & Goldstein, I. P. WUMPUS Advisor I: A First Implementation of a program that tutors logical and probabilistic reasoning skills, MIT AI Memo 381, October 1976.
- Stevens, A. L. & Collins, A. Multiple Conceptual Models of a Complex System (BBN Rep. No. 3923). Cambridge, Mass.: Bolt Beranek & Newman, 1978. To appear in R. Snow, P. Federico and W. Mantague (eds.), *Aptitude, learning and instruction: Cognitive Process Analysis*.
- Stevens, A. L., & Collins, A. The Goal Structure of a Socratic Tutor (BBN Rep. No. 3518). Cambridge, Mass.: Bolt, Beranek, & Newman, 1977.
- Stevens, A. L., Collins, A., & Goldin, S. Diagnosing Student's Misconceptions in Causal Models (BBN Rep. No. 3786). Cambridge, Mass.: Bolt, Beranek, & Newman, 1978.
- Suppes, P. *Introduction to logic*. New York: Van Nostrand Reinhold, 1957.
- Suppes, P. *Axiomatic set theory*. New York: Van Nostrand, 1960. (Slightly rev. ed. publ. by Dover, New York, 1972)
- Suppes, P., & Morningstar, M. *Computer-assisted instruction at Stanford, 1966-68: Data, models, and evaluation of the arithmetic programs*. New York: Academic Press, 1972.
- Wescourt, K. T., and Hemphill, L. Representing and teaching knowledge for troubleshooting/debugging. IMSSS Tech. Report No. 292, Stanford University, 1978.
- Wexler, J. D. Information networks in generative computer-assisted instruction. *IEEE Trans. Man-Machine Systems*, 1970, 11, 181-190.
- Yob, G. Hunt the Wumpus. *Creative Computing*, Sept.-Oct. 1975, pp. 51-54.

## Index

anaphoric references 639  
 Anderson, Bonnie 642  
 arithmetic skills 655  
 articulate expert 620, 640  
 Atkinson, Richard C. 618

BIP 620, 621  
 Brown, John Seely 619, 620, 636, 642,  
 655  
 BUGGY 622, 655-659  
 Burton, Richard 619, 636, 642, 655

CAI 1  
 Carbonell, Jaime 618, 619, 626  
 Carr, Brian 649  
 case grammar 627, 628  
 case method tutor 624, 631  
 Clancey, William 624  
 closed sets 630  
 Collins, Allan 619, 623, 626, 631  
 computer coach 621, 624, 642, 646-649  
 computer coach. 624  
 computer gaming 624, 640, 642, 649  
 conceptual bugs 655  
 constructive bugs 642  
 courseware author 618, 630

diagnosis of student errors 618, 628, 642  
 diagnostic model 643, 656, 658  
 diagnostic modelling 623  
 diagnostic models 655  
 diagnostic strategies 642  
 dialogue management 624, 628, 632, 633,  
 646, 651  
 differential modeling 643  
 discourse model 628, 646, 651  
 discussion agenda 628

education applications 1-666  
 ellipsis 639

EXCHECK 619, 659-666  
 EXCHECK informal proof 663  
 expert program 649  
 expertise module 620-621, 643, 651  
 explanation 619, 646, 651, 654, 663

frame-oriented CAI 1, 622, 640  
 functional relationships 634

generate-and-test 639  
 generative CAI 618, 620  
 geography tutor 626  
 Goldstein, Ira 622, 624, 649  
 GUIDON 620, 624

heuristics 631  
 How the West Was Won 642  
 hypothesis evaluation 639, 640  
 hypothesis generation 639, 640

ICAI 1, 618-620-666  
 importance tags 627  
 incomplete knowledge 630  
 individualization of instruction 618  
 inference 619  
 inference procedures 659  
 inference strategies 629, 639  
 informal mathematical reasoning 659  
 informal proofs 659  
 intersection search 629  
 issue evaluators 644  
 issue recognizers 644  
 issues and examples tutoring 644

Koffman, Elliot 618

learning by discovery 642  
 LISP 639

## LOGO 1

meta-level knowledge 630  
 mixed-initiative dialogue 624, 626, 636  
 multiple representations 620  
 MYCIN 663

natural deduction 659  
 natural inference procedures 661  
 natural language interface 659  
 natural language understanding 619, 628,  
 639, 659  
 NLS-SCHOLAR 626

open problems 630  
 open sets 630  
 overlay model 622, 643, 649, 658

Papert, Seymour 1  
 pattern matching 622, 661  
 PLATO Project 642  
 plausible reasoning 626, 629, 630  
 problem-solving expertise 619, 620-621,  
 636, 643, 651  
 procedural knowledge 649  
 procedural net 656  
 procedural networks 658  
 procedural representation of  
 knowledge 620  
 production rules 620, 624, 651  
 proof checking 659  
 proof summary 659  
 protocol analysis 627, 633

reactive learning environment 618, 636,  
 640  
 reasoning from incomplete knowledge 626  
 representation of knowledge 619, 620-  
 621, 626, 632, 635

rule-based representation 662  
 rule-based systems 651

SCHOLAR 619, 620, 624, 626-631  
 scripts 632  
 search 656  
 semantic grammar 639  
 semantic net 620, 622, 626, 628  
 set theory 659  
 simulation 640  
 Socratic method 623, 627, 631  
 SOPHIE 619, 622, 636-642  
 SOPHIE-I 620, 621, 636, 637-639  
 SOPHIE-II 636, 640-641  
 Stevens, Albert 623, 631  
 stochastic learning models 622  
 student model 1, 619, 621-623, 624, 644,  
 647, 649, 653  
 Suppes, Patrick 619, 659  
 syllabus 646

teaching strategies 624  
 temporal relations 632  
 text generation 628  
 tutorial goals 633  
 tutorial programs 1-620-666  
 tutorial rules 632  
 tutoring principles 646  
 tutoring strategies 619, 623-625, 626,  
 627, 642, 643, 644, 646

Wescourt, Keith 623  
 WEST 621, 624, 642-649  
 WHY 620, 623, 624, 630, 631-636  
 WUMPUS 620, 621, 624, 646, 649-655,  
 663  
 WUSOR 649-655



# Automatic Programming

# Automatic Programming

## Table of Contents

A. An Overview of Automatic Programming . . . . .	675
B. Methods of Specification . . . . .	684
C. Basic Approaches . . . . .	689
D. PSI . . . . .	699
E. SAFE . . . . .	708
F. Programmer's Apprentice . . . . .	714
G. PECOS . . . . .	721
H. DEDALUS . . . . .	726
I. PROTOSYSTEM I . . . . .	734
J. NLPQ: Natural Language Programming for Queuing Simulations . . . . .	739
K. LIBRA . . . . .	744
References . . . . .	748

## A. An Overview of Automatic Programming

Automatic Programming (AP) is a new, dynamic, and not precisely defined area of artificial intelligence. This overview discusses the definitions, history, motivating forces and goals of AP, along with a brief description of the basic characteristics and central issues of AP systems. The article begins with a section discussing the various possible definitions of AP, along with a brief presentation of the background and some of the general motivating forces and goals of AP. The next section describes four characteristics of all AP systems: the method by which a user of such a system specifies or describes the desired program, the target language in which the system writes the program, the problem or application area to which the system is addressed, and the approach or operational method employed by the system. Next, a section discusses four basic issues. All AP systems are primarily involved with one or more of these: the representation and processing of partial or incomplete information; the transformation of structures, and especially the transformation of program descriptions into other descriptions (in this chapter, the term program description includes the user's specification of the desired program, any internal representations of the program, as well as the target language implementation); the efficiency of the target language implementation; and the system's capabilities for understanding the desired program. Following this overview, the reader will find articles on the methods of specifying programs in AP systems, on some of the basic operational methods employed in such systems, and then eight articles describing most of the major AP projects.

### Definition

The bulk of the research in AP has appeared in the 1970s, and it is not surprising that there is lack of agreement as to the definition, scope, and direction of the endeavor. Several brief definitions of automatic programming have been suggested in the literature, but considering the newness of the area, one should not expect these definitions to be precise. One definition says simply that AP is something that will save people from the chores of programming Biermann (1976a). Another states that an AP system carries out part of the programming activity currently performed by a human in constructing, from the definition of a problem to be solved, a program written in some machine executable language; here, the essence of an AP system is that it assumes some responsibilities otherwise borne by a human, and thereby reduces the person's task Hammer & Ruth (1979). Yet another states that AP means having the computer help write its own programs Heidorn (1977). AP is the application of a computing system to the problem of effectively utilizing that or another computing system in the performance of a task specified by the user Balzer (1973b). To summarize, perhaps we can define AP here as an automation of some part of the program writing activities that, currently, are typically performed by people and not yet performed by machine. Therefore the definition excludes such systems and software environments as assembly languages and high level languages such as FORTRAN, COBOL, PL/1, ALGOL, or LISP; and such programming aids as symbol tables, cross reference generators, text editors, and debugging systems.

Other more extensive definitions have been suggested. One such definition Balzer (1973b) would "rate" AP systems according to a measure of merit, which includes the following factors:

- (a) the amount of time and effort needed by the programmer to formulate and specify the desired program;

- (b) the efficiency of the decisions made by the system in designing the program, and consequently the overall efficiency of the program that is produced by the system;
- (c) the ease with which future modifications can be incorporated in the program;
- (d) the reliability and ruggedness of the program;
- (e) the amount of computer resources, including time and memory, used by the system to produce that program; and
- (f) the range, as well as the complexity, of the tasks that can be handled by the system.

Notice that, according to such a measure, a FORTRAN language compiler would rank as an AP system. However, its rank would be significantly less than the potential of current AP research projects.

Another source (see article D3) lists some specific factors that bear on factor (a) above, the factor concerned with the effort required of the programmer. The specific factors are informality, language level, and executability. Informality refers to the degree of informality with which the user may specify the desired program to the AP system. It is informal to the degree that the user can be ambiguous (various interpretations of the specification are possible) and partial or incomplete (pieces of information, including perhaps information about referencing and sequencing, have been omitted). Language level refers to the degree to which the AP system can accept specifications in a terminology natural to the problem area under consideration. Executability refers to the degree to which the system can achieve a desired program state on the basis of a description of that state, that is, the user need only specify what is wanted rather than how to obtain it.

Another definition of AP is obtained by first defining the development phases of a software system (software development refers to the creation of a program or collection of programs, from their inception to the completed product). On this basis, it would follow that AP assists the programmer with one or more of these phases. For example, in a later article that describes the PROSYSTEM research project D7, the development of data-processing systems (programs) is seen as passing through five phases. First, the programming problem is defined by clearly identifying and understanding what the desired software is to accomplish; second, what the program is to do in order to alleviate this problem is clearly and precisely determined; third, the organization, flow of control, and data representations are selected from standard implementation possibilities; fourth, this very high-level specification in terms of standard implementations is transformed into code in some high-level language, and; fifth, this code is compiled.

These, then, are some of the more detailed definitions that have been presented for AP. Altogether, they define a somewhat amorphous direction of research. In this new area there is still no widespread agreement as to exactly what constitutes AP.

The present period is not the first time the term automatic programming has been used. The term was employed once before, about twenty years ago, to mean writing a program in a

high-level language (e.g., FORTRAN), and having a compiler transform the program into machine language code. Thus, one finds "Automatic Coding," Franklin Institute, January 1957 (see *Automatic Coding*, 1957), or *The Annual Review of Automatic Programming*, first appearing in 1959 (see *The Annual Review in Automatic Programming*, 1960). At that time, when "real" programming referred to writing a program in machine or assembly language, AP meant writing a program in FORTRAN. Today, when most programming is done in high-level languages, AP means programming in a software environment much more advanced than the ones created by these high-level languages.

Though the early meaning of the term automatic programming differs from the current meaning, nevertheless, at both times AP meant assisting and automating the process of writing programs.

In a general way, the forces responsible for AP twenty years ago are similar to those responsible for the appearance of AP today. At both times there was a feeling that programmers were burdened with the need to specify many details, with the need to keep track of the many relations between these details, and with a programming environment that was not, perhaps, natural to the way in which they thought about the problem. At both times there was a feeling among some that new programming environments might be within grasp (twenty years ago the new environments were high-level languages) and that the required software technologies required to realize such environments might be feasible. Out of the desire for new programming environments and out of the feeling that these new environments might be attainable, there appeared, in each period, an endeavor called AP.

The current motivations for AP, while similar to those twenty years ago, are more intense. Today software is costly and unreliable. So much time, money, and effort is currently being expended, with even greater expenditures forecast for the future. Too infrequently is software produced within budget or on time. Too often the supposedly finished product, when delivered, fails to meet specifications. As programming applications of increasingly greater complexity are addressed, not only does reliability become more difficult to attain, but the costs of software, in terms of time, money, and effort, spiral upward.

To help alleviate these problems, AP should aim at certain general goals. AP would like to restyle, in more natural terms, the way in which the programmer specifies the desired program. This restyling should allow the programmer to think of the problem at a higher and more natural level. AP would like to relieve the programmer of mundane portions of programming, thereby relieving the programmer of the need to keep track of too many details. By changing the programming environment, AP could allow programmers to construct, with greater ease and with greater accuracy, the programs of the present and the more complex programs of the future.

These last goals circle back to a succinct definition of AP: The computer itself writes its own programs, or parts of its own programs, that is, the computer performs a portion of the program-writing activities. Neither the goals nor this definition are especially precise, but the next sections are more specific. They describe the common characteristics and primary issues of AP systems.

### Characteristics of AP Systems

All AP systems have a specification method, a target language, a problem area, and an approach or method of operation.

Users of an AP system must be given some means or method for conveying to the system the program that they desire. This means is referred to as the *specification method* of the AP system. As will be seen in the remainder of this chapter, AP systems possess a variety of specification methods. Formal specification methods are those that might be considered to be *very high-level* programming languages. In general, the syntax and semantics of such methods are precisely and definitely defined. Formal methods also tend to be *complete*; that is, the specification, within this method, of the desired program will completely and precisely indicate what it is that the program is to accomplish, though, of course, the specification may not indicate the form of the program or how the program is to accomplish it. Formal specification methods are usually not *non-interactive*, which is to say, the system does not interact with the user in order to obtain missing information, to verify hypotheses, or to point out inconsistencies in the specification. For example, it is comparable to the passive acceptance of a program's specification by a compiler of a high-level language (e.g., FORTRAN).

A different method of specification is by examples. Here the user would specify the desired program by simply giving examples of what the desired program is to do; the AP system would then construct the desired program. The specification might consist of examples of the input/output behavior of the desired program, or it might consist of traces of the desired program's behavior (a trace is an example showing how the program should process a given input). Specification by examples (or traces) is certainly not complete: The examples do not fully describe in all cases the behavior of the desired program.

Natural language (e.g., English) is another method of specification. The user specifies in natural language what the desired program is to do. These are often interactive (cf. articles on PSI and NGPS), since they check hypotheses, point out inconsistencies, and ask for further information.

A more detailed discussion of specification, including some advantages and disadvantages of the various methods, is presented in the article on program specification. Examples of program specification are found in most of the remaining articles of this chapter.

If the specification method refers to the input to the AP system, the target language is concerned with the system's output of the finished program. The language in which the AP writes the finished program, or parts of the finished program, is called the *target language*. The target languages of the AP systems described in this chapter are high-level languages such as LISP, PL/1, or GPSS. As an example, suppose that the target language of an AP system were LISP. The user, possibly employing a very high-level language, or examples, or natural language, would specify to the AP system what the desired program is to do. Then the AP system would eventually output a LISP program to do just that.

It is possible to view specification method and target language as relative terms. In an AP system that carries the process of writing programs through several phases, the input method to each phase could be thought of as a specification method, and the output specification as being written in a target language, which then becomes the input

specification method to the next phase. However, in this chapter, target language is usually reserved for the language in which the output program of the whole AP system is written.

Another characteristic of an AP system is its problem area or area of intended application. Problem area, problem domain, application area, and application domain are synonymous terms. For some AP systems, the scope of its problem area is relatively precise. Thus, the problem area of the NLPQ system is that of simple queuing problems. The problem area of the PROTOSYSTEM project (see article below) is that of input/output intensive data-processing systems, including inventory control, payroll, and other record-keeping systems. On the other hand, the problem area of some AP systems can be relatively large; the application domain of the PSI system (see article below) is symbolic computation, including list processing, searching and sorting, data storage and retrieval, and concept formation. The problem area of a system can have a bearing on the method of specification, introducing terminology that is relevant to the problem area, influencing the method of operation or approach used by the AP system, etc.

The fourth characteristic of AP systems is the approach or method of operation. The AP area is too new for there to be very many clear-cut categories of methods of operation. While there are some systems whose approach falls into a basic, though general, category, the approach of most systems is not easily categorized. A separate article on basic approaches discusses some of the more clear-cut categories, including theorem proving, program formation, knowledge engineering, automatic data selection, traditional problem solving, and induction.

In the theorem-proving approach, the user specifies the conditions that must hold for the input data (to the desired program) and the conditions that the output data should satisfy: The conditions are specified in some formal language, often the predicate calculus. A theorem prover is then asked to prove that for all given inputs, there exists an output that satisfies the output conditions. The proof, then, yields a program. Briefly, theorem proving uses theorem provers to produce certain kinds of proofs from which the desired program can be extracted.

The program transformation approach refers to transforming a specification or description of a program into an equivalent description of the program. The reason for the transformation might be to convert a specification that can be easily written and read into one that is more complicated but more efficient; or the reason might be to convert a very high-level description of the program into a description closer to a target language implementation.

Knowledge engineering (see section Applications), applicable to many areas in addition to AP, refers to identifying and explicating knowledge; and it often means "realizing" the knowledge as specific rules that can be added to or removed from the *knowledge base* of a system.

Traditional problem solving (see section Search), also applicable to many areas, refers to the use of goals to direct the choice and application of a set of operators.

These approaches or paradigms overlap, and many systems utilize a method that may, in part, draw on elements from several. While it is hard to categorize the approaches of AP systems, there are now enough systems so that it is possible to identify some common issues, and these are the topic of the next section.

### Basic Issues

In the article on basic approaches and in all the articles describing the individual research projects, the reader will find one or more of several explicit basic issues addressed: partial information, transformation, efficiency, and understanding.

Partial information pertains to systems whose methods of specification allow for partial or fragmentary descriptions of the desired program: Not all of the required information is present in the specification, or, where it is present, it may not be explicit. Since the problem of partial information does not apply to systems that have complete methods of specification, systems such as DEDALUS, PROSYSTEM I, LIBRA, and PECOS are not concerned with this problem. On the other hand, systems that accept incomplete specifications, especially natural language specifications, are very much concerned with partial information. The NLPQ, PSI, and SAFE systems fall in this category. In the article on the SAFE project, one will find a classification of the different kinds of missing information that might occur in a natural language specification.

Usually going hand in hand with the problem of partial information is the problem of consistency. Incomplete methods of specification often permit inconsistency between different parts of the same specification. In such cases, the system must check for inconsistencies and, if they are found, resolve them.

In trying to fill in missing information in one part of the specification or checking for consistency between different parts and resolving any discovered inconsistency, the system may use information that occurs either explicitly or implicitly in other parts of the specification. Also, it might utilize a knowledge base containing information about the problem area. Finally, the system may consult the user in an attempt to gain the sought-for information. One of the explicit devices for utilizing such information is *constraints*. For examples of these, see the article on PSI and especially the article on SAFE.

Another issue addressed by AP systems is transformation. The term refers, simply, to transforming a program description, or part of a program description, into another form. All AP systems use transformation, if only to transform an internal description of the program into a target language implementation (description). Even a compiler of high-level languages (e.g., FORTRAN, PL/I, ALGOL) will often transform a program description several times, taking it through several internal representations, the last of which is the machine language description. However, a compiler differs from an AP system in that it applies the transformations in a rigid, predetermined manner; in an automatic programming system there might be no predetermined manner of applying the transformations, the application depending on an analysis and exploration or the results of applying various transformations. Such systems as DEDALUS and PECOS, systems that use extensive transformation on the program description, have a knowledge base containing many transformation rules that convert parts of a higher level description into a lower level description closer to a target language implementation. Such rules are repeatedly applied to parts of the program description with the goal of eventually producing descriptions within the target language. These systems develop a tree of possible descriptions of the program, with each descendant of a node being the result of a transformation. One of the goals, then, in developing the tree is to find a description that is a target language implementation of the desired program. Another goal might be to find an efficient target language implementation.



Other AP systems may use transformation rules in various ways. For instance, the NLPQ system uses transformation rules to parse the natural language input from the user, to generate natural language output to the user, and to generate the target language program from an internal description.

Another concern of AP systems is the efficiency of the target language implementation. The two projects that dealt with this issue are PROSYSTEM I and the PSI subsystem LIBRA. While the PROSYSTEM approach to creating efficient programs combines artificial intelligence with the mathematical technique of dynamic programming, the LIBRA approach uses a more extensive range of artificial intelligence techniques, employing a variety of heuristics, estimates, and kinds of knowledge to guide its search for an efficient program.

When it is said that an AP system optimizes a program for efficiency, it does not mean that the system finds the absolutely most efficient implementation; combinatorial explosion makes such a task impossible. Instead, optimizing means making some reasonable choices in the implementation so as to achieve a reasonably efficient program.

A basic concern of all AP systems is *understanding*. Understanding a program might be defined as that which enables a system to talk about, analyze, modify, or write parts of a program. One of the central concerns of the PROGRAMMER'S APPRENTICE described below is understanding through the explicit use of *plans*. A plan represents one particular understanding or way of viewing a program, or part of a program (for a more detailed explanation, see the article on PROGRAMMER'S APPRENTICE). Understanding in the other systems is relatively implicit and does not reside in any one particular class of structures.

### Overview of the System Articles

The projects described in the system articles cover much of the current research in AP, including the four basic issues just discussed: transformation rules, search for efficiency, handling partial information, and explicit understanding.

The NLPQ system is the first AP system to utilize natural language dialogue as a specification method. The user specifies part of a simple queuing simulation problem in English, and then the system, as is necessary, answers questions posed by the user, as well as queries the user in order to complete missing information or to resolve inconsistencies. The partial knowledge that the system has obtained about the desired program is represented as a semantic net that is eventually used to generate the program in the target language GPSS. Transformation or production rules analyze the user's natural language specification, build and modify the semantic net, produce natural language responses, and finally generate the target language program.

The PSI system is more recent and consists of many subsystems; it stresses the integration of a number of different processes and sources of knowledge. The problem application area is that of symbolic programming, including information retrieval, simple sorts, and concept formation. The user can specify the desired program with a mixture of examples and mixed-initiative natural language dialogue; for an easier and more natural interaction with the user, the system maintains and utilizes a tree of the topics that occur during the specification dialogue. Through such a dialogue, PSI creates a complete, consistent description of the desired program. In the last phase, the system explores

repeated application of transformation rules in order to convert the description into a target language implementation. This last phase, the synthesis phase, is carried out by two subsystems: PECOS provides suitable transformation rules and LIBRA directs and explores the application of the rules with the goal of obtaining an efficient target implementation. PECOS and LIBRA are described in separate articles.

Both PECOS and DEDALUS are examples of full-fledged, dynamic transformation systems. They each start out with a complete specification of the desired program. Each has a knowledge base of many transformation rules that are repeatedly applied to the specification. These repeated applications produce a sequence of specifications that eventually terminate with a specification that is a target language implementation. Because more than one transformation rule can apply in some cases, each system actually develops a tree of specifications (descriptions), with eventually one or more of the final nodes of the tree being a program implementation within the target language. Part of the differences between these two systems lies in the fact that DEDALUS is concerned with the logic of such programming concepts as recursion and subroutine. On the other hand, PECOS is more concerned with the multiplicity of implementations of very high-level programming constructs and operations, because that is the task of PECOS within the PSI system. Though PECOS stresses knowledge of various implementations and DEDALUS stresses knowledge of programming constructs, both are systems where transformation is the primary emphasis.

The SAFE system article contains an extensive description of constraints and their use in handling partial information. SAFE processes a variety of different kinds of constraints, in order to fill in different kinds of information in the specification of the desired program, and employs different methods of processing these constraints. There are constraints related to type of object referenced in the specification, as well as related to sequencing of steps. Constraints are processed by backtracking and by carrying out a form of symbolic execution.

One of the ideas of the SAFE project is that a completely specified program satisfies a very large number of constraints. Information in the user's partial, fragmentary specification (partial and fragmentary since the specification does not mention all objects explicitly, or partially mentions other objects, and may not contain explicit sequencing of actions) combined with the many constraints that a formal program satisfies (and possibly with information from a knowledge base of the application area or, in special cases, from information obtained from queries to the user), taken together, fully determine a complete and formal description of the program. No other system deals in so central a way with partial information and constraints as does the SAFE system.

The LIBRA and PROTOSYSTEM I projects are concerned with efficiency of the target language implementation. LIBRA uses an artificial intelligence approach, while PROTOSYSTEM I uses a combination of some artificial intelligence with primarily the mathematical approach of dynamic programming. Dynamic programming, modified by approximations and heuristics, produces an optimized target language implementation. On the other hand, LIBRA guides the application of the transformation rules furnished by the PECOS subsystem of PSI and directs the growth of the resulting tree (see above discussion of PECOS) with the goal of finding an efficient target implementation. LIBRA determines and utilizes estimates of what it is likely to achieve by exploring the development of a particular node. LIBRA has knowledge about how its own allocation of space and time should influence its strategy in searching for an efficient implementation. Though both LIBRA and PROTOSYSTEM I are concerned with producing efficient implementations, they approach the problem in different contexts. The first

explores configurations of a data-processing program and the second explores applications of transformation rules.

The PROGRAMMER'S APPRENTICE does not necessarily write the program, but instead functions as an apprentice to the user, with the user writing none, some, or all of the program and the apprentice assisting with such tasks as writing parts of the program, checking for consistency, explaining pieces of program, and helping the user modify programs. The central concern of this project is *understanding*, understanding through the explicit device of *plans*. A plan may be thought of as a template that expresses an understanding or viewpoint. Matching the plan to a part of a program description corresponds to understanding the part in that way. Several plans can match the same part of a program, corresponding to different ways of understanding that part. Plans can also be built up in a hierarchical fashion. The goal is that the PROGRAMMER'S APPRENTICE, with the understanding attained through the use of plans, can assist the programmer with correcting mistakes, writing parts of the program, and effecting modifications.

All of these are research projects: At present none have been responsible for an AP production system. Much research remains before most of these systems can be of use to programmers.

#### References

See The Annual Review in Automatic Programming (1960), Automatic Coding (1957), Balzer (1973a), Balzer (1973b), Balzer (1973c), Biermann (1976a), Hammer (1977), Hammer & Ruth (1979), Heidorn (1976); and Heidorn (1977).

Further references for specific research areas are listed with the other articles in this chapter.

### B. Methods of Specification

There must be some means or method by which the user conveys to the AP system what kind of program the user wants. This method is called the program specification. It might entail fully specifying the program in some formal programming language, or possibly just specifying certain properties of the program. It might involve giving examples of the input and the output of the desired program, giving formal constraints on the program in the predicate calculus, or giving interactive descriptions of the program at increasing levels of detail in English. (The subject of specification is introduced in general terms in the overview article.)

#### Formal Specifications

One method of formal specification is that used with the basic approach of theorem proving (see below for this basic approach). Here one might specify a program as

$$(1) \quad \forall s_1 (P(s_1) \supset \exists s_2 Q(s_1, s_2))$$

where  $s_1$  are the input variables, and  $s_2$  the output variables.  $P(s_1)$  is the input predicate (or input specification); it gives the conditions the inputs  $s_1$  can be expected to satisfy at the beginning of program execution.  $Q(s_2)$  is the output predicate (specification); it gives the conditions the outputs  $s_2$  of the desired program are expected to satisfy.

Expression (1) states that for all  $s_1$  the truth of  $P$  implies there is an  $s_2$  such that  $Q(s_1, s_2)$  is true. If there are no restrictions on the inputs, one may simply write

$$\forall s_1 \exists s_2 Q(s_1, s_2) .$$

For example, a program that computes the greatest common divisor of two integers  $x$  and  $y$  might be specified by taking  $P(x, y)$  as the condition that  $x$  and  $y$  are positive, and  $Q(x, y, z)$  as the condition that  $z$  is the greatest common divisor.  $P(x, y)$  could be written as

$$x > 0 \text{ and } y > 0 ,$$

and  $Q(x, y, z)$  could be written as

$$\text{divide}(z, x) \text{ and } \text{divide}(z, y) \text{ and} \\ \forall r ((r > 0 \text{ and } \text{divide}(r, x) \text{ and } \text{divide}(r, y)) \supset z \geq r) .$$

The expression

$$\forall x y \exists z (P(x, y) \supset Q(x, y, z))$$

would then state that for all positive integers  $x$  and  $y$ , there is a  $z$  such that  $z$  is their greatest common divisor.

In the basic approach for this kind of specification, the above expression is given to a theorem prover that produces a proof from which a program can be extracted (see basic approach of theorem proving below). One is required to give to the theorem prover enough

facts concerning any predicates and functions that occur in P and Q so that (1) is provable. Thus, in the above, one would have to specify a number of facts concerning the predicates "divide", "<", and ">" over the integers.

Another very similar method of specification is that used with the basic approach of program transformation and of very high-level languages. This specification method stresses the use of entities that are not immediately implementable on a computer, or at least not implementable with some desired degree of efficiency. There is considerable leeway in this classification. For instance in some program transformation systems the entities employed may be quite abstract without any hint of the desired algorithm. In other systems the algorithm most naturally suggested by the specification of the program could be inefficient, but the AP system will produce an efficient but perhaps convoluted program.

One example of a specification used with program transformation is (see article D6)

$\text{gcd}(x,y) = \text{compute max } (z: \text{divide}(z,x) \text{ and } \text{divide}(z,y))$   
*where x and y are nonnegative integers greater than zero .*

This expression states that the gcd (greatest common divisor) of x and y is the maximum of all those z such that z divides x and y. Furthermore it is assumed that x and y are nonnegative integers one of which is non-zero. By successive transformations of this definition of gcd, the system would produce an efficient recursive program. Another example (Darlington & Burstall, 1973, p. 280) is

$\text{factorial}(x) := \text{if } x=0 \text{ then } 1 \text{ else } \text{times}(x, \text{factorial}(x-1)) .$

The system, then, by various transformations produces a more efficient non-recursive though more tortuous program.

### Advantages and Disadvantages of Formal Specifications

The first specification method, that involving the input and the output predicates, and based on formal logic, is completely general: anything can be specified. On the other hand the user must have a sufficient understanding of the desired program in order to give a full formal description of the input and output. This understanding can sometimes be difficult, even for simple programs. Also, the present form of theorem provers and problem reduction methods makes synthesis of longer programs difficult.

The second type of formal specification does not have such arbitrary generality, but the terminology used in the specification often is closer to our way of thinking about a particular subject, and so it should be easier to create such specifications.

Even though some of the above formal methods are arbitrarily general and others are not, they all are complete: there the specification of the desired program fully and completely specifies what the program is to do. This is not true of some of the other methods discussed below, where the specification does not uniquely determine what the program is to do. With such methods it becomes a concern whether the program produced by the system is actually what the user desires. Sometimes a system employing such a method may have to verify whether the program it produces is the program the user wants. On the other hand, with the specification methods discussed here, there is no such problem. For further reading on this subject see Sibel, Furbach, & Schreiber (1978).

### Specification by Examples

Some simple programs are most easily described using examples of what the program is supposed to do.

**Examples of INPUT/OUTPUT Pairs** In this specification method, the user gives examples of typical inputs and the corresponding outputs. Consider specifying or describing a concatenation of lists to someone who is unfamiliar with the term "concatenation". It might be most straightforward to use an example:

$$\text{concat} [(A B C), (D E)] = (A B C D E) ,$$

which states that when the input of the function "concat" consists of the two lists (A B C) and (D E), then the corresponding output is (A B C D E).

Given certain common sense assumptions, this example input/output pair should suffice to specify what it is that the desired program is to do. In more complicated cases, where the common sense assumptions are not sufficient, more examples must be given in order to specify the program. For instance, the above example could be misinterpreted as a "constant" program that always gave (A B C D E) as output:

$$\text{concat} [x,y] = (A B C D E) .$$

In such a case, giving an additional example

$$\text{concat} [(L M),(N O P)] = (L M N O P) ,$$

would probably clear up any confusion.

Another instance of this method is the specification of the function "prime" by a set of input/output pairs:

```

prime(1) = 1
prime(2) = 2
prime(3) = 3
prime(4) = 5
prime(5) = 7
prime(6) = 11

```

**Generic Examples of INPUT/OUTPUT Pairs** In certain cases, generalizations of specific examples or generic examples are more useful in order to avoid the problems inherent in partial specifications. For instance, the generic example

$$\text{reverse} [(X_1 X_2 X_3 \dots X_n)] = (X_n \dots X_3 X_2 X_1)$$

describes a list reversal function. Here, the  $X_1, X_2, \dots, X_n$  are variables which may be anything. This specification is still partial (must  $n > 0$  hold, or could  $n = 0$  ?), but it is more complete than any specification of this function given by example of input/output pairs.

**Program Traces** Traces allow more imperative specifications than do example pairs. A sorting program may be specified with input/output pairs (e.g., Green et al., 1974):

sort [(3 1 4 2)] = (1 2 3 4) ,

but it would be hard to specify an insertion sort program in the same way. Yet, a program trace could express such a program as follows:

```

sort [(3 1 4 2)] --> ( )
(1 4 2) --> (3)
(4 2) --> (1 3)
(2) --> (1 3 4)
() --> (1 2 3 4)

```

Another example of specification by traces might be

```

gcd(12,18) ->
(6,12) ->
(0,6) ->
6

```

for the specification by trace of the Euclidean algorithm which computes the greatest common divisor. An example of using a trace to specify part of a concept formation program is presented in  $\square 2$ .

More formally, a trace may be defined as follows. A programming domain can be thought of as consisting of a set of abstract objects, a set of possible representations (called *data structures*) for these abstract objects, a basic set of operators to transform these representations, and a class of questions or predicates that can be evaluated on these data structures. Thus, a programming domain characterizes a class of programs that might be constructed to operate on representations of the set of abstract objects in the domain. A *trace* is a sequence of data structure changes and control flow decisions that have caused these changes. Traces are usually expressed in terms of domain operators and tests (or functional compositions of these). Traces are classified as *complete* if they carry all information about operators applied, data structures changed, control decisions taken, etc.; otherwise, they are called *incomplete*. An interesting subclass of the latter is the class of *protocols*, in which all data flow information is explicit but all control information is omitted.

**Generic Traces** Like generic examples of input/output pairs, these may also be useful. In general, there is a whole spectrum of trace specifications depending on how much imperative information and how much descriptive information is present in the trace. For instance, the trace above is completely descriptive; traces that contain function applications and/or sequencing information tend to be more imperative.

#### Advantages and Disadvantages of Specification by Examples

As stated above, generic examples are less ambiguous than non-generic examples. Traces are less ambiguous than input-output pairs, but the user is required to have in mind some idea of how the desired program is to function. On the other hand, traces do allow some imperative specification of the flow of control.

Specification by examples can be natural and easy for the user to formulate Manna

(1977). Examples have the limitations inherent to informal program specifications: the user must choose examples so as to unambiguously specify the desired program. The AP system must be able to determine when the user's specification is consistent and complete, and that the system's "model" of what the user wants is indeed the right program.

### **Natural Language Specifications**

Given an appropriate conceptual vocabulary, English descriptions of algorithms are often the most natural method of specification. Part of the reason for this is that natural language allows greater flexibility in dealing with basic concepts than do, say, very high-level languages. This flexibility requires a fairly sophisticated representational structure for the model, with capabilities for representing the partial (incomplete) and often ambiguous descriptions users provide. In addition, it may be necessary to maintain a database of domain dependent knowledge for certain applications. Experience with implemented systems such as SAFE Baizer, Goldman, & Wile, 1977a; see also D3, suggests that the relevant issues are not in the area of natural language processing but in how the specifications are modeled in the system and what "programming knowledge" the system must have.

### **Mixed-Initiative Natural Language Dialogue**

More versatile, this specification method involves interaction between the user and the system as the system builds and tries to fill in the details in its model of the algorithm. In addition to maintaining a model of the algorithm, such systems sometimes for purposes of the dialogue will even maintain a kind of model of the user to help the system tailor the dialogue to a particular user's idiosyncracies. Various techniques mentioned previously, such as examples or traces, could be used in the dialog as a description of some part of the algorithm. The system might be so designed as to allow users to be as vague or ambiguous as he pleases; the system will ultimately ask them enough to fill in the model.

This method is probably the closest to the usual method of program specification used by people, allowing both the specifier and the programmer to make comments and suggestions. Users do not have to keep every detail in mind, nor do they have to present them in a certain order. The system will eventually question the user for missing details or ambiguous specifications. On the other hand, this method requires a system that deals with many problems of natural language translation, generation, and representation. A representation is also required for the system's model of the algorithm.

The PSI system Green, 1976b; see also D2 and the NLPQ system Heidorn, 1974; see also D8 use this method of program specification. Floyd (1972) and Green (1977) give hypothetical dialogs with such a system, illustrating the problems that researchers have encountered with this approach.

### **References**

See Biermann (1976a) and Heidorn (1977). For examples of individual specification methods see the remaining articles of this chapter.



### C. Basic Approaches

The following are some of the basic approaches used in Automatic Programming (AP) systems to synthesize desired programs from user specifications. There is not always a clear distinction between synthesis and specification. Furthermore, as will be seen from the later articles, some systems employ primarily one approach while others employ more elaborate paradigms that use several approaches. (This material is introduced in the overview article above.)

#### Theorem Proving

The theorem-proving approach is used for the synthesis of programs whose input and output conditions can be specified in the formalism of the predicate calculus. As stated in the section on formal specifications, the user specifies the desired program for the theorem prover in the form:

$$\forall s_1 ( P(s_1) \supset \exists s_2 Q(s_1, s_2) ) ,$$

where  $s_1$  is one or more input variables,  $s_2$  is one or more output variables,  $P$  is the predicate that  $s_1$  is expected to satisfy, and  $Q$  is the predicate that  $s_2$  is expected to satisfy after execution of the desired program. In addition to the above expression, the theorem prover must also be given enough axioms to make the above expression provable.

From the proof produced by the theorem prover, a program is extracted. For instance, certain constructs in the proof will produce conditional statements, others sequential statements; and occurrences of induction axioms may produce loops or recursion. There are several variant methods of accomplishing this (see Green, 1969, Waldinger & Levitt, 1974, Kowalski, 1974, Clark & Sickel, 1977).

Although any interesting example would be far too long to work out in all of its detail here, it may be worthwhile to show how such a problem is set up. Consider the very simple problem of sorting the dotted pair of two distinct numbers, in LISP. The axioms that would prove useful for this synthesis would be:

- 1)  $x = \text{car}(\text{cons}(x,y))$
- 2)  $y = \text{cdr}(\text{cons}(x,y))$
- 3)  $x = \text{nil} \supset \text{cond}(x,y,z) = z$
- 4)  $x \neq \text{nil} \supset \text{cond}(x,y,z) = y$
- 5)  $\forall x,y (\text{lessp}(x,y) \neq \text{nil} \leftrightarrow x < y)$

The specification of the desired program, and the theorem to be proved, would be:

$$\forall x. \exists y. [\text{car}(x) < \text{cdr}(x) \supset y = x] \wedge$$

$$[\text{car}(x) \geq \text{cdr}(x) \supset \text{car}(x) = \text{cdr}(y) \wedge \text{cdr}(x) = \text{car}(y)] .$$

This says that for every dotted pair input  $x$ , there is a dotted pair output  $y$  such that if  $x$  is already sorted, then  $y$  is the same as  $x$ , and if  $x$  is not sorted, then  $y$  is the interchange of the two elements of  $x$ . Using the techniques of resolution theorem proving (Theorem Proving.C), we would obtain the following program:

$$y = \text{cond}(\text{lessp}(\text{car}(x), \text{cdr}(x)), x, \text{cons}(\text{cdr}(x), \text{car}(x)))$$

In general, programs to be synthesized will not be as simple as the one above. One of the major problems that more complicated programs introduce is that they require some form of iteration or recursion for solution. To form a recursive program, one needs the proper induction axioms for the problem. A general schema for the induction axiom sufficient for most programs is:

$$[P(h(\text{nil}), \text{nil}) \wedge \forall x[\text{ATOM}(x) \wedge P(h(\text{cdr}(x)), \text{cdr}(x)) \supset P(h(x), x)]] \\ \supset \forall z [P(h(z), z)]$$

where  $P$  is any predicate and  $h$  is any function. Somehow this predicate and function must be determined. Necessitating the user to supply the induction axioms for each program to be synthesized somewhat defeats the purpose of the synthesis, yet having the system generate induction axioms until one of them works takes up far too much time and memory. Systems that determine the  $P$  and  $h$  usually use various heuristics to limit search.

There are several constraints inherent to the approach of theorem proving. First, for more complicated programs, it is often more difficult to correctly specify programs in the predicate calculus than it is to write the program itself. Second, the domain must be axiomatized completely, that is, one must give enough axioms to the theorem prover so that any statement that is true of the various functions and predicates that occur in the specification of the program can actually be proved from the axioms--otherwise, the theorem prover may fail to produce a proof, and thereby fail to produce the program. Third, present theorem provers lack the power to produce proofs for the specification of very complicated programs. To summarize, the user must fully and correctly specify the desired program, the theorem prover must be given enough axioms so that the specification is provable, and the theorem prover must be strong enough to prove the specification.

It should be noted that this approach does not allow partial specification: Users cannot specify the program partially with the system helping them to fill in details. On the other hand, when a theorem prover does succeed in producing a proof of the specification, the correctness of the extracted program is guaranteed. Thus, AP systems might incorporate theorem proving where it is either convenient or where correctness is an important requisite.

### Program Transformation

The transformation approach has been used to automatically convert an easily written, easily understood LISP function into a more efficient, but perhaps convoluted program. One such system, described in Darlington & Burstall (1973), performs recursion removal, the elimination of redundant computation, expansion of procedure calls, and reuse of discarded list cells.

The recursion removal transforms a recursive program into an iterative one, which is generally more efficient, avoiding the overhead of the stacking mechanism. Candidates for recursion removal were determined by pattern matching the parts of the program against a recursive schema input pattern. If the match is successful and if certain preconditions are met, then the program is replaced by an iterative schema. A simple example of such a transformation rule is:

```

input pattern: f(x) ::= if a then b else h(d,f(e));
precondition: h is associative, x does not occur free in h;
result pattern: f(x) ::= if a
                    then result ← b
                    else begin
                        result ← d;
                        x ← e;
                        while not a
                        do begin
                            result ← h(result,d);
                            x ← e
                        end;
                        result ← h(result,b)
                    end

```

where a, b, d, e, f, and h in the input pattern are matched against arbitrary expressions in the candidate functions. For example, the function,

```

FACTORIAL(x) ::= if( x=1) then 1 else TIMES (x, FACTORIAL (x-1))

```

would match the above input pattern with  $f = \text{FACTORIAL}$ ,  $a = (x=1)$ ,  $b = 1$ ,  $h = \text{TIMES}$ ,  $d = x$ , and  $e = (x-1)$ . The resulting program would be the resulting pattern with these values substituted for a, b, d, e, f, and h.

Eliminating redundant computations includes traditional subexpression elimination as well as combining loops that iterate over the same range. The latter includes implicit iteration. Thus, if A, B, and C are represented as linked lists, the sequence:

```

X ← INTERSECTION (A,B)
Y ← INTERSECTION (A,C) ,

```

is really two implicit iterations, each over the set A. A suitable transformation rule would convert this into a single iteration over the set A.

Expanding procedure calls generally involves substituting the body of a procedure for each of the calls to it. The potential benefit arises from simplifications made possible by use of the local context. This technique is the starting point for a general class of transformations explored in Burstall & Darlington (1975), Wegbreit (1975a).

Program transformation is also used to convert very high-level specifications into target language implementations (see D6, D5, as well as summaries of these articles in A).

### Knowledge Engineering

AP systems are said to be "knowledge-based" when they are built by identifying and codifying the knowledge that is appropriate for the program synthesis and understanding (i.e., ability to manipulate and analyze programs) and by embedding this knowledge in some representation. Many of these systems use large amounts of many kinds of knowledge to

analyze, modify, and debug large classes of problems. While the distinction is relative, it is possible to divide this knowledge into two types: programming knowledge and domain knowledge.

Programming knowledge includes both *programming language knowledge*, which is knowledge about the semantics of the target language in which the system will write the desired program, and *general programming knowledge*, which is knowledge about such things as generators, tests, initialization, loops, sorting, searching, and hashing. Programming knowledge includes: (a) optimization techniques, (b) high-level programming constructs (loops, recursion, branching), and (c) strategy and planning techniques.

Domain knowledge is what is necessary for a system to infer how to go from the problem description or specification to what needs to be done to solve the problem. This "know-how" includes how to structure the concepts in the domain or problem area and find interrelationships among them. It must also include knowledge about how to achieve certain results in the problem domain (cf., HACKER's learning of procedures Problem Solving.B5). Moreover, it should be able to define the problem in alternative ways and find alternative ways to solve the task--such knowledge represents an "understanding" of the domain.

Knowledge-based systems need a method of reasoning. Since they are not restricted to using the traditional formalisms of logic, they often supply their own flexible reasoning techniques for guiding the synthesis. Some of these techniques include inference, program simplification, illustration and simplification for the user, decision trees, problem-solving techniques, and refinement.

The basic concern in representing the knowledge is that the knowledge be structured in such a way that the search for relevant facts does not cause a combinatorial explosion. Various representations employed include:

- PLANNER-like procedural experts (AI Languages.C1),
- Refinement rules (D5),
- Modular, frame-like experts (OWL (Martin, 1974) and BEINGS (Lenat, 1975))
- Semantic nets (D8), and
- Amorphous systems that try several ad hoc techniques ((Biggerstaff, 1976)).

Methods of accessing knowledge bases include: pattern invocation (D5), "when needed" (Sussman, 1975); frame relations and assertions, including the filling in of process models (Martin, 1974, Green, 1969, Lenat, 1975, D8 D2, D3); and subgoal or case analysis (Green, 1977 and D6).

#### Automatic Data Selection

This approach refers to the selection of efficient low-level data-structure implementations for a program specified in terms of high-level abstract information structures

(e.g., sets). Generally, programming languages containing abstract data types have default representations that are a compromise between all likely uses of the structures; these data types are typically far from efficient in any one particular program. But a system with automatic data selection would choose, from a collection of possible implementations, an implementation more efficient for the particular program under consideration. For example, the abstract data type *set* could be represented in low-level implementations as a linked list, a binary tree, a hash table, a bit string, or as property list markings. Various operations on sets are easier in one representation than in another--e.g., set intersection using bit strings is simply a logical AND operation, while iteration over a set is easier when it is represented as a linked list--and some representations may not even be applicable in a given case (e.g., bit strings require that the domain of set elements be fixed and reasonably small, since one bit position is used for each possible element). Also, some representations may not permit all needed operations (e.g., the only way to enumerate the items in a set represented with property markings is to enumerate all atoms in the system.) By tailoring the representation to the particular programmer's intention, it is possible to produce much better code.

One such system performing data-structure selection for the user is Low (1974), Low (1978). This system handles simple programs written in LEAP, a sublanguage of SAIL. It selects representations for sets, sequences, and relations from the fixed library of low-level data structures available in LEAP. The selection is guided by the goal of minimizing the product of the memory and time required to execute the resulting program.

The system begins with an information-gathering phase that searches out the relevant characteristics of the program's data structures, such as their expected size, number, the operations performed on them and their interactions. Some of this information is obtained by questioning the user, and some is obtained by monitoring the actual execution of the program on typical data, using default representations for each structure. Then the system partitions into equivalence classes the variables whose values will be of the same type of data structure. The system employs a method similar to hill climbing Search.Overview in order to determine a good assignment of data structures to the equivalence classes (i.e., the representations assigned to the equivalence classes are repeatedly varied, one at a time, to see if an improvement will result). For further details, see the above references.

Other AP systems are also concerned with the selection of an efficient set of data structures or file structures, but this concern is part of the general goal of writing an efficient program (see Articles D7 and D9).

### Traditional Problem Solving

Traditional problem solving refers to using goals to direct the application of operations in a state space (see Search). The Heuristic Compiler (Simon, 1972) regards the task of writing a program as a problem-solving process using heuristic techniques, like those of GPS (see Article Search.D2). This pioneering work recognized the value of both a *state language*, to describe problem states and goals, and a *process language*, to represent the solver's actions.

In the Heuristic Compiler, the *State Description Compiler* is quite similar to later work on synthesis from examples. The program being synthesized is defined by specifying input/output conditions on the memory cells that it affects. The difference between the

current state and the desired state is looked up in a table that specifies which operators to apply to transform the contents of the cells appropriately. The *Functional Description Compiler* is an important precursor to later work in automatic modification and debugging of programs. It uses a means-ends analysis to transform a known (compiled) routine into a new (desired) routine.

HACKER, a system by Sussman (1975), adds to Simon's work, detecting and generalizing new differences (bugs) and defining appropriate operators to resolve them (patches). This system uses many significant AI techniques and language features: learning through practice how to write and debug programs; modular, pattern-invoked expert procedures (chunks of procedural knowledge); and hypothetical world models for subgoal analysis. Sussman's emphasis on generalizing from experience (trying old techniques in new situations), acceptance of the fact that users have an incomplete understanding of the desired program, and his goal-purpose annotation technique are all interesting directions in the development of Automatic Programming.

However, HACKER's preference for ruthless generation of "buggy" code without detailed planning has led to inadequate handling of subgoal conflicts. The user must carefully schedule the training sequences and be ready for the combinatorial explosion as the system exhaustively searches its base of world facts and programming knowledge. Such systems must constrain the search problem of large knowledge bases. Other attempts to distribute knowledge among interacting specialists have encountered the same difficulty (Lenat, 1975).

We find that systems such as HACKER, which are designed to operate like human programmers, have experienced a moderate degree of success compared to knowledge-impooverished formal methods. However, these systems are still often hampered by the rigid formalism that governs their application: In what order are operators to be applied? How can domain-specific information be specified as differences? The formalisms used to incorporate the various knowledge sources in these systems seem too methodical; the method is space and time bound because it is based on search.

### Induction

Induction or inductive inference refers to the system's "educated guess" at what the user wants on the basis of program specifications that only partially describe the program's performance. Such specifications are often the examples of input/output pairs and program traces, in both regular and generic form (B). For each of these kinds of specification, the corresponding AP system must determine the general rules on the basis of a specification that contains only a few examples (or in the generic specifications, a limited class of examples) of the program behavior.

The work in *program synthesis from specification by examples* had its origin in research dealing with grammatical inference, where the objective was to infer a grammar that described a language, given several examples of strings of the language. The crucial issue for programming synthesis is to develop a generalized program, that is, one that can account for more than the examples given in the program specification. To do this, these programs break down the input, looking for recursively solvable subparts (Shaw, Swartout, & Green, 1975) or computation repetitions that can be fitted into a known program scheme (Hardy, 1975).

The work in *program synthesis from trace specifications* seeks to invert the transformations observed in a trace protocol to create abstractions that generalize into loops and variables (Bauer, 1975). Bliermann & Krishnaswamy (1974) has built a system that interprets traces as directions through a developing flowchart.

All inductive inference systems are dependent upon a good *axiomatization of operations*. In other words, the system must know about all of the possible primitive operations that can be applied to the data structures if it is to hope to construct, by composition of these primitives, the desired program. Furthermore, a harmonious relation between the nature of the constructs in the specification and the most basic constructs in the target language is essential; for example, in Siklossy & Sykes (1975), the tasks of tree traversal and repetitive robot maneuvers are directly translatable into LISP recursion. Moreover, these programs are required to know quite a bit about generalization. After synthesizing the program, they test it on other examples, sometimes by generating test cases and sometimes by asking the user for approval. For certain classes of programs, examples and traces provide a natural way for the user to specify what the desired program is to do.

#### Induction For Input/Output Pairs

The synthesis of programs from a specification consisting of examples of input/output pairs is strongly related to the problem domain to which these programs belong (e.g., sorting, concept formation). A set of program schemata characterize the entire class of programs for the domain. These schemata are like program skeletons. A schema defines the general structure of a program, omitting some details. The synthesis of the program, then, amounts to (a) selecting a given schema that is representative of the program specified by the set of example pairs, and then (b) using the information present in the examples to instantiate the unfilled slots of the schema. So, there are two steps: a *classification* process, which selects the general structure (schema) of the target program, and an *instantiation* process, which completes the details of the target program.

What does the classification process require? Every schema defines a subclass of programs in the problem domain. Every set of example pairs defines a program in the domain. Thus, the classification process must associate these programs with one of the subclasses of programs in the domain. In order to accomplish this task, a set of characteristics is associated with each schema (subclass) that, if present in the set of example pairs, guarantees that the set specifies a program of this type. Usually this task is accomplished by (a) providing a set of *difference measures* to be applied to the inputs and outputs of an example pair, as well as to different example pairs in the input collection (if it consists of more than one), and (b) providing a set of heuristics for each program schema that determine a *fit* measure of the example set that accompanies it. The task of classifying the example set is then simply reduced to choosing the schema with the highest fit value.

During the instantiation process, in addition to the difference fit measures described above, every schema has an associated set of rules for filling its empty slots through extracting necessary features from the examples. For instance, in the domain of list manipulation functions, cases where the output list contains all elements in the input and cases where the output list contains only every other element, etc., suggest different methods of constructing the output incrementally from the input. In the first case the function maps down the input list; in the second case it maps down the input using the LISP

CDDR function. Slots are instantiated by these rules in terms of primitive operators of the domain and their functional compositions (in the above case, the basic LISP functions and their compositions).

Once a schema has been selected and instantiated, the synthesis algorithm must *validate* its hypothesis. This task is usually done either by generating some new examples for the program, evaluating the synthesized program on the example set, and checking the results with the user or presenting the program to the user and letting him/her verify its correctness.

In summary, the basic algorithm is:

- (1) Apply the difference measures to the example set.
- (2) Based on this application, classify the set into a particular schema class.
- (3) Using heuristics associated with the particular schema, hypothesize a complete instantiation of the selected schema.
- (4) Validate this hypothesis.

In this basic algorithm, if there is a single I/O-pair in the specification, the difference measures are just a set of feature-detecting heuristics. If there is more than one pair, the pairs may be ordered according to the complexity of the input. Difference measures will fall into two classes: those that associate the structure of a pair with a schema class, and those that find differences between pairs. The latter are perhaps more crucial in the inference of a program. From these differences, a theory for the operation of the program is inductively inferred or, what is the same, a formation rule is derived. This operational theory might take the form of a certain schema class or of a recurrence equation that, in turn, specifies a schema class. In the classification phase it may be necessary to apply the classification rule to all pairs in order to infer the corresponding schema correctly. When several different schemas have been inferred, a decision rule is required to select the correct one.

An alternative approach is to reduce the whole problem to another paradigm for synthesizing programs. For example, if the problem domain has been formalized, so that there is a set of operators for the domain, it is possible to use a traditional problem solver to generate a solution to the input/output pair (considered as initial-state, goal-state). The solution so obtained can be considered a trace of the program to be synthesized and a trace-based paradigm may be employed.

Specification by examples is suitable for synthesizing a program only in those cases where the task domain is small and easily axiomatizable. It may also be a feasible approach in the case where the domain is repetitious enough that a small set of pairs is sufficient to specify the program.

Specification by example is quite limited and does not lend itself to useful generalization to large domains. Nevertheless, the power of examples for clarifying concepts is unquestionable. It seems that the main application that this specification formalism will have in future automatic programming systems is restricted to the annotation and clarification of more formal program descriptions.



### Induction From Traces

The primary approach is first to enumerate, in order of increasing size, all possible programs constructed from domain operators, tests, and their functional compositions; then after each new program is generated, to validate the given traces against the program. If the program generated produces the traces, then it is the required solution. Note that a measure of program size is needed for the enumeration (e.g., number of instructions in program). This paradigm is clearly not generally practical and is suited only to the inference of small programs in very simple domains. It has been applied with moderate success to the inference of programs from memory traces. These traces usually consist of register assignments, tests, and memory modification instructions and, as such, are not very complex. Programs as complex as Hoare's FIND algorithm have been synthesized in this manner (Petry & Biermann, 1976). There are certain other special inference paradigms for some trace classes.

**Problem-solver generated traces** If the domain is fully axiomatized, as may be the case for simple domains like those for robots, it may be possible to synthesize programs from example pairs using a problem solver that produces a solution to the input pair in the form of a trace.

- (1) Synthesize trace from example pair via problem solver.
- (2) Using the trace, a set of program schemas for the domain, and a set of schema selection and instantiation heuristics that operate on trace steps, produce a program in terms of domain operators and domain predicates that realizes the example pair.

All these paradigms work only for complete traces and protocols. The problem of program inference from incomplete specifications is still under investigation. It is possible that the techniques outlined may be extended to cover the incomplete case by coupling the program synthesizer to a domain-based theory formation module that could, so to speak, "fill in" the missing elements from the original specification. At this point, then, the methodology discussed above could be used.

Traces have the limitations inherent to informal program specifications, namely, the difficulty of specifying the required program uniquely with respect to the limited amount of information conveyed to the synthesizer. Thus, the problem of choosing a good description is left, as a burden, to the user. This problem might be alleviated by the use of greater domain expertise--to produce the program that more nearly resembles the user's desired result.

Traces, and informal specification methods, will be a useful for algorithm description and correction in future automatic programming systems. Clearly, the reason for this is that these methods closely reflect the form in which we humans understand and describe programs. Current applications include the synthesis of calculator-like programs from memory-register traces (Biermann & Krishnaswamy, 1974).

**References**

For theorem proving, see Green, 1969, Waldinger & Levitt, 1974, Kowalski, 1974, Clark & Sichel, 1977; for program transformation, (Darlington & Burstall, 1973), (Wegbreit, 1975a), D6, D5 and A; for knowledge engineering, (Martin, 1974), (Lenat, 1975), (Biggerstaff, 1976), (Sussman, 1975), (Green, 1977), D5, D8, D2, D3, D6; for automatic data selection, (Iref Low78); for traditional problem solving, (Simon, 1972), Sussman, 1975; For induction from input/output pairs Amarel (1972), Green (1975a), Hardy (1975), Shaw, Swartout, & Green (1975), Siklossy & Sykes (1975), and Summers (1977); and for induction from traces, Bauer (1975), Biermann (1972a), Biermann (1976a), Petry & Biermann (1976), and Siklossy & Sykes (1975).

#### D. PSI

The goal of the PSI system being developed by Cordell Green and his colleagues at Systems Control, Inc., and at Stanford is the integration of the more specialized methods of automatic programming into a total system. This system then would incorporate specification by examples, by traces, or by interactive natural language dialogue; knowledge engineering; model acquisition; program synthesis; and efficiency analysis. Research objectives include the organization of such a system, the determination of the amount and type of knowledge such a system would require, and the representation of this knowledge.

The program is specified by means of an interactive, mixed-initiative dialogue, which may include as a subpart the specification by example of a trace. Plans are also underway to add specification by means of a loose, very high-level language. The different specification methods can usually be intermixed.

When the specification is interactive natural language dialogue, the user furnishes both a description of what the desired program is to do and an indication of the overall control structure of the program.

The problem area of PSI is symbolic computation, including list processing, searching and sorting, data storage and retrieval, and concept formation.

The overall operation of the system, illustrated in Figure 1, may be divided into two phases: acquisition of a description of the program, and synthesis of the program. During the acquisition phase, several modules of the system--including the parser/interpreter, example/trace, explainer, and moderator--will jointly interact with the user to obtain and construct a net, called the program net, that describes the desired program. Then the program model-builder module converts the net into a complete and consistent description of the program. Afterwards, during the synthesis phase, the coding and efficiency modules, interacting with each other, convert the program model, through the use of repeated transformations, into an efficient program written in the target language.

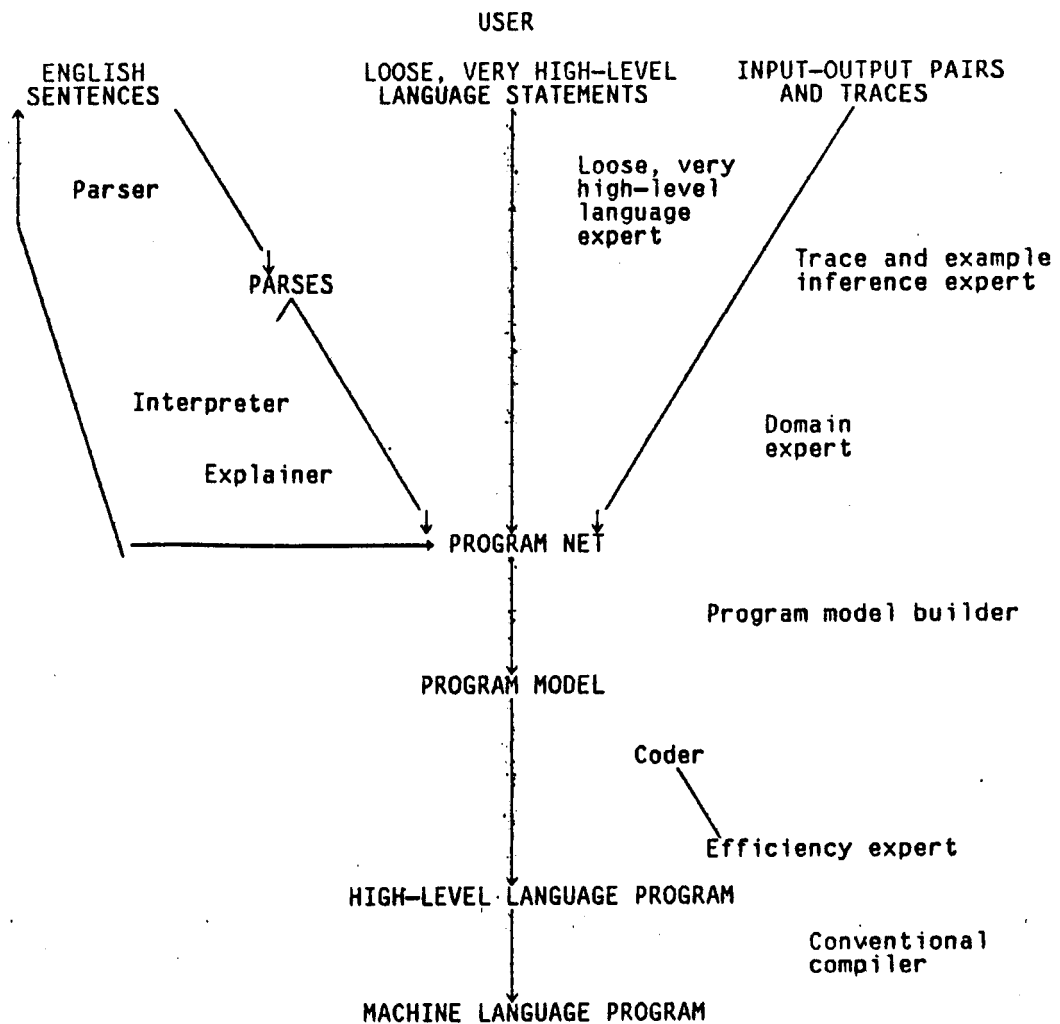


Figure 1: Major paths of information flow in PSI

There were three reasons for separating the operation into acquisition and synthesis phases. First, the problems of designing such a system are more tractable because of the separation. Second, it was envisioned that code generators for different target languages and domain experts for different problem areas could be implemented to result in a versatile modular system. Third, acquisition requires interaction with the user, whereas, in PSI, synthesis does not.

In the overall operation, two of the primary interfaces within the PSI system are the program net and the program model. Both are very high-level program and data structure description languages. The program net forms a looser description of the program than does the program model. Fragments of the program net can be accessed in the order of occurrence in the dialogue, rather than in execution order, which allows a less detailed, local, and partial specification of the program. Since these fragments correspond rather closely to what the user says, they ease the burden of the parser/interpreter as well as the example/trace inference module. As opposed to the program net, the program model includes complete, consistent, and interpretable very high-level algorithmic and information structures. Further description of the program model occurs in the section below on the program model builder.

The remainder of this article briefly describes the PSI modules, presents the status of PSI, and then describes several examples (Figures 2 through 5) from the acquisition phase. The latter includes a specification by interactive natural language dialogue, the resulting program net and model, and a specification by trace.

### Experts

PSI is a knowledge-based system organized as a set of closely interacting modules, also called experts. These experts include:

- parser/interpreter expert, explainer expert,
- dialogue-moderator expert,
- applications domain expert, example/trace inference expert,
- program model-building expert, coding expert, and the
- algorithm analysis and efficiency experts.

### Parser/Interpreter

In the acquisition phase, the parser/interpreter expert (Ginsparg, 1978) first parses sentences and then interprets these parses into less linguistic and more program-oriented terms, which are then stored in the program net. This expert efficiently handles a very large English grammar and has knowledge about data structures (e.g., sets, records), control structures (e.g., loops, conditionals, procedures), and more complicated algorithm ideas (e.g., interchanges between the user and the desired program, set construction, quantification). The parser/interpreter can sometimes assign a concept to an unknown word on the basis of the context in which the word appears.

### Dialogue Moderator Expert

This expert (Steinberg, 1976) models the user, the dialogue, and the state of the system and selects appropriate questions and statements to present to the user. It also

determines whether the user or the expert has the initiative, and at what level on what subject, and attempts to keep PSI and the user in agreement on the current topic. It provides review and preview when the topic changes. This expert decides which of the many questions being asked by the other experts should be passed on to the user. Since experts phrase questions in an internal form based on relations, the dialogue-moderator expert gives questions to the explainer expert which, in turn, converts them into English and gives them to the user.

### **Explainer Expert**

The explainer expert, developed by Richard Gabriel, phrases questions in terms that the user finds meaningful (i.e., in terms related to the problem domain and the previous sentences in the dialogue), rather than using the more programming-oriented terms used in the program net or by the model builder. For example, rather than asking for the definition of "A0018," PSI asks what does it mean for "a scene to fit a concept." The explainer also generates English descriptions of the net.

### **Example/Trace Expert**

PSI also allows specification by traces and examples, since these are useful for inferring data structures and simple spatial transformations. This expert Phillips (1977) handles simple loop and data structure inference and uses several of the techniques discussed in the last three articles. The final section of this article illustrates how the PSI user can specify part of a program using traces.

### **Domain Expert**

The domain expert, developed by Jorge Phillips, uses knowledge of the application area to help the parser/interpreter and example/trace experts fill in missing information in the program net.

### **Model Builder**

The program model-building expert McCune (1977) applies knowledge of what constitutes a correct program to the conversion of the program net into a complete and consistent program model, which then will be transformed during the synthesis phase into the target language implementation. The model-building expert completes the model by filling in the various pieces of required information and by analyzing the model for consistency; it checks to see that its parts are legal both with respect to each other and with respect to the semantics of the program-modeling language. Information is filled in either by default, by inference mechanisms (which are in the form of rules and which make use of consistency requirements), or by queries to other experts, which may eventually result in a query to the user. As an example, suppose that the program net contains "x part of y" and that the model builder needs to fill in whether "part of" is to mean set membership, subset inclusion, component of y, the image of x under some correspondence relation with y, or whether there might be an unspecified intervening subpart. Such information may be deducible from the structures of x and y, if these structures are known or when they become known.

The model builder also corrects minor inconsistencies, adds cross-references, and generalizes parts of the program description so that the synthesis phase has more freedom in looking for a good implementation. Thus, if the program net specifies that a certain object is to be a set of ordered pairs, the program model may, if appropriate, indicate that the object is to be a correspondence (i.e., a functional mapping).

### **Coding and Efficiency Experts**

These two experts are responsible for the synthesis phase. The coding expert's knowledge base contains rules that transform parts of a program description to forms closer to the target language. It is the goal of the efficiency expert to guide the choice of the different rules so that an efficient target language implementation eventually results. These two experts, also known as the systems LIBRA and PECOS, are discussed more explicitly in articles devoted to their description. See the articles D905.

### **Status**

The entire PSI system can now be used by a knowledgeable user. PSI has successfully produced LISP code that implements the specifications given in English dialogues for four variants of CLASS, a simple pattern classification program. In addition, other tasks are handled by individual experts in a stand-alone mode; see, for example, D5 D9.

The next section concludes this article by describing examples of PSI at work.

### **Examples**

Figure 2 illustrates an interactive natural language dialogue held with the PSI system in which the user specifies a simple program, to be called CLASS. CLASS includes many programming constructs necessary for more complex programs.

What is the name of the program you want to write?

\*\* CLASS.

Does CLASS have a special program type?

\*\* Classification.

Please describe CLASS.

\*\* CLASS first inputs a concept. Then it repeatedly accepts an input \*\*scene from the user, classifies it based on whether or not it fits the \*\*concept, and outputs this classification to the user.

What is the structure of the concept?

\*\* A concept is a collection of relations. Each relation is just a string.

What is the exit test of the loop?

\*\* Stop when the user types "QUIT" instead of a scene.

What is the structure of the scene?

\*\* The same as the concept.

Please define what it means for a scene to fit a concept.

\*\* A scene fits a concept if all the relations in the concept are in the scene.

What is the format of the classification output?

\*\* Output "FIT" if the scene fit, "DIDN'T FIT" if it didn't.

Are we finished?

\*\* Yes.

Figure 2. Specification by interactive natural language dialogue: CLASS

The user intends that the CLASS program input a sample set of items (e.g., a list of job qualifications) and then repeatedly input a trial set (e.g., the qualifications of an applicant), each time testing whether the sample set (required qualifications) is a subset of the trial set (applicant qualifications) and printing "FIT" or "DIDN'T FIT," accordingly. The user further intends that a person will be able to terminate CLASS simply by typing the word "QUIT," instead of a trial set.

Based upon its understanding of the dialogue, the parser/interpreter expert produces the program net, which is summarized in Figure 3 (the algorithmic part of the net is shown in an ALGOL-like notation). Then the program-model building expert creates the very high-level complete and consistent model of Figure 4. After repeated application of transformation rules during the synthesis phase, the coding and efficiency experts will convert this model into an efficient target language implementation.

A2 is either a set whose generic element is a string or a string whose value is "QUIT".

A1 is a set whose generic element is a string.

A4 is the generic element of A1.

A3 is either TRUE or FALSE.

B1 is a variable bound to A2.

B2 is a variable bound to A1.

B3 is a variable bound to A4.

```

CLASS
  PRINT("Ready for the CONCEPT")
  A1 ← READ()
LOOP1:
  PRINT("Ready for the SCENE")
  A2 ← READ()
  IF EQUAL(A2,"QUIT") THEN GO _ TO EXIT1
  A3 ← FIT(A2,A1)
  CASES: IF A3 THEN PRINT("FIT")
        ELSE IF NOT(A3) THEN PRINT("DIDN'T FIT")
        GO _ TO LOOP1
EXIT1:

```



```
FIT(B1,B2)  
FOR_ALL B3 IMPLIES(MEMBER(B3,B2),MEMBER(B3,B1))
```

Figure 3. Summary of the program net.

```

program CLASS;
  type
    a0032 : set of string ,
    a0053 : alternative of [<string = >"QUIT" , a0032];
  vars
    a0011 , a0014 , a0035 , a0036 : a0032 ,
    a0055 , m0080 : a0053 ,
    m0095 : string = "DIDN'T FIT" ,
    m0092 : string = "FIT" ,
    m0091 : Boolean ,
    m0081 : string = "QUIT" ;
  procedure a0067(a0036 , a0035 : a0032) : Boolean ;
    a0035 K a0036 ;
  procedure a0065(a0055 : a0053) : Boolean ;
    a0055 = "QUIT" ;
  begin
    a0011 ← input(a0032 , user , "READY FOR CONCEPT" ,
      "Illegal Input. Input again: ") ;
    until A0051
    repeat
      begin
        m0080 ← input(a0053 , user , "READY" , "Illegal Input.
          Input again: ") ;
        if a0065(m0080) then assert_exit_condition(A0051) ;
        a0014 ← m0080 ;
        m0091 ← a0067(a0014 , a0011) ;
        case
          & m0091 : inform_user("DIDN'T FIT") ;
          m0091 : inform_user("FIT") ;
        endcase
      end
    finally
      A0051 :
    endloop
  end ;

```

Figure 4. The program model.

Traces are another method of specification allowed by the PSI system. Figure 5 shows the use of a trace to specify part of the behavior of a program called TF ("Theory Formation"). A simplified version of Pat Winston's concept formation program, (Winston, 1975), TF builds and updates an internal model of a concept. A concept is a collection of "may" and "must" conditions. TF builds and updates the model by repeatedly reading in a scene, guessing whether the scene is an instance of the concept, verifying with the person using TF whether the guess was correct or incorrect, and updating the model of the concept accordingly. The trace in Figure 5 shows the specification for only a part of the behavior of TF, the part that describes how TF is to update the model, given that a scene does or does not fit a concept. The other parts of TF can be specified by trace or by interactive natural language dialogue.

```

Concept:      []
Scene:        [(block a)(block b)(on a b)]

```

Result of fit:	True
Updated concept:	[[ (block a) may ] [ (block b) may ] [ (on a b) may ]]
Concept:	[[ (block a) may ] [ (block b) may ] [ (on a b) may ]]
Scene:	[ (block a) (block b) ]
Result of fit:	False
Updated concept:	[[ (block a) may ] [ (block b) may ] [ (on a b) must ]]
Concept:	[[ (block a) may ] [ (block b) may ] [ (on a b) must ]]
Scene:	[ (block a) (block b) (block c) (on a b) ]
Result of fit:	True
Updated concept:	[[ (block a) may ] [ (block b) may ] [ (block c) may ] [ (on a b) must ]]

Figure 5. A specification by trace.

From this specification the example/trace inference expert generates the following information about the desired program: If the scene fits the concept, then add all relations in the scene but not present in the concept to the concept and mark them with "may." Otherwise, if the scene doesn't fit the concept, then change the marking of all relations marked "may" in the concept and not appearing in the scene from "may" to "must."

#### References

See Barstow (1977a), Barstow (1977b), Barstow (1977c), Barstow & Kant (1977), Ginsparg (1978), Green (1975a), Green (1975b), Green (1976a), Green (1976b), Green (1976c), Green (1977), Green (1978), Green & Barstow (1975), Green & Barstow (1977a), Green & Barstow (1977b), Green & Barstow (1978), Kant (1977), Kant (1978), McCune (1977), Phillips (1977), and Shaw, Swartout, & Green (1975).

### E. SAFE

The SAFE system, developed at USC Information Sciences Institute by Robert Balzer, Neil Goldman, David Wile, and Chuck Williams (with the recent addition of Lee Erman and Phil London), accepts a program specification consisting of pre-parsed English, with limited syntax and vocabulary, including terms from the problem domain. The phrases and sentences of this specification, however, may be ambiguous and may fail to explicitly provide all the information required in a formal program specification. Therefore, using a large number of built-in constraints (that must be satisfied by any well-formed program), any specified constraints on the problem domain, and an occasional interaction with the user, SAFE resolves ambiguities, fills in missing pieces of information, and produces a high-level, complete program specification. To decide on missing pieces of information, SAFE uses a variety of techniques, including backtracking (see article *AI Languages*) and a form of symbolic execution.

The SAFE system views the task of Automatic Programming as the production of a program from a description of the desired *behavior* of that program. There are four major differences between a conventionally specified program and a program described in terms of its desired behavior.

**Informality:** The behavioral description is informal. It contains ambiguity (alternative interpretations yielding distinct behaviors) and "partial" constructs (constructs missing pieces of information that must be supplied before any interpretation is possible). A conventionally specified program, on the other hand, is formal; its meaning is completely and unambiguously defined by the semantics of the programming language.

**Vocabulary:** The primitive terms used in the behavioral description are those of the problem domain. General-purpose programming languages, on the other hand, provide a primitive vocabulary that is significantly more independent of particular problem areas.

**Executability:** Informality aside, it is possible, and sometimes desirable, to describe behavior in terms of relationships between desired and achieved states of a process, rather than by rules that specify how to obtain the desired state. Conventionally specified programs must specify an algorithm for reaching the desired state.

**Efficiency:** Conventionally specified programs contain many details of operation beyond the desired input/output behavior. Among these are data representation, internal communication protocols, store-recompute decisions, etc., that affect a program's efficiency (utilization of computer resources and time). In general, these details should not appear in the description of input/output behavior.

When one writes a program in the conventional manner, one must formalize the behavioral specification, translate the terms of the problem domain into those of a general programming language, guarantee that the specified algorithms actually achieve the desired results, and make a myriad of decisions for the sake of an efficient implementation.

The ISI group has attempted to split the task of creating a program into two separate parts by designing a formal, complete specification language Balzer & Goldman, 1979 that allows behavioral specifications to be stated in terms specific to the problem domain while avoiding efficiency and representational concerns. This formal specification language acts as an interface between two projects that deal respectively with the first issue, translation from informal to formal specifications, and the last issue, optimization of a formal specification. The former project is the subject of this article, while the latter is described elsewhere Balzer, Goldman, & Wile (1976). The other issues, domain-specific vocabulary and executability, are addressed within the formal specification language.

The SAFE project has concentrated on only the first of the above specification issues: automatically producing a formal description from an informal description. It is not, therefore, a complete automatic programming system. The user of the SAFE system provides a behavioral description in a pre-parsed, limited subset of English, including terms from the problem area. SAFE then seeks to determine a way of resolving all ambiguities and of filling in all missing information in a way that satisfies SAFE's knowledge of the constraints that all programs must satisfy. The result is a complete, unambiguous, very high-level program specification in a language called AP2.

### Partial Descriptions

After studying many examples of program specifications written in English, the SAFE research group concluded that the main semantic difference between these specifications and their formal equivalent is that partial descriptions rather than complete descriptions were used. When such partial descriptions were used, it was because the missing information could be determined from the surrounding context. These partial descriptions possess some of the useful properties of natural language specifications that are lacking in formal languages. They focus both the writer's and reader's attention on the relevant issues and condense the specification. Furthermore, the extensive use of context almost totally eliminates bookkeeping operations from the natural language specification.

A partial description may have zero or one or more valid interpretations in a given context. If a single valid interpretation is found for a description, it is unambiguous in that context. Multiple valid interpretations indicate that there is not sufficient information from the context to complete the description and that interaction with the user is required to resolve the ambiguity. If a partial description possesses no valid interpretation, it is inconsistent within the existing context.

The SAFE system incorporates the most prevalent forms of partial descriptions found in natural language specifications:

**Partial sequencing:** Operations are not always described in the order of execution. While sequencing may sometimes be described explicitly, it is frequently implicit in the relationships between operations. Example: "Output generated while compiling is sent to a scratch file. This file must be opened in *write only* mode. (file should be opened before compiling commences)."

**Missing operands:** The operands of operations are frequently omitted because they are recoverable from context. Recovering them may involve considering

the operation's definition, other operands, and the procedural context. Example: "Do not mount a tape for a job unless the tape drive has been assigned (to that job)."

**Incomplete reference:** A description of an object(s) may match several objects whereas it was intended to refer to only one or possibly a subset of these objects. A complete description may be recovered by methods similar to that for missing operands. Example: "When the mail program starts, it opens the file named MESSAGE (in the directory of the job running the program)."

**Type coercions:** Often, people using natural language do not precisely specify the object intended, but instead specify an associated object or a subpart of an object. This situation can be recognized by a mismatch between the type of object actually specified and the type of object expected. Example: "Information messages are copied to each logged-in user (to the terminal of the job of each logged-in user)."

#### Operation of SAFE

The goal of SAFE is to complete the various partial descriptions in the user's specification of the desired program so as to produce a formal specification of the whole program. SAFE goes through several phases, but in all phases the system uses a variety of constraints to achieve the goal of completing the partial descriptions. These include built-in criteria that any formal program must meet (e.g., information must be produced before it is consumed), built-in heuristics that "sensible" programs will meet (e.g., the value of a conditional must depend on the program data), as well as any known or discovered constraints particular to a program's domain (e.g., each file in a directory has a distinct name). In fact, since programs are highly constrained objects, there are a large number of constraints that any "well-formed" program must satisfy, and this is one reason programs are hard to write.

In general, each partial description has several different possible completions. Based on the partial description and the context in which it occurs, an ordered set of possible completions is created for it. But one decision cannot be made in isolation from the others; decisions must be consistent with one another and the resulting program must make sense as a whole, satisfying all the criteria of well-formed programs.

The problem of finding viable completions for a collection of partial descriptions provides a classical backtracking situation, since there are many interrelated individual decisions that, in combination, can be either accepted or rejected on the basis of the constraints. SAFE utilizes the constraints so that early rejection possibilities can be realized.

The operation of SAFE consists of three sequential phases: the linguistics, planning, and meta-evaluation phases. The cumulative effect of these phases is to produce a formal specification that is composed of declarative and procedural portions. The declarative part, or domain model, specifies the types of objects manipulated by the process, the various ways they may relate to one another, the actions that may be performed on various object types, and other global regularities of the problem domain. The procedural portion specifies the controlled application of actions to objects.

The linguistic phase, using production rules, transforms the parse trees of the English specification into fragments that retain the semantic content while discarding the syntactic detail. The production rules capture many context-sensitive aspects of natural language such as various uses of the verb "be" and of quantifiers. The production rules may also add declarations to the domain model, with user approval, when this is required for interpretation of the input. This procedure is accomplished by distinguishing two sets of conditions on each rule: those relating to the linguistic form of the phrase being processed, and those relating a form to the domain model. If the linguistic form conditions are not satisfied (e.g., a clause using a transitive verb) but the domain model conditions are (e.g., the verb names an action in the problem domain that has operands of types compatible with the verb arguments), then the domain model conditions are assumed.

The planning phase determines the overall sequencing of the operations in the program. It also determines which fragments belong together and how they are to interact. It does this by using explicit sequencing information in the description, such as "A is executed immediately after B," "A is invoked whenever the condition C becomes true," as well as static flow constraints on well-formed processes such as:

Before information is consumed (used by one fragment), it must be produced (created by the same or another fragment).

Expected outputs of the whole program or of a subprogram must be produced somewhere within that program.

The results of each described operation must be used or referenced somewhere.

The final phase, meta-evaluation, uses dynamic constraints to help determine the proper completion of partial descriptions. Dynamic constraints are those that apply, or at least relate to, the program during execution. Examples of such constraints are:

It must be possible (in general) to execute both branches of a conditional statement (otherwise why would the user have specified a conditional).

The constraints of a domain must not be violated.

Since no actual input data is available for testing the execution of the program and since the program must be well-formed for all allowable inputs, inputs are represented symbolically. Instead of actual execution, the program is symbolically executed on the inputs, which provides a much stronger test of the constraints than would execution on any particular set of inputs. The result is a database of relationships between the symbolic values and, implicitly, a database of relationships between program variables that are bound to these values.

All decisions concerning the proper interpretation of partial descriptions that affect the computation to some point in the execution (but not beyond) must be made before these dynamic criteria can be tested at that point in the execution. Thus, decisions are made as they are needed by the computation of the program, and the symbolic state of the program is examined at each stage of the computation. This arrangement allows the dynamic state-of-computation criteria to be used to obtain early rejection of infeasible alternatives.

There is an additional point worth noting. Representing the complete state of a computation during symbolic execution is very difficult (e.g., it is quite hard to determine the state after execution of a loop or conditional statement) and more detailed than necessary for testing the constraints. Therefore, the SAFE system uses a weaker form of symbolic interpretation called Meta-Evaluation, which only partially determines the program's state as the computation proceeds (e.g., loops are executed only once for some "generic" element).

Notice that symbolic execution requires that the sequential relationships between the fragments be known; therefore the meta-evaluation phase must follow the planning phase.

Finally, the global referencing constraints (such as "The body of a procedure must make use of the procedure's parameters") test the overall use of names within the program and, thus, cannot be tested until all decisions have been made. These criteria can be tested only after the Meta-Evaluation is complete.

### Status

The prototype system has successfully handled the 75-200 word specifications of three quite distinct programs. In these cases the SAFE output of a completed specification, including domain structure definition, requires approximately two pages. One example concerned part of a system for scheduling transmissions in a communications network. Given a table (SOL) containing entries for various network subscribers and for various unassigned time slots (RATS), a schedule of absolute times when a particular subscriber could broadcast on the network was tabulated. The input specification to SAFE is:

```
((THE SOL)
 (IS SEARCHED)
 FOR
 (AN ENTRY FOR (THE SUBSCRIBER)))

(IF ((ONE)
 (IS FOUND))
 ((THE SUBSCRIBER'S (RELATIVE TRANSMISSION TIME))
 (IS COMPUTED) ACCORDING TO ("FORMULA-1"))))

((THE SUBSCRIBER'S (CLOCK TRANSMISSION TIME))
 (IS COMPUTED) ACCORDING-TO ("FORMULA-2"))))

WHEN ((THE TRANSMISSION TIME))
 (HAS BEEN COMPUTED))
 ((IT)
 (IS INSERTED)
 AS (THE (PRIMARY ENTRY))
 IN (A (TRANSMISSION SCHEDULE))))

FOR (EACH RATS ENTRY)
 (PERFORM)
 (: ((THE RATS'S (RELATIVE TRANSMISSION TIME))
 (IS COMPUTED) ACCORDING TO ("FORMULA-1"))))
```



```
((THE RATS'S (CLOCK TRANSMISSION TIME))
 (IS COMPUTED) ACCORDING TO ("FORMULA-2"))))

((THE RATS (TRANSMISSION TIMES))
 (ARE ENTERED)
 INTO (THE SCHEDULE))
```

Figure 1. Actual input for link scheduling example.

In formalizing this description, SAFE encountered and resolved the following characteristics of informal specifications:

number of missing operands	= 7
number of incomplete references	= 12
number of implicit type coercions	= 3
number of implicit sequencing decisions	= 4

Robustness of the system has been increased by processing a number of perturbations of each of the major examples. These have involved specifying the same process but varying the syntax and vocabulary used, the partial descriptions used, and the formal knowledge provided about the problem domain.

### Future Developments

The key technical restrictions of the prototype system appear to be (a) the sequential application of the three phases, which prohibits adequate interactions between the expertise embodied in each, and (b) the backtracking within the meta-evaluation phase, which corresponds to restarting the symbolic execution from an earlier point, which can lead to much unnecessary search. To correct these limitations, a reformulation of the system architecture within a framework derived from the HEARSAY II speech understanding system (see article *Speech.C*) is currently in progress. This framework consists of a number of cooperating experts interacting via a "blackboard" database.

Simultaneously, the system is being scaled up to handle larger practical specifications (approximately 20 pages). Later, the project will consider the formalization of incremental informal specifications so that it can also provide help during both specification formulation and maintenance activities.

### References

See Balzer, Goldman, & Wile (1976), Balzer, Goldman, & Wile (1977a), Balzer, Goldman, & Wile (1978), and Balzer & Goldman (1979).

### F. Programmer's Apprentice

The Programmer's Apprentice (PA) is an interactive system for assisting programmers with the task of programming. It is being designed and implemented at MIT by Charles Rich, Howard Shrobe, and Richard Waters. Currently, most, but not all, of the modules that comprise the PA system are running. It should be kept in mind that the scenario described here illustrates the projected operation of the system, not the present operation. The intent of the PA is that the programmer will do the hard parts of design and implementation, while the PA will act as a junior partner and critic, keeping track of details and assisting the programmer in the documentation, verification, debugging, and modification of his program. In order to cooperate with the programmer in this fashion, the PA must be able to "understand" what is going on. From the point of view of Artificial Intelligence, the central development of the Programmer's Apprentice project has been the design of a representation (called a "plan") for programs and for knowledge about programming that serves as the basis for this "understanding." Developing and reasoning about plans is the central activity of the PA.

The "plan" for a program represents the program as a network of operations interconnected by links explicitly representing data flow and control flow. The advantage of this aspect of the plan formalism is that it abstracts away from the specific syntactic constructs used by various programming languages in order to implement control flow and data flow. The most novel aspect of the plan formalism is that it goes beyond this level in order to create a vehicle for expressing the *logical* interrelationships in a program. First, a plan is not just a graph of primitive operations. Rather, it is a hierarchy of segments within segments, where each segment corresponds to a unit of behavior and has an input/output specification that describes features of this behavior. The plan specifies how each nonterminal segment is constructed out of the segments contained within it. This segmentation is important because it breaks the plan up into localities that can be understood in isolation from each other. Second, the behavior of a segment is related to the behavior of its subsegments. This interrelationship is represented by explicit dependency links that record the goal-subgoal and prerequisite relationships between the input-output specification for a segment and those for its subsegments. Taken together, the links summarize a proof of how these specifications for a segment follow from the specifications of its subsegments and from the way the subsegments are interconnected by control flow and data flow. A final aspect of the plan formalism is that there may be more than one plan for a given segment of a program, with each plan representing a different point of view on the segment. The data structures used by a program are represented by specifying their parts, properties, and the relationships between them in a method similar to data abstractions (Zilles, 1975; Liskov, 1977).

Knowledge about programming in general is also represented using plans and data structure descriptions. This knowledge is stored in the PA in a database of common algorithms and data structure implementations called the "plan library." The PA's "understanding" of a program is embodied in a hierarchical plan for it. In general, the subplan for each individual segment in terms of its subsegments will be an instance of some plan stored in the plan library. This structure gives the PA access to all of the information stored in the plan library about the particular subplan as soon as it can make a guess as to what the subplan is.

### A Scenario of Use of the Programmer's Apprentice

The following imagined conversation between a programmer and the PA is presented in order to illustrate the intended operation of the system. (Comments discussing the scenario are printed in *italics*.) The scenario illustrates the following four basic areas in which the PA can assist a programmer:

- (1) **Documentation:** One of the primary services the PA provides is automatic, permanent, and in-depth documentation of the program. The PA remembers not only explicit commentary supplied by the programmer with the code, but also a substantial body of derived information describing the logical structure underlying the program, such as the dependency relationships between parts of the program.
- (2) **Verification:** The development of a program is accompanied by the construction of a sequence of plans at various levels of abstraction. At each step, the PA attempts to verify that the current plan is both consistent and sufficient to accomplish the desired goal. As more information is specified, the PA's reasoning about these plans approaches a complete verification of the program.
- (3) **Debugging:** Any discrepancy between the PA's understanding of the programmer's intent and the actual operation of the program is reported to the programmer as a potential bug.
- (4) **Managing Modification:** Perhaps the most useful aspect of the PA is that it can help a programmer modify his program without introducing new bugs. Based on its knowledge of the logical relationships between parts of a program, the PA is able to determine what parts of a program can be affected by a proposed change, and how they can be affected. It can use this information to warn the programmer of impending difficulties.

The scenario traces the design, coding, and subsequent modification of a program that deletes an entry from a hash table. The scenario picks up in the middle of a session, at a point where the programmer has already made many design choices and conveyed them to the PA. In particular, he has stated the input-output specifications for the subroutine DELETE and chosen the basic data structures to be used. The hash table is implemented as an array of linked lists called "buckets." Each of the entries in a bucket is a dotted-pair whose CAR is the key for the entry.

**Programmer:** The plan for the routine DELETE has three main steps: The first one hashes the input key to get an index; the second one fetches the bucket corresponding to that index from the table; and the third splices out the entry in the bucket whose key part is equal to the input key.

*The only feature of this scenario that is not a current research goal of the Programmer's Apprentice project is the use of natural English dialogue. Although the system should be able to support a sophisticated natural language "front-end," no effort has been expended in this direction to date. As a result, the programmer would actually type in something like:*

```
"(subsegments-of-DELETE step1-step2-step3)
(specifications-for-step1 (inputs-key1) (outputs-index)
(assert-(=-index-(hash-key1)))) ..."
```

*The PA uses the programmer's outline of the design in order to build a plan representing the program's intended structure. References to basic programming techniques such as "splicing out" (deleting elements from a linked list) trigger the retrieval of plans from the plan library.*

PA: I can't verify that this plan satisfies the specifications for DELETE. In particular, when the input key is equal to the key part of the first entry in a bucket, I can't prove that the entry is no longer in the table after DELETE is executed. Note that in this situation, the bucket after splicing out entries has a different initial cell from the bucket before splicing out entries, and that therefore the table isn't changed.

*After constructing a plan, the PA uses its reasoning abilities in order to check for bugs. Above, the PA finds one and reports it to the programmer. In general, it is the responsibility of the programmer to supply corrections. Below, the programmer corrects the plan by issuing semantically based editing commands to the PA.*

Programmer: Modify the plan as follows. Add a final step that compares the bucket after splicing with the bucket before splicing and stores the new bucket in the table under the index if the two initial cells are not identical.

PA: OK, the plan now seems correct.

*The programmer then types in the code for DELETE. In cases where the programmer specifies the plan for a program as completely as he does here, he could ask the PA to write the code. It is expected that the programmer will actually only write code in situations where it is easier to write the code than to fully explain the plan.*

```
Programmer: (DEFUN DELETE (KEY) ; hash table delete routine.
(PROG (BKT1 BKT2)
(SETQ BKT1 (TBL (HASH KEY)) BKT2 (BKT-DELETE KEY BKT1))
(COND ((NOT (EQ BKT1 BKT2)) ; check if first entry in bucket deleted.
(STORE (TBL (HASH KEY)) BKT1))))))
```

*The PA analyzes the code written by the programmer in order to construct a plan for it. Comparing this plan with the plan directly described by the programmer, the PA discovers an implementation bug: In the last line of code the user probably meant to store into the hash table not the first bucket, BKT1, but the second bucket, BKT2.*

PA: This code is not consistent with your plan. When the first entry is deleted, the code stores the bucket before splicing, rather than the bucket after splicing. Did you mean

"(STORE-(TBL-(HASH-KEY))-BKT2)"?

Programmer: Yes, fix the code. Also, use the standard list deletion algorithm to code BKT-DELETE so that it splices out of the bucket all entries whose key equals the indicated value.

PA: (DEFUN BKT-DELETE (KEY BUCKET) ; list deletion routine coded by the PA.  
 (PROG (CURRENT PREVIOUS)  
 (SETQ CURRENT BUCKET PREVIOUS NIL)  
 LP (COND ((NULL CURRENT) (RETURN BUCKET))  
 ((EQUAL (CAAR CURRENT) KEY) ; if keypart equal, splice out entry.  
 (COND (PREVIOUS (RPLACD PREVIOUS (CDR CURRENT)))  
 (T (SETQ BUCKET (CDR BUCKET)))))) ;special if first element deleted.  
 (T (SETQ PREVIOUS CURRENT)))  
 (SETQ CURRENT (CDR CURRENT))  
 (GO LP)))

*The PA is able to code a program like this itself, even though the programmer did not specify a plan for it, because it is an implementation of a standard algorithm that corresponds to a plan in the plan library. Code written by the PA can be freely mixed with code written by the programmer and each can edit the other's work.*

*Some time later it is necessary to modify the representation of the buckets. The PA helps the (perhaps new) programmer maintain consistency.*

Programmer: I want to make a change in the representation of buckets in the hashing programs. A bucket now has two parts: a count field and a list of entries as before. The count field will contain an integer equal to the length of the list of entries. Let a bucket be a dotted-pair whose CAR is the count and whose CDR is the list of entries.

*When the PA inspects the logical dependencies in the plans, it discovers that a type requirement has been violated.*

PA: The proposed change will affect DELETE as follows: After the bucket is fetched, it is passed to BKT-DELETE which does the actual deletion. However, BKT-DELETE requires a list of entries while in this new implementation, a bucket is no longer just a list of entries.

Programmer: Explain the plan for BKT-DELETE please.

PA: BKT-DELETE is implemented following the standard list deletion plan. The inputs are a key and a list of entries. The plan is a search loop using two pointers: a pointer to the current entry, which is initialized to the input list, and a trailing pointer, which is initialized to NIL. On each iteration the loop tests the key of the current entry. If it is equal to the input key, the element is spliced out of the list by RPLACDing the trailing pointer.

*Acting in its role as documentation, the PA describes the structure of BKT-DELETE by reporting out the structure of its plan. The programmer then decides how to modify BKT-DELETE.*

Programmer: Change the specifications of BKT-DELETE so that it expects as input a bucket of the new type, rather than a list of entries. Also change the initialization of the pointers to read:  
"(SETQ-PREVIOUS-BUCKET- CURRENT-(CDR-BUCKET))".

*The PA reanalyzes the program in order to verify that this change fixes the problem without causing any other problems to arise. This analysis reveals the presence of inaccessible code which the PA reports to the programmer.*

PA: OK this fixes the problem. However, now the special case tests in BKT-DELETE and DELETE are not necessary since PREVIOUS is never NIL and therefore elements are always deleted by side-effect.

*... and so the session continues with the PA looking over the programmer's shoulder.*

### Operation of the System

The design of the PA is based on four modules, a surface analyzer, a recognizer, an interactive module, and a deductive module; and two data bases, the plan library and a scratch pad called the "design notebook." Only the first three modules have been implemented so far. As described above, the plan library contains the PA's knowledge of programming in general. The design notebook contains the PA's evolving knowledge of the particular programs being worked on and serves as the communication center for the system as a whole. The modules communicate with one another solely by making assertions in the design notebook. Each module has predefined trigger patterns which cause it to perform specific tasks (such as making a deduction or querying the user) whenever appropriate assertions appear in the notebook. Every assertion added to the notebook is also accompanied by a justification of its presence. These justifications make it possible for the PA to account for its actions.

The surface analyzer is used to construct simple surface plans for sections of code written by the programmer. It is the only module whose implementation depends on the particular programming language being used. To date, surface analyzers have been implemented for both LISP and FORTRAN. The recognition module takes over where the surface analyzer leaves off in order to construct a detailed plan for a piece of code. It first breaks up the surface plan by identifying weakly interacting subsegments that can be further analyzed in isolation from each other. It then compares these subsegments with the plans in the library in order to determine more detailed plans for the program.

The interactive module is the communication link between the PA and the programmer. It converts the programmer's input (which can consist of code, direct specification of a plan, or various requests) into assertions in the design notebook and decides what to say to the programmer based on the information currently in the notebook. The deductive module

operates in the background in cooperation with all of the other modules. It performs the deductions necessary to verify a proposed match between a program and a plan, to detect bugs in a plan, and to determine the ramifications of a proposed modification to a program or plan.

At a given moment, the design notebook holds the sum total of what the PA knows about the program being worked on. This information triggers additional activity by the modules. If the recognizer and deductive modules are strong enough and the program is simple enough, this process will culminate in a complete understanding and verification of the program. However, typically, this will not be the case, and some questions (such as the exact plan for a segment or the correctness of a specification) will remain unresolved in the notebook. The flexible architecture chosen for the PA makes it possible for the PA to exhibit useful partial performance in this situation. It is able to ignore what it doesn't understand and work constructively with what it does understand. The programmer can be called upon to fill in the gaps.

#### **Current Status of the Programmer's Apprentice**

Rich and Shrobe (1976) laid out the basic idea of a plan and the initial design of the PA. Since that time Rich, Shrobe, and Waters have been working together on further aspects of the theory along with design and implementation of the PA.

Rich's work (forthcoming) centers on the plan library and the recognition process. He is using the plan representation in order to codify a large body of common programming strategies in the domain of non numerical programming. He is also designing a recognition module that will be able to identify instances of plans in the library as they occur in combination in a programmer's program.

Shrobe (1978) has implemented a prototype deductive module that can reason about programs represented by plans. An important aspect of its operation is that it maintains a record of the dependency relationships embodied in its deductions. In doing this it builds up some of the logical structure that is a vital part of a plan for a program. He is currently designing an improved version of this deductive module.

Waters (1976, 1978) has implemented a system that can analyze the code for a program and produce the basic structure of a plan for the entire program. The system corresponds to the surface analysis module and the initial phase of the recognition process. The basic idea behind Waters' work is that plans for typical programs are built up in a small number of stereotyped ways and that features in the code for a program can be used to determine how the plan for the program should be built up.

The goal for the immediate future is to construct a prototype system that can exhibit the kind of behavior shown in the scenario. To do this, an interactive module must be built, and the other modules must be connected together into an integrated system. Looking further ahead, additional modules (such as a simple program synthesis module, and one dealing with efficiency issues) will be added to the PA, and the existing ones will be strengthened so that the PA can assume an even larger part of the programming process.

**References**

See Liskov et al. (1977), Rich & Shrobe (1976), Rich & Shrobe (1978), Rich (1979), Shrobe (1978), Waters (1976), Waters (1978), Waters (1979), and Zilles (1975).



## G. PECOS

Developed in 1976 by David Barstow (Barstow, 1976), the automatic programming system PECOS serves as the coding expert of Stanford's PSI project (see article D2 and Barstow, 1979). Now the PSI project is also being developed at Systems Control Inc.. Though PECOS can act in conjunction with the PSI system, it can also stand on its own and interact directly with the user. The original problem area of PECOS was symbolic programming, which includes simple list processing, sorting, database retrieval, and concept formation. This domain has recently been extended to graph theory and simple number theory. Programs are specified in terms of very high-level constructs including *data structures*, like collections or mappings, and *operations*, like testing for membership in a collection or computing the inverse image of an object under a mapping. Knowledge about programming in the problem area has been codified (i.e., made explicit and put into machine useable form) primarily in the form of transformation rules, and these have been entered into PECOS's knowledge base. Most of the rules describe how constructs and operations can be represented or implemented in terms of other constructs and operations that are closer to, or actually in, the target language LISP (actually a subset of INTERLISP Teitelman et al. (1978)). These rules can identify design decisions and can also serve as limited explanations.

The operation of the system proceeds by the repeated selection and application of the transformation rules in the knowledge base to parts of the program. Also referred to as *gradual refinement*, this transformation process reduces the high-level specification to an implementation fully within the target language. Each application of a rule is said to produce a partial implementation or *refinement* of the program, and the transformation rules are called *refinement* rules.

### Conflict Resolution

At some points during the transformation process, a conflict may arise because several rules apply to the same part of the program. The handling of this situation is important: the application of the several rules ultimately results in different target language implementations that often vary significantly in terms of efficiency. There are three ways to handle this situation.

- (1) If PECOS is interacting directly with the user, the user may select which rule should be applied (and thus which implementation will be constructed).
- (2) For the convenience of the user, PECOS can choose one of the applicable rules, using about a dozen heuristics it has to pick the rule that leads to the *more* efficient implementation. These heuristics handle about two-thirds of the choices that typically arise.
- (3) When no heuristic applies and the user is uncertain about which rule is "best" for his or her purposes, PECOS can apply each in parallel, constructing a separate implementation for each rule applied.

When PECOS functions as the Coding Expert of the PSI program synthesis system Green, 1976b, D2, choices between rules are made by an automated Efficiency Expert known as LIBRA (see article D9, Kant (1977)), which incorporates more sophisticated analytic

techniques than the simple heuristics used by PECOS. The capability of developing different implementations in parallel is used extensively in the interactions between PECOS and LIBRA Barstow & Kant, 1977.

### PECOS's Knowledge Base

PECOS's knowledge base consists of about 400 rules dealing with a variety of symbolic programming concepts. The most abstract concepts are those of the specification language (e.g., collection, inverse image, enumerating the objects in a collection, etc.). The implementation techniques covered by the rules include the representation of collections as linked lists, arrays (both ordered and unordered), and Boolean mappings, and the representation of mappings as tables, sets of pairs, property list markings, and inverted mappings (indexed by range element). As a natural by-product, these rules also cover sorting within a transfer paradigm that includes simpler sorts such as insertion and selection. While some of the rules are specific to LISP, about three-fourths of the rules are independent of LISP or any other target language.

Internally, PECOS's rules are represented as condition-action pairs. The conditions are particular configurations of abstract operations and data structures that are matched against parts of the developing program. Where the match is successful, the actions replace parts of the abstract concepts with refinements of those parts.

In the system of refinement rules, intermediate-level abstractions play a major role. One benefit of such intermediate-level concepts is a certain economy of knowledge. Consider, for example, the construct of a *sequential collection*: a linearly ordered group of locations in which the elements of a collection can be stored. Since there is no constraint on how the linear ordering is implemented, the construct can be seen as an abstraction (or generalization) of both linked lists and arrays. Much of what programmers know about linked lists is in common to what they know about arrays, and hence can be represented as one rule set about sequential collections, rather than as two, one about linked lists, and one about arrays. Another benefit of these intermediate-level concepts is that the process of choosing between alternative (valid) rules is facilitated: Attention can be focused on the essential aspects of a choice while ignoring irrelevant details.

### Rules about Programming Knowledge

Most currently available sources of programming knowledge (e.g., books and articles) lack the precision required for effective use by a machine. The descriptions are often informal, with details omitted and assumptions unstated. Before this programming knowledge can be made available to machines, it must be made more precise; the assumptions must be made explicit; and the details must be filled in.

PECOS's rules provide much of this precision for the domain of elementary symbolic programming. For example, consider the following rule (an English paraphrase of PECOS's internal representation):

*A collection may be represented as a mapping of objects to Boolean values; the default range object is FALSE.*

Most programmers know this fact: that a collection may be represented by its *characteristic function*. Without knowing this rule, or something similar, it is almost impossible to understand why a bitstring can be used to represent a set (or, for that matter, why property list markings work). Yet this rule is generally left unstated in discussions of bitstring representations. As another example, consider the following rule:

*An association table whose keys are integers from a fixed range may be represented as an array subregion.*

The fact that an array is simply a way to represent a mapping of integers to arbitrary values is well known and usually stated explicitly. The detail that the integers must be from a fixed range is usually not stated. Note that if the integers are not from a fixed range, then an array is the wrong representation and something like a hash table should be used.

PECOS's rules also identify particular design decisions involved in programming. For example, one of the crucial decisions in building an enumerator of the objects in a sequential collection is selecting the order in which they should be enumerated. This decision is often made only implicitly. For example, the use of the LISP function MAPC to enumerate the objects in a list assumes implicitly that the stored (or "natural") order is the right order in which to enumerate them. While this is often correct, there are times when some other order is desired. For example, the selector of a selection sort involves enumerating the objects according to a particular ordering relation. A second major decision in building an enumerator involves selecting a way to save the state of the computation between calls to the enumerator. The use of a location (e.g., index or list cell) to specify the current state is based on knowing the following rule:

*If the enumeration order is the same as the stored order, the state of an enumeration may be represented as a location in the sequential collection.*

Were the enumeration order different from the stored order (as in a selection sort), then some other state-saving scheme would be needed, such as deleting the objects or marking them in some fashion.

Another interesting aspect of PECOS's rules is that they have a certain kind of explanatory power. Consider, for example, a well-known trick for computing the intersection of two linked lists of atoms in linear time: Map down the first list and put a special mark on the property list of each atom; then map down the second list collecting only those atoms whose property lists contain the special mark. This technique can be understood on the basis of the following four of PECOS's rules (in addition to the rules about representing collections as linked lists):

*A collection may be represented as a mapping of objects to Boolean values; the default range object is FALSE.*

*A mapping whose domain consists of atoms may be represented using property list markings.*

*The intersection of two collections may be implemented by enumerating the objects in one, and while enumerating them, collecting those that are members of the other.*

*If a collection is input, its representation may be converted into any other representation before further processing.*

Given these rules, the trick works by first converting the representation of one collection from a linked list to property list markings with Boolean values, and then computing the intersection in the standard way, except that a membership test for property list markings involves a call to GETPROP rather than a scan down a linked list.

### Status

PECOS is able to implement abstract algorithms (i.e., a very high-level specification) in a variety of domains, including elementary symbolic programming (simple classification and concept formation algorithms), sorting (several versions of selection and insertion sort), graph theory (a reachability algorithm), and even simple number theory (a prime number algorithm). In each case, PECOS's knowledge about different implementation techniques enabled the construction of a variety of alternative implementations, often with significantly different efficiency characteristics.

PECOS's success demonstrates the viability of the knowledge-based approach to automatic programming. In order to develop this approach further, two research directions seem particularly useful.

First, programming knowledge for other domains must be codified. In the process, rules developed for one domain may be found to be useful in other domains. With the hope of verifying the wider utility of PECOS's rules about collections and mappings, Yale's Knowledge-based Automatic Programming Project Barstow, 1978 is currently codifying the programming knowledge needed for elementary graph algorithms.

As an example, consider the common technique of representing a graph as an adjacency matrix. In order to construct such a representation, only one rule about graphs need be known:

*A graph may be represented as a pair of sets: a set of vertices (whose elements are primitive objects) and a set of edges (whose elements are pairs of vertices).*

The rest of the necessary knowledge is concerned with sets and mappings and is independent of its application to graphs. For example, in order to derive the bounds on the matrix, one need only know that primitive objects may be represented as integers, that a set of otherwise unconstrained integers may be represented as a sequence of consecutive integers, and that a sequence of consecutive integers may be represented as lower and upper bounds. To derive the representation of the matrix itself, one need only know PECOS's rules about Boolean mappings and association tables, plus the fact that a table whose keys are pairs of integers in fixed ranges may be represented as a two-dimensional matrix.

Second, different types of programming knowledge need to be codified. Two types seem particularly important: efficiency knowledge and strategic knowledge. LIBRA (article D9), which acts together with PECOS in PSI's synthesis phase, embodies a large amount of efficiency knowledge; but much remains to be done. Very little work on the use of general strategies (e.g., divide and conquer) in program synthesis has been done. The latter seems

an especially important direction, since such strategies seem to play a major role in human programming.

#### References

See Barstow & Kant (1977), Barstow (1978), Barstow (1979), Green (1976b), Kant (1977), and Teitelman et al. (1978).

## H. DEDALUS

DEDALUS, the DEDuctive ALgorithm Ur-Synthesizer, accepts an unambiguous, logically complete, very high-level specification of a desired program and through repeated application of transformation rules seeks to reduce it to an implementation within a simple LISP-like target language. This target language implementation is guaranteed to be correct (i.e., logically equivalent to the high-level specification) and to terminate. The knowledge that ultimately relates the constructs of the specification language to those in the target language is expressed in the transformation rules. But of special importance are certain rules that express general programming principles that are independent of the particular specification language and target language. These rules, which have constituted a major component of the DEDALUS effort, form conditional statements and recursive and nonrecursive procedures; they also generalize procedures, construct well-founded orderings to guarantee the termination of recursive calls, and write code that simultaneously achieves two or more goals. These general programming principles are described in detail in a subsequent section, with examples illustrating their application. As pointed out in the STATUS section, some of the principles are fairly well understood, while others require further study. Not all the principles are implemented in the current DEDALUS system.

The DEDALUS specification language can contain constructs that are close to how the user actually thinks about the problem. Thus, the DEDLAUS specification of the program `lessall(x l)`, which tests whether a number  $x$  is less than every element of a list  $l$  of numbers, and the program `gcd(x y)`, which computes the greatest common divisor of two nonnegative integers  $x$  and  $y$ , are specified as follows:

```
lessall(x l) <= compute x < all(l)
    where x is a number and l is a list of numbers,
gcd(x y) <= compute max (z:z|x and z|y)
    where x and y are nonnegative nonzero integers .
```

The `all` construct in  $P(\text{all}(l))$ , indicating that the condition  $P$  holds for all elements of the list  $l$ , and the set constructor  $\{u: P(u)\}$ , indicating the set of elements for which  $P$  is true, are constructs that, through the repeated application of transformation rules will eventually be converted into target language code that, for the particular program, is logically equivalent to the original specification. The specification language is not fixed: New constructs can be introduced by modifying or adding transformation rules.

The operation of DEDALUS consists of the repeated application of transformations to expressions in order to produce expressions that are closer to, or within, the target language. In DEDALUS, the expressions that occur during the transformation process specify not only programs; they can also specify conditions to be proved, as well as conditions to be made true. All these expressions are treated as goals to be achieved: For an expression that specifies a program, the goal is to convert that program into a target language implementation; for an expression that is a condition to be proved, the goal is to convert it to the logical constant `true`; for an expression that is a condition to be made true, the goal is to construct a program that will make that condition true.

Transforming a subexpression (of an expression) into another subexpression, require rules of the form

$$t \Rightarrow t' \text{ if } P ,$$

the condition  $P$  being optional. This rule indicates that the subexpression  $t$  can be replaced by  $t'$ . If  $P$  is present, then the rule can only be applied provided that the system first prove that  $P$  is true; which is to say, before the rule can be applied, the system must succeed in achieving the subgoal

Goal: prove  $P$  .

For example, consider

$$P(\text{all}(l)) \Rightarrow P(\text{head}(l)) \text{ and } P(\text{all}(\text{tail}(l))) \text{ if not empty } (l) ,$$

which expresses the fact that a property  $P$  holds for every element of a nonempty list  $l$  if it holds for the first element  $\text{head}(l)$  and for every element of the list  $\text{tail}(l)$  of the other elements. Before the system can apply this rule to some part of an expression, it would have to succeed in proving that  $l$  is not empty.

The application of transformation rules results in a tree of goals and subgoals. Initially the top-level goals of this tree are established by program specifications. Thus, the common form of program specification

$$f(x) \Leftarrow \begin{array}{l} \text{compute } P(x) \\ \text{where } Q(x) , \end{array}$$

establishes its output description as the top-level goal

Goal: compute  $P(x)$  ,

and in trying to achieve this goal, the system assumes the truth of  $Q(x)$ . If the top-level goals of trees are established by program specifications, most goals are established as the result of transformations. Thus, by applying the transformation rule

$$u|v \text{ and } u|w \Rightarrow u|v \text{ and } u|w-v$$

to the top-level goal of the gcd program

Goal 1: compute  $\max(z:z|x \text{ and } z|y)$ ,

the system establishes

Goal 2: compute  $\max(z:z|x \text{ and } z|y-x)$

as a subgoal. Such transformations express knowledge about specific constructs. In the DEDALUS system there is also knowledge of a more general sort.

### General Programming Principles

This section describes five general programming principles and presents several

examples to illustrate their application. The principles express knowledge about how to form conditionals and procedures (recursive and nonrecursive), how to replace two or more procedures by a generalized procedure, and how to achieve simultaneous goals. As explained in the STATUS section, the current implementation of DEDALUS does not incorporate the generalization of procedures or the achievement of simultaneous goals.

**Conditional formation.** Many of the transformation rules impose some condition  $P$  (e.g.,  $l$  is nonempty,  $x$  is nonnegative) that must be satisfied for the rule to be applied. Suppose that in attempting to apply a particular rule, the system failed to prove or disprove the condition  $P$ , where  $P$  is expressed entirely in terms of the primitive constructs of the target language; in such a situation, the conditional formation rule is invoked. This rule allows the introduction of case analysis to consider separately the cases in which  $P$  is true and in which  $P$  is false. Suppose the result is both a program segment  $S1$  that achieves the goal under the assumption that  $P$  is true and another program segment  $S2$  that achieves the goal under the assumption that  $P$  is false. The conditional formation principle puts these two program segments together into a conditional expression

$$\text{if } P \text{ then } S1 \text{ else } S2 ,$$

which solves the problem regardless of whether  $P$  is true or false. During the generation of  $S2$ , the system could discover that a conditional expression was unnecessary: The generation of  $S2$  may not have required the assumption that  $P$  was false. In such a case, the program constructed would be simply  $S2$ .

**Recursion formation.** Suppose, in constructing a program with specifications

$$f(x) \Leftarrow \begin{array}{l} \text{compute } P(x) \\ \text{where } Q(x) , \end{array}$$

the system encounters a subgoal

$$\text{compute } P(t) ,$$

which is an instance of the output specification,  $\text{compute } P(x)$ . Because the program  $f(x)$  is intended to compute  $P(x)$  for any  $x$  satisfying its input specification  $Q(x)$ , the recursion formation rule proposes achieving the subgoal by computing  $P(t)$  with a recursive call  $f(t)$ . For this step to be valid, it must ensure that the input condition  $Q(t)$  holds when the proposed recursive call is executed. To ensure that the new recursive call will not cause the program to loop indefinitely, the rule must also establish a termination condition, showing that the argument  $t$  is strictly less than the input  $x$  in some well-founded ordering. (A well-founded ordering is an ordering in which no infinite strictly decreasing sequences can exist.) This condition precludes the possibility that an infinite sequence of recursive calls occur during the execution of the program.

**Example: lessall.** The DEDALUS system derived the program  $\text{lessall}(x \ l)$ , which tests whether a given number  $x$  is less than every element of a give list  $l$  of numbers. The specifications for this program are

$$\text{lessall}(x \ l) \Leftarrow \begin{array}{l} \text{compute } x < \text{all } (l) \\ \text{where } x \text{ is a number and } l \text{ is a list of numbers} . \end{array}$$



In deriving this program, the system develops a subgoal

compute  $x < \text{all}(\text{tail}(l))$  ,

in the case that  $l$  is nonempty. This subgoal is an instance of the output specification of the original specification, with the input  $l$  replaced by  $\text{tail}(l)$ ; therefore, the recursion formation principle proposes that the subgoal be achieved by introducing a recursive call  $\text{lessall}(x \text{ tail}(l))$ . To ensure that this step is valid, the rule establishes an input condition that

$x$  is a number and  $\text{tail}(l)$  is a list of numbers ,

and a termination condition that the argument pair  $(x \text{ tail}(l))$  is less than the input pair  $(x \ l)$  in some well-founded ordering. This termination condition holds because  $\text{tail}(l)$  is a proper sublist of  $l$ .

As the final program the system obtains

$\text{lessall}(x \ l) \Leftarrow$  if  $\text{empty}(l)$  then true  
 else  $x < \text{head}(l)$  and  $\text{lessall}(x \ \text{tail}(l))$  .

**Procedure formation.** Suppose that while developing a tree for a specification of the form

$f(x) \Leftarrow$  compute  $P(x)$   
 where  $Q(x)$  ,

the system encounters a subgoal

Goal B: compute  $R(t)$  ,

which is an instance not of the output specification compute  $P(x)$  but of some previously generated subgoal

Goal A: compute  $R(x)$  .

Then the procedure formation principle introduces a new procedure,  $g(x)$ , whose output specification is

$g(x) \Leftarrow$  compute  $R(x)$  .

In this way, both Goals A and B can be achieved by calls  $g(x)$  and  $g(t)$  to a single procedure. In the case where Goal B has been derived from Goal A, the call to  $g(t)$  will be a recursive call; otherwise, both calls will be simple procedure calls.

**Example: cart.** The specification of the program  $\text{cart}(s \ t)$  to compute the Cartesian product of two sets,  $s$  and  $t$ , is

$\text{cart}(s \ t) \Leftarrow$  compute  $((x \ y) : x \in s \text{ and } y \in t)$   
 where  $s$  and  $t$  are finite sets .

While deriving the tree for the program, the system obtains a subgoal

Goal A: compute  $((x\ y) : x=\text{head}(s) \text{ and } y\text{t})$  ,

given that  $s$  is nonempty. Developing Goal A further, the system derives

Goal B: compute  $((x\ y) : x=\text{head}(s) \text{ and } y\text{tail}(t))$  ,

given that  $t$  is nonempty. Goal B is an instance of Goal A; therefore, the procedure formation rule proposes introducing a new procedure  $\text{carhead}(s\ t)$  whose output specification is

$\text{carhead}(s\ t) \Leftarrow \text{compute } ((x\ y) : x=\text{head}(s) \text{ and } y\text{t})$

so that Goal A can be achieved with a procedure call  $\text{carhead}(s\ t)$ , and Goal B, with a (recursive) call  $\text{carhead}(s\ \text{tail}(t))$ .

Constructing the  $\text{carhead}$  procedure by the techniques already described, the final system of programs becomes,

$\text{cart}(s\ t) \Leftarrow \text{if empty}(s) \text{ then } ()$   
 $\text{else union}(\text{carhead}(s\ t)\ \text{cart}(\text{tail}(s)\ t))$  ,

$\text{carhead}(s\ t) \Leftarrow \text{if empty}(t) \text{ then } ()$   
 $\text{else union}(\text{head}(s)\ \text{head}(t))$   
 $\text{carhead}(s\ \text{tail}(t))$  .

**Generalization.** Suppose, in deriving a program, that we obtain two subgoals

Goal A: compute  $R(a(x))$

Goal B: compute  $R(b(x))$  ,

neither of which is an instance of the other, but both of which are instances of the more general expression

compute  $R(y)$  .

In such a case the extended procedure formation rule proposes the introduction of the new procedure, whose output specification is

$g(y) \Leftarrow \text{compute } R(y)$  .

Thus, Goal A and Goal B can be achieved by procedure calls to  $g(a(x))$  and  $g(b(x))$ , respectively.

**Example: reverse.** In constructing a program  $\text{reverse}(l)$ , to reverse a list  $l$ , we first derive two subgoals:

Goal A: compute  $\text{append}(\text{reverse}(\text{tail}(l)))$

```
cons(head(l)nil))
```

```
Goal B: compute append(reverse(tail (tail(l)))
cons(head(tail(l))
cons(head(l) nil))) .
```

skip

Each is an instance of the more general expression

```
compute append(reverse(tail(l))
cons(head(l) m)) ;
```

therefore, the extended procedure formation rule proposes introducing a new procedure `reversegen(l m)`, whose output specification is the more general expression:

```
reversegen(l m) <= compute append(reverse(tail(l))
cons(head(l) m)) .
```

Although this procedure, which reverses a nonempty list `l` and appends the result to `m`, is a more general problem than the original `reverse` program, it turns out that `reversegen` is actually easier to construct. The final system of programs obtained is

```
reverse(l) <= if empty(l) then nil
else reversegen(l nil)
```

```
reversegen(l m) <= if empty(tail(l)) then cons(head(l) m)
else reversegen(tail(l) cons(head(l) m)) .
```

**Simultaneous goals.** In order to deal with operations that produce side-effects such as modifying the structure of data objects (e.g., assignment statements), DEDALUS introduces constructs such as `achieve P`, to denote a program intended to make the condition `P` true.

In constructing a program to achieve two conditions, `P1` and `P2`, it is not sufficient to decompose the problem by constructing two independent programs to achieve `P1` and `P2`, respectively. The concatenation of the two programs might not achieve both conditions because the program that achieves `P2` may in the process make `P1` false, and vice versa.

For example, suppose a program is desired to sort the values of three variables `x`, `y`, and `z`; in other words, to permute the values of the variables to achieve the two conditions `xzy` and `yzz` simultaneously. Assume the given primitive instruction `sort2(u v)`, which sorts the values of its input variables `u` and `v`. The concatenation

```
sort2(x y)
sort2(y z)
```

of these two segments will not achieve both conditions simultaneously; the second segment `sort2(y z)` may, by sorting `y` and `z`, make the first condition `xzy` false.

The simultaneous goal principle, which was introduced to circumvent such difficulties, states that to satisfy a goal of form

achieve P1 and P2 ,

first construct a program F to achieve P1, then modify F to achieve P2 while protecting P1 at the end of F. A special "protection mechanism" (cf. (Sussman, 1975)) ensures that no modification is permitted that destroys the truth of the protected condition P1 at the end of the program.

Example: sort. To apply this principle to the goal

achieve  $x < y$  and  $y < z$

In the sorting problem, a system would first achieve  $x < y$ , by using the segment  $\text{sort2}(x y)$ . This program would then be modified to achieve the second condition  $y \leq z$ . But adding  $\text{sort2}(y z)$  at the end of the program will not work because it destroys the truth of the protected condition  $x \leq y$ .

However, in general, a goal may be achieved by inserting modifications at any point in the program, not merely at the end. Introducing the two instructions

if  $y < x$  then  $\text{sort2}(x y)$

if  $x < y$  then  $\text{sort2}(y z)$

at the beginning of the program segment would simultaneously achieve both conditions  $x < y$  and  $y < z$ . The resulting program would be

if  $y < x$  then  $\text{sort2}(x z)$

if  $x < y$  then  $\text{sort2}(y z)$

$\text{sort2}(x y)$  .

### Status

Currently, the DEDALUS implementation incorporates the principles of conditional formation, recursion formation (including the termination proofs), and procedure formation, but it does not include generalization or the formation of structure-changing programs. The techniques for deriving straight-line structure-changing programs were implemented in a separate system (see (Waldinger, 1977)).

Conditional formation and recursion formation are well understood. The method for proving termination of ordinary recursive calls does not always extend to the multiple-procedure case. The generalization mechanism and the extended procedure formation principle are just beginning to be formulated.

The derivation of straight-line programs with simple side-effects is fairly well

understood, but much work needs to be done on the derivation of structure-changing programs with conditional expressions and loops, as well as on the derivation of programs that alter list structures and other complex data objects.

The DEDALUS system is implemented in QLISP (Wilber, 1976), an extension of INTERLISP (Teitelman et al., 1978) that includes pattern-matching and backtracking facilities. The full power of the QLISP language is available in expressing each rule since the rules are represented as QLISP programs in a fairly direct manner.

To date, these are some of the representative samples of the programs constructed by the current DEDALUS system:

#### Numerical Programs:

- the subtractive gcd algorithm,
- the Euclidean gcd algorithm,
- the binary gcd algorithm, and
- the remainder of dividing two integers.

#### List Programs:

- finding the maximum element of a list,
- testing if a list is sorted,
- testing if a number is less than every element of a list of numbers (lessall), and
- testing if every element of one list of numbers is less than every element of another.

#### Set Programs:

- computing the union or intersection of two sets,
- testing if an element belongs to a set,
- testing if one set is a subset of another, and
- computing the cartesian product of two sets (cart).

#### References

See Balzer (1972), Balzer, Goldman, & Wile (1977b), Boyer & Moore (1975), Buchanan & Luckham (1974), Burstall & Darlington (1977), Dijkstra (1975), Dijkstra (1976), Green (1976b), Guttag, Horowitz, & Musser (1976), Heidorn (1976), Manna & Waldinger (1978), Siklossy (1974), Sussman (1975), Teitelman et al. (1978), Waldinger (1977), Warren (1974), Warren (1976), and Wilber (1976).

## I. PROSYSTEM I

PROSYSTEM I, an automatic programming system designed by William Martin, Gregory Ruth, Robert Baron, Matthew Morgenstern, and others of the MIT Laboratory for Computer Science, is part of a larger research project aimed at modeling, understanding, and automating the writing of a data-processing system. Hereafter the data-processing system is referred to as a *data-processing program*, in accord with this chapter's terminology, which refers to the output of an automatic programming system as a program. A model of the larger research project was developed that consists of five phases. The successive phases can be viewed as a series of transformations of the descriptions of the target program, beginning with a global conceptual description of the problem at hand and progressing, through increasing specificity, toward a detailed machine-level solution. The aim of the project is to develop stages of an automatic programming system where each corresponds to one of the five phases of the model and each embodies the particular knowledge and expertise for that phase.

**Phase 1: Problem Definition**--The specification of the data-processing program is expressed in domain-dependent terms in English.

**Phase 2: Specification Analysis and System Formulation**--The specification in Phase 1 is viewed as a data-processing problem. This problem is solved, yielding a data-processing formulation of the desired program.

**Phase 3: Implementation**--The procedural steps, data representation, and organization of the target are determined by intelligent selection from, and adaptation of, a set of standard implementation possibilities.

**Phase 4: Code Generation**--The implementation of Phase 3 is transformed into code in some high-level language (e.g., PL/I).

**Phase 5: Compilation and Loading**--The high-level code is transformed into a form that can be "understood" and executed by the target computer.

The first two phases involve such AI areas as natural language comprehension, program model formation, and problem solving. Since these areas are still in the process of evolution, the development of the first two phases has been deferred. At present, PROSYSTEM I is limited to the automation of phases 3 and 4 since it was felt that these phases were much more amenable to solution. Thus, the current PROSYSTEM I accepts a specification in terms of abstract relations (in a very high-level language called SSL), and then designs an optimized data-processing program and generates code for an efficient implementation. In automatic programming it is usually impossible for a system to carry out a search for the absolutely optimal implementation; instead, a system works at optimizing a program only to a degree.

The particular problem area of PROSYSTEM I is that of I/O intensive (file manipulation and updating), batch-oriented, data-processing programs. Included in this area are programs for inventory control, payroll, and other record-keeping systems.

The specification method uses a description of the desired data-processing program in the SSL language. An SSL specification consists of a data and a computation division. The

data division gives the names of data sets (conceptual aggregations or groupings of data), their keys, and their period of updating. The computation division specifies for each computed file the calculations to be performed when it is computed. Figure 1 illustrates an SSL specification of a data-processing program for a warehouse inventory. In the proposed problem, the warehouse stocks a number of different kinds of items that are sent out daily to various stores. The data-processing program's task is to keep track of inventory levels, which items and how many of each item should be reordered from the producer (an item is reordered when less than 100 remain in stock), and how many items are received from the producer. In the data division are data sets (e.g., shipments-received, beginning-inventory, total-items, etc.), and in the computation division are the computation steps that involve these data sets (e.g., for each item, the beginning inventory is computed by adding the shipments received to the final inventory from the previous day).

After receiving the SSL specification of the desired program, PROTOSYSTEM transforms it into an efficient target language implementation consisting of a collection of PL/I programs and its JCL ("Job Control Language") for the IBM 360 system. To accomplish this transformation, the following specific design decisions are made with the goal of achieving an efficient implementation:

- (a) Design each keyed file, deciding what are to be its data items, organization (consecutive, index sequential, regional), storage device, associated sort ordering, and number of records per block;
- (b) design each job step, determining which computations the step is to include, its accessing method (sequential, random, core table), its driving data set(s), and the order (by key values) in which the records of its input data sets are to be processed;
- (c) determine whether sorts are necessary and where they should be performed;  
and
- (d) determine the sequence of job steps.

Generally, these design decisions, especially the central ones of determining the final target data sets, computation steps, and sequencing of computation steps, are made by exploring the different ways of combining data sets and computation steps. The system carries out these explorations with the goal of minimizing the number of file accesses made during the run-time of the target implementation. Sometimes, as explained below, the system also will seek to minimize a more detailed cost estimate of the target implementation.

Described in greater detail in the next section, the method employed by PROTOSYSTEM for achieving an efficient implementation does not rely solely on heuristics but instead uses what is essentially a dynamic programming algorithm with heuristics added to the algorithm, so that it can finish in a reasonable amount of time. An advantage of dynamic programming is that it can provide a good handle on global optimization when the results of individual decisions have far-reaching and compounding effects throughout the design of the data-processing program.

### Operation

Although the actual optimization process is performed by the optimizer module, several other modules provide preparatory and support services. First, the structural analyzer module generates predicates for the operations in the SSL computation division. These predicates indicate the conditions under which data items in a data set will be either accessed or generated during an operation. For example, the condition

$$(\text{DEFINED A (k1)}) = (\text{OR} (\text{DEFINED B (k1)}) (\text{DEFINED C (k1)}))$$

would indicate that there is a record in data set A for a value of the key, k1, only when at least one of the data sets B or C has a record for that value of the key. The structural analyzer also produces candidate driving data sets for each operation in the computation division. A driving data set of an operation is a data set whose records are "walked through" once in order of their occurrence--i.e., the operation is executed once at each step (record)--to drive the operation.

The predicates produced by the structural analyzer are then used by the question-answering module to provide information to the optimizer about the average number of I/O accesses implied by tentative configurations (i.e. tentative choices for the data sets and computation steps) of the target implementation. The question-answering module maintains a knowledge base consisting of the predicates, characteristics of the data, as well as information obtained from interaction with the user, such as average data set size or the probability of a predicate fragment being true. This knowledge, along with knowledge about the probability calculus, is used to answer questions about the size of a data set and about the average number of items in the data set that are likely to satisfy a certain predicate (e.g., an access predicate). When the knowledge is insufficient to answer an optimizer question, the question answerer initiates a dialogue with the user in order to elicit enough additional information to proceed.

The optimization process itself is performed by the optimizer module. This module intermittently obtains information from the question answerer about I/O accesses of tentative configurations of parts of the data-processing program, in order to explore the effects of such design parameters as the number of records per block, the file organization, the data items that are collected into a single data set, and the computations that are performed during a single reading of a file or files. Since the problem area of PROTSYSTEM is that of I/O intensive programs, the optimizer explores the various design parameters with the goal of minimizing the number of file accesses of the target language implementation (of the data-processing program). Sometimes, however, after a number of more important design decisions have been made, the optimizer will explore design decisions by computing a more detailed cost estimate that attempts to approximate the charging structure of the particular installation on which the target system is to run (e.g., disc space, core residency charges, explicit I/O, etc.).

The central part of the optimization process is concerned with the the exploration of various ways of setting up data sets and computation steps. Basically, the optimization module starts with the data sets and computation steps in the data division and computation division of the SSL specification. Then, with the goal of minimizing the number of file accesses, the module looks at data-processing programs that use various aggregations of these initial data sets and computation steps (an aggregation of two or more data sets is a



data set that has all the data items of the original data sets, while an aggregation of several computation steps is a computation step that performs the functions of the original steps). The optimizer explores aggregating data sets and aggregating computation steps and develops and utilizes constraints on the sort order of both data sets and computation steps (an example of a sort order constraint on a data set would be when the data set should have its records sorted on a particular key first).

To avoid the problem of combinatorial explosion, the module uses a form of dynamic programming with heuristics. Loosely speaking, one may say that dynamic programming is a set of parameterized recursive equations, which, in this case, express the cost of optimized longer segments of the program in terms of optimized shorter segments. A pure dynamic programming algorithm, though it would find the absolute optimum target implementation, would require an extreme amount of time to do so. Therefore, in order that the algorithm finish in a reasonable time, a number of heuristics have been employed in the algorithm, including decoupling decisions where possible (and sometimes even where it is not completely possible) and carrying out local optimizations before making adjustments for global concerns. A full explanation of the algorithm is found in Morgenstern (1976).

**Status**

The SSL specification language has been completely defined and there is an operational implementation of PROTOSYSTEM in MACLISP on the MIT LCS PDP-10. The system is capable of producing acceptable target language implementations. From a larger perspective, the PROTOSYSTEM I project has developed a 5-phase model of the process of writing a data-processing program (system), from its conception to its implementation as executable code. Twenty years ago, the fifth phase, compilation and loading, was automated. At present, a preliminary theory and automation of the third and fourth phases, the generation of the system and translation into high-level code, are embodied in PROTOSYSTEM I. It is felt that within the next decade the theory and automation of the remaining two phases, including problem definition, specification analysis, and system formulation, should easily fall within the realm of presently developing AI technologies.

**DATA DIVISION**

FILE SHIPMENTS-RECEIVED  
 KEY IS ITEM  
 GENERATED EVERY DAY  
 FILE BEGINNING-INVENTORY  
 KEY IS ITEM  
 GENERATED EVERY DAY  
 FILE TOTAL-ITEM-ORDERS  
 KEY IS ITEM  
 GENERATED EVERY DAY  
 FILE QUANTITY-SHIPPED-TO-STORE  
 KEY IS ITEM, STORE  
 GENERATED EVERY DAY

FILE QUANTITY-ORDERED-BY-STORE  
 KEY IS ITEM, STORE  
 GENERATED EVERY DAY  
 FILE TOTAL-SHIPPED  
 KEY IS ITEM  
 GENERATED EVERY DAY  
 FILE FINAL-INVENTORY  
 KEY IS ITEM  
 GENERATED EVERY DAY  
 FILE REORDER-AMOUNT  
 KEY IS ITEM  
 GENERATED EVERY DAY

**COMPUTATION DIVISION**

```
BEGINNING-INVENTORY IS
  FINAL-INVENTORY (from the previous day) + SHIPMENTS-RECEIVED
TOTAL-ITEM-ORDERS IS SUM OF QUANTITY-ORDERED-BY-STORE FOR EACH ITEM
QUANTITY-SHIPPED-TO-STORE IS
  QUANTITY-ORDERED-BY-STORE          IF BEGINNING-INVENTORY IS
                                     GREATER THEN TOTAL-ITEM-ORDERS
ELSE
  QUANTITY-ORDERED-BY-STORE
  * (BEGINNING-INVENTORY / TOTAL-ITEM-ORDERS)
  IF BEGINNING-INVENTORY IS NOT
    GREATER THEN TOTAL-ITEM-ORDERS
TOTAL-SHIPPED IS SUM OF QUANTITY-SHIPPED-TO-STORE FOR EACH ITEM
FINAL-INVENTORY IS BEGINNING-INVENTORY - TOTAL-SHIPPED
REORDER-AMOUNT IS 1000 IF FINAL-INVENTORY IS LESS THAN 100.
```

Figure 1: SSL relational description for a data processing program.

#### References

See Baron (1977), Morgenstern (1976), Ruth (1976a), Ruth (1978), and Ruth (1979).

## J. NLPQ: Natural Language Programming for Queuing Simulations

The Natural Language Programming for Queuing Simulations (NLPQ) project was begun by George Heidorn at Yale University in 1967 as a doctoral dissertation and completed at the Naval Postgraduate School during the years 1968-1972. The problem area is that of simulation programs for simple queuing problems. The queuing problem's specification occurs during an English dialogue in which the user and the NLPQ system each can furnish information to, and request information from, the other. From this dialogue, the NLPQ system creates and maintains a partial internal description of the queuing problem. This partial description is used to answer any questions that the user may ask; it is used to generate questions that are to be asked of the user; and when eventually completed by the dialogue activity, it is used to generate the implementation of the simulation program in the target language GPSS. The system's processing -- including creating the problem description and generating the GPSS program, as well as translating and generating sentences--is specified by production rules.

### Specification

In the English dialogue that constitutes NLPQ's method of specification, the user can make statements, give commands, ask questions, and answer questions. During this process the system can ask and answer questions and respond to commands. In the very brief dialogue of Figure 1, most of the features of the specification method are illustrated in a simple way (other more complex dialogues are presented in Heidorn, 1972, Heidorn, 1974, Heidorn, 1975b, and Heidorn, 1976). In the dialogue, line numbers have been added for purposes of reference; and the lower case typing was done by the user, while the UPPER CASE typing was done by the computer.

The particular specification concerns a queuing problem about cars that arrive at a station, get serviced, and leave. The user initially volunteered some information about how often the cars arrive and about the sequencing of actions, in lines 1 and 2, and then asked the system what other information was needed to complete the queuing model, in line 3. In lines 4 through 9, the system asked questions about times and actions, which the user answered, until the problem statement was complete, in line 10. Then in line 11 the user asked a specific question about the queuing model concerning the mean time needed to service a car, a question that the system answered in line 12. In line 13 the user told the system to produce a description of the entire problem in English for verification purposes, which it did, in line 14. Finally, in response to the user command in line 15, the system produced the GPSS program, in line 16, complete with English comments (preceded by \*) and meaningful symbolic names. The time required for the NLPQ system to process this sample specification and to produce the GPSS program was 8 seconds of virtual CPU time on an IBM 370/168 running LISP.

```
1 ** EVERY 6 MINUTES A CAR ARRIVES AT A STATION.  
2 ** AFTER ARRIVING, IT IS SERVICED AT A PUMP.  
3 ** IS THE PROBLEM STATEMENT COMPLETE?  
4 How long are the cars serviced at the pump?  
5 ** FROM 3 TO 7 MINUTES.  
6 After being serviced at the pump, what do the cars do?  
7 ** THEY LEAVE THE STATION.
```

- 8 How long shall the simulation be run?  
 9 \*\* 10 HOURS; USE 1 MINUTE AS THE BASIC TIME UNIT.  
 10 The problem statement is complete.  
 11 \*\* WHAT IS THE MEAN OF THE TIME TO SERVICE CARS?  
 12 The mean of the uniform distribution is 5 minutes.  
 13 \*\* DESCRIBE THE PROBLEM IN ENGLISH.  
 14 The cars arrive at the station every 6 minutes.  
 After arriving at the station, the cars are serviced at the pump. The time for the cars to be serviced at the pump is uniformly distributed, with a mean of 5 minutes and a half-range of 2 minutes. After being serviced at the pump, the cars leave the station.  
 The simulation is to be run for 10 hours, using a basic time unit of 1 minute.

15 \*\* WRITE THE GPSS PROGRAM.

```

SIMULATE
STAT1 EQU    1,F,Q
PUMP2 EQU    2,F,Q
CAR1 EQU     1,T
1  TABLE    M1,1,1,2
*
*   THE CARS ARRIVE AT THE STATION.
GENERATE 6
ASSIGN 1,CAR1
*
*   THE CARS ARE SERVICED AT THE PUMP.
ACT1 QUEUE  PUMP2
SEIZE  PUMP2
DEPART PUMP2
ADVANCE 5,2
RELEASE PUMP2
*
*   THE CARS LEAVE THE STATION.
ACT3 TABULATE P1
TERMINATE
*
*   TIMING LOOP
GENERATE 600
TERMINATE 1
START 1
END

```

Figure 1. A very brief NLPQ dialogue.

### Operation

The processing to be done by NLPQ is specified by sets of production rules written in a

language designed especially for this system. "Decoding" rules specify how strings of English text are to be converted into records in a semantic net, and "encoding" rules specify how records are to be converted into text. These rules are basically phrase structure grammar rules Natural Language.B1, but they are augmented with arbitrary conditions and structure-building actions .

The representation of the internal description of the simulation problem as well as the representation of the syntactic and semantic structures are in the form of a semantic network Representation.B2. A network consists of records that represent such things as concepts, words, physical entities, and probability distributions. Each record is a list of attribute-value pairs, where the value of an attribute is usually a pointer to another record but may sometimes be simply a number or character string.

Prior to a queuing dialogue, the system is given a network of about 300 "named" records containing information about words and concepts relevant to simple queuing problems. Also, it is furnished with a set of about 300 English decoding rules and 500 English and GPSS encoding rules. As the dialogue progresses, the system uses the information it obtains from the English dialogue to build and complete a partial description of the desired simulation, a description that is in the form of a network called the Internal Problem Description (IPD).

Basically, an IPD network describes the flow of mobile entities, such as vehicles, through a framework consisting of stationary entities, such as pumps, by specifying the actions that take place in the framework and their interrelationships. Each action is represented by a record whose attributes furnish such information as the type of action, the entity doing the action (i.e., the agent), the entity that is the object of the action, the location where it happens, its duration, its frequency of occurrence, and what happens next. For example, the action "The men unload the truck at a dock for two hours" could be represented by the record:

```

R1:  Type      unload
      Agent     men
      Object    truck
      Location  dock
      Duration  2 hours

```

From the English dialogue the NLPQ system must obtain all the information needed to build the IPD. Thus, the user must describe the flow of mobile entities through the queuing model by making statements about the actions that take place and about the relations between these actions. Each mobile entity must "arrive" at or "enter" the model. Then it may go through one or more other actions, such as "service," "load," "unload," and "wait." Then, typically, it "leaves" the model. The order in which these actions take place must eventually be made explicit by the use of subordinate clauses beginning with such conjunctions as "after," "when," and "before," or by using the adverb "then." If the order of the actions depends on the state of the queuing model, an "if" clause may be used to specify the condition for performing an action; a sentence with an "otherwise" in it is used to give an alternative action to be performed when this condition is not met.

The information needed to simulate the problem, including the various times involved, must also be furnished by the English dialogue. It is necessary to specify the time between

arrivals, the time required to perform each activity, the length of the simulation run, and the basic time unit to be used in the GPSS program. Inter-event and activity times may be given as constants or as probability distributions, such as uniform, exponential, normal, or empirical. The quantity of each stationary entity should also be specified, unless 1 is to be assumed.

The user may either furnish this information in the form of a complete problem statement or state some part of it and then let the system ask questions to obtain the rest of the information, as was done above in lines 1 through 10 of Figure 1. The latter method results in a scan of the partially built IPD for missing or erroneous information and the generation of appropriate questions. Each time the system asks a question, it is trying to obtain the value of some specific attribute that will be needed to generate a GPSS program. To furnish a value for the attribute, the question may be answered by a complete sentence or simply by a phrase.

The user may ask the system specific questions about the queuing model, and then the system generates the answers from the information in the appropriate parts of the IPD. In order to check the entire IPD as it exists at any time, the user may request that an English problem description be produced. Such a description consists of all the information in the IPD as it is converted into English by the encoding rules (see line 14 of Figure 1). Specifically, for each action in the IPD, the system generates one or more statements describing the type of action, its agent, object, location, what action if any follows (if none, a new paragraph is started), and, if applicable, an inter-event time or duration. Conditional successor actions may result in two sentences, with the first one having an "if" clause in it and the second one beginning with "otherwise." After all of the actions have been described, a separate one-sentence paragraph is produced with the values of the run time and the basic time unit.

After the dialogue is finished and all the required information is obtained, NLPQ uses the IPD and the GPSS encoding rules to produce the desired program in the GPSS target language. Such a program was listed in 16 of Figure 1. At the beginning of this program, the definitions for the stationary entities, mobile entities, and distributions are given. Then, for each action, a comment consisting of a simple English action sentence is produced, followed by the GPSS statements appropriate to this action. For example, an "arrive" usually produces a GENERATE and an ASSIGN; a "leave" produces a TABULATE and a TERMINATE; and most activities produce a sequence like QUEUE, SEIZE, DEPART, ADVANCE, and RELEASE. These are usually followed by some sort of TRANSFER, depending upon the type of value that the action's successor attribute has. Finally, the GPSS program closes with a "timing loop" to govern the length of the simulation run.

### Status

Though this project was "completed," a system ready for production use was not developed. The NLPQ prototype, however, was demonstrated several times on a variety of problems. Although the capabilities of the implemented system are limited, the research did establish an overall framework for such a system, and useful techniques were developed. Enough details were worked out to enable the system to carry out interesting interactions, as evidenced by the longer more complicated dialogues found in the first four references at the end of this article. More details of the processing done by this system can be found in any of the references, especially Heidorn, 1972, which is a 376-page technical report.

**References**

See Heidorn (1972), Heidorn (1974), Heidorn (1975a), Heidorn (1975b), and Heidorn (1976).

### K. LIBRA

LIBRA, the efficiency analysis expert of the PSI system D2 is being developed by Elaine Kant in conjunction with the PSI project at Systems Control, Inc., and at Stanford. The PSI system, through interaction with the user, constructs a very high-level program specification called the program model. Then LIBRA, working together with the PECOS coding expert D5, converts the program model into a target language implementation. The PECOS system supplies the transformation rules that can convert the program model into various target language implementations. Using global efficiency analysis ("global analysis" is analysis with access to the entire program, as opposed to only a local segment), LIBRA directs and explores the application of the transformation rules so as to produce an efficient implementation.

The transformation process itself consists of repeated applications of transformation rules to parts of the program, where every application results in a specification closer to a target language implementation. Each such application of a rule is said to produce a partial implementation or *refinement* of the program, and the transformation rules are called *refinement rules*. Thus refinement rules applied to refinements produce further refinements. Because more than one refinement rule may be applicable to the same part of a refinement, the transformation process produces a tree of possible refinements (the actual situation is slightly more complicated since the order in which the rules are applied can affect the tree that is produced). To avoid the problem of combinatorial explosion, LIBRA develops only part of the tree. A discussion of the details of this process follows.

It is LIBRA's function to analyze and guide the development of the refinement tree in order to achieve an efficient implementation. LIBRA determines what parts of the program to expand next and what parts not to expand at all. In particular, when more than one refinement rule is applicable, LIBRA may decide to apply them all so that the resulting refinements can be considered in greater detail; or LIBRA may decide to apply only one of the rules. In the latter case, the refinement is implemented directly in the current node of the tree, and the other possibilities are permanently forgone.

One of the most important ways in which LIBRA attacks the problem of combinatorial explosion is by *estimating* the efficiency of possible target language implementations. For each refinement in the tree, LIBRA maintains two cost estimates; the estimates are in the form of symbolic algebraic expressions that give the time and space requirements needed to execute a certain kind of target language implementation. The first estimate is the default cost that might result if all the constructs and operators in the refinement were assigned default implementations. The second is the optimistic cost estimate that might result assuming: (a) certain efficient implementation techniques that have worked in similar situations will prove successful in the present situation, and (b) LIBRA expends enough of its own resources of time and space to carry out these implementation techniques.

Treating these two costs as upper and lower bounds on the costs of possible target language implementations of the refinement, LIBRA obtains important guidance in directing the growth of the refinement tree. These upper and lower bounds can be used to prune a branch of the refinement tree (without further consideration of the branch) or to calculate the effect of a partial implementation decision on the global program cost. As discussed below in the RULES section, the upper and lower bounds are used to direct attention to high impact areas, those areas where effort is likely to yield the greatest increases in overall efficiency.



Another feature of the LIBRA system, a feature implicit in the above, is the knowledge LIBRA has about the use and limits of its own resources of available time and space. This feature is important because no system can devote unlimited effort to finding an efficient implementation. Effort must be allocated. The way in which LIBRA performs this allocation is to assign available resources to high impact areas, where the resources will do the most good. The RULES section will present the method used to compute impact, as well as examples and uses of resource knowledge.

LIBRA also includes mechanisms to assist in the acquisition of new programming concepts. When new high-level constructs are added (such as new types of sorts, or trees), new efficiency knowledge is needed to analyze these concepts (their subparts, running times, data structure accesses, and so on). LIBRA has a model of programming concepts that is consulted when new concepts are added. Some of the necessary information can be deduced automatically, and the user is asked specific questions to obtain the rest. To help construct these estimation functions, LIBRA provides a semi-automatic procedure for deriving cost estimation functions from the set of cost functions for the target language constructs.

The knowledge for managing resources, computing upper and lower cost estimates, directing attention to different parts of the tree, making implementation decisions, and, in general, for analyzing and directing the growth of the tree is in the form of rules. Each rule consists of a condition and an action to be performed if the condition is met. The knowledge that a rule expresses can easily be modified since the rules are replaceable and can be added, deleted, or altered without requiring a modification to the system itself.

## Rules

The rules in LIBRA's knowledge base generally can be divided into three groups: attention and resource management rules, plausible-implementation rules, and cost-analysis rules.

Attention and resource management rules describe when to shift attention to other nodes in the tree and also how to set priorities for refining the different constructs and operations within a refinement node. Some of the more important of these rules determine how LIBRA's own resources of available time and space are to be allocated, on the basis of where they will have the greatest impact. One of the ways of determining impact is to consider the difference between the upper bound cost estimate (assuming default implementations) and the optimistic lower bound cost estimate (assuming both the successful application of efficiency techniques that have worked in similar situations and the sufficient expenditure of resources to carry the techniques to completion). Other rules in this group state how to shift attention among nodes. These rules (a) cause complex programs to be expanded early in order to see what decisions are involved, (b) postpone trivial decisions until important ones are made, (c) look at all refinements in the tree and select for development the one whose optimistic cost estimate is least (when resources for developing a particular refinement are exhausted), and (d) apply a form of branch and bound which states that (when resources allocated for considering a particular decision are exhausted) attention should be directed to the whole tree and that all nodes whose optimistic cost estimate is worse than the default estimate of some other node should be eliminated. As described later, when cost analysis rules compare estimates, they take into account the degree of uncertainty in the estimate.

Plausible implementation rules express heuristics about when to limit expansion of nodes, by making a decision about some part of an implementation. For example, when the question of how to represent a set first arises, LIBRA performs a global examination of the program to determine all uses of the set. If there are many places where the program checks for membership in the set, then a hash-table representation may be suggested. In general, plausible implementation rules express knowledge derived by human or machine analysis of commonly occurring situations, such as which sorting techniques are best for different size inputs. These rules also contain heuristics to make quick decisions. Thus, if LIBRA is running out of resources, heuristics that are not as dependable as the one just described are used to make decisions on the spot, without creating any new nodes. These heuristics generally express defaults, such as "use lists rather than arrays if the target language is LISP"; they are used to make the less important decisions or to make all decisions if the total resources for writing a program are nearly exhausted.

The final group, the cost-analysis rules, express how to compute, update, and compare upper and lower bound estimates of the cost of the final implementation. The cost estimates are in the form of symbolic algebraic expressions that may involve variables representing set sizes. The cost estimates are not computed once and for all: Whenever a refinement in the tree is further refined (i.e., a refinement rule is applied to some part of a node in the subtree whose root is the refinement), then the cost estimates associated with the refinement are *incrementally updated* so as to produce estimates that are more accurate in view of the new information. Cost estimates are constructed from a knowledge base that includes information on upper and lower bounds on costs for time and space usage by individual constructs and operations, and on how to combine such cost estimates for composite programs. The knowledge needed to incrementally update the cost estimates is contained in rules corresponding to the particular construct or operation. The method of comparing the cost estimates of different refinements involves the addition of a bonus to the refinement that has a greater degree of completion and that consequently has a greater certainty in its cost estimates (default and optimistic). This feature favors a nearly complete refinement that has a slightly worse lower bound over a less complete (more abstract) refinement that has a slightly better lower bound. Such a preference is desirable since the cost estimate of the more abstract refinement is less certain and therefore may not be achievable. By giving a bonus for the degree of completion, the cost analysis rules take into account the likelihood of being able to achieve the cost estimate.

### Status

LIBRA has guided the application of the PECOS refinement rules to produce efficient implementation of several variants of simple database retrieval, sorting, and concept formation programs (see PSI article for an example of a concept formation program). Current plans include extending the problem area to include simple algorithms for finding prime numbers and for reaching nodes in a graph. For an efficiency expert to be of use in a complete automatic programming system, a good deal more research is needed. Higher level optimizations, extended symbolic analysis and comparison capabilities, and more domain expertise are some obvious extensions. Automatic bookkeeping of heuristics and perhaps even automatic generation of heuristics from an analysis of symbolic cost estimates of target language concepts are some long-range goals. In order to write more complex programs such as compilers or operating systems, more efficiency rules would have to be added to the system, rules about concepts such as bit-packing, machine interrupts, and multiprocessing.

However, even with such additions, the efficiency techniques employed by the LIBRA system should be significant in controlling the problem of combinatorial explosion that occurs during the search for efficient implementations.

This article closes with the description of an example illustrating LIBRA's present operation producing a simple sort program.

### Example

Suppose that a SORT is specified as a transfer of elements from a SOURCE sequential collection to a TARGET sequential collection that is ordered by some relation such as LESS-THAN. After the application of some preliminary refinement rules that do not require any decisions as to alternative choices, three choice points remain: choosing a transfer order, and choosing representations for SOURCE and for TARGET.

Since the transfer order is selected as the most important decision, LIBRA directs attention first to that choice point. A heuristic rule is applied that suggests the use of either an insertion sort from list to list or array to array, or a selection sort from list to array. The different refinement possibilities are added to the tree accordingly. Each of the branches is given a limited amount of resources and told to focus attention only on the parts of the program directly relevant to the transfer order decision.

After these branches are refined within the limits of the assigned resources, the nodes of the tree are compared. Branch and bound does not eliminate any of the alternatives here, but the insertion branch is selected as it has the best lower bound (taking into account factors related to uncertainty of estimates).

Refinement then proceeds in that node. The choice of a list or array representation for the TARGET is made by a heuristic that says that lists are easier to manipulate than arrays in LISP. This heuristic was applied because much of the time and space resources allocated for finding an implementation had been consumed in the above tasks and a quick decision was required. The choice of a list representation for TARGET forces a list representation for SOURCE because of a suggestion made under the transfer-order heuristic. Thereafter, the refinement process is basically straightforward, though several choices of whether to store or recompute local variables are made.

### References

See Barstow (1979), Barstow & Kant (1977), Green (1976b), Green (1977), Green & Barstow (1978), Kant (1977), Kant (1978), Kant (1979), and McCune (1977).

## References

- Automatic Coding. Proc. of the Symposium, Franklin Institute, Philadelphia, PA, January 1957.
- Aho, A. V., Hopcroft, J. E., & Ullman, J. D. *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, 1974.
- Allen, F. E., & Cocke, J. A catalogue of optimizing transformations. In R. Rustin (Ed.), *Design and Optimization of Compilers, Proceedings of the Courant Computer Science Symposium 5*. Englewood Cliffs, N.J.: Prentice-Hall, 1972. Pp. 1-30.
- Allen, F. E., & Cocke, J. A program data flow analysis procedure. *Communications of the ACM*, 1976, 19(3), 137-147.
- Amarel, S. Representation and modeling in problems of program formation. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 6*. New York: American Elsevier, 1972. Pp. 411-466.
- Balzer, R. M. Dataless programming. *Proceedings FJCC*, 1967, 31, 535-544.
- Balzer, R. M. *Automatic Programming*. Information Sciences Institute Tech. Memo 1, University of Southern California, Marina Del Rey, 1972.
- Balzer, R. M. CASAP: A testbed for program flexibility. *IJCAI 3*, 1973, 601-605. (a)
- Balzer, R. M. A global view of automatic programming. *IJCAI 3*, 1973, 494-499. (b)
- Balzer, R. M. *A Language-independent programmer's interface*. Information Sciences Institute Report RR-73-15, University of Southern California, Marina Del Rey, November 1973. (c)
- Balzer, R. M. *Human Use of World Knowledge*. Information Sciences Institute Report USC-ISI RR-73-07, University of Southern California, Marina Del Rey, March 1974 (ARPA Order 2223/1).
- Balzer, R. M., & Goldman, N. Principles of Good Software Specification and Their Implications for Specification Languages. *Proc. of the IEEE Specifications of Reliable Software Conf.*, Cambridge, April 1979.
- Balzer, R. M., Goldman, N., & Wile, D. On the Transformational Implementation Approach to Programming. *2nd Int. Conf. on Software Engineering*, October 1976, pp. 337-344.
- Balzer, R. M., Goldman, N., & Wile, D. Informality in program specification. *IJCAI 5*, 1977, 389-397. (a)
- Balzer, R. M., Goldman, N., & Wile, D. Meta-evaluation as a tool for program understanding. *IJCAI 5*, 1977, 398-403. (b)

- Balzer, R. M., Goldman, N., & Wile, D. On the Use of Programming Knowledge to Understand Informal Process Descriptions. *SIGART Newsletter*, No. 63, June 1977, pp. 72-75. (c).
- Balzer, R. M., Goldman, N., & Wile, D. Informality in Program Specifications. *IEEE Transactions on Software Engineering*, 1978, SE-4(2), 94-103.
- Balzer, R. M., Greenfeld, N., Kay, M., Mann, W., Ryder, W., Wilczynski, D., & Zobrist, A. Domain independent automatic programming. *IFIP*, 1974, 326-330.
- Baron, R. V. *Structural Analysis in a Very High Level Language*, Master's thesis, MIT, 1977.
- Barstow, D. A Knowledge based system for automatic program construction. *IJCAI 5*, 1977, 382-388. (a)
- Barstow, D. A knowledge base organization for rules about programming. *Proc. of the Workshop on Pattern Directed Inference Systems*. *SIGART Newsletter*, No. 63, June 1977, pp. 18-22. (b)
- Barstow, D. Automatic Construction of Algorithms and Data Structures using a Knowledge Base of Programming Rules, AI Memo 308, Computer Science Dept., Stanford University, November 1977. (c)
- Barstow, D. Codification of programming knowledge: Graph algorithms, TR-149, Computer Science Dept., Yale University, December 1978.
- Barstow, D. *Knowledge-based Program Construction*. Elsevier: North Holland, 1979.
- Barstow, D. R., & Kant, E. Observations on the interaction between coding and efficiency knowledge in the PSI system. *Proc. of the 2nd Int. Conf. on Software Engineering*, Computer Society, Institute of Electrical and Electronics Engineers, Inc., Long Beach, CA, October 1977, pp. 19-31.
- Barth, J. M. An Interprocedural data flow analysis algorithm. *Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, CA, January 1977.
- Bauer, M. A basis for the acquisition of procedures from protocols. *IJCAI 4*, 1975, 226-231.
- Biermann, A. W. Computer program synthesis from computation traces. *Symposium on Fundamental Theory of Programming*, Kyoto University, Kyoto, Japan, October 1972. (a)
- Biermann, A. W. On the inference of Turing machines from sample computations. *Artificial Intelligence*, 1972, 3, 181-198. (b)
- Biermann, A. W. The Use of Examples in Program Construction and Debugging. *ACM '75: Proceedings of the National Conference, Association for Computing Machinery*, New York, 1975. Pp. 242-247.

- Biermann, A. W. Approaches to automatic programming. In M. Rubinoff & M. C. Yovits (Eds.), *Advances in Computers* (vol. 15). New York: Academic Press, 1976. Pp. 1-63. (a)
- Biermann, A. W. *Regular LISP Programs and Their Automatic Synthesis from Examples*, CS-1976-12, Dept. of Computer Science, Duke University, June 1976. (b)
- Biermann, A. W., Baum, R. I., & Petry, F. E. Speeding up the synthesis of programs from traces. *IEEE Transactions on computers*, February 1975, C-24, 122-136.
- Biermann, A. W., & Krishnaswamy. *Constructing programs from example computations*, OSU CISRC TR-74-5, August 1974.
- Biggerstaff, T. J. *C2: A Super-compiler Approach to Automatic Programming*. Doctoral dissertation, Tech. Rep. 76-01-01, Dept. of Computer Science, University of Washington, 1976.
- Bobrow, D. G., & Wegbreit, B. A model of control structures for Artificial Intelligence programming languages. *IEEE Transactions on Computers*, 1976, C-25(4), 347-353.
- Bobrow, D. G., & Winograd, T. An overview of KRL, a knowledge representation language. *Cognitive Science*, 1977, 1(1), 3-46.
- Boyer, R. S., & Moore, J. S. Proving theorems about LISP Functions. *JACM*, 1975, 22(1), 129-144.
- Brown, G. P. *A Framework for Processing Dialogue*, TR-182, Laboratory for Computer Science, MIT, June 1977.
- Brown, R. *Use of Analogy to Achieve New Expertise*, AI-TR-403, MIT AI Lab, April 1977.
- Buchanan, J. R., & Luckham, D. C. *On Automating the Construction of Programs*, TR-CS-433, Artificial Intelligence Laboratory, Stanford University, Stanford, CA, May 1974. (Also Stanford AI Memo 236)
- Burstall, R. M., & Darlington, J. Some transformations for developing recursive programs. *International Conference on Reliable Software*, IEEE Computer Society, April 1975, pp. 465-472.
- Burstall, R. M., & Darlington, J. A Transformation System for Developing Recursive Programs. *Journal of the Association for Computing Machinery*, 1977, 24(1), 44-67.
- Chandrasekaran, B. AI--The past decade--Automatic programming. In M. Rubinoff & M. C. Yovits (Eds.), *Advances in Computers* (vol. 13). New York: Academic Press, 1975. Pp. 170-232.
- Chang, C., & Lee, R. *Symbolic Logic and Mechanical Theorem Proving*. New York: Academic Press, 1969.

- Cheatham, T. E., Jr., & Wegbreit, B. A laboratory for the study of automating programming. *Proceedings of AFIPS Spring Joint Computer Conference*, 1972, pp. 11-21.
- Cheatham, T. E., & Townley, J. A. *Symbolic Evaluation of Programs: A Look at Loop Analysis*, TR-11-76, Center for Research in Computing Technology, Harvard University, 1976.
- Clark, K. and SICKEL, ? Predicate logic: A calculus for deriving programs. *IJCAI* 5, 1977, 419-420.
- Dahl, O. J., Dijkstra, E. W., & Hoare, C. A. R. *Structured Programming*. New York: Academic Press, 1972.
- Dahl, O. J., Myrhaug, B., & Nygaard, K. *SIMULA67 Common Base Language*, Publ. No. S-2, Norwegian Computing Centre, Oslo, 1968.
- Darlington, J. *A Semantic approach to automatic program improvement*. Doctoral dissertation, University of Edinburgh, Scotland, 1972.
- Darlington, J. Automatic program synthesis in second-order logic. *IJCAI* 3, 1973, 537-542.
- Darlington, J. Applications of program transformation to program synthesis. In G. Huet & G. Kahn (Eds.), *Proving and Improving Programs*. Rocquencourt, France: Institut de Recherche d'Informatique et d'Automatique, July 1975. Pp. 133-144.
- Darlington, J. *A Synthesis of Several Sorting Algorithms*, Research Report 23, Dept. of Artificial Intelligence, University of Edinburgh, Scotland, July 1976.
- Darlington, J., & Burstall, R. M. A System which automatically improves programs. *IJCAI* 3, 1973, 479-485.
- Dershowitz, N., & Manna, Z. On automating structured programming. In G. Huet & G. Kahn (Eds.), *Proving and Improving Programs*. Rocquencourt, France: Institut de Recherche d'Informatique et d'Automatique, July 1975. Pp. 167-193.
- Dershowitz, N., & Manna, Z. The evolution of programs: A system for automatic program modification. *Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, CA, January 1977.
- Deutsch, B. G. The structure of task-oriented dialogs. In L. Erman (Ed.), *IEEE Symposium on Speech Recognition: Contributed Papers*, IEEE Group on Acoustics, Speech, and Signal Processing. The Institute of Electrical and Electronics Engineers, Inc., New York, April 1974. Pp. 250-254.
- Deutsch, B. G. *Establishing Context in Task-Oriented Dialogs*, Tech. Note 114, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, CA, September 1975.
- Dijkstra, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 1975, 18(8), 453-457.

- Dijkstra, E. W. *A discipline of programming*. Englewood Cliffs, N.J.: Prentice-Hall, 1976.
- Earley, J. Relational level data structures in programming languages. *Acta Informatica*, 1973, 2, 293-309.
- Earley, J. High-level Iterators and a Method for Automatically Designing Data Structure Representation, Memo ERL-M425, Electronics Research Laboratory, University of California, Berkeley, 1974. (a)
- Earley, J. High-level operations in automatic programming. *Proceedings of the SIGPLAN Symposium on Very High-level Languages*, March 1974. *SIGPLAN Notices*, 1974, 9(4), 34-42. (b)
- Elcock, E. W., Foster, J. M., Gray, P. M. D., McGregor, J. J., & Murray, A. M. ABSET: A programming language based on sets: Motivation and examples. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 6*. Edinburgh: Edinburgh University Press, 1971. Pp. 467-492.
- Feldman, J. A. *Towards Automatic Programming*. Preprints of the NATO Software Engineering Conference, Rome, Italy, October 1969.
- Feldman, J. A. *Automatic Programming*, AIM-160, STAN-CS-72-255, Stanford AI Lab, Computer Science Dept., Stanford University, February 1972.
- Fenichel, R. R., Weizenbaum, J., & Yochelson, J. C. A program to teach programming. *CACM*, 1970, 13(3), 141-146.
- Floyd, R. W. Toward interactive design of correct programs. In C. V. Freiman (Ed.), *Foundations and Systems, Information Processing 71: Proceedings of IFIP Congress 71* (vol. 1). Amsterdam: North-Holland Publishing Co., 1972. Pp. 7-10. (Also Memo AIM-150, Report STAN-CS-71-235, AI Lab, Computer Science Dept., Stanford University, September 1971.)
- Ginsparg, J. M. *Natural Language Processing in an Automatic Programming Domain*. Doctoral dissertation and Memo AIM-316, Rep. STAN-CS-78-671, AI Lab, Computer Science Dept., Stanford University, Stanford, CA, June 1978.
- Goldberg, P. C. *Automatic Programming*, RC 5148, Computer Sciences Dept., Thomas J. Watson Research Center, IBM, Yorktown Heights, New York, September 1974.
- Goldberg, P. C. *The Future of Programming for Nonprogrammers*, RC 5975, Watson Research Center, IBM, Yorktown Heights, New York, May 1976.
- Goldman, N., Balzer, R. M., & Wile, D. The Inference of Domain Structure from Informal Process Descriptions, *Workshop on Pattern-Directed Inference Systems*, Hawaii, May 1977. *SIGART Newsletter*, No. 63, June 1977, pp. 75-82.
- Goldstein, I., & Sussman, G. J. *Some projects in automatic programming*, Working Paper 67, AI Lab, MIT, March 1974.



- Goodman, R. (Ed.) *The Annual Review in Automatic Programming* (Papers of the Working Conference on Automatic Programming of Digital Computers, Brighton, April 1959). New York: Pergamon Press, 1960.
- Green, C. *The Application of Theorem Proving to Question-answering Systems*. Doctoral dissertation, Electrical Engineering Dept., Memo AIM-96, Report STAN-CS-69-138, AI Lab, Computer Science Dept., Stanford University, June 1969.
- Green, C. Unpublished lecture surveying Automatic Programming. Stanford University, Computer Science Dept., 1975. (a)
- Green, C. *Whither automatic programming*, invited tutorial lecture. IJCAI 4, Tbilisi, USSR, September 1975. (b)
- Green, C. An informal talk on recent progress in Automatic Programming. *Lectures on Automatic Programming and List Processing*, PIPS-R-12, Electrotechnical Laboratory, Tokyo, Japan, November 1976, pp. 1-69. (a)
- Green, C. The design of the PSI program synthesis system. *Proc. 2nd International Conference on Software Engineering*, October 1976. Pp. 4-18. (b)
- Green, C. The PSI Program Synthesis System, 1976. *ACM '76: Proceedings of the Annual Conference, Association for Computing Machinery*, New York, N.Y., October 1976, pp. 74-75. (c)
- Green, C. A Summary of the PSI Program Synthesis System. *IJCAI 5*, 1977, 380-381.
- Green, C. The PSI Program Synthesis System, 1978: An Abstract. In S. P. Ghosh, & L. Y. Liu (Eds.), *AFIPS Conference Proc.: National Computer Conf.*, 1978, 47, 673-674.
- Green, C., et al. *Progress Report on Knowledge Based Programming*. Systems Control, Inc., Computer Science Division, Palo Alto, CA, September 1978.
- Green, C., & Barstow, D. A hypothetical dialogue exhibiting a knowledge base for a program understanding system. In E. W. Elcock & D. Michie (Eds.), *Machine Intelligence 8: Machine Representations of Knowledge*. New York: Halsted Press, John Wiley & Sons, 1977. Pp. 335-359. (a)
- Green, C., & Barstow, D. *On Program Synthesis Knowledge*, Memo AIM-306, Report STAN-CS-77-639, AI Lab, Computer Science Dept., Stanford University, Stanford, CA, November 1977. (b)
- Green, C., & Barstow, D. *On program synthesis knowledge*. *Artificial Intelligence*, 1978, 10(3), 241-279.
- Green, C., Waldinger, R., Barstow, D., Elschlager, R., Lenat, D., McCune, B., Shaw, D., & Steinberg, L. *Progress Report on Program Understanding Systems*, Memo AIM-240, AI Lab, Stanford, CA, August 1974.
- Green, C., & Barstow, D. Some rules for the automatic synthesis of programs. *IJCAI 4*, 1975, 232-239.

- Gries, D. **Programming by Induction**, TR 71-106, Computer Science Dept., Cornell University, September 1971.
- Guttag, J. V., Horowitz, E., & Musser, D. R. **Abstract Data Types and Software Validation**, Tech. Report ISI-RR-76-48, Information Sciences Institute, Marina del Rey, CA, August 1976.
- Hammer, M. **Automatic Programming: An Assessment**. Unpublished paper, MIT Lab for Computer Science, Cambridge, Mass., December 1977.
- Hammer, M., & Ruth, G. **Automating the Software Development Process**. In P. Wegner (Ed.), **Research Directions in Software Technology**. Cambridge: MIT Press, 1979. Pp. 767-792.
- Hammer, M., Howe, W. G., Kruskal, V. J., & Wladawsky, I. **A Very High-Level Programming Language for Data Processing Applications**, RC 5583, Computer Sciences Dept., Thomas J. Watson Research Center, IBM, Yorktown Heights, New York, August 1975.
- Hardy, S. **Synthesis of LISP functions from examples**. IJCAI 4, 1975, 240-245.
- Heidorn, G. **The End of the User Programmer? The Software Revolution**, Infotech State of the Art Conf., Copenhagen, Denmark, October 1977. (To appear in **Future Programming**, Infotech, England, 1979.)
- Heidorn, G. E. **Natural Language Inputs to a Simulation Programming System**, Report 55hd72101A, Naval Postgraduate School, Monterey, CA, October 1972.
- Heidorn, G. E. **English as a very high level language for simulation programming**. IBM Research 4536, September 1973.
- Heidorn, G. E. **English as a very high level language for simulation programming**, Proceedings Symposium on Very High Level Languages. SIGPLAN Notices, 1974, 9(4), 91-100.
- Heidorn, G. E. **Augmented Phrase Structure Grammars**. In B. L. Nash-Webber & R. C. Schank (Eds.), **Theoretical Issues in Natural Language Processing**. Association for Computational Linguistics, June 1975. Pp. 1-5. (a)
- Heidorn, G. E. **Simulation programming through natural language dialogue**. Amsterdam: North-Holland Studies in the Management Sciences, 1975. (b)
- Heidorn, G. E. **Simulation programming through natural language dialogue**. In M. A. Geisler (Ed.), **TIMS Studies in the Management Sciences, Logistics** (vol. 1). Amsterdam: North Holland, 1975. Pp. 71-85. (c)
- Heidorn, G. E. **Automatic programming through natural language dialogue: A survey**. IBM Journal of Research and Development, 1976, 20(4), 302-313.
- Henderson, P., & Morris, J. H., Jr. **A lazy evaluator**. Third ACM Symposium on Principles of Programming Languages, Atlanta, GA, January 1976. Pp. 95-103.

- Hewitt, C. **Teaching Procedures in Humans and Robots**, Memo 208, AI Lab, Massachusetts Institute of Technology, April 1970.
- Hewitt, C. **Viewing Control Structures as Patterns of Passing Messages**, Working paper 92 (rev. ed.), AI Lab, Massachusetts Institute of Technology, April 1976.
- Hewitt, C., & Smith, B. C. **Towards a programming apprentice**. *IEEE Transactions: Software Engineering*, 1975, 1(1), 26-45.
- Hill, I. D. **Wouldn't it be nice if we could write computer programs in ordinary English--or would it?** *Computer Bulletin*, 1972, 16(6), 306-312.
- Hobbs, J. R. **From Well Written Algorithm Descriptions into Code**, Research Rep. 77-1, Dept. of Computer Science, City College, City University of New York, July 1977.
- Kant, E. **The selection of efficient implementations for a high level language**, *Proceedings of Symposium on Artificial Intelligence and Programming Languages*. *SIGPLAN Notices*, 12(8); *SIGART Newsletter*, No. 64, August 1977, pp. 140-146.
- Kant, E. **Efficiency Estimation: Controlling Search in Program Synthesis**. In S. P. Ghosh & L. Y. Leonard (Eds.), *AFIPS Conf. Proc.: National Computer Conf.*, 1978, 47, 703.
- Kant, E. **Efficiency Considerations in Program Synthesis: A Knowledge-based Approach**. Doctoral dissertation, Stanford University, Computer Science Dept., 1979.
- Kowalski, R. **Predicate Logic as a Programming Language**, Information Processing, North Holland, Amsterdam, 1977.
- Lenat, D. B. **Synthesis of large programs from specific dialogues**. In G. Huet & G. Kahn (Eds.), *Proving and Improving Programs*. Rocquencourt, France: Institut de Recherche d'Informatique et d'Automatique, July 1975. Pp. 225-241.
- Liskov, B. H., Snyder, A., Atkinson, R., & Schaffert, C. **Abstraction Mechanisms in CLU**. *CACM*, 1977, 20(8), 564-576.
- Lomet, D. B. **Data Flow Analysis in the Presence of Procedure Calls**, RC 5728, Thomas J. Watson Research Center, IBM, Yorktown Heights, New York, November 1975.
- Long, W. J. **A Program Writer**. Doctoral dissertation, TR-187, LCS, Massachusetts Institute of Technology, November 1977.
- Low, J. R. **Automatic coding: Choice of data structures**, Stanford AI Memo AIM-242, Stanford University, August 1974.
- Low, J. R. **Automatic Coding: Choice of Data Structures**, ISR16, Birkhauser Verlag, 1976. (a)
- Low, J. R. **Automatic Data Structure Selection: An Example and Overview**, TR-14, Computer Science Dept., University of Rochester, September 1976. (b)

- Low, J. R. Automatic data structure selection: An example and overview. *CACM*, 1978, 5, 21-25.
- Manna, Z., & Waldinger, R. DEDALUS--The DEDuctive ALgorithm UR-Synthesizer. *National Computer Conference*, Anaheim, CA, June 1978. Pp. 683-690.
- Manna, Z., & Waldinger, R. *Synthesis: Dreams Programs*, Memo AIM, AI Lab, Stanford, CA, November 1977.
- Manna, Z., & Waldinger, R. J. Toward automatic program synthesis. *Communications of the ACM*, 1971, 14(3), 151-165.
- Manna, Z., & Waldinger, R. Knowledge and reasoning in program synthesis. *Artificial Intelligence*, 1975, 6(2), 175-208.
- Manna, Z., & Waldinger, R. DEDALUS--The DEDuctive ALgorithm Ur-Synthesizer. *AFIPS National Computer Conference*, 1978, 47, 683-690.
- Martin, W. A. *OWL Notes: A System for Building Expert Problem Solving Systems Involving Verbal Reasoning*, M.I.T. Project MAC, 1974.
- McCune, B. P. The PSI program model builder: Synthesis of very high-level programs. *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*. *SIGPLAN Notices*, 12(8), 130-139; *SIGART Newsletter*, No. 64, August 1977, 130-139.
- McCune, B. P. *Building Program Models Incrementally from Informal Descriptions*. Doctoral dissertation, AI Lab Memo, Computer Science Dept., Stanford University, in press.
- Michie, D. Memo functions and machine learning. *Nature*, 1968, 218(No. 5136), 19-22.
- Miller, L. A., & Becker, C. A. *Programming in Natural English*, Research Report RC 5137, Thomas J. Watson Research Center, IBM, Yorktown Heights, New York, November 1974.
- Mitchell, J. G. *The Design and Construction of Flexible and Efficient Interactive Programming Systems*. Doctoral dissertation, Dept. of Computer Science, Carnegie-Mellon University, June 1970.
- Morgenstern, M. *Automated Design and Optimization of Information Processing Systems*. Doctoral dissertation, MIT, 1976.
- Persson, S. *Some Sequence Extrapolating Programs: A Study of Representation and Modeling in Inquiring Systems*. Doctoral dissertation, School of Business Administration, University of California, Berkeley; Memo AIM-46, Report STAN-CS-66-50, AI Lab, Computer Science Dept., Stanford University, September 1966.
- Petry, F. E., & Biermann, A. W. Reconstruction of algorithms from memory snapshots of their execution. *ACM '76: Proceedings of the Annual Conference*, Association for Computing Machinery, New York, October 1976, pp. 530-534.

- Phillips, J. V. Program Inference from Traces Using Multiple Knowledge Sources. *IJCAI* 5, 1977, p. 812.
- Pratt, V. R. The Competence/Performance Dichotomy in Programming, TM-400, AI Lab, Massachusetts Institute of Technology, January 1977.
- Project MAC, Automatic composition of functions from modules (Section III E.1). *Project MAC Progress Report X*, Massachusetts Institute of Technology, July 1972-July 1973.
- Reisser, J. F. (Ed.) *SAIL*, Stanford AI Memo No. 289, August 1976.
- Rich, C. A Library of Programming Plans with Applications to Automated Analysis, Synthesis and Verification of Programs. Doctoral dissertation, MIT, Cambridge, MA, 1979.
- Rich, C., & Shrobe, H. E. Initial Report on a LISP Programmer's Apprentice, TR-354, AI Lab, Massachusetts Institute of Technology, December 1976.
- Rich, C., & Shrobe, H. Initial Report on a LISP Programmer's Apprentice. *IEEE Trans. on Soft. Eng.*, 1978, 4(6), 456-467.
- Rivest, R. L. Two-Dimensional Programming Languages. Dept. of Electrical Engineering and Computer Science Dept., MIT, April 1975.
- Rosen, B. K. Data Flow Analysis for Procedural Languages, RC 5948, Computer Sciences Dept., Thomas J. Watson Research Center, IBM, Yorktown Heights, New York, April 1976.
- Rosen, B. K. Applications of high-level control flow. *Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, CA, January 1977.
- Rovner, P. D. Automatic Representation Selection for Associate Data Structures, TR-10, Computer Science Dept., University of Rochester, September 1976.
- Ruth, G. Analysis of Algorithm Implementations. Doctoral dissertation, TR-130. Project MAC, Massachusetts Institute of Technology, May 1974.
- Ruth, G. Automatic Design of Data Processing Systems, *Proc. of the Third ACM Symposium on Principles of Programming Languages*, Atlanta, Georgia, 1976 (also MIT Comp. Sci. TR TM-070). (a)
- Ruth, G. Intelligent program analysis. *AI*, 1976, 7, 65-85. (b)
- Ruth, G. Protosystem I: An Automatic Programming System Prototype, *Proc. of the National Computer Conf.*, Anaheim, CA, 1978. *AFIPS*, 1978, 47, 675-681.
- Ruth, G. Automating the Software Development Process. In P. Wegner (Ed.), *Research Directions in Software Technology*. Cambridge: MIT Press, 1979.
- Sacerdoti, E. D. *A Structure for Plans and Behavior*. New York: Elsevier, 1977.

- Schwartz, J. T. On Programming: An Interim Report on the SETL Project (rev. ed.). Computer Science Dept., Courant Institute of Mathematical Sciences, New York University, June 1975.
- Shaw, D., Swartout, W., & Green, C. Inferring LISP programs from examples. *IJCAI* 4, 1975, 260-267.
- Shrobe, H. E. Reasoning and Logic for Complex Program Understanding. Doctoral dissertation, MIT, Cambridge, MA, August 1978.
- Sibel, W., Furbach, U., & Schreiber, J. F. Strategies for the synthesis of algorithms. *Informatik-Fachbereich*, 1978, 5, 97-109.
- Siklossy, L. The synthesis of programs from their properties, and the insane heuristic. *Proceedings of the Third Texas Conference on Computing Systems*, Austin, TX, 1974, pp. 5-2-1 - 5-2-5.
- Siklossy, L., & Sykes, D. Automatic program synthesis from example problems. *IJCAI* 4, 1975, 268-273.
- Simon, H. A. Experiments with a heuristic compiler. *JACM*, 1963, 10(4), 493-503.
- Simon, H. A. The heuristic compiler. In H. A. Simon & L. Siklossy (Eds.), *Representation and Meaning*. Englewood Cliffs, N. J.: Prentice-Hall, 1972. Pp. 9-43.
- Summers, P. D. A Methodology for LISP Program Construction from Examples. *JACM*, 1977, 24(1), 161-175.
- Sussman, G. J. *A Computer Model of Skill Acquisition*. New York: American Elsevier, 1975.
- Szolovits, P., Hawkinson, L. B., & Martin, W. A. An Overview of OWL, A Language for Knowledge Representation, TM-86, LCS, Massachusetts Institute of Technology, June 1977.
- Teitelman, W. PILOT: A Step Toward Man-Computer Symbiosis. Doctoral dissertation, MAC-TR-32, Project MAC, Massachusetts Institute of Technology, September 1966.
- Teitelman, W. Toward a programming laboratory. *IJCAI* 1, 1969, 1-8.
- Teitelman, W. Automated programming--The programmer's assistant. *Proceedings Fall Joint Computer Conference* (Vol. 41), December 1972, pp. 917-921.
- Teitelman, W. *Interlisp Reference Manual*. Xerox Corp., Palo Alto, CA, 1974.
- Teitelman, W. A Display Oriented Programmer's Assistant, CSL 77-3, Palo Alto Research Center, Xerox Corp., Palo Alto, CA, March 1977.
- Teitelman, W., et al. *INTERLISP Reference Manual*. Xerox PARC, Palo Alto, CA, October 1978.

- Van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., & Koster, C. H. A. Report on the algorithmic Language ALGOL68. *Numerische Mathematik*, 1969, 14, 79-218.
- Waldinger, R. **Constructing Program Automatically Using Theorem Proving**. Doctoral dissertation, Carnegie-Mellon University, Pittsburgh, Penn., 1969.
- Waldinger, R. Achieving several goals simultaneously. In E. W. Elcock & D. Michie (Eds.), **Machine Intelligence 8: Machine Representations of Knowledge**. New York: Halsted Press, John Wiley & Sons, 1977. Pp. 94-136.
- Waldinger, R. , & Lee, R. PROW: A step toward automatic program writing. *IJCAI* 1, 1969, 241-252.
- Waldinger, R. , & Levitt, K. N. Reasoning about programs. *Artificial Intelligence*, 1974, 5(3), 235-316.
- Warren, D. H. D. **WARPLAN: A System for Generating Plans**, Memo No. 76, Dept. of Computational Logic, School of Artificial Intelligence, University of Edinburgh, Scotland, June 1974.
- Warren, D. H. D. Generating conditional plans and programs. *Proceedings of the Conference on Artificial Intelligence and Simulation of Behavior*, Edinburgh, Scotland, July 1976, pp. 344-354.
- Warren, D. H. D. **Implementing PROLOG: Compiling Predicate Logic Programs** (vols. 1-2), Research Reports 39-40, Dept. of AI, University of Edinburgh, Scotland, May 1977.
- Warren, H. S., Jr. **Data Types and Structures for a Set Theoretic Programming Language**, RC 5567, Thomas J. Watson Research Center, IBM, Yorktown Heights, New York, August 1975.
- Waterman, D. A. Generalization Learning Techniques for Automating the Learning of Heuristics. *Artificial Intelligence*, 1970, 1, 121-170.
- Waters, R. C. **A System for Understanding Mathematical FORTRAN Programs**, MIT-AIM-168, MIT, Cambridge, MA, August 1976.
- Waters, R. C. **Automatic Analysis of the Logical Structure of Programs**, MIT-AI-TR-492, December 1978 (based on doctoral dissertation, **A Method for Automatically Analyzing the Logical Structure of Programs**, August 1978).
- Waters, R. C. **A Method for Analyzing Loop Programs**. To appear in *IEEE Trans. on Soft. Eng.*, 1979.
- Wegbreit, B. **Studies in Extensible Programming Languages**. Doctoral dissertation, Center for Research in Computing Technology, Harvard University, January 1972.
- Wegbreit, B. **Goal-directed program transformation**, CSL-75-8, Xerox PARC, Palo Alto, CA, September 1975. (a)
- Wegbreit, B. **Mechanical program analysis**. *CACM*, 1975, 9(18), 528-539. (b)

- Wilber, B. M. A QLISP Reference Manual, AI Center Tech. Report 118, SRI International, Inc., Menlo Park, CA, March 1976.
- Winograd, T. Five Lectures on Artificial Intelligence, Stanford AIM-246, CS459, Computer Science Dept., Stanford University, September 1974.
- Winograd, T. Breaking the complexity barrier again. SIGPLAN Notices, 1975, 10(1), 13-30.
- Winston, P. H. Learning structural descriptions from examples. In P. Winston (Ed.), The Psychology of Computer Vision. New York: McGraw-Hill, 1975.
- Wirth, N. The programming language PASCAL. Acta Informatica, 1971, 1, 35-63.
- Zilles, S. Abstract Specification for Data Types. IBM Research Laboratory, San Jose, CA, 1975.
- Zimmerman, L. L. On-line program debugging: A graphic approach. Computers and Automation, 1967, 16(11), 30-34.