Second Edition

ENCYCLOPEDIA OF ARTIFICIAL INTELLIGENCE

Volume 1
A-L

Awarded

American Library Association's

Outstanding Reference Source

Association of American Publishers Award

Best New Professional and Scholarly Publication

Stuart C. Shapiro
Editor-in-Chief

AI Overview

Stuart C. Shapiro, ARTIFICIAL INTELLIGENCE, 54-57

From Shapiro, Stuart C., Editor-in-Chief, Encyclopedia of Artificial Intelligence, 2nd Ed., John Wiley & Sons, Inc., NY, 1992.

ARTIFICIAL INTELLIGENCE

There have been many definitions of artificial intelligence (AI) offered over the years. Perhaps a good one is:

> Artificial Intelligence is a field of science and engineering concerned with the computational understanding of what is commonly called intelligent behavior, and with the creation of artifacts that exhibit such behavior.

This may be examined more closely by considering the field from the points of view of three goals that AI researchers have, which might be called *computational psychology, computational philosophy,* and *advanced computer science.*

Computational Psychology

The goal of computational psychology is to understand human intelligent behavior by creating computer programs that behave in the same way people do. For this goal, it is important that the algorithm expressed by the program be the same algorithm that people actually use, and that the data structures used by the program be the same data structures used by the human mind. The program should do quickly what people do quickly, should do more slowly what people have difficulty doing, and should even tend to make mistakes where people tend to make mistakes. If the program were put into the same experimental situations that human subjects are subjected to, the program's results should be within the range of human variability.

Computational Philosophy

The goal of computational philosophy is to form a computational understanding of human-level intelligent behavior, without being restricted to the algorithms and data structures that the human mind actually does (or conceivably might) use. By *computational understanding* is meant a model that is expressed as a procedure that is at least implementable (if not actually implemented) on a computer. By "human-level intelligent behavior" is meant behavior that, when engaged in by people is commonly taken as being part of human intelligent cognitive behavior. It is acceptable, though not required, if the implemented model perform some tasks better than any people would. Bearing in mind Church's Thesis (qv), this goal might be reworded as asking the question, is intelligence a computable function?

In the AI areas of vision and robotics, computational philosophy is replaced by computational natural philosophy (science). For example, computer vision researchers are interested in the computational optics question of how can the information contained in light waves reflected from an object be used to reconstruct the object. Notice that this is a different question from the computational psychology question of how the human visual system uses light waves falling on the retina to identify objects in the world.

Advanced Computer Science

The goal of advanced computer science is to push outwards the frontier of what we know how to program on computers, especially in the direction of tasks that, although we don't know how to program them, people can perform them. This goal led to one of the oldest definitions of AI: the attempt to program computers to do what until recently only people could do. Although this gets across the idea of pushing out the frontier, it is also perpetually self-defeating in that as soon as a task is conquered, it no longer falls within the domain of AI, AI is left only with its failures as its successes become other areas of computer science. The most famous example of this is the area of symbolic calculus. When Slagle wrote the SAINT (qv) program, it was the first program in history that could solve symbolic integration problems at the level of freshman calculus students, and was considered an AI project. Now that there are multiple systems on the market that can do much more than what SAINT did, these systems are not considered to be the results of AI research.

Heuristic Programming

Computational psychology, computational philosophy, and advanced computer science are subareas of AI divided by their goals. Any given AI researcher probably wanders among two or all three of these areas throughout his or her career, and may even have a mixture of these goals at the same time.

Another way of distinguishing AI as a field is by noting the AI researcher's interest in heuristics rather than in algorithms. Here I am taking a wide interpretation of a *heuristic* as any problem solving procedure that fails to be

an algorithm, or that has not been shown to be an algorithm, for any reason. An interesting view of the tasks that AI researchers consider to be their own may be gained by considering those ways in which a procedure may fail to be an algorithm.

The common definition of an algorithm for a general problem P is: an unambiguous procedure that, for every particular instance of P, terminates and produces the correct answer. The most common reasons that a heuristic H fails to be an algorithm are: it doesn't terminate for some instances of P; it has not been proved correct for all instances of P because of some problem with H; or it has not been proved correct for all instances of P because P is not well-defined. Common examples of AI heuristic programs that don't terminate for all instances of the problem they have been designed for are search and theorem proving programs. Any search procedure will run forever if given an infinite search space that contains no solution state. Gödel's Incompleteness Theorem states that there are formal theories that contain true but unprovable propositions. In actual practice, AI programs for these problems stop after some prespecified time, space, or work bound has been reached. They can then only report that they were unable to find a solution—in any given case, a little more work might have produced an answer. An example of an AI heuristic that has not been proved correct is any static evaluation function used in a computer chess program. The static evaluation function returns an estimate of the value of some state of the board. To be correct, it would return $+\infty$ if the state were a sure win for the side to move, $-\infty$ if it were a sure win for the opponent, and 0 if it were a forced stalemate. Moreover, for any state it is theoretically possible to find the correct answer algorithmically by doing a full minimax search of the game tree rooted in the state being examined. However, such a full search is infeasable for almost all states because of the size of the game tree. Static evaluation functions are still useful, even without being proved correct. An example of an AI heuristic program that has not been proved correct because the problem for which it has been designed is not well-defined is any natural language understanding program or natural language interface. Since no one has any well-defined criteria for whether a person understands a given language, there cannot be any well-defined criteria for programs either.

EARLY HISTORY

Although the dream of creating intelligent artifacts has existed for many centuries, the field of artificial intelligence is considered to have had its birth at a conference held at Dartmouth College in the summer of 1956. The conference was organized by Minsky and McCarthy, and McCarthy coined the name "artificial intelligence" for the proposal to obtain funding for the conference. Among the attendees were Simon and Newell, who had already implemented the Logic Theorist program at the Rand Corp. These four people are considered the fathers of AI. Minsky and McCarthy founded the AI Laboratory at the Massachusetts Institute of Technology; Simon and Newell founded the AI laboratory at Carnegie Mellon University; and McCarthy later moved from M.I.T. to Stanford and founded the AI laboratory there. These three universities, along with Edinburgh University, whose Department of Machine Intelligence was founded by Michie, have remained the premier research universities in the field. The name artificial intelligence remained controversial for some years, even among people doing research in the area, but it eventually was accepted.

The first AI text was *Computers And Thought,* edited by Feigenbaum and Feldman and published by McGraw-Hill in 1963. *Computers and Thought* is a collection of 21 papers, some of them short versions of Ph.D. dissertations, by early AI researchers. Most of the papers in this collection are still considered classics of AI, but of particular note is a reprint of Turing's 1950 paper in which the Turing Test was introduced.

Regular AI conferences began in the mid to late 1960s. The Machine Intelligence Workshops series began in 1965 in Edinburgh. A conference at Case Western University in the Spring of 1968 drew many of the U.S. AI researchers of the time, and the first biennial International Joint Conference on Artificial Intelligence was held in Washington, D.C. in May, 1969. *Artificial Intelligence,* still the premier journal of AI research, began publishing in 1970. For a more complete history of AI, see McCorduck (1979). (see LITERATURE, AI).

NEIGHBORING DISCIPLINES

Artificial Intelligence is generally considered to be a subfield of computer science, though there are some computer scientists who have only recently and grudgingly accepted this view. There are several disciplines outside computer science, however, that strongly impact AI and which AI strongly impacts.

Cognitive psychology (qv) is the subfield of psychology that uses experimental methods to study human cognitive behavior. The goal of AI called computational psychology above is obviously closely related to cognitive psychology, differing mainly in the use of computational models rather than experiments on human subjects. However, most AI researchers pay some attention to the results of cognitive psychology, and cognitive psychologists tend to pay attention to AI as suggesting possible cognitive procedures that they might look for in humans.

Cognitive science (qv) is an interdisciplinary field that studies human cognitive behavior under the hypothesis that cognition is (or can usefully be modeled as) computation. AI and cognitive science overlap in that there are researchers in each field that would not consider themselves to be in the other. AI researchers whose primary goal is what was called advanced computer science above generally do not consider themselves to be doing cognitive science. Cognitive science contains not only AI researchers, but also cognitive psychologists, linguists, philosophers, anthropologists, and others, each using the methodology of his or her own discipline on a common problem—that of understanding human cognitive behavior.

Computational linguistics (qv) researchers use computers, or at least the computational paradigm, to study and/or to process human languages. Like cognitive science, computational linguistics overlaps AI. It includes those areas of AI called natural language understanding (qv), natural language generation (qv), speech understanding (qv), and machine translation (qv), but also non-AI areas such as the use of statistical methods to find index keywords useful for retrieving a document.

AI-COMPLETE TASKS

There are many subtopics in the field of AI, as one can see by contemplating the individual articles in this encyclopedia. These subtopics vary from the consideration of a very particular, technical problem, to broad areas of research. Several of these broad areas can be considered *AI-complete,* in the sense that solving the problem of the area is equivalent to solving the entire AI problem: producing a generally intelligent computer program. A researcher in one of these areas may see himself or herself as attacking the entire AI problem from a particular direction. These areas are also ways of organizing the articles of this encyclopedia. The following sections discuss some of the AI-complete areas covered by this encyclopedia, and point to some of the articles relevant to those areas. Not every article in the encyclopedia is included in the following lists.

Natural Language

The AI subarea of natural language is essentially the overlap of AI and computational linguistics (see above). The goal of the area is to form a computational understanding of how people learn and use their native languages, and to produce a computer program that can use a human language at the same level of competence as a native human speaker. Virtually all human knowledge has been (or could be) encoded in human languages (consider this encyclopedia and others, textbooks, etc). Moreover, research in natural language understanding has shown that encyclopedic knowledge is required to understand natural language. Therefore, a complete natural language using system will also be a complete intelligent system. The articles in this Encyclopedia relevant to natural language include: ARGUMENT COMPREHENSION; COMPUTATIONAL LINGUISTICS; CONVERSATIONAL IMPLICATURE; DEEP STRUCTURE; DICTIONARY/LEXICON; DISCOURSE UNDERSTANDING; ELLIPSIS; GRAMMAR, AUGMENTED TRANSITION NETWORK; GRAMMAR, CASE; GRAMMAR, GENERALIZED PHRASE STRUCTURE; GRAMMAR, PHRASE STRUCTURE; GRAMMAR, SEMANTIC; GRAMMAR, SYSTEMIC; HERMENEUTICS; LEXICAL DECOMPOSITION; MACHINE TRANSLATION; MORPHOLOGY; NATURAL LANGUAGE GENERATION; NATURAL LANGUAGE UNDERSTANDING; PARSING; PARSING, WORD EXPERT; PREFERENCE SEMANTICS; PRESUPPOSITION; QUESTION ANSWERING; SPEECH RECOGNITION; SPEECH SYNTHESIS; SPEECH UNDERSTANDING; STORY ANALYSIS; TEXT SUMMARIZATION; THESAURUS.

Problem Solving and Search

Problem solving is the area of AI that is concerned with finding or constructing the solution to a problem. That sounds like a very general area, and it is. The distinctive characteristic of the area is probably its approach of seeing tasks as problems to be solved, and of seeing problems as spaces of potential solutions that must be searched to find the true one, or the best one. Thus the AI area of search is very much connected to problem solving. Since any area investigated by AI researchers may be seen as consisting of problems to be solved, all of AI may be seen as involving problem solving and search. The articles in the encyclopedia that are most directly about problem solving and search include: A* ALGORITHM; AND/OR GRAPHS; BACKTRACKING; BRANCHING FACTOR; DISTRIBUTED PROBLEM SOLVING; HEURISTICS; MEANS-ENDS ANALYSIS; MINIMAX PROCEDURE; PROBLEM REDUCTION; PROBLEM SOLVING; SEARCH; SEARCH, BEAM; SEARCH, BEST-FIRST; SEARCH, BI-DIRECTIONAL; SEARCH, BRANCH-AND-BOUND; SEARCH, DEPTH-FIRST; SIMULATED ANNEALING.

Knowledge Representation and Reasoning

Knowledge representation is the area of AI concerned with the formal symbolic languages used to represent the knowledge (data) used by intelligent systems, and the data structures used to implement those formal languages. However, one cannot study static representation formalisms and know anything about how useful they are. Instead, one must study how they are helpful for their intended use. In most cases, the intended use is to use explicitly stored knowledge to produce additional explicit knowledge. This is what reasoning is. Together, knowledge representation and reasoning can be seen to be both necessary and sufficient for producing general intelligence—it is another AI-complete area. Although they are bound up with each other, knowledge representation and reasoning can be teased apart according to whether the particular study is more about the representation language–data structure, or about the active process of drawing conclusions.

The articles in this encyclopedia that are most concerned with knowledge representation include: BELIEF REPRESENTATION SYSTEMS; CONCEPTUAL DEPENDENCY; DYNAMIC MEMORY; EPISODIC MEMORY; FRAMES; KNOWLEDGE REPRESENTATION; LOGIC; LOGIC, CONDITIONAL; LOGIC, HIGHER ORDER; LOGIC, MODAL; LOGIC, ORDER-SORTED; LOGIC, PREDICATE; LOGIC, PROPOSITIONAL; MEMORY ORGANIZATION PACKETS; MENTAL MODELS; SEMANTIC NETWORKS.

The articles in this encyclopedia that are most concerned with reasoning include: ABDUCTION; BAYESIAN INFERENCE METHODS; CIRCUMSCRIPTION; DEMPSTER-SHAFER THEORY; EQUALITY INFERENCING; FUZZY SETS AND FUZZY LOGIC: AN OVERVIEW; META-KNOWLEDGE, META-RULES, AND META-REASONING; QUALITATIVE PHYSICS; REASONING, CASE-BASED; REASONING, CAUSAL; REASONING, COMMONSENSE; REASONING, DEFAULT; REASONING, MEMORY-BASED; REASONING, NONMONOTONIC; REASONING, PLAUSIBLE, REASONING, SPATIAL; REASONING, TEMPORAL; RESOLUTION; RESOLUTION, GRAPH-BASED; RESOLUTION, THEORY; THEOREM PROVING; TRUTH MAINTENANCE; UNIFICATION.

Learning

Learning is often cited as the criterial characteristic of intelligence, and it has always seemed like the easy way to producing intelligent systems: Why build an intelligent

system when we could just build a learning system and send it to school? The articles in this encyclopedia that are most concerned with learning include: CONCEPT LEARNING; INDUCTIVE INFERENCE; LEARNING, MACHINE.

Vision

Vision, or image understanding, has to do with interpreting visual images that fall on the human retina or the camera lens. The actual scene being looked at could be 2-dimensional, such as a printed page of text, or 3-dimensional, such as the world about us. If we take "interpreting" broadly enough, it is clear that general intelligence may be needed to do the interpretation, and that correct interpretation implies general intelligence, so this is another AI-complete area. The articles in this encyclopedia that are most concerned with vision include: CHARACTER RECOGNITION; COLOR VISION; EARLY VISION; EDGE AND LOCAL FEATURE DETECTION; HOUGH TRANSFORMS; IMAGE MODELS; IMAGE PROPERTIES; IMAGE UNDERSTANDING; ISPECTION; OBJECT RECOGNITION; PATTERN RECOGNITION; RANGE DATA ANALYSIS; SCALE SPACE; SEGMENTATION; SENSORS AND SENSOR FUSION; SHAPE; STEREO VISION; VISUAL MOTION ANALYSIS; VISUAL PERCEPTION; VISUAL RECOVERY.

Robotics

The area of robotics is concerned with artifacts that can move about in the actual physical world, and/or that can manipulate other objects in the world. The articles in this encyclopedia that are most concerned with robotics include: PROTHESES; ROBOT CONTROL SYSTEMS; ROBOT HANDS AND END EFFECTORS; ROBOT MANIPULATORS; ROBOT PATH PLANNING AND OBSTACLE AVOIDANCE; ROBOTICS; ROBOTS, LEGGED; ROBOTS, MOBILE; TELEOPERATORS.

APPLICATIONS

Throughout its existence as a field, AI research has produced spin-offs into other areas of computer science. Lately, however, programming techniques developed by AI researchers have found application to many programming problems. This has largely come about through the subarea of AI known as expert systems. Whether or not any particular program is intelligent, or is an expert is largely irrelevant pragmatically. From the point of view of the field as a whole, probably the best thing about this development is that after many years of being criticized as following an impossible dream by inappropriate and inadequate means, AI has been recognized by the general public as having applications to everyday problems.

The articles in this encyclopedia that are most relevant to expert systems as a subarea of AI include: BLACKBOARD SYSTEMS; CERTAINTY FACTORS; EXPERT SYSTEMS; RULE-BASED SYSTEMS. The articles in this encyclopedia that discuss past, current, and potential applications of AI include: ART, AI IN; CHEMISTRY, AI IN; EDUCATION, AI IN; ENGINEERING, AI IN; LAW, AI IN; MANUFACTURING, AI IN; MEDICINE, AI IN; MILITARY APPLICATIONS OF AI; MUSIC, AI IN; PROGRAMMING ASSISTANTS.

BIBLIOGRAPHY

E. A. Feigenbaum and J. Feldman, *Computers and Thought,* McGraw-Hill Book Company, New York, 1963.

P. McCorduck, *Machines Who Think,* W. H. Freeman and Company, San Francisco, 1979.

A. M. Turing, "Computing Machinery and Intelligence," *Mind* **59,** 433–460 (October 1950).

STUART C. SHAPIRO
SUNY at Buffalo

ASSOCIATIVE MEMORY

Memory architectures can be classified as random, sequential, and associative (Hwang and Briggs, 1984). First introduced by Bush (1945), associative memories have found considerable use in hardware and have been discussed in innumerable papers. This section overviews the use of associative memories from controllers to database memories and concludes with their importance to AI systems.

An associative memory is composed of a memory that is a two-dimensional array of bits $\mathbf{M}[1, \ldots, i; 1, \ldots, j]$ of i rows and j columns and a search mechanism that can search this array and extract information from it. The array of bits can be considered a set of equal-length words $\mathbf{M}[n; 1, \ldots, j]$ for $n = 1, \ldots, i$. A comparand $\mathbf{C}[1, \ldots, j]$ and a mask $\mathbf{M}[1, \ldots, j]$ can be used to search $\mathbf{M}$ and set bits in a result register $\mathbf{R}[1, \ldots, i]$, see Figure 1. For each $n = 1, \ldots, i$, if for $m = 1, \ldots, j$, $\mathbf{H}[m] = 1$ or $(\mathbf{M}[n; m] = \mathbf{C}[m])$, one can say word $\mathbf{M}[n; 1, \ldots, j]$ matches $\mathbf{C}$ under $\mathbf{H}$, and $\mathbf{R}[n]$ of set to 1; otherwise $\mathbf{R}[n]$ is cleared. Matched words can be output to a bus $\mathbf{B}[1, \ldots, j]$; for $m = 1, \ldots, j$, $\mathbf{B}[m]$ is the OR of $\mathbf{M}[n; m]$ wherever $\mathbf{R}[n]$ is 1.

Consider this simple example of a telephone directory searched in an associative memory. Each word is (person's name, telephone number, address). Suppose one wants to know Mr. Smith's address or telephone number. The search and output operation would use $\mathbf{C}$ = (SMITH, xxx, xxx) and $\mathbf{H}$ = (00000, 111, 111). In the search operation the row having SMITH in the leftmost part will match, and the result bit for that row is set. In the output part that row is put on the bus $\mathbf{B}$, where the unknown parts can be obtained.

MEMORY MANAGEMENT AND HARDWARE CONTROL

The search and output operation is often combined in a single step in an associative memory, which is the MMU used in virtual memories (Hwang and Briggs, 1984). In such a system a processor (CPU) reads and writes data in RAM. The processor sends an address to RAM in order to read or write a word in it. This address is put into the MMU, as a comparand, like the name SMITH in the example above. The output on the bus, rather like the telephone number of the example, is the actual address that is

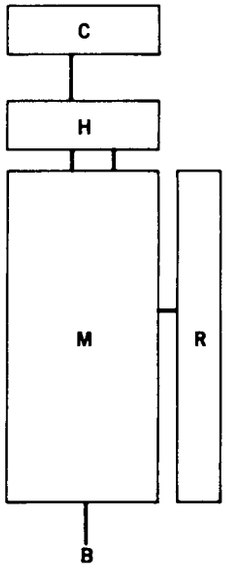


Figure 1. Basic associative memory.

sent to RAM. An MMU lets the address maintained and sent by the CPU (called the virtual address) be different from the real address used in RAM.

Another common application, where the associative memory is a read-only memory, is the PLA. Used in the control logic of a processor, microprogrammed commands are put in as comparands, and the outputs of the bus are sent to control registers, adders, and other parts of the hardware. The PLA lets the microprogram use commands that are efficiently encoded, which control a vast number of hardware units.

STARAN

The associative memory may be extended with a rewrite function so that each word appears to be a processor. Matched words may be partially rewritten using the comparand **C** and mask **H**. For each $n = 1, \ldots, i$, if $\mathbf{R}[n] = 1$, then for each $m = 1, \ldots, j$, if $\mathbf{H}[m] = 0$, $\mathbf{C}[m]$ is put into $\mathbf{M}[n; m]$. Searching and rewriting permit arithmetic and logical operations on all words simultaneously in a SIMD parallel computer such as STARAN (Batcher, 1974). A large number of programs have been written for STARAN, and highly parallel SIMD algorithms, such as those found in radar-signal processing, have been considerably faster in STARAN than in a conventional machine. But large database searches are not suited to STARAN because the time to load **M** dominates the time to search it; even if the search time were zero, STARAN is slowed to the speed of conventional machines by the time to load its memory.

SET SEARCH

A plethora of papers (Minker, 1971; Maryanski, 1980) have been written on the use of associative memories for searching databases. If the words are considered ordered (whereas they are unordered in the examples above), sequential addressing (eg, the notion of next lower word) can be combined with associative addressing. After a search that sets the result bit in **R**, a "next search" can be made, where only words below a word where the **R** bit was set are searched to set the **R** bit at the end of the search. A string of characters can be searched, one character after another, forming the basis for text-oriented information retrieval. Partitioning the rows into contiguous blocks and using a delimiter to mark the beginning of each block, a "set search" can be made, where only words in a block, where the **R** bit was set in some word in the block, can be searched to set the **R** bit at the end of the search. This is the basis of most relational, hierarchical, and network database machines.

ASSOCIATIVE DISKS

Although it may become feasible to build large associative memories using integrated circuits for the memory, associatively addressed disk memories have been proposed for large databases (Bray and Freeman, 1979). As the data on a disk track pass over a read head, searching may be accomplished essentially as described above. Using additional techniques (eg, storing the data on a track in RAM called a disk cache), data can be modified and later written back on the track. If each disk head has these capabilities, a large amount of data can be associatively searched where it is stored rather than transported to and from a large mainframe computer.

OUTLOOK

Finally, the associative memory continues to find new applications as new problems are studied. Associative memory integrated circuits have been sold for over 15 years now, but they have not been widely used because they were considerably slower and smaller than RAMs. However, custom VLSI is becoming commercially accepted, and an associative memory can be integrated with other processor logic and memory in these custom chips with less relative cost than in current systems. Also, considerable work is in progress in the design of database processors. In these systems the disk is made associative, as discussed above. A group of researchers in Japan and in the Microelectronics and Computer Corporation in the United States are competing to make these database machines commercially useful. These advances should contribute to better AI systems. For instance, in PROLOG (see LOGIC PROGRAMMING) the database can be searched, the Horne clauses can be matched, variables can be instantiated, and control can be effected using associative memories from associative disks to associative VLSI memories and associative controllers (scoreboards and PLAs).

BIBLIOGRAPHY

K. E. Batcher, "STARAN Parallel Processor System Hardware," *Proceedings of AFIPS-NCC* **43,** 405–410 (1974).

O. H. Bray and H. A. Freeman, *Data Base Computers*, Lexington Books, Lexington, Mass., 1979.

V. Bush, "As We May Think," *Atl. Mo.* **176**(1), 101 (1945).

K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984, pp. 57–80.

F. Maryanski, "Backend Database Systems," *ACM Comput. Surv.*, **12**(1), 3–27 (March 1980).

J. Minker, "An Overview of Associative Memory or Content-Addressable Memory Systems and a KWIC Index to the Literature," *Comput. Rev.*, 453–504 (Oct. 1971).

G. J. LIPOVSKI
University of Texas

ASSOCIATIVE NETWORKS. See ASSOCIATIVE MEMORY; SEMANTIC NETWORKS.

AUGMENTED TRANSITION NETWORKS. See GRAMMAR, AUGMENTED TRANSITION NETWORK.

AURA

AURA, an automated reasoning program developed in 1978, was used to answer a previously open question in algebra (Winker, 1978). Written in IBM assembly language, AURA was designed and implemented by Overbeek with contributions from Smith, Winker, and Lusk. Although the use of AURA led to a number of useful results in mathematics and logic, its lack of portability proved to be a serious disadvantage. LMA, a collection of subroutines written in Pascal for the purpose of implementing automated reasoning programs tailored to the user's specification, was designed and implemented to address the concern of portability (see RESOLUTION, BINARY; B. Smith, *Reference Manual for the Environmental Theorem Prover, an Incarnation of AURA*, Technical Report ANL-88-2, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1988; S. Winker and L. Wos, "Automated Generation of Models and Counterexamples and Its Application to Open Questions in Ternary Boolean Algebra," *Proceedings of the Eighth International Symposium on Multiple-Valued Logic*, IEEE, Rosemont, Ill., 1978, pp. 251–256.

L. WOS
Argonne National Laboratory

AUTOMATIC PROGRAMMING

Computer programming is the process of constructing executable code from fragmentary information. This information may come in many forms including vague ideas of how the output should look, the nature of the expected input, the type of algorithm to be used, and possibly, examples of the target behavior. The result of the programming is a section of code which is capable of receiving inputs from the target domain and processing them to yield appropriate outputs.

When computer programming is done by a machine, the process is called automatic programming. Artificial intelligence researchers are interested in studying automatic programming for two reasons: First, it would be highly useful to have a powerful automatic programming system which could receive casual and imprecise specifications for a desired target program and then correctly generate that program (Fig. 1*a*). Second, automatic programming is widely believed to be a necessary component of any intelligent system and is therefore a topic for fundamental research in its own right. Thus a system might discover a successful way to achieve a given result and program itself to achieve that result. For example, a legged being might assemble code to enable itself to walk after a series of walking experiences (Fig. 1*b*).

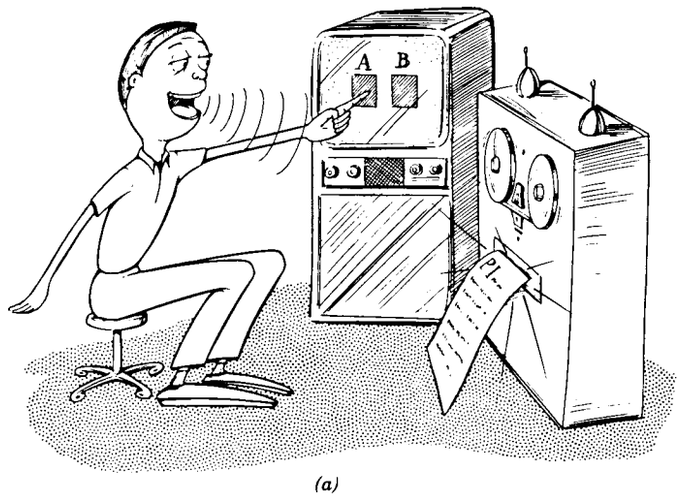
A number of approaches to automatic programming have been developed over the years and the most important ones will be described here. The following sections, as illustrated in Figure 2, describe methodologies for synthesis from formal input-output specifications, from examples of the desired program behavior, from natural language dialogue, and from cooperative interaction between a human programmer and a mechanical programmer's assistant.

The methodologies based upon synthesis from formal specifications utilize predicate calculus notation and derive the target program in a sequence of logical steps. Because the resulting program is mathematically derived from its specification, its correctness with respect to the specification is assured. Thus the methodologies are very attractive and their development has implications for the foundations of computer science as well as artificial intelligence.

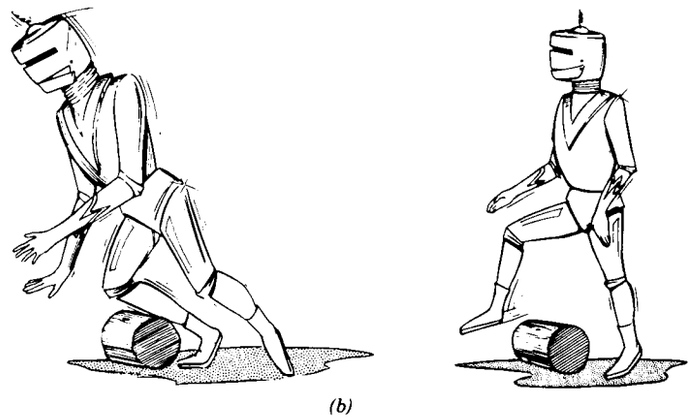
The synthesis-from-example methodologies involve generalization and learning behaviors. Since the examples do not completely specify the target program, the initially generated program may not achieve all of the desired behaviors. But the addition of appropriate examples can force the synthesizer to efficiently converge to a satisfactory program. The attractiveness of this approach comes partly from the ease with which a user can provide examples. These techniques are also important for artificial intelligence researchers to understand because they seem to be fundamental to certain kinds of intelligent behavior (see INDUCTIVE INFERENCE).

The third approach, program generation from natural language dialogue, involves translating informal descriptions into formal specifications which can be programmed using formal methods. This approach uses those technologies mentioned above as well as natural language processing, knowledge representation, and artificial intelligence systems design.

The final approach, program synthesis using an automated programmer's assistant, assumes that a human will be the primary programmer and that the proper role for a machine is to supplement his or her efforts. The human's role is to develop and refine a set of formal specifications and make some implementation decisions with the machine assisting by checking consistency, retrieving in-



(a)



(b)

Figure 1. Examples of automatic programming. (*a*) Generating the user's program. The computer automatically generates code from casual specifications. (*b*) Learning new behaviors. The intelligent being internally assembles code to enable itself to function in the world.

formation from libraries, and so forth. At some point in the specification process, the machine can take the primary role in selecting data structures and building the code and documentation for the software product.

The following sections thus describe these four approaches to automatic programming.

SYNTHESIS FROM INPUT-OUTPUT SPECIFICATIONS

Stating the Problem

The first approach to the study of automatic programming assumes that a specification of the input-output behavior is given and that the automatic system is to find a program to implement the specification. Very often the specification is written as:

$$\forall a(P(a) \Rightarrow \exists z R(a,z))$$

Here $P(a)$ is an input predicate that is true if and only if a is an acceptable input for the target program. $R(a,z)$ is an input-output relation that is true if and only if z is the desired output when the target program reads a as its input. The specification states that for all a such that the input requirement P is met, there is a z such that the input-output relation $R(a,z)$ holds. The program synthesis will proceed by proving this theorem. It turns out that in order to complete the proof, the theorem prover must constructively find the z that is asserted to exist and the method for finding z is the target program (see also THEOREM PROVING).

As an illustration, suppose it is desired to automatically generate a program to add up a list of numbers a to obtain their sum z. Then one should define the input predicate $P(a)$ to be true if and only if a is a list of numbers of length zero or more. Thus $P((4\ 2\ 9))$ is true and $P((A(B)))$ is false. The input-output relation $R(a,z)$ should be defined to be true if and only if z is the sum of the numbers in a. For example, $R((4\ 2\ 9),15)$ is true and $R((4\ 2\ 9),16)$ is false (z is zero if a has length zero). Then the proof of the specifying theorem,

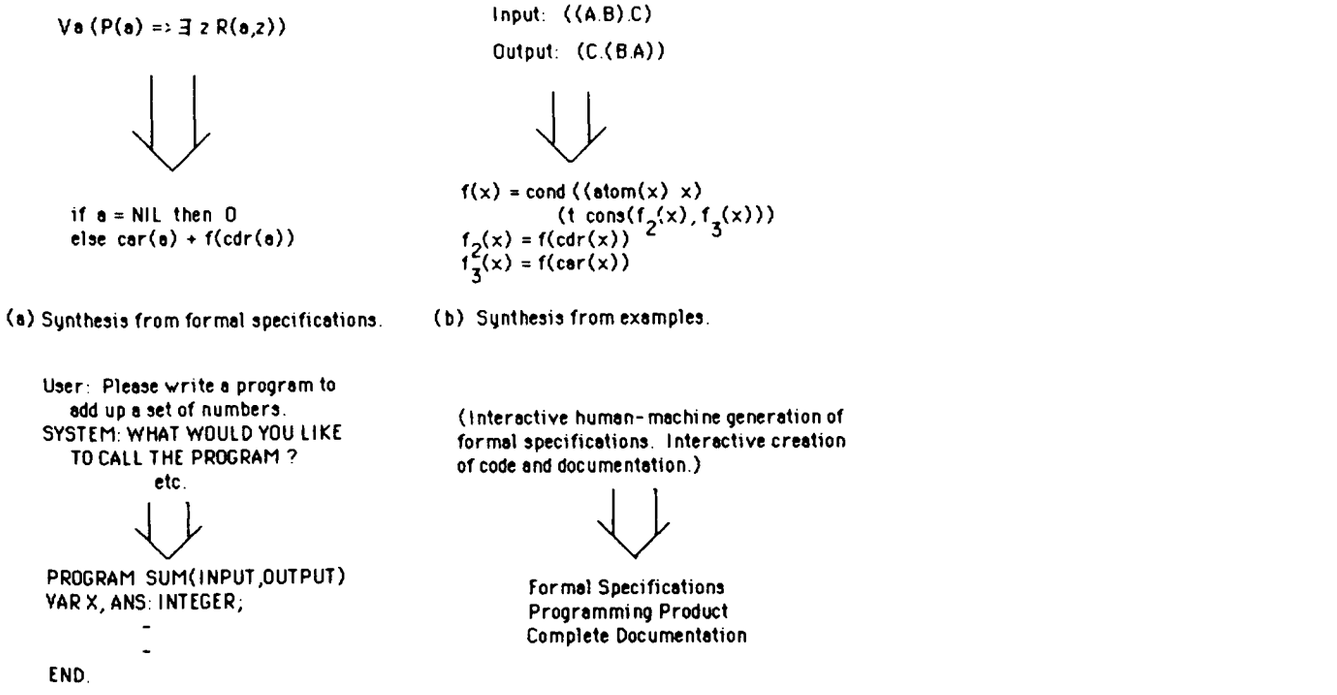


Figure 2. Four research areas in automatic programming.

$$\forall a(P(a) \Rightarrow \exists z R(a,z))$$

requires the system to find a way of constructing z for every acceptable a. This proof will thus yield the target program.

Of course, it is not possible to carry out the proof unless many facts are known about P and R. For example, one needs the fact that if a is the list of length zero, that is a = NIL, Then $R(a,0)$ is true.

$$R(a,0) = \text{true if } a = \text{NIL}.$$

Also, let $car(a)$ be defined to be the first element of list a and let $cdr(a)$ be defined to be list a with its first element removed. The synthesis will use the fact that the sum of a is simply obtained by adding $car(a)$ to the sum of $cdr(a)$ if a is not NIL.

$$R(a, car(a) + z') = \text{true if } R(cdr(a),z') \text{ and not } (a = \text{NIL})$$

The program synthesis methodology uses these kinds of facts to prove the above theorem and generate the following program:

$$f(a) = \text{if } a = \text{NIL then } 0 \quad \text{else } car(a) + f(cdr(a))$$

The proof of the specifying theorem will involve manipulation of formulas of the form:

$$\text{if } \forall x A_1(x) \text{ and } \forall x A_2(x) \text{ and } \cdots \text{ and } \forall_x A_n(x) \text{ then}$$
$$\exists x G_1(x) \text{ or } \exists x G_2(x) \text{ or } \cdots \text{ or } \exists x G_m(x)$$

Following the methodology of Manna and Waldinger (1980), such formulas can be written into a tableau of columns labelled assertions, goals, and outputs as follows.

Assertions	Goals	Outputs
$A_1(x)$		
$A_2(x)$		
.		
.		
.		
$A_n(x)$		
	$G_1(x)$	
	$G_2(x)$	
	.	
	.	
	.	
	$G_m(x)$	

This notation enables one to write such formulas omitting quantifiers and connectives. If an entry $t_i(x)$ appears in the output column in the same row as some goal $G_i(x)$, that output $t_i(x)$ expresses the desired program output when $G_i(x)$ is true.

Using this tabular convention, the specifying theorem can be written as follows:

Assertions	Goals	Outputs
$P(a)$		
	$R(a, z)$	z

The Manna and Waldinger (1980) program synthesis procedure involves adding rows to this table such that the

correctness of the above meaning formula is maintained. If a goal can be deduced which is always true and such that its corresponding output entry is in terms of the input and primitive functions, that output entry will be the target program:

Assertions	Goals	Outputs
	True	Target program

The Deductive Mechanism

Once the problem is stated and appropriately entered into the above table, methods are needed for deducing new entries in the table so that progress toward the target program can be made. Following the methodology of Manna and Waldinger (1980), two kinds of rules are introduced here, transformations which convert portions of an assertion or goal to a new form, and resolution rules which allow one to combine assertions and/or goals to obtain new assertions or goals.

Transformations have the form:

$$r \Rightarrow s \text{ if } Q$$

which means that r may be converted to s if Q is true.

In order to illustrate usage, suppose there is a goal G that contains a subexpression r. It can be written as $G[r]$. Then the transformation $r \Rightarrow s$ yields $G[s]$, and this substitution can be made if Q is true.

This gives a way to generate a new goal in the deductive table if $G[r]$ is an existing goal. The new goal is $G[s]$ and Q. It means that if $G[r]$ is a valid goal, then $G[s]$ is one also provided that Q is true. In terms of the deductive table, the transformation $r \Rightarrow s$ if Q enables one to begin with:

Assertions	Goals	Outputs
	$G[r]$	p

and deduce the new entry:

Assertions	Goals	Outputs
	$G[s]$ and Q	p

An example of a transformation is:

$$U(x,-x) \Rightarrow \text{true if } x < 0$$

and its usage can be shown on the following example goal and output:

Assertions	Goals	Outputs
	$U(b,y)$	y

The transformation can convert $U(x,-x)$ to true, but this expression does not occur in the given goal. However, one can make substitutions into the transformation and the goal and output so that the transformation is applicable; that is, unification is being used as described elsewhere in this volume. Specifically, one can substitute $x = b$ into the transformation to obtain:

$$U(b,-b) \Rightarrow \text{true} \qquad \text{if } b < 0$$

and one can substitute $y = -b$ into the goal and output to obtain:

Assertions	Goals	Outputs
	$U(b,-b)$	$-b$

Now the transformation can be applied using the above rule to obtain a new row in the tableau.

Assertions	Goals	Outputs
	True and $b < 0$	$-b$

Another way to construct new entries in a deductive table is to resolve (see RESOLUTION) two goals to yield a new one. Suppose one has two goals F with associated output p_1 and G with associated output p_2. Further suppose that F and G both have the same predicate subexpression e so they will be written $F[e]$ and $G[e]$.

Assertions	Goals	Outputs
	$F[e]$	p_1
	$G[e]$	p_2

In the following, the notation $F[e \leftarrow \text{true}]$ stands for the goal F with subexpression e replaced by true. $G[e \leftarrow \text{false}]$ is similarly defined. Then the two goals $F[e]$ and $G[e]$ can be combined to obtain a new goal $F[e \leftarrow \text{true}]$ and $G[e \leftarrow \text{false}]$ with associated output if e then p_1 else p_2. The new row in the tableau is:

Assertions	Goals	Outputs
	$T[e \leftarrow \text{true}]$ and $G[e \leftarrow \text{false}]$	if e, then p_1 else p_2

An example of such a resolution occurs if one has these goals.

Assertions	Goals	Outputs
	$a < 0$	$-a$
	Not $(a < 0)$	a

Then resolution of these two goals where e is $a < 0$ yields the following new entry:

Assertions	Goals	Outputs
	True and not (false)	If $a < 0$, then $-a$, else a

Two kinds of deductive rules have been described here: transformations and goal-goal resolutions. Manna and Waldinger (1980) have given numerous other deductive rules but these examples illustrate the nature of the technique. Once the problem is properly represented and rules

are available for deduction for new table entries, one can proceed to synthesize programs as shown in the next section.

Synthesizing Programs

The program synthesis procedure follows the problem solving paradigm so well known in the AI community. An initial state is given and transitions are available for moving from one state of the domain to another. The problem solving system attempts to find a sequence of applicable transitions that will transform the world from the initial state to an acceptable final state.

In the program synthesis domain, the initial state is the specification of the program input-output characteristics. The applicable transitions are the transformations, resolution schemes, and other available rules for deducing new forms from the original specification. An acceptable final state is one that gives a program in a machine executable language which meets the original specification. In terms of the deductive table, a final state has a goal which is true and an associated output in terms of the input and primitive machine operations.

One can illustrate the whole process by synthesizing a program to compute the absolute value function h:

$$h(x) = \begin{cases} x \text{ if not } (x < 0) \\ -x \text{ otherwise} \end{cases}$$

Then the input specification must require x to be a real number: $V(x)$ is true if and only if x is real. The input-output relation $U(x,z)$ will be true whenever z is the absolute value of x. This information must be available to the system in the form of transformations:

T1: $U(x,-x) \Rightarrow$ true if $x < 0$

T2: $U(x,x) \Rightarrow$ true if not $(x < 0)$

The synthesis proceeds by applying the available transformations and deductive rules until a program is synthesized.

Following the methodology described above, one begins with the original specification:

Assertions	Goals	Outputs
$V(b)$		
	$U(b,y)$	y

Applying T1 and T2 to the goal $U(b,y)$ obtains the following two goals:

Assertions	Goals	Outputs
	$b < 0$	$-b$
	Not $(b < 0)$	b

Resolving these two goals results in the final program:

Assertions	Goals	Outputs
	True	If $b < 0$, then $-b$, else b

A more interesting example appears in Figure 3 where a program is generated to add a list of numbers. In this case, $P(a)$ is true if and only if a is a list of integers of length zero or more. $R(a,z)$ is true if and only if z is the sum of the numbers in a. If a has length zero, then $R(a,0)$ is true. Two transformations carry the critical information:

T3: $R(a,0) \Rightarrow$ true if a = NIL

T4: $R(a,car(a) + z') \Rightarrow$ true if $R(cdr(a),z')$ and not $(a$ = NIL)

Step 5 is the inductive hypothesis for an inductive proof which introduces a looping behavior into the synthesis. It states that the synthesized program f works properly (ie, if

step	assertions	goals	outputs	remarks
(1)	P(a)			input spec.
(2)		R(a,z)	z	i-o relation
(3)		a = NIL	0	T3 on (2)
(4)		R(cdr(a),z') and not (a = NIL)	car(a) + z'	T4 on (2)
(5)	if v < a then if P(v) then R(v,f(v))			inductive hypothesis
(6)		not(if cdr(a) < a then if P(cdr(a)) then false) and not (a = NIL)	car(a) + f(cdr(a))	resolving (4) and (5)
(7)		not (a = NIL)	car(a) + f(cdr(a)	simplifying (6)
(8)		true	if a = NIL then 0 else car(a) + f(cdr(a))	resolving (3) and (7)

Figure 3. Synthesizing the program to add a list of numbers.

$P(v)$ then $R(v,f(v))$) for all lists shorter than the input a (ie, $v < a$). The proof that f then also works on a completes the inductive argument and enables the introduction of recursion in the generated program. Step 6 is a goal-assertion resolution that functions similarly to the goal-goal resolution above. The final synthesized program is:

$$f(x) = \text{if } x = \text{NIL then 0 else car}(x) + f(cdr(x)).$$

Manna and Waldinger (1987) show a detailed example of these mechanisms in the generation of binary search programs.

Searching for Loops

One of the central problems in program synthesis is the discovery of loops and information is often available to help find them. Bibel (1980) and Bibel and Hornig (1984), for example, suggest breaking the input into parts and attempting to discover the contribution of the parts to the output. If the synthesizer can find a way to do one part of the calculation on a pass through the body of the loop, then perhaps repeated passes will enable it to consume the rest of the input and complete the target output.

This strategy is easy to illustrate on the example of Figure 3, adding up a list of numbers. In this example, the input is a list of numbers, and Bibel and Hornig (1984) suggest that one should guess at a way to break this input into parts. If a method can be found to isolate the effects of the parts on the result of the computation, perhaps code can be found to process the parts individually and combine the results to yield the answer. An example is the method of splitting input a into $car(a)$ and $cdr(a)$. Observing the function definition for f, we see

$$f(a) = \sum_{i=1}^{length\ (a)} a[i]$$

and this can be separated to observe the effects of $car(a)$ (or $a[1]$) and $cdr(a)$ (or $a[2]$ . . . , $a[\text{length}(a)]$).

$$f(a) = a[1] + \sum_{i=2}^{length\ (a)} a[i]$$

But the last term of this expression is simply the function f applied to the rest of the list, $cdr(a)$:

$$f(a) = a[1] + f(cdr(a))$$

So the program for addition needs to only add its first entry to the result of applying itself to the rest of the list. Then if we include code to handle the trivial case, the program can be completed:

$$f(a) = \text{if length } (a) = 0 \text{ then } 0$$
$$\text{else } car(a) + f(cdr(a))$$

The reason to try to program such mechanisms is that they may automatically discover the strategy for making a loop work and eliminate the requirement that knowledge of the type represented by T3 and T4 be explicitly coded in the system.

If the synthesizer had guessed another way to break up the input, another algorithm might result. Thus one could break a into $a[1], \ldots, a[\text{length}(a)/2]$ and $a[\text{length}(a)/2 + 1], \ldots, a[\text{length}(a)]$ and find an algorithm that recursively calls f on both the first and second halves of its input list until a trivial list of length 0 or 1 is found. Bibel and Hornig (1984) implemented this methodology in their LOPS system and demonstrated it on many examples. Their system included many features including methodologies for guiding the search to obtain a final program. One of its more novel mechanisms generated examples in the problem domain and generalized from them to produce hypothesized theorems to be used in the synthesis.

Other work using this insight is described by Smith (1987) on the automatic construction of divide and conquer algorithms. This work includes an interesting study of sorting methods and shows how a variety of algorithms result from different initial decompositions of the input.

Transformational Methodologies

A type of transformation different from the transformations discussed above has been investigated in the literature (Bauer and co-workers, 1989; Broy, 1983; Burstall and Darlington, 1977; Gerhart, 1976; Manna and Waldinger, 1979). In this methodology, the original specification of the program is transformed repeatedly until a final program is derived. Such transformations have the form shown in Figure 4. An initial specification gives the format of the schema to be modified. An applicability condition specifies all relationships that are prerequisite to the use of the transformation. A final specification gives the new format for the schema after the application of the transformation. Program synthesis involves the repeated application of such transformations, beginning with the initial specification, until the final program is created.

The approach assumes the existence of a library of such transformations and synthesis proceeds by automatically or manually selecting the sequence to be used. Some transformations may perform rather small changes to the existing form while others may create dramatic changes and enable great leaps toward the final product.

An example of a transformation for creating simple linearly recursive code from a specification appears in Figure 5. This transformation can solve many standard synthesis examples including the list summing problem mentioned above. For this example, one needs to discover the substitutions $C(x)$ is $\text{length}(x) = 0$, $T(x) = 0$, $H_1(x) = car(x)$, $H_2(x) = cdr(x)$, and $G(x,y) = x + y$. With these substitutions, the applicability conditions hold, and the final program can be created in a single step.

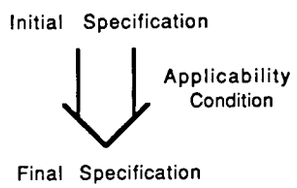


Figure 4. The format for a program transformation.

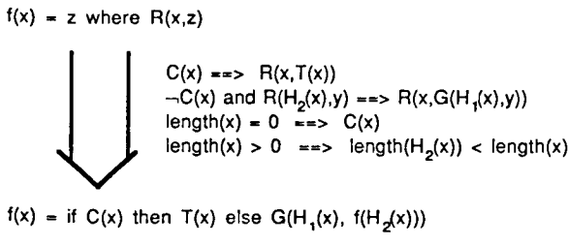


Figure 5. A program transformation to create a recursive program.

Historical Remarks

It was discovered in the late 1960s (Green, 1969; Waldinger and Lee, 1969) that the proof of a theorem with existential quantifiers will implicitly contain the sequence of operators required to find the objects asserted to exist. This sequence of operators can be considered to be a program for finding those objects and the discovery became the basis for much research in automatic programming. The task of theorists then became one of finding methods to extract the operators from the proofs and developing theorem proving strategies which would properly introduce the desired looping, branching, and subroutine constructions into the code. The methodology of Manna and Waldinger (1980, 1987) systematizes and generalizes on the techniques that had been developed earlier.

Simultaneously, other work in program synthesis has shown how program loops are formed as described above and has created the transformational synthesis methodology. The total of this literature has had a substantial effect on the theory of programming and on software engineering.

PROGRAM SYNTHESIS FROM EXAMPLES

This section will first give an algorithm for synthesizing flowcharts from traces and an example of its usage, the generation of a Turing machine program to sort A's and B's. Then it shows how this leads to a program synthesis methodology for LISP code. Also methods will be given for creating LISP programs from recurrence relations and for factoring behavior graphs to create real time programs. The last methodology shows how PROLOG programs can be created from example behaviors. The section concludes with some general observations regarding program synthesis from examples.

Constructing Flowcharts from Example Traces

Suppose a computer program has executed the following computation trace in completing a particular calculation.

Time	Condition	Instruction
1		I_1
2	a	I_2
3	b	I_2
4	c	I_2
5	b	I_2
6	b	I_3

That is, at the first instant of time, I_1 was executed. Then condition a was tested and found to be true and I_2 was executed. This proceeds until the final statement at time 6 when I_3 was executed. The instructions I_j may be any instructions such as READ(x) or $x = x + 1$ and the conditions a, b and c may be any predicates such as $x > 3$ or $x < y$. It is desired to find an algorithm capable of building a program that can do this trace.

One can, in fact, begin building the desired program by starting at the beginning of the trace and moving downward building the code to account for each step. From the first instruction (Time = 1), the beginning of the target program can be created (Fig. 6a). Next Condition a is observed and Instruction I_2 executed. I_2 is added to the flowchart (Fig. 6b). The trace indicates the next condition is b out of I_2 and this leads to another execution of I_2. Since an example of I_2 already exists, this transition is sent to it (Fig. 6c). Examining Condition c from this instruction to I_2 at Time 4, again this transition is sent to the existing version of I_2 (Fig. 6d). At Time 5, the trace indicates a b transition to I_2 and the existing program already has such a b transition to I_2. Whenever the current trace condition matches the condition on an outgoing transition of the active node of the program, one has what is called a forced move. If the current version of the program is correct, as it is here, that transition will properly predict the next instruction in the trace. At Time 6, however, a contradiction occurs: the move is forced and the program uses the b transition to predict the next instruction to be I_2, but, the trace indicates the next instruction should be I_3. Apparently an error has occurred in the synthesis.

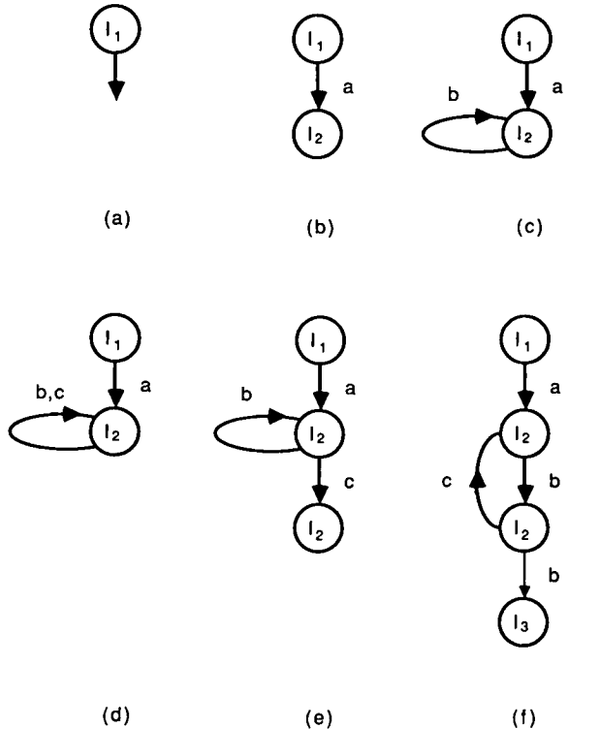


Figure 6. The generation of a program to execute the example trace.

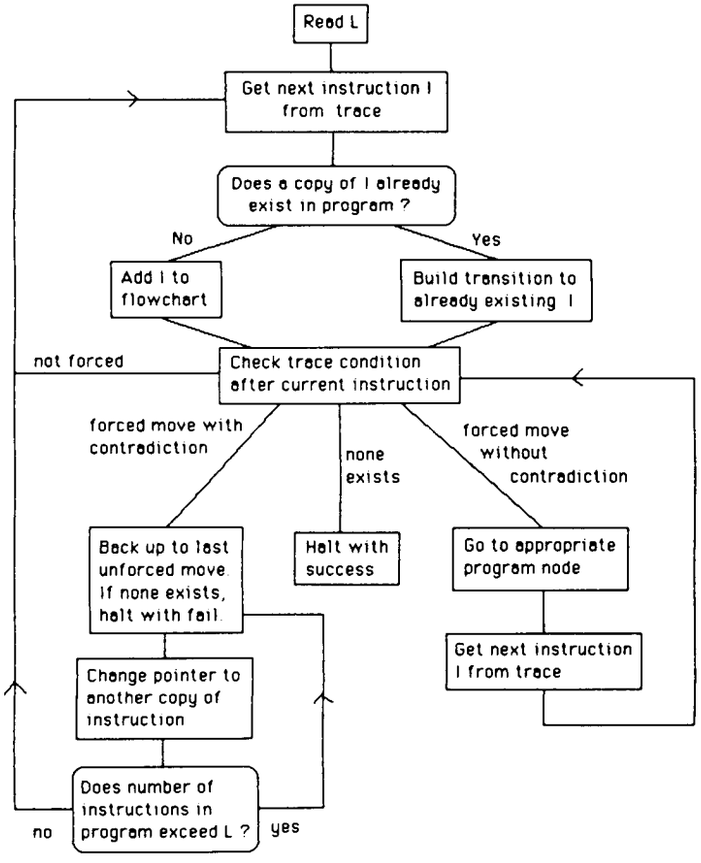


Figure 7. The algorithm for program creation from trace information.

At this point, the procedure backs up to the last unforced move and changes its decision. Returning to Time 4, the c transition will not be directed backward but instead to a new copy of I_2 (Fig. 6e). Moving forward again, a second contradiction will be found, another back up, and then the final flow chart of Figure 6f will be built.

A general algorithm for creating flowcharts from traces appears in Figure 7. This algorithm requires the user to specify a limit L on the number of nodes to appear in the target program and will always find a program of that size or less which can execute the trace. If none exists, backup will occur to the first instruction and above resulting in a termination with failure. The algorithm can then be restarted with a larger L. The following section shows some applications of the method.

The Trainable Turing Machine

An interesting application of this methodology is the trainable Turing machine as described by Biermann (1972) (Fig. 8). This system resembles the traditional model of a Turing machine with an infinite two-way tape and a tape-head that can move left or right reading and writing symbols. This machine differs from the traditional model in that it begins without any finite-state controller; it builds its own controller on the basis of examples.

Suppose, as an illustration, it is desired to train the machine to sort a string of A's and B's so that the A's all precede the B's. In order to train the machine to do this calculation, one must put the machine in learn mode and physically force its read-write head through one or more sample computations. These examples will enable the machine to construct a general program, using the algorithm of the previous section, to sort any string of A's and B's. A simple example calculation appears in Figure 9.

The algorithm of Figure 9 starts the head at the left end of the nonblank symbols of the tape. It moves right in search of an A, and, on finding it, prints a B and moves left. When it finds the end of the string or another A, it moves right one step, prints the A, and moves right again looking for another A. It continues moving A's back toward the front until no more symbols, A's or B's, are found.

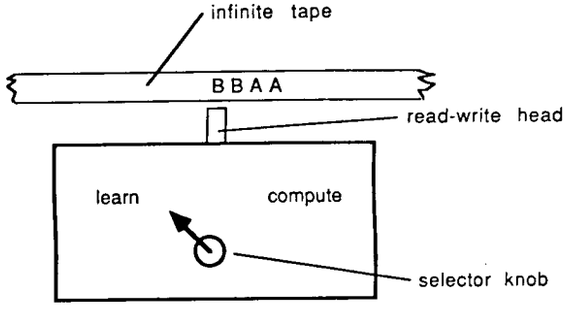


Figure 8. The trainable Turing machine.

Time	Current Tape	Condition	Instruction
1	BBAA		start
2	BBAA	B	BR
3	BBAA	B	BR
4	BBAA	A	BL
5	BBBA	B	BL
6	BBBA	B	BL
7	_BBBA	ø	øR
8	BBBA	B	AR
9	ABBA	B	BR
10	ABBA	B	BR
11	ABBA	A	BL
12	ABBB	B	BL
13	ABBB	B	BL
14	ABBB	A	AR
15	ABBB	B	AR
16	AABB	B	BR
17	AABB	B	BR
18	AABB_	ø	halt

Figure 9. The trace of a Turing machine computation to sort a string of A's and B's.

To the right of the tape configurations in Figure 9, the symbols being read are shown in the column labelled condition. These correspond to the conditions described in the algorithm synthesis method of the previous section. The instructions to the tape head give the symbol to be printed: A, B, or blank, and the direction of the next head move, left L or right R. The combination of these two columns is the needed information for the synthesizer of Figure 7. In fact, when it is executed, the controller of Figure 10a results. It is the proposed program for sorting any string of A's and B's into ascending order. This program has a small bug in that it will not sort a string that begins with an A. The omission can be removed if one more example is processed, the sorting of the list AA, and the result appears in Figure 10b. The machine can now be put into compute mode and used to sort any string of A's and B's.

The methodology is algorithmic and capable of creating any Turing machine. For example, Biermann, Baum, and Petry (1975) showed that it can create a universal Turing machine from a single example computation. The major problem with the approach is its high execution time. A variety of strategies were developed by Biermann, Baum, and Petry (1975) for speeding it up.

The algorithm for synthesis has had many other applications. Another example is the trainable desk calculator of Biermann and Krishnaswamy (1976) that enables a user to do hand calculations at a machine display with a light pen and which creates programs from the resulting traces. Using this system, many programs have been generated including various matrix manipulation routines, sorting programs, a finite state machine minimizer, and a compiler for a small ALGOL-like language. In a sequel to this work, Biermann (1978) showed that certain types of instructions for indexing can be omitted from such example traces while still allowing for correct program synthesis.

An application for this approach was also developed by Waterman and co-workers (1984) in the construction of a

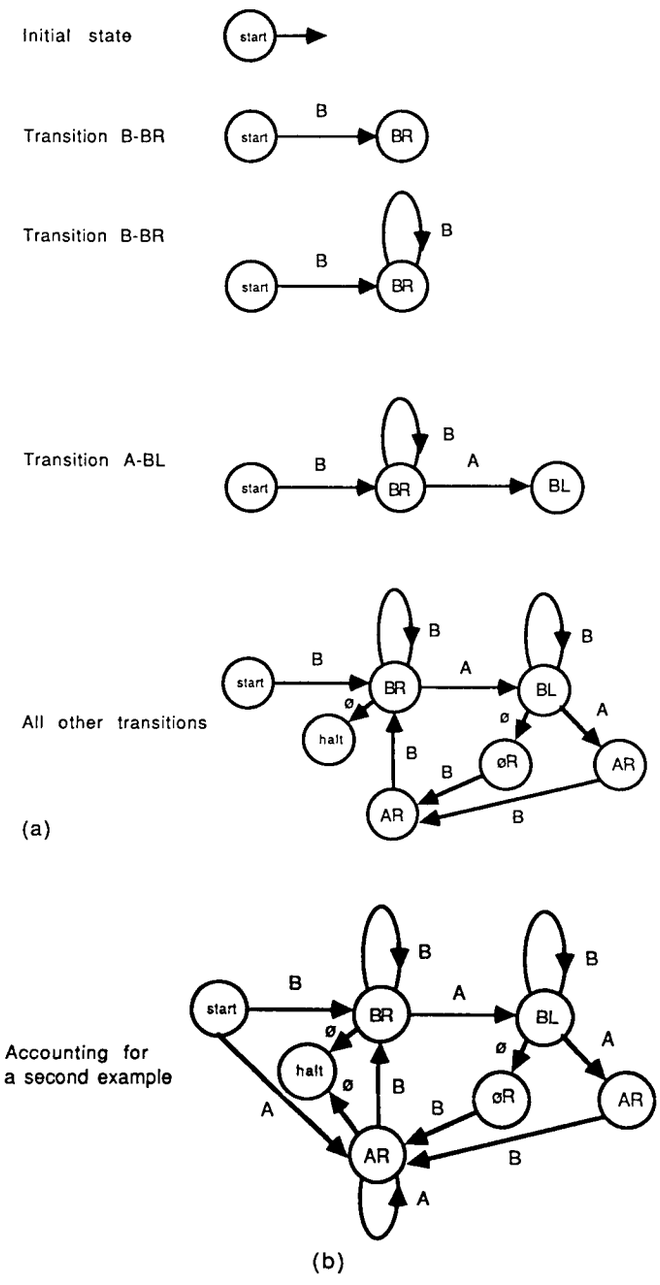


Figure 10. (*a*) Constructing the Turing machine controller to sort A's and B's; (*b*) completing the controller.

programmer's helper called EP. This system observes the user in typical daily applications and builds programs to automatically mimic the user behaviors. Then when repetitive tasks occur, the user can release control to the machine which has built code to finish them automatically. Fink and Biermann (1986) used the idea to create a dialogue acquisition mechanism in a natural language processor.

Synthesis of LISP Code

A popular area for research in recent years (Smith, 1984) has centered around the creation of LISP programs from

examples of their behaviors. Thus, one might be given the fact that the input x = ((A . B) . C) is to yield output z = (C .(B . A)). The goal is to construct a LISP program that is capable of executing this and all similar examples.

The synthesis of the code begins with the discovery of the LISP operations required to yield the output from the input.

$$z = cons(cdr(x), cons(cdr(car(x)), car(car(x))))$$

Here $cons(u,v)$ is defined to be the dotted pair $(u \cdot v)$. $car(x)$ and $cdr(x)$ are defined to be the left and right sides of dotted pair x. Thus, if $x = (A.B)$ then $car(x) = A$, $cdr(x) = B$, and $cons(x,x) = ((A.B).(A.B))$. The breakdown of output z in terms of primitive functions is unique and easy to find. Furthermore, it corresponds to the trace of instructions employed in the previous examples. In fact, the concatenated operators can be broken apart into primitives in preparation for the program synthesis:

$$\begin{aligned} z &= f_1(x) \\ f_1(x) &= \text{cons}(f_2(x),f_3(x)) \\ f_2(x) &= f_4(cdr(x)) \\ f_3(x) &= f_5(car(x)) \\ f_4(x) &= x \\ f_5(x) &= cons(f_6(x),f_7(x)) \\ f_6(x) &= f_8(cdr(x)) \\ f_7(x) &= f_9(car(x)) \\ f_8(x) &= x \\ f_9(x) &= x \end{aligned}$$

Here the flowchart construction methodology can again be applied; instead of merging separate instructions from the trace, however, different f_i's will be merged. Also, conditional tests need to be created during the synthesis procedure.

The process of merger is made possible by the existence of the cond operator in LISP which is written as follows:

$$\text{cond}((p_1g_1)(p_2g_2)(p_3g_3) \ . \ . \ . \ (p_ng_n))$$

The predicates p_1, p_2, . . . , p_n are evaluated sequentially and when the first p_j is found that evaluates to true, cond returns g_j as its value. LISP has some built–in predicates such as atom(x) which is defined to be true if and only if x is a LISP atom. Suppose x is a dotted pair of the atoms A and B (ie, $x = (A.B)$) then:

$$\begin{aligned} f(x) = \text{cond}&((\text{atom}(x)\ \text{car}(x)) \\ &(\text{atom}(car(x))\ cdr(x)) \\ &(\text{T}\ cons(x,x))) \end{aligned}$$

will evaluate to B. That is, atom(x) is false, and atom ($car(x)$) is true, so $cdr(x)$ is returned as the result.

One can thus see how the cond operator can be used to merge functions from the above trace. Suppose, for example, it is desired to merge $f_1(x)$ and $f_4(x)$ and that a predicate generator has discovered that function f_4 should be selected if x is an atom. Then f_1 and f_4 can be merged to produce f.

$$f(x) = cond((atom(x)\ x)(\text{T}\ cons(f_2(x), f_3(x))))$$

In fact, the flow chart synthesis procedure with automatic predicate generation can merge f_1, f_4, f_5, f_8, and f_9 to produce f as shown. It will also merge f_6 into f_2 and f_7 into f_3 leaving them unchanged:

$$f_2(x) = f(cdr(x)) \qquad f_3(x) = f(car(x))$$

The combination of the three functions f, f_2, and f_3 comprise a program that will achieve the target behavior. In fact, it will reverse any LISP S-expression of any level of complexity. Thus a complete program for reversing LISP S-expressions has been synthesized from one example.

This function merging technique is capable of generating any member in the class of regular LISP programs (Biermann, 1978). These programs include most LISP functions that have only one parameter, no auxiliary variables, and use only the atom predicate. The process reliably generates programs from randomly selected examples and always converges to a correct regular program if one exists and if enough examples are given. Its main disadvantage is that it is a searching procedure which becomes very expensive to execute if the target program is large.

LISP Synthesis Using Recurrence Relations

Summers (1977) has developed a LISP synthesis methodology based upon the discovery of recurrence relations in a sequence of examples. This methodology has the advantage that it creates programs more quickly than the above method but it also requires more carefully constructed training examples. Suppose it is desired to create a program which will delete the negative numbers in a list. One might present the system with these examples:

Example	Input	Output
1	NIL	NIL
2	(2)	(2)
3	(−1,2)	(2)
4	(0,−1,2)	(0,2)
5	(1,0,−1,2)	(1,0,2)
6	(−2,1,0,−1,2)	(1,0,2)

The Summers method involves discovering relationships between the sequential examples and the construction of single loop recursive programs to implement them. One can begin by writing each output in terms of its corresponding input using LISP primitives:

Example	Input	Output
1	NIL	$f_1(x)$ = NIL
2	(2)	$f_2(x)$ = *cons*(*car*(*x*), NIL)
3	(−1,2)	$f_3(x)$ = *cons*(*car*(*cdr*(*x*)), NIL)
4	(0,−1,2)	$f_4(x)$ = *cons*(*car*(*x*), *cons*(*car*(*cdr*(*cdr* (*x*))),NIL))
5	(1,0,−1,2)	$f_5(x)$ = *cons*(*car*(*x*), *cons*(*car*(*cdr*(*x*)), *cons*(*car*(*cdr*(*cdr*(*cdr* (*x*)))), NIL)))
6	(−2,1,0,−1,2)	$f_6(x)$ = *cons*(*car*(*cdr*(*x*)), *cons*(*car*(*cdr*(*cdr*(*x*))) *cons*(*car*(*cdr*(*cdr*(*cdr* (*cdr*(*x*))))), NIL)))

Although this step was quite straightforward, the next one requires a key discovery. Specifically, if one studies each f_i in the above sequence, it can be seen that each can be rewritten in terms of the previous f_i in a very systematic way. In fact, the following pattern arises:

Example	Input	Output
1	NIL	$f_1(x)$ = NIL
2	(2)	$f_2(x)$ = *cons*(*car*(*x*),f_1(*cdr*(*x*)))
3	(−1,2)	$f_3(x)$ = f_2(*cdr*(*x*))
4	(0,−1,2)	$f_4(x)$ = *cons*(*car*(*x*),f_3(*cdr*(*x*)))
5	(1,0,−1,2)	$f_5(x)$ = *cons*(*car*(*x*),f_4(*cdr*(*x*)))
6	(−2,1,0,−1,2)	$f_6(x)$ = f_5(*cdr*(*x*))

Examining this sequence, one can see two recurrence relations arising consistently:

$$f_i(x) = cons(car(x), f_{i-1}(cdr(x)))$$

and

$$f_i(x) = f_{i-1}(cdr(x)).$$

One can then run a test generation procedure on the inputs to determine when each recurrence relation is appropriate. In fact, it is easy to discover that if x is an atom, $f_i(x)$ is NIL, and if the first entry of x is negative then $f_i(x)$ takes the second recurrence form given above. Otherwise, $f_i(x)$ takes the first form given. The synthesized program is thus:

$$f(x) = \text{cond}((\text{atom } (x)\ \text{NIL})\ (\text{neg}(car(x))\ f(cdr(x)))\ (T\ cons(car(x), f(cdr(x)))))$$

Summers has given a basic synthesis theorem that specifies the nature of the required recurrence relations and a recursive program schema that will implement the observed recurrences. He also shows a fascinating strategy for introducing new auxiliary variables into the synthesized program if they are needed.

Synthesizing Programs Using Factorization of the Behavior Graph

Fahmy (1988) has given an algorithm for creating real time acceptors using a graph factorization technique. The procedure begins with a set of examples of the target behavior and assembles these into a behavior graph. Then the behavior graph is factored into two graphs, a finite-state controller and a data structure graph. The data structure may be any from a library of such structures and the factorization technique seeks one that will yield a successful decomposition.

Suppose, as an illustration, one would like to create an acceptor for strings of the form a^nb^n where n could be any natural number. Specifically, it is desired that a small set of examples of the target behavior be input to the synthesis system and the system should print a program to recognize the whole set. This problem can be better understood if one writes down a few examples of the behavior and then creates the smallest possible flowchart that will accept them. If the examples are *ab, aabb, aaabbb,* and *aaaabbbb*, then the flowchart for the acceptor is shown in Figure 11. Clearly if a larger number of examples had been included, the graph would extend farther to the right. If the whole language were to be represented, the graph would extend infinitely far.

The next concept in this theory is of the data structure as a graph. Two examples appear in Figure 12. The first is a counter: it begins in the 0–state, moves right on each increment instruction, and moves left on each decrement. The second data structure is a stack: it starts in the empty state and each push of an *a* or *b* causes it to move to the appropriate lower state. A pop instruction causes it to move upward to a previous state.

The third idea in this theory is the concept of a program as a finite state controller. This idea is used in several of the approaches given above and is not new. Figure 13 shows two finite-state controllers that are capable of recognizing a^nb^n. One uses the counter data structure and the other uses the stack. A computation begins with the controller in its initial state. The input string is said to be accepted if the controller is in a final state when the last symbol has been processed. Otherwise the input is not accepted.

It is easy to see how these controllers work. At each instant of time, a symbol is read from the input, a condition is checked on data structure, and a transition in the controller is made. This transition results in an instruc-

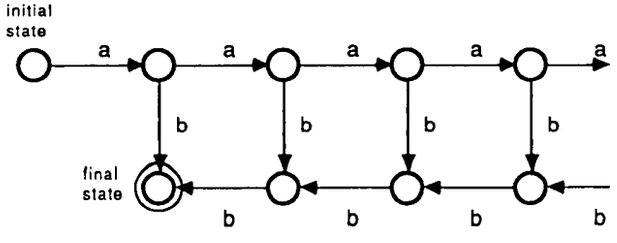
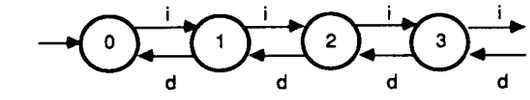
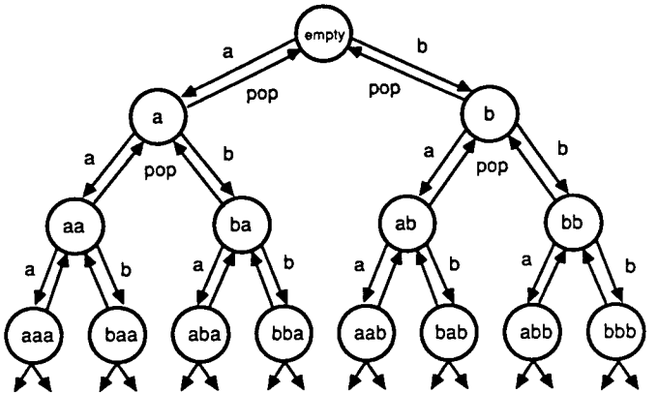


Figure 11. The behavior graph that accounts for the computations of a^nb^n.

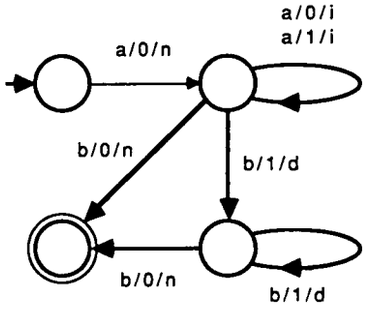


(a) Counter

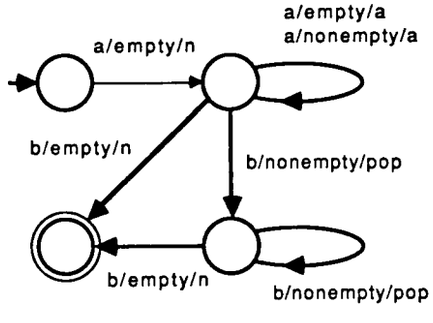


(b) Pushdown stack

Figure 12. Representing data structures with graphs.



(a) Controller using a counter.



(b) Controller using a stack.

Figure 13. Two controllers for the computation of $a^n b^n$.

tion being sent to the data structure possibly changing its state. As an illustration, one can follow the acceptance of string *aabb* by the first controller in Figure 13. The first *a* will be received with the controller in the initial state and the counter in state *0*. The controller transition asserts that the move should be to the state on the right with no instruction *n* sent to the counter. Next a second *a* is read and the transition *a/0/i* tells the controller to remain in that state and increment the counter to 1. Next a *b* will arrive moving the controller to another state and decrementing the counter to *0*. Finally, the next *b* will move the controller to the final state. The string *aabb* has been accepted.

The Fahmy (1988) theory of synthesis is based upon the discovery that the graph product of the control structure C and data structure D shown above is the behavior graph B: $B = C \times D$. This graph product is defined as follows: For each state c in C and each state d in D, create a state (c,d) in B. If c has transition $a/b/e$ to state c', and d in condition b has transition e to d', then state (c,d) in B should have transition a to state (c',d').

The synthesis procedure receives a set of strings of the target behavior, all of the strings of some length j or less. Then it builds the behavior graph B and attempts to find a data structure D such that the factorization $B = C \times D$ will hold. The factorization methodology is adapted from Hartmanis-Stearns (1966) machine decomposition theory. The procedure will create any real time acceptor if an appropriate data structure is available.

Synthesizing PROLOG Programs

Shapiro (1982) has developed a methodology for creating PROLOG logic programs from examples. The operation of his system will be illustrated here by showing how it creates the member function in PROLOG. Here member (X,Y) will be defined to be a predicate which is true if and only if X is a member of list Y. Following PROLOG notation, lists will be written with square brackets. Thus $[a,b,c]$ is the list containing entries a, b, and c and so member $(a,[a,b,c])$ and member $(b,[a,b,c])$ will be true while member $(d,[a,b,c])$ will be false.

PROLOG programs will be written here as sets of clauses of the form $p_1 \leftarrow p_2, p_3, \ldots, p_n$ where the p_i are predicates. The meaning of such a clause is that p_1 is true if $p_2, p_3, \ldots, p_{n-1}$, and p_n are true. A PROLOG program is executed by asserting such a p_1 and having the processor prove $p_2, p_3, \ldots, p_n$. Typically these latter proofs involve calls to other clauses in the program with very deep nestings possible.

The example to be studied here is the following program which has two clauses. The notation $[X|Y]$ stands for a list whose first element is X and whose other elements are contained in Y.

$$\{\text{member}(X,[X|Z]) \leftarrow \text{true},$$
$$\text{member}(X,[Y|Z]) \leftarrow \text{member}(X,Z)\}$$

The operation of this program can be understood by observing its action on some of the above example behaviors. Thus member $(a,[a,b,c])$ can be proved using the first

clause with $X = a$ and $Z = [b,c]$. The case member $(b,[a,b,c])$ can be proved by invoking the second clause which asserts that member $(b,[a,b,c])$ if member $(b,[b,c])$ and then using the first clause to prove member $(b,[b,c])$. The concern here is to show how such a program can be generated automatically.

The synthesis methodology is shown in Figure 14. The user introduces facts in the form of ground instances of the predicates. Each such predicate must be accompanied by an indication of whether it is true or false. Thus the user might enter the facts "member $(a,[a,b])$ is true" and "member $(c,[a,b])$ is false". The system proposes various clauses which might be parts of the target program and stores them into the data structure called PROLOG program. Then the PROLOG interpreter executes the currently proposed program on the available facts and determines whether it produces the desired result in each case. That is, given the above two facts, the current program should evaluate to true for member $(a,[a,b])$ and false for member $(c,[a,b])$. If the current program is not able to prove a desired result as in member $(a,[a,b])$, it needs an additional clause and it calls for one. If the current program proves something which the facts indicate is wrong, the system finds the offending clause in the program and removes it.

The clause generator at the top of Figure 14 is basically enumerative in nature. However, Shapiro has designed it carefully to avoid any unnecessary enumeration. First the user must declare the predicate symbols to be used and then specify what predicates can appear on the right hand sides of clauses. The clause generator also omits the creation of many "refinements" of clauses that have been shown to be unsatisfactory. In the example of this section, the clause generator will produce the following series of proposed clauses:

$$\text{member } (X,Y) \leftarrow \text{true}$$

$$\text{member } (X,[X|Z]) \leftarrow \text{true}$$

$$\text{member } (X,[Y|Z]) \leftarrow \text{member } (X,Z)$$

$$\text{member } (X,Y) \leftarrow \text{member } (Y,X)$$

$$\text{etc.}$$

A dialog for doing this synthesis on the Shapiro system is as follows:

System: Declare the routine to be synthesized.

USER: member (,).

System: What routines does "member" call?

USER: member

System: The guessed program is { }, the empty program.

USER: member $(a,[a])$ is true.

System: The guessed program is incomplete. Member $(a,[a])$ is not covered. The revised program is (member $(X,Y) \leftarrow$ true}. (A call has been made to the clause generator.)

USER: member $(a,[b])$ is false.

System: The guessed program is incorrect. Member $(a,[b])$ is covered and it should not be. The revised program is { }. The guessed program is incomplete.

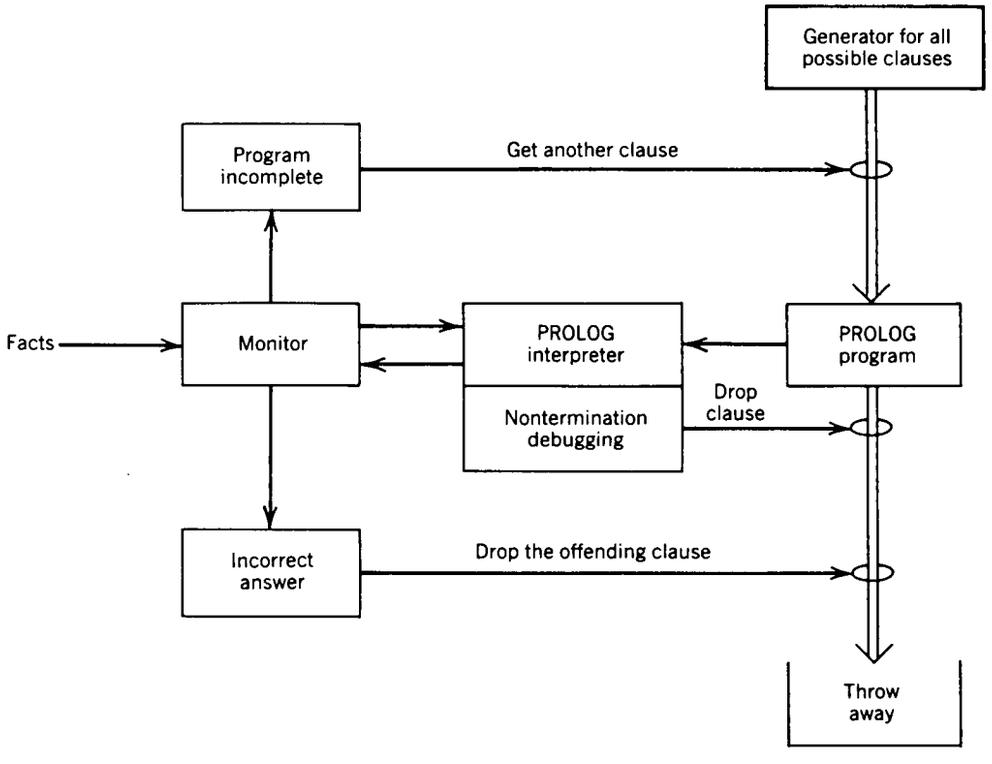


Figure 14. The Shapiro synthesis algorithm.

Member $(a,[a])$ is not covered. The revised program is {member $(X,[X|Z]) \leftarrow$ true}. (A second call to the clause generator has been made.)

USER: member $(a,[b,a])$ is true.

System: The guessed program is incomplete. Member $(a,[b,a])$ is not covered. The revised program is {member $(X,|X|Z) \leftarrow$ true, member $(X,[Y|Z]) \leftarrow$ member (X,Z)}. (A third call to the clause generator has been made.)

This example is somewhat contrived in order to show the mechanisms of the system in a short space. However, on more complicated examples, the system very efficiently uses the available facts to sort through the generated clauses until a solution is found.

The overall strategy is thus one of building approximations to the target program and then debugging them using facts provided by the user. Three kinds of errors may occur:

1. The program may compute a result which is undesired, an incorrect answer, as in the above dialog when the guessed program was able to prove member $(a,[b])$.
2. The program may be unable to compute a desired answer as was the case above when member (a,[a]) could not be proved.
3. The program may not terminate.

In the first type of error, the debugging mechanisms in the system simulate the incorrect computation and query the user and/or the data base of facts to check the correctness of every step in the computation. When a clause is found that computes an incorrect result from correct premises, that clause is discarded from the program. Thus in the above dialog, when the incorrect result member $(a,[b])$ was proven, the clause member $(X,Y) \leftarrow$ true was shown to be incorrect and was discarded. In the second type of error, a simulation of the failed computation is performed to find what predicate p_1 needed to be proven which was not proven. Then a call is made to the clause generator to find a new clause which will yield p_1 in the given computation. In the third type of error where nontermination occurs, the interpreter halts after a prespecified limit on the computation size has been exceeded. The processor than looks for an unending loop where the same computation state is reentered repeatedly, and it may also query the user concerning violations to a well founded ordering needed to insure termination. This debugging procedure leads to the discovery and removal of a clause in the program.

Shapiro tested this system in a variety of problem domains and compared it with various other program generation systems in the literature. For example, consider the problem solved by Biermann's LISP synthesizer (1978): Construct a program to find the first elements of lists in a list of atoms and lists. Thus the target program should be able to read input $[a,[b],c,[d],[e],f]$ and compute the result $[b,d,e]$. Shapiro's system needed 25 facts to solve this problem and generated the following code after 38 seconds of computation time.

{heads([ ],[ ]) $\leftarrow$ true,

heads([[$X|Y$]|Z], [$X|W$]) $\leftarrow$ heads (Z,W),

heads([$X|Y$],Z) $\leftarrow$ atom (X), heads (Y,Z)}

Biermann's system used only the single example given above and produced a correct regular LISP program after one half hour of computation.

Tinkham (1990) has developed a related methodology for PROLOG synthesis. It uses the insight that many target programs for synthesis resemble other well known programs, and synthesis need not begin with the empty program as does the Shapiro system. Tinkham uses a generalization technique to build a hierarchical tree of program schemas. At the bottom are a variety of well known programs organized in groups with similar structures. Above each group is a PROLOG schema which is a generalization of all of the members of its group. These schemas are organized into similarity groups that are generalized again by more abstract schemas higher in the tree. At the top of the tree is the most general schema from which all programs can be derived.

Synthesis proceeds by starting at the lowest level schema on the tree that is believed to be applicable to the task. If a schema can be chosen that is very near the target program, synthesis will be very fast. If the initial schema is very abstract and far from the target, synthesis may require substantially more time.

Theoretical Issues in Synthesis From Examples

A program synthesis system is called sound if whenever a program is generated from a set of examples, it can properly do all of those examples. The system is called complete for a class C of programs if it can generate all of the programs in the class. The properties of soundness and completeness are desirable for a program synthesis algorithm because they guarantee at least a minimal degree of behavioral acceptability for that algorithm.

An example of a synthesis method that is both sound and complete is the algorithm that simply enumerates all the members of a class C until a program is found that properly executes the given example behaviors. Two restrictions on the class C are needed before the algorithm will work: C must be enumerable, call its members P_1, P_2, P_3, etc., and it must be decidable for each behavior B and each program P_j in C whether P_j achieves B. The algorithm can be stated more precisely as shown below:

Algorithm

Input. A finite set S behaviors for the target program.

Output. A program P from class C with the property that P can execute each B in S.

1. j $\leftarrow$ 1.
2. while there is B in S such that P_j cannot execute B, increment j.
3. *return with result* P_j.

This algorithm is sound by its very construction. One can show it is complete on C by considering its behavior in attempting to synthesize an arbitrary program in C. Suppose P_T is the first program in the enumeration P_1, P_2, P_3, . . . that is capable of executing all the behaviors of the target program. Then one can give the algorithm randomly selected behaviors of P_T and observe which P_j is generated. If P_j is not P_T, the user will detect the problem either by testing P_j or by studying its code. Then more examples can be given until the enumeration is forced to find P_T. There will always be examples to achieve this, because P_T is, by definition, the first program capable of all of the target behaviors. So the algorithm is complete. An interesting pragmatic discovery that has come out of this research is that very few examples are needed to achieve synthesis of most programs, even some very large ones.

Another important characteristic of the enumerative algorithm, as shown by Gold (1967), is that it is input optimal in the following sense: If another algorithm is proposed for generating programs in class C on the basis of behaviors, it will not be true that all programs in C will be generated from fewer behaviors than with the enumerative algorithm.

These results have practical significance because the flowchart synthesis algorithm of Figure 7 is functionally equivalent to the enumerative strategy if it is executed repeatedly for $L = 1, 2, 3, \ldots$ until a program is synthesized. This means that the flowchart synthesis method is sound, complete, and input optimal on the class of all flowcharts. Furthermore, many of its variations have similar properties. For example, the function merging technique of LISP program synthesis is sound, complete, and input optimal on the class of regular LISP programs. Thus these methodologies are not heuristic in the sense that their abilities to converge to a solution are in any way unpredictable.

The Summers synthesis method is sound, and a variation of it has been proved to be complete over a class of programs defined by Smith (1977). The Shapiro methodology is sound and complete over the class of programs that can be constructed with rules from the rule generation routines.

Historical Remarks

One of the earliest papers on synthesis from examples was done by Amarel (1962). Later Solomonoff (1964) and Gold (1967) proposed the grammatical inference problem which resulted in a series of studies on the construction of grammars from their generated strings (Biermann and Feldman, 1982; Angluin, 1978; Blum and Blum, 1975; Feldman, Gips, Horning, and Reder, 1969). In the early 1970s, Biermann (1972, 1978) and Biermann and Krishnaswamy (1976) developed strategies for program synthesis from traces while a number of researchers were beginning to study synthesis procedures for LISP code (Biggerstaff, 1976; Hardy, 1975; Kodratoff and Jouannaud, 1984; Shaw, Swartout, and Green, 1975; Summers, 1977). Biermann and Smith (1979) developed a strategy for hierarchically decomposing examples and generating LISP code using production rules. The synthesis from schemas technique of Tinkham used ideas from Dershowitz (1983).

PROGRAM SYNTHESIS THROUGH NATURAL LANGUAGE DIALOGUE

Although the techniques given above provide fundamental mechanisms for program synthesis, they need to be embedded in a larger system which can acquire the information for synthesis, provide the needed domain and programming knowledge, coordinate the various synthesis processes, and generate an acceptable output. Several of these large systems were constructed during the 1970s with very ambitious goals. The systems were to interview the user in natural language, acquire a model of the computational process to be undertaken, verify its correctness through further dialog, select data structures for efficient execution, and code the output in a traditional programming language. The goals of the research were twofold: To learn the nature of the problems associated with assembling a wide variety of technologies into a single automatic programming system, and to provide an environment within which these technologies could be further studied.

System Design

An example of this type of system is the PSI automatic programmer (Green, 1976) which is organized as shown in Figure 15. Here the first set of modules handle the acquisition phase of the synthesis when the user is being interviewed and a high level version of the program is being assembled. The lower portion of the figure shows the coding phase where efficiency decisions are made and code is generated.

The acquisition phase begins with a parser-interpreter which receives natural language input from the user and constructs a semantic net representation of what the user has said. The discourse module monitors the input, attempts to discover the user's intentions, and coordinates the various system functions to achieve the desired result. The explainer generates user friendly questions posed by the system or outputs a description of the program model. The domain expert builds fragmentary pieces of high level code for solving parts of the problem and passes them on to the model builder which assembles fragments into a high level version of the target program. The trace expert can interpret illustrative inputs from the user and usefully supplement other information sources. The coding phase of the processing can involve considerable revision to the program model received from the acquisition phase. The coder and efficiency expert work together to evaluate various alternative data structures for the target program, make choices on representation, and create the final code.

In the following sections, the flavor of each of these stages is illustrated by an example. However, the descriptions are necessarily approximate and oversimplified and the original sources should be consulted for more accurate and detailed information.

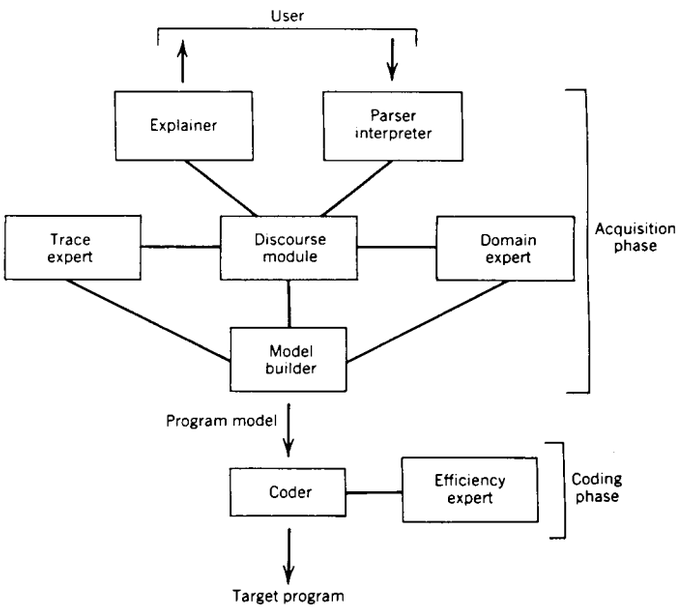


Figure 15. Functional diagram for the PSI system.

Example Dialog

The PSI system is designed to handle dialogs of the following kind where we are assuming the user wants a program to add a column of numbers.

USER: Please write a program to add up a set of numbers.

System: WHAT WOULD YOU LIKE TO CALL THE PROGRAM?

USER: Sum.

System: WHAT NUMBERS ARE TO BE ADDED UP?

USER: The program should read nonnegative integers from the user, sum them, and then print the answer.

System: WHEN SHOULD THE PROGRAM STOP READING?

USER: When a negative number is read.

System: IS THERE ANYTHING ELSE?

USER: The program should work as follows:

Output	READ:
Input	7
Output	READ:
Input	12
Output	READ:
Input	−1
Output	THE SUM IS 19.

System: Okay, how is this?

```
PROGRAM SUM (INPUT, OUTPUT);
VAR X, ANS: INTEGER;
BEGIN
ANS:=0;
WRITELN ('READ:');
READ (X);
WHILE X >= 0 DO
        BEGIN
        ANS: = ANS + X;
        WRITELN ('READ:');
        READ (X);
        END;
WRITELN ('THE SUM IS', ANS)
END.
```

The Functional Modules: Acquisition

The first module to process user inputs is the parser-interpreter (Ginsparg, 1978) which receives incoming sentences, parses them to identify their constituents, and then builds a meaning representation. The reader should consult portions of this volume on natural language processing to better understand the nature of this problem. The many issues concerning the analysis of complex sentences, construction of a meaning structure, resolution of pronominal and elliptical structures, handling of quantifiers and other problems are handled at this level. The output of this stage of processing is a semantic net that assembles as well as possible the individual parts of the dialogue into a cohesive unit. Figure 16 shows the nature of the meaning representation that would be constructed from the example dialogue. The individual sentences each contribute a few nodes and connections to the total network. As the information arrives, it is properly attached to the existing structure which gradually evolves toward a complete representation of the target program. It may not be possible for the parser-interpreter to complete the attachment of all parts of the description because domain or programming knowledge may be needed in places, but most of the primary connections can be made.

The discourse module (Steinberg, 1980) maintains a sense of progress in the dialogue by attempting to build a representation of the user's desires and initiating actions to satisfy them. It has communications with all acquisi-

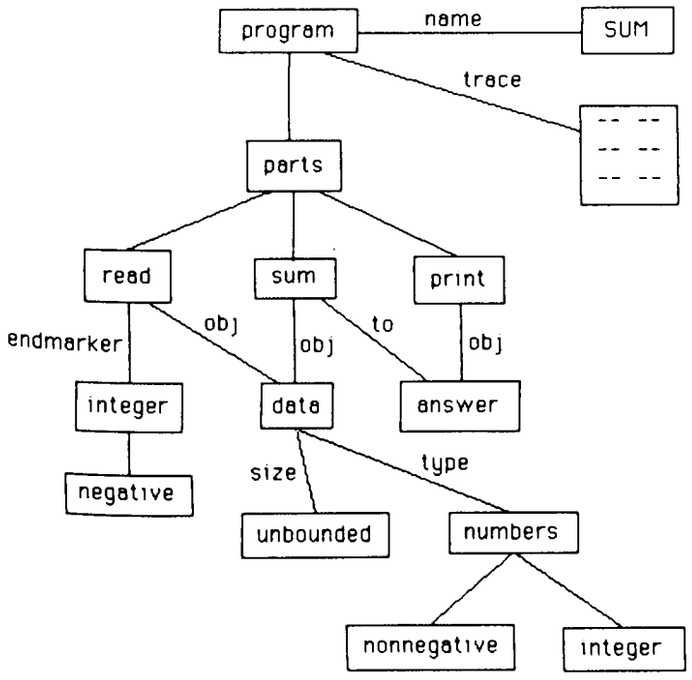


Figure 16. Representation of dialogue meaning structure.

tion modules and attempts to coordinate activities to achieve a cohesive interaction. The user may introduce concepts that need to be clarified and fit into the total theme. The system domain expert and model builder may recognize where information is needed to complete portions of the program description, and the discourse module must formulate queries to the user through the explainer (Gabriel, 1981) to obtain this information. Thus in the dialogue above, the system recognizes upon the mention of a program in the first sentence, that such a program should have a name, a body of code, and other related information. This causes it, for example, to formulate and return the next query. If the system finds all its internally generated questions answered, it may then pass control back to the user as is illustrated above in the last system query: "Is there anything else?" This is called a mixed initiative dialogue where one party introduces an issue that is resolved in subsequent interaction, then the other party mentions a point that requires discussion, and so forth.

The domain expert (Phillips, 1977) has the task of converting the semantic net class of information (Fig. 16) into code fragments written in a very high level language. Thus a generic input routine of the following form might be retrieved to handle the read function called for in Figure 16.

```
S ← 0
input (x)
while x is valid data
        S ← S ∪ {x}
        input x
```

And a generic collection routine might be instantiated to do the required summing operation.

```
ans ← 0
while more data
        retrieve y
        ans ← ans + y
```

Finally, high level codes would be produced for the print routine,

```
output (z)
```

and all of these code fragments would be transferred to the model builder for assembly of the high level program. Thus the domain expert instantiates somewhat vague information from the earlier stages in concrete though rather high level code. It fills in some information where domain knowledge is needed but certain connections between the fragments are still not made.

The trace expert (Phillips, 1977) is designed to receive example input-output pairs for the target program, traces of snapshots of the program's behavior, or high level traces expressed in natural language. This expert generates a sequence of state–characterizing schemata which are then used in the creation of code fragments for the model builder. In the example dialogue, two effects result from the given trace. First, it is noted that the user wants the target program to output a read prompt before each input so this code must be merged into the given code segment.

```
S ← 0
output ('READ:')
input (x)
while x is valid data
        S ← S ∪ {x}
        output ('READ:')
        input (x)
```

Second, the trace gives the system a way of checking its generated program for acceptability.

The model builder (McCune, 1977) receives fragments from the other processes and attempts to assemble a high level version of the target program. This assembly may involve very complex processing including the compilation of all the information units and control structures needed and their proper coordination. The model builder, besides receiving information, can return information to the earlier stages regarding the portion of the code currently being discussed and thus provide possible referents for unresolved incoming noun phrases. In the current example, the processor must collect the above three code segments and coordinate them by noting the set being read in the first segment is identical to that being added in the second. Also, the correct data object must be printed at the end.

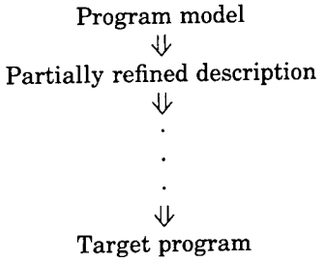
```
*read*
        S ← 0
        output ('READ:')
        input (x)
        while x ≥ 0
                S ← S ∪ {x}
                output ('READ:')
                input (x)
*sum*
        ans ← 0
        while not empty S
                y ← remove from (S)
                ans ← ans + y
*print*
        output (ans)
```

The program model is then sent to the coding phase for the creation of efficient machine code.

The Functional Modules: Coding

The coding module (Barstow, 1979) contains a large amount of detailed programming knowledge in the form of production rules. These rules are capable of proposing wide variations in the form of the data structures and the code. At selected points in the generation process, possible alternatives are created and handed to the efficiency expert (Kant, 1977) for evaluation. The efficiency expert has tools similar to those of a human being for making choices: analysis of algorithms techniques, general knowledge, and simulation. The system uses probabilistic information about data and the costs of machine operations to compute space-time cost functions for various alternatives and passes results back to the coder. Thus through a process of generating alternatives, evaluation, and movement down the search tree, the production rule system refines the program model into machine executable form.

The general organization of the coding module follows the tradition of expert system technology. It receives the program model from the acquisition phase and has several hundred production rules for modifying this model and converting it one step at a time to concrete code.



Many possible rules may be applicable at a given stage in the development. For example, if a set of objects is represented in the program model, the coder must make a decision concerning how the set is to be represented in the target programming language. Production rules will be available to select specific representations such as arrays, linked lists, bit maps, and so forth. An agenda orders the tasks to be addressed in the coding process and guides the selection of the rule to be tried next. Evaluation of each new partially refined program description is done using heuristic methods and calls to the efficiency expert. The more attractive paths in the sequential search for an acceptable program are moved toward the top of the agenda for continued expansion and refinement.

The steps that the coding phase might follow in completing the example of this section will be described next. The production rules given here are not actually taken from the system but give a feeling for how it works.

The system might have a production rule for combining loops that scan the same set.

If two separate loops increment through the same set and their code segments have independent effects, then they can be combined into a single loop.

Following the style of the original author, the rules will be given here in English rather than in a detailed notational form. The result of this rule applied to the two loops in the program model for reading and summing would be the following:

```
S ← 0
output ('READ:')
input (x)
ans ← 0
while x ≥ 0
        S ← S ∪ {x}
        y ← remove from (S)
        ans ← ans + y
        output ('READ:')
        input (x)
print (ans)
```

At this point, the system could notice the redundancy of the S data structure and employ the following rule to delete it:

If a single data structure is loaded and then emptied without any intermediate references, it can be removed.

```
output ('READ:')
input (x)
ans ← 0
while x ≥ 0
        ans ← ans + x
        output ('READ:')
        input (x)
print (ans)
```

Finally, a long series of rules is needed to actually create the executable code. The initialization lines, declarations, and all other special syntax must be properly assembled to achieve the target code.

```
PROGRAM SUM (INPUT, OUTPUT);
VAR X, ANS: INTEGER;
BEGIN
ANS:=0;
WRITELN ('READ:');
READ (X);
WHILE X >= 0 DO
        BEGIN
        ANS: = ANS + X;
        WRITELN ('READ:')
        READ (X);
        END;
WRITELN ('THE SUM IS', ANS);
END.
```

The Implementation

The PSI system was completed in the mid 1970s and is capable of constructing programs of several types including some concept formation programs and some numerical programs. A number of example dialogues have been published (Green, 1976) including interactions of up to about fifty sentences which result in several dozen lines of LISP code.

Historical Remarks

Heidorn (1974) built the first and one of the most impressive natural language automatic programming systems, NLPQ, which was aimed at the solution of operations research queuing problems. The system translated incoming sentences into a semantic network problem representation which then could be translated back to the user in paraphrase for verification. Then the network was compiled into the GPSS simulation language and run on conventional software.

Simultaneously with the Green project (1976), a synthesizer called SAFE was built by Balzer and co-workers (1976, 1978) which emphasized the automatic acquisition of domain knowledge and the creation of software from informal specifications. Also Martin and co-workers (1974) built a system which placed heavy emphasis on high quality processing of natural language.

Biermann and Ballard (1980) constructed an interpreter for "natural language programs" which was robust enough to be used by college students in solving programming problems (Biermann and co-workers, 1983; Geist and co-workers, 1982).

PROGRAM CONSTRUCTION USING A MECHANIZED ASSISTANT

More recently researchers have been examining the role that artificial intelligence can play in industrial programming environments where large software systems are specified, coded, evaluated, and maintained. Here the whole life cycle of the software system is under consideration: The client and the professional systems analyst discuss informally a proposed software product. Then more formal specifications are derived, performance estimates are made, and a model of the system evolves. Many times specifications are modified or redefined as analysis proceeds. The next phase is the actual construction, documentation, and testing of the product. After release into the user environment, the system may be debugged and changed or improved on a regular basis over a period of years.

A developing idea in some current automatic programming projects (Rich and Shrobe, 1978; Balzer and co-workers, 1983) envisions a mechanized programmer's assistant that would intelligently support all of the above activities. It would provide a programming environment for the user capable of receiving many kinds of information from programmers including formal and informal specifications, possibly natural language assertions regarding goals, motivations, and justifications, and code segments. It would assist the programmer in debugging these inputs and properly fitting them into the context of the programming project. It would be knowledge based and thus capable of fully understanding all of the above inputs. It would provide library facilities for presenting the programmer with standardized program modules or with information concerning the current project. It would be able to generate code from specifications, program segments, and other information available from the programmer and other sources. It would be able to understand program documentation within the code and to generate documentation where necessary. Finally, it would maintain historical notes related to what was done, by whom, when, and most importantly why. All of these functions are envisioned as operating strictly in a supportive role for human programmers who are expected to carry on most high level tasks.

Thus, the concept of the automatic programmer's assistant places the human programmer in the primary position of specifying the program and guiding progress towards successful implementation and maintenance. The task of the assistant is to maximally utilize available technologies to automate as many lower level functions as possible.

This view emphasizes the decomposition of the programming task into two stages (Fig. 17), systems analysis and programming. The first stage involves the development of formal specifications and deals primarily with what performance is required; the latter includes the decomposition of the task into an appropriate hierarchy of subparts, the selection of data structures, and the coding and documentation of the product. The former is assumed to be the appropriate domain for considerable human involvement whereas the latter is expected to be more amenable to automation.

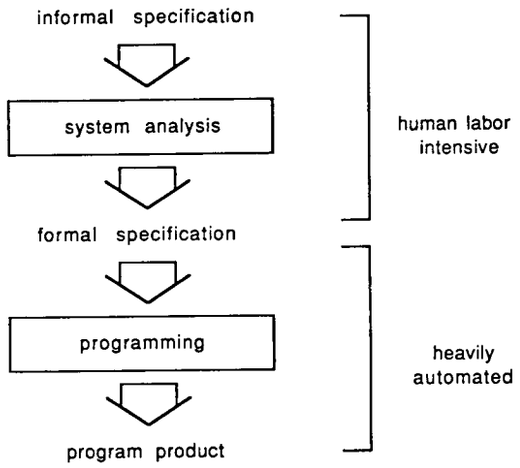


Figure 17. Stages in program construction.

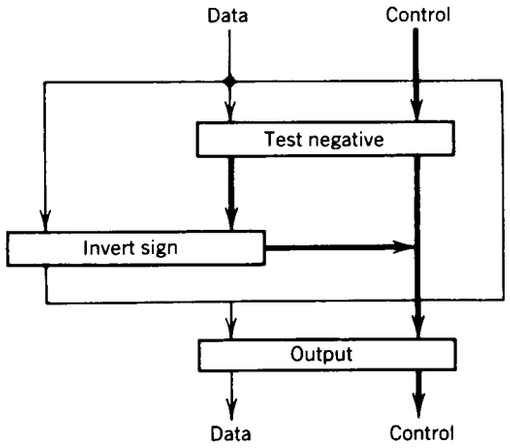


Figure 18. Example plan for computing absolute value showing both flow of data and flow of control.

All of this has implications for the program life cycle. The initial programming effort involves the human-machine team in developing the specification-system analysis followed by a largely automated generation of the programming product. Later stages in the program life cycle may involve revised specifications and improved approaches for doing the computation. These are implemented by replaying the complete construction process. Revisions are made to the informal specifications and these become reflected in the formal specifications. Then the automatic generation can be replayed possibly with revised decisions to account for the updated requirements.

The Programmer's Assistant

In order to begin implementing such an assistant, it is necessary to have appropriate languages to handle the many kinds of information that appear in this application. One approach is to introduce the concept of a wide-spectrum language that can be used at all levels of implementation from the specification of requirements to high-level coding of the actual target program. An example of such a language is REFINE (Smith and Westfold, 1987), which has as primitives sets, mappings, relations, predicates, enumerations, state transformation sequences, and other constructions.

Another approach (Rich and Waters, 1988) is based primarily on the concept of plans for programs that contain the essential data and control flow but exclude programming language details. An example of a plan appears in Figure 18, where the computation of absolute value is represented. The advantages of such plans are that because they locally contain essential information, they can be glued together arbitrarily without global repercussions. A complete plan will include both the graphical representation shown here and predicate calculus specifications on the plan behaviors. Commonly used plans, called cliches, can be stored in a library and can provide the building blocks for the assembly of large plans.

This approach uses code and plans as parallel representations for the program and allows the user to deal easily with either one. If the user chooses to work in the plan domain, each action in creating or modifying a given plan results in appropriate updates in the code domain. The coder module translates the current version of the plan into code. If the user wishes to work with the code, the analyzer appropriately revises the associated plan.

The methodology utilizes plan transformations, called overlays, to do reasoning with plans. An example of an overlay appears in Figure 19 where the relationships between two plans are given as described in (Rich and Waters, 1988): the left side represents a plan for adding to an indexed sequence by incrementing an index and entering a data item; the right side represents a plan for pushing a data item on a stack. The overlay specifies the relation-

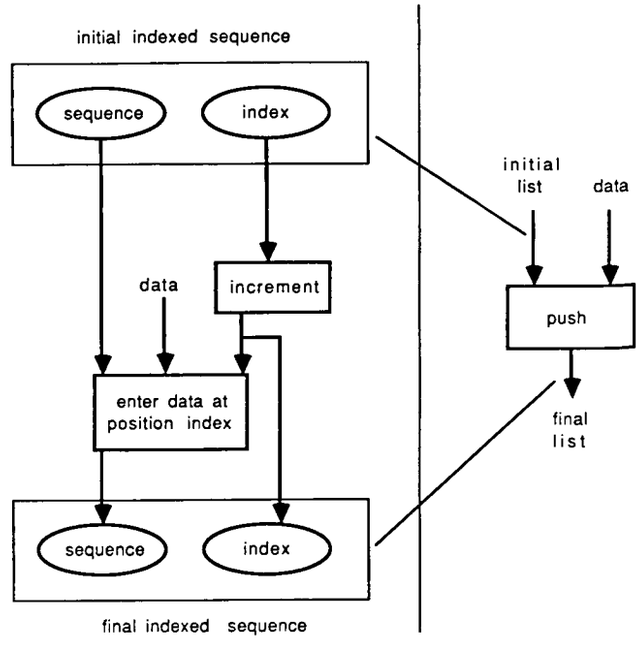


Figure 19. A plan transformation (or overlay).

ship between the two operations and allows a plan formulated in one paradigm to be revised to handle the other. The transformation can be used in either direction.

A knowledge-based editor KBEmacs has been developed (Waters, 1985) to create and manipulate plans and their associated code. It includes a layered reasoning system that works at the plan calculus level as well as at algebraic and predicate calculus levels. It has been demonstrated by creating a 54 line report generator program for a repair application in Ada from a five line specification as follows:

Define a simple_report procedure UNIT_REPAIR_REPORT.
Fill the enumerator with a chain_enumeration of UNITS and REPAIRS.
Fill the main_file_key with a query_user_for_key of UNITS.
Fill the title with ("Report of Repairs on Unit" & UNIT KEY).
Remove the summary.

The system in this example contained an elaborate report cliche and substantial information related reporting the repair problem. The details of the generation appear in (Rich and Waters, 1988).

Automatically Generating Search Programs

Many of the ideas introduced here are illustrated in the KIDS system described by Smith (1990). This system utilizes algorithm theories which correspond to the plans of Rich and Waterman; they are stereotypic programming structures with associated input-output specifications. Included in the system is a reasoning mechanism that is used to instantiate such code structures with the appropriate details for the user's current problem. First, an inefficient prototype of the target program is created. Then it is refined repeatedly to improve its efficiency and quality until the final program is produced.

Although the approach is general, Smith has concentrated his efforts on the automatic creation of search programs (Smith, 1990) and divide-and-conquer programs (Smith, 1987). The approach will be illustrated here with the semi–automatic creation of a binary search routine. The synthesis will go through these steps:

1. Specification of the problem.
2. Selection of a theory from a library.
3. Specialization of the theory to the current problem.
4. Instantiation of the theory into a first draft program.
5. Refinement of the draft program.
6. Compilation into object code.

Suppose the problem is to create a program to search a sorted array for a given key and to return the index of the item when it is found. Step 1 of the process is to specify the problem in the KIDS language.

F_F: Ord Search
D_F: $\{\langle h, A, key\rangle \mid h \in \text{Nat} \wedge 1 \leq h \wedge A \in map(\{1..h\}, \text{Nat}) \wedge \text{Ordered}(A) \wedge key \in \text{Nat}\}$
R_F: $\{index \mid index \in \text{Nat}\}$
O_F: $\lambda\langle h, A, key\rangle, index.\ A(index)=key \wedge index \in \{1..h\}$

Thus the program is to have name "OrdSearch" and it is to accept three inputs h, A, and key as specified. A is a sorted array of size h and key is a natural number. The output as specified by R_F is to be a natural number and the relation O_F between the input $\langle h, A, key\rangle$ and output $index$ is that $A(index) = key$ where $index$ is in $\{1..h\}$. (The lambda notation λ specifies arguments followed by a period followed by the relationship.)

Step 2 is to select a theory for the computation and Smith has developed seven choices which cover most of the well known searching paradigms. For example, his system can semi–automatically create programs to solve k-queens, traveling salesman, *0–1* integer linear programming, propositional satisfiability, hamiltonian circuit, knapsack, topological sort, and long lists of other search programs. In the current example, a reasonable choice is to use the binary-split-of-integer-subrange theory as given here:

F_G: gs_binary_split_of_integer_subrange
D_G: $\{\langle m,n\rangle \mid m \in \text{integer} \wedge n \in \text{integer} \wedge m \leq n\}$
R_G: $\{k \mid k \in \text{integer}\}$
O_G: $\lambda\langle m,n\rangle, k.\ k \in \{m..n\}$
$\hat{R}$: $\lambda\langle m,n\rangle.\ \{\langle i,j\rangle \mid i \in \text{integer} \wedge j \in \text{integer} \wedge m \leq i \leq j \leq n\}$
Satisfies: $\lambda k, \langle i,j\rangle.\ i \leq k \leq j$
$\hat{f}_0$: $\lambda\langle m,n\rangle\ .\ \langle m,n\rangle$
Split: $\lambda\langle i,j\rangle, \langle i',j'\rangle.$ $i<j \wedge (\langle i',j'\rangle = \langle i,(i+j) \text{ div } 2\rangle \vee \langle i',j'\rangle = \langle 1+ (i+j) \text{ div } 2, j\rangle)$
Extract: $\lambda k, \langle i,j\rangle.\ i=j \wedge k=i$

In words, this theory has the name gs_binary_split_of_integer_subrange and it receives a pair of integers $\langle m,n\rangle$ as input. The output is an integer k where k must be in the interval $\{m..n\}$ $\hat{R}$ specifies the set of objects to be searched, the set of intervals $\langle i,j\rangle$. An integer k will satisfy an interval $\langle i,j\rangle$ if $i \leq k \leq j$. The initial interval in the search $\hat{r}_0$ will be $\langle m,n\rangle$ which is the whole interval. The method of search will be to repeatedly split the current interval using the rule given: $\langle i,j\rangle$ is split into $\langle i,(i+j) \text{ div } 2\rangle$ and $\langle 1 + (i+j) \text{ div } 2, j\rangle$. A solution will be extracted when interval $\langle i,j\rangle$ is reached where $i=j$. The solution will be $k=i$. This theory of search is applicable to such problems as integer square root, maximum of a unimodal function, and binary search of a sorted array. The latter problem is the one being addressed here.

The third step of the process is largely done automatically. The theory given above is to be specialized to solve the specific problem at hand. The method is to prove the following theorem which has the side effect of producing a unifier θ. This unifier provides the substitution needed to produce the specialized theory.

$$\forall x \in D_F\ \exists y \in D_G\ \forall z \in R_F [O_F(x,z) \Rightarrow O_G(y,z)]$$

This theorem is instantiated in the current problem as follows:

$\forall x \in \{\langle h,A,key\rangle \mid$ (see D_F above)$\}$

$\exists y \in \{\langle m,n\rangle \mid$ (see D_G above)$\}$

$\forall z \in \{\langle index \mid$ (see R_F above)$\}$

[$\{index \mid index \in$ Nat$\}$ is a subset of $\{k \mid k \in$ integer$\}$ and

$(A(index) = key \wedge index \in \{l..h\}) \Rightarrow k \in \{m..n\}$]

The proof of this theorem yields the unifier θ: $n \leftarrow h$, $m \leftarrow 1$, $k \leftarrow index$. Making these substitutions into the general binary search theory yields a search theory for sorted linear arrays.

The fourth step of the procedure utilizes a synthesis theorem that gives a generalized program to compute the search. Smith has proved a number of such theorems and the one needed here says that the program to input x as specified in D_F and compute z meeting the requirement $O_F(x,z)$ is given here:

function $F(x : D_F)$: set (R_F)
returns $\{z \mid O_F(x, z)\}$
$= F\text{-}gs(x,\hat{r}_0(x))$

function $F\_gs\ (x : D_F, \hat{r} : \hat{R})$: set (R_F)
returns $\{z \mid$ Satisfies $(z, \hat{r}) \wedge O_F(x, z)\}$
$= \{z \mid$ Extract $(z, \hat{r}) \wedge O_F(x, z)\} \cup$
$\{F\_gs(x, s) \mid$ Split $(x, \hat{r}, s)\}$

Making the correct substitutions from above, the program becomes:

function

OrdSearch (h:Nat,A:map($\{1..h\}$,Nat), key:Nat): set (Nat)
returns $\{index \mid A(index) = key \wedge index \in \{1..h\}\}$
=OrdSearch_gs ($h,A,key,1,h$)

function

OrdSearch_gs(h:Nat,A:map($\{1...h\}$,Nat),
key : Nat, i : Nat, j : Nat) : set (Nat)
returns $\{index \mid A(index) = key \wedge index \in \{i..j\}\}$
$= \{index \mid i=j \wedge index = i \wedge A(index) = key\}$
$\cup \{index \mid i<j \wedge \langle i',j'\rangle = \langle i, (i+j) \text{ div } 2\rangle \wedge$
$index \in$ OrdSearch_gs(h,A,key,i',j')$\}$
$\cup \{index \mid i<j \wedge \langle i',j'\rangle = \langle 1+(i+j) \text{ div } 2:j\rangle \wedge$
$index \in$ OrdSearch_gs(h,A,key,i',j')$\}$

This is a draft of the target program. It receives the specified input, executes the binary split strategy of search, and returns the correct answer. It has a shortcoming, however; it searches the complete list requiring an examination of every entry in A. There is a major improvement possible because A is sorted, and this leads to step 5 of the process, refinement of the program.

The primary efficiency improving technique for search programs is the deletion of fruitless subsearches. In most search problems, there are many subspaces that probably have no solutions, and tests must be inserted into the program to prevent useless excursions. One such strategy is the derivation of a formula $\Phi(x, \hat{r})$ such that

$$\forall x \in D_F\ \forall \hat{r} \in \hat{R}\ \forall z \in R_F\ [\text{Satisfies}(z,\hat{r}) \wedge O(x,z) \Rightarrow \Phi(x,\hat{r})]$$

The formula $\Phi(x,\hat{r})$ provides a necessary condition for subspace $\hat{r}$ to contain a solution. If it fails to hold, the subspace can be discarded as a possible place for problem solutions.

In the current problem, the program OrdSearch_gs returns $\{index \mid A(index) = key \wedge index \in \{i,j\}\}$. An additional and previously unused fact in this problem is that A is sorted:

$$1 \le i \le j \le h \Rightarrow A(i) \le A(j)$$

These two facts combine to yield

$$A(i) \le A(index) \le A(j)$$

or

$$A(i) \le key \le A(j)$$

The insertion of this guard into every entrance to a new subspace results in the pruning of a large fraction of the tree. It has changed the computation time from order n to order $log_2 n$.

function

OrdSearch (h:Nat,A:map($\{1..h\}$,Nat),key:Nat): set(Nat)
returns $\{index \mid A(index)=key \wedge index \in \{l..h\}\}$
=OrdSearch_gs ($h,A,key,1,h$)

function

OrdSearch_gs(h:Nat,A:map($\{1..h\}$,Nat),key:Nat,i:Nat,
:Nat): set (Nat)
returns $\{index \mid A(index) = key \wedge index \in \{i..j\}\}$
$= \{index \mid i=j \wedge index=i \wedge A(index)=key\}$
$\cup$ {index $\mid i<j \wedge \langle i',j'\rangle = \langle i, (i+j)$div 2$\rangle$
$\wedge A(i') \le key \le A(j')$
$\wedge index \in$ OrdSearch_gs (h,A,key,i',j')$\}$
$\cup \{index \mid i<j \wedge \langle i',j'\rangle = \langle 1+(i+j)$div 2,$j\rangle$
$\wedge A(i) \le key \le A(j')$
$\wedge index \in$ OrdSearch_gs(h,A,key,i',j')$\}$

The KIDS system is capable of finding many other optimizations in this program, with some interactions from the user. For example, one can simplify expressions and pull out common subexpressions. The derivation outlined here is given in substantially more detail in Smith (1987). One interesting speedup that can be made in some programs is called finite differencing. Sometimes an expression that is computed on each cycle through a loop can be eliminated by saving its results from the previous cycle. If the new value of the expression can be computed by updating its results from the previous cycle, a complete reevaluation of the expression is not necessary.

The product of this processing is a REFINE program (Smith and Westfold, 1987, Smith and co-workers, 1985) that is then compiled into LISP and is executable. Additional examples of the system's capabilities are described in Smith (1987). Many such derivations required about an hour or less of the user's time, and this may be faster than

traditional methods of programming. An interesting derivation of a k-queens program appears in Smith (1990). The initial unoptimized global search program in this case required approximately one hour to enumerate the 92 solutions of the 8-queens problem. The optimized version could enumerate them in less than one second.

Other Work

Lubars and Harandi (1987) have built a knowledge-based system aimed at specification and design. Neighbors (1984) has defined two expert roles, the domain analyst and the domain designer, and attempted to code their knowledge so that new domains can be easily implemented from knowledge derived from earlier ones. Kelly and Nonnenmann (1987) have built a specification acquistion system for the creation of telephone switching software. McCartney (1987) has developed a synthesis system aimed at special problems in computational geometry. Kant (1985), Kant and Newell (1984), and Steier (1989) have studied humans in the process of algorithm design and attempted to emulate the process they observed on the Soar system. Bates and Constable (1985) have built a program development system that emphasizes logical specification of the target behavior and manipulation of the proof to derive a program. A survey of many of these projects appears in Lowry and Duran (1989).

CONCLUSION

Automatic programming is the process of mechanically assembling fragmentary information about target behaviors into machine executable code for achieving those behaviors. This article has described the four main approaches to the field followed by researchers in recent years. The field is still very much in its infancy but already many useful discoveries have been made. Because of its tremendous importance, it is clear automatic programming will be a research area central to artificial intelligence in the years to come.

Additional overviews of the subject are found in Balzer (1985), Barr and Feigenbaum (1982), Biermann (1976, 1985), Biermann and co-workers (1984), Goldberg (1986), Heidorn (1976), Partridge (1989), and Rich and Waters (1986).

BIBLIOGRAPHY

S. Amarel, "On the Automatic Formation of a Computer Program which Represents a Theory," in M. Yovits, G. T. Jacobi, and A. D. Goldstein, eds., *Self-Organizing Systems*, Spartan Books, 1962, pp. 107–175.

D. Angluin, "On the Complexity of Minimum Inference of Regular Sets," *Inf. Control* **39,** 337–350 (1978).

R. Balzer, "A 15–Year Perspective on Automatic Programming," *IEEE Trans. on Software Eng.* **SE–11**(11), 1257–1267 (1985).

R. Balzer, T. E. Cheatham, Jr., and C. Green, "Software Technology in the 1990s: Using a New Paradigm," *Computer* **16,** 39–45 (Nov. 1983).

R. M. Balzer, N. Goldman, and D. Wile, "Informality in Program Specifications," *IEEE Trans. on Software Eng.* **SE–4,** 94–103 (1978).

R. M. Balzer, N. Goldman, and D. Wile, "On the Transformational Implementation Approach to Programming," *Second International Conference on Software Engineering*, 1976, pp. 337–344.

A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*, William Kaufmann, Inc., 1982.

D. R. Barstow, *Knowledge-based Program Construction*, Elsevier North-Holland, 1979.

J. L. Bates and R. L. Constable, "Proofs as Programs," *ACM Trans. on Prog. Lang. Sys.* **7**(1), 113–136 (1985).

F. L. Bauer, B. Moller, H. Partsch, and P. Pepper, "Formal Program Construction by Transformations—Computer Aided, Intuition Guided Programming," *IEEE Trans. on Software Eng.* **SE–15**(2), 165–180 (1989).

W. Bibel, "Syntax-Directed, Semantics-Supported Program Synthesis," *Artif. Intell.* **14**(3), 243–261 (1980).

W. Bibel and K. M. Hornig, "LOPS—A System Based On A Strategical Approach to Program Synthesis," in A. Biermann, G. Guiho, and Y. Kodratoff, eds., *Automatic Program Construction Techniques*, Macmillan Publishing Co., 1984, pp. 69–90.

A. W. Biermann, "Approaches to Automatic Programming," in M. Rubinoff and M. C. Yovits, eds., *Advances in Computers*, Academic Press, 1976.

A. W. Biermann, "Automatic Insertion of Indexing Instructions in Program Synthesis," *Int. J. Comput. Inf. Sci.* **7,** 65–90 (1978).

A. W. Biermann, "Formal Methodologies in Automatic Programming: A Tutorial," *J. Symb. Comp.* **1** (1985).

A. W. Biermann, "On the Inference of Turing Machines from Sample Computations," *Artif. Intell.* **3,** 181–198 (1972).

A. W. Biermann, "The Inference of Regular LISP Programs from Examples," *IEEE Trans. on Systems, Man, and Cybernetics* **SMC–8,** 585–600 (1978).

A. W. Biermann and B. W. Ballard, "Towards Natural Language Programming," *Am. J. Computational Linguistics* **6,** 71–86 (1980).

A. W. Biermann and D. R. Smith, "A Production Rule Mechanism for Generating LISP Code," *IEEE Trans. on Systems, Man, and Cybernetics* **SMC–9,** 260–276 (1979).

A. W. Biermann and J. A. Feldman, "A Survey of Results in Grammatical Inference," in Y. H. Rao and G. W. Ernst, eds., *Context-Directed Pattern Recognition and Machine Intelligence Technologies for Information Processing*, IEEE Computer Society Press, 1982.

A. W. Biermann and R. Krishnaswamy, "Constructing Programs from Example Computations," *IEEE Trans. on Software Eng.* **SE–2,** 141–153 (1976).

A. W. Biermann, B. W. Ballard, and A. H. Sigmon, "An Experimental Study of Natural Language Programming," *Int. J. Man-Mach. Stud.* **18,** 71–87 (1983).

A. W. Biermann, I. R. Baum, and F. E. Petry, "Speeding Up the Synthesis of Programs from Traces," *IEEE Trans. Comput.* **C–24** (1975).

A. W. Biermann, G. Guiho, and Y. Kodratoff, eds., *Automatic Program Construction Techniques*, Macmillan Publishing Co., 1984.

T. J. Biggerstaff, *C2: A Super Compiler Model of Automatic Programming*, Ph.D. dissertation, University of Washington, Seattle, Wash., 1976.

L. Blum and M. Blum, "Toward a Mathematical Theory of Inductive Inference," *Inf. Control* **28,** 125–155 (1975).

M. Broy, "Program Construction by Transformations: A Family Tree of Sorting Programs," in A. W. Biermann and G. Guiho, eds., *Computer Program Synthesis Methodologies*, D. Reidel Publishing Co., 1983, pp. 1–50.

R. M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs," *JACM* **24,** 44–67 (1977).

N. Dershowitz, *The Evolution of Programs*, Birkhauser, Boston, 1983.

A. F. Fahmy, *Synthesis of Real Time Programs*, Doctoral dissertation, Dept. of Computer Science, Duke University, Durham, N.C., 1988.

J. A. Feldman, J. Gips, J. J. Horning, and S. Reder, *Grammatical Complexity and Inference*, Tech. Report CS–125, Computer Science Dept., Stanford University, 1969.

P. K. Fink and A. W. Biermann, "The Correction of Ill-Formed Input Using History Based Expectation with Application to Speech Understanding," *Computational Linguistics* **12**(1) (1986).

R. Gabriel, *An Organization for Programs in Fluid Dynamics*, Rep. No. STAN–CS–81–856, Computer Science Dept., Stanford University, 1981.

S. L. Gerhart, "Proof Theory of Partial Correctness Verification Systems," *SIAM J. Comp.* **5,** 355–377 (1976).

R. Geist, D. Kraines, and P. Fink, "Natural Language Computing in a Linear Algebra Course," *Proceedings of the National Educational Computing Conference*, 1982.

J. M. Ginsparg, *Natural Language Processing in an Automatic Programming Domain*, Rep. No. STAN–CS–78–671, Computer Science Dept., Stanford University, 1978.

E. M. Gold, "Language Identification in the Limit," *Inf. Control* **10,** 447–474 (1967).

A. T. Goldberg, "Knowledge-based Programming: A Survey of Program Design and Construction Techniques," *IEEE Trans. on Software Eng.* **SE–12**(7), 752–768 (1986).

C. Green, "The Design of the PSI Program Synthesis System," *Proceedings of the Second International Conference on Software Engineering*, San Francisco, 1976, pp. 4–18.

C. C. Green, "Application of Theorem Proving to Problem Solving," *Proceedings of the International Joint Conferences on Artificial Intelligence*, Washington, D.C., Morgan-Kaufmann, San Mateo, Calif., May 1969.

C. Green and S. Westfold, "Knowledge-based Programming Self-applied," *Mach. Intell.* **10,** Halsted Press, N.Y., 1981.

C. Green, J. Phillips, S. Westfold, T. Pressburger, B. Kedzierski, S. Angebranndt, B. Mont-Reynaud, and S. Tappel, *Research on Knowledge-based Programming and Algorithm Design—1981*, Tech. Rep. KES.U.81.2, Kestrel Institute, Palo Alto, Calif., 1981.

S. Hardy, "Synthesis of LISP Functions from Examples," *Proceedings of the Fourth IJCAI*, 1975, Morgan-Kaufmann, San Mateo, Calif., 1975, pp. 240–245.

J. Hartmanis and R. E. Stearns, *Algebraic Structure Theory of Sequential Machines*, Prentice-Hall, 1966.

G. E. Heidorn, "Automatic Programming Through Natural Language Dialogue: A Survey," *IBM J. Res. Dev.*, 302–313 (1976).

G. E. Heidorn, "English as a Very High Level Language for Simulation Programming," *Proceedings of the Symposium on Very High Level Languages*, SIGPLAN notices **9,** 91–100 (1974).

E. Kant, "The Selection of Efficient Implementations for a High Level Language," *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, SIGART newsletter **64,** 140–146 (1977).

E. Kant, "Understanding and Automating Algorithm Design," *IEEE Trans. on Software Eng.* **SE–11**(11), 1361–1374 (1985).

E. Kant and A. Newell, "Problem Solving Techniques for the Design of Algorithms," *Inf. Processing and Management* **20**(1–2), 97–118 (1984).

V. Kelly and U. Nonnemann, "Inferring Formal Software Specifications from Episode Descriptions," *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, AAAI, Menlo Park, Calif., 1987.

Y. Kodratoff and J. P. Jouannaud, "Synthesizing LISP Programs Working on the List Level of Embedding," in A. Biermann, G. Guiho, and Y. Kodratoff, eds., *Automatic Program Construction Techniques*, 1984, pp. 325–374.

M. R. Lowry and R. Duran, *Knowledge-based Software Engineering*, Tech. Rep. KES.U.89.4, Kestrel Institute, Palo Alto, Calif., 1989.

M. D. Lubars and M. T. Harandi, "Knowledge-based Software Design Using Design Schemas," *Ninth International Conference on Software Engineering*, IEEE Computer Society Press, 1987, pp. 253–262.

Z. Manna and R. Waldinger, "A Deductive Approach to Program Synthesis," *Trans. Prog. Lang. Sys.*, 1980.

Z. Manna and R. Waldinger, "Synthesis: Dreams=>Programs," *IEEE Trans. Software Eng.* **SE–5,** 294–328 (1979).

Z. Manna and R. Waldinger, "The Origin of a Binary-Search Paradigm," *Sci. Comput. Prog.* **9,** 37–83 (1987).

W. A. Martin, M. J. Ginzberg, R. Krumland, B. Mark, M. Morgenstern, B. Niamir, and A. Sunguroff, *Internal Memos*, Automatic Programming Group, Massachusetts Institute of Technology, Cambridge, 1974.

R. McCartney, "Synthesizing Algorithms with Performance Constraints," *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, AAAI, Menlo Park, Calif., 1987.

B. P. McCune, "The PSI Program Model Builder: Synthesis of Very High-level Programs," *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, SIGART newsletter **64,** 130–139 (1977).

J. Neighbors, "The DRACO Approach to Constructing Software from Reusable Components," *IEEE Trans. on Software Eng.* **SE–10**(5), 5–27 (1984).

H. Partach and R. Steinbruggen, *ACM Computing Surveys*, **15**(3), 199–236 (1983).

D. Partridge, ed., *Artificial Intelligence and Software Engineering*, Ablex Publishers, New York, 1989.

J. V. Phillips, "Program Reference from Traces Using Multiple Knowledge Sources," *Inter. Joint Conf. on Artif. Intell.* **5,** 812 (1977).

C. Rich and H. E. Shrobe, "Initial Report on a LISP Programmer's Apprentice," *IEEE Trans. on Software Eng.* **SE–4,** 456–467 (1978).

C. Rich and R. C. Waters, eds., *Readings in Artificial Intelligence and Software Engineering*, Morgan-Kaufmann, Los Altos, Calif., 1986.

C. Rich and R. C. Waters, "The Programmer's Apprentice," *IEEE Comput.* **21**(11), 10–25 (1988).

E. Y. Shapiro, *Algorithmic Program Debugging*, M.I.T. Press, Cambridge, Mass., 1982.

D. Shaw, W. Swartout, and C. Green, "Inferring LISP Programs from Examples," *Inter. Joint Conf. on Artif. Intell.* **4,** 260–267 (1975).

D. R. Smith, *A Class of Synthesizeable LISP Programs*, A. M. thesis, Duke University, 1977.

D. R. Smith, "Applications of a Strategy for Designing Divide-and-Conquer Algorithms," *Sci. Comp. Prog.* **8,** 213–229 (1987).

D. R. Smith, "KIDS: A Semi-Automatic Program Development System," *IEEE Trans. on Software Eng.* (Sept. 1990).

D. R. Smith, *Structure and Design of Global Search Algorithms*, Tech. Rep. KES.U.87.12, Kestral Institute, November 1987.

D. R. Smith, "The Synthesis of LISP Programs from Examples: A Survey," in A. Biermann, G. Guiho, and Y. Kodratoff, eds., *Automatic Program Construction Techniques*, Macmillan Publishers, 1984, pp. 307–324.

D. R. Smith and S. Westfold, "Application of REFINE to Knowledge-based Modeling," Application Note 1.3, Reasoning Systems, Inc., 1987.

D. R. Smith, G. B. Kotik, and S. J. Westfold, "Research on Knowledge-based Software Environments at Kestrel Institute," *IEEE Trans. on Software Eng.* **SE–11** (1985).

R. Solomonoff, "A Formal Theory of Inductive Inference," *Inf. Control* **1–22,** 224–254 (1964).

L. Steinberg, *A Dialogue Moderator for Program Specification Dialogues in the PSI System*, Doctoral dissertation, Computer Science Dept., Stanford University, 1980.

D. M. Steier, *Automating Algorithm Design Within a General Architecture for Intelligence*, Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, 1989.

P. D. Summers, "A Methodology for LISP Program Construction from Examples," *JACM* **24,** 161–175 (1977).

N. L. Tinkham, *Induction of Schemata for Program Synthesis*, Doctoral dissertation, Dept. of Computer Science, Duke University, Durham, N.C., 1990.

R. J. Waldinger and R. C. T. Lee, "PROW: A Step Toward Automatic Program Writing," *Proceedings of the International Joint Conferences on Artificial Intelligence*, Washington, D.C., Morgan-Kaufmann, San Mateo, Calif., May 1969.

D. A. Waterman, W. S. Faught, P. Klahr, S. J. Rosenschein, and R. Wesson, "Design Issues for Exemplary Programming," in A. Biermann, G. Guiho, and Y. Kodratoff, eds., *Automatic Program Construction Techniques*, Macmillan Publishing Co., 1984, pp. 433–461.

R. C. Waters, "The Programmer's Apprentice: A Session with KBEmacs," *IEEE Trans. on Software Eng.* **SE–11**(11), 1296–1320 (1985).

R. C. Waters, "The Programmer's Apprentice: Knowledge-based Program Editing," *IEEE Trans. on Software Eng.* **SE–8**(1), 1–12 (1982).

ALAN W. BIERMANN
Duke University

Preparation of this article was funded partially by Army Research Office Grant Number DAAG-29-84-K-0072 and the National Science Foundation Grant Number IRI 8803802).

ENCYCLOPEDIA OF ARTIFICIAL INTELLIGENCE

This extensively revised and expanded *Second Edition* of the *Encyclopedia of Artificial Intelligence* defines the discipline by bringing together the core of knowledge from all fields encompassed by AI. It covers the latest developments in current AI topics such as neural networks, fuzzy logic, machine vision, natural language generation, and many more. Includes:

- Over 450 articles—all entries written expressly for the *Encyclopedia*
- Over 5,000 literature references; 454 illustrations and color photographs
- Over 50% new and revised material
- Exemplary indexing and cross-referencing for easy, complete information access to all topics

Praise for the *First Edition*...

"The *Encyclopedia* is a wonder of clarity and scope: surprisingly easy to read...the clarity is an especially pleasant surprise, considering the articles were all written by AI experts...It's a treasure house of easily accessible knowledge."
—*Language Technology*

"Excellent bibliographies are attached to most of the articles, and diagrams and sketches are clear and helpful. The indexing and cross-indexing are exemplary. As the editor points out, the reader will be led by the extensive cross-references to almost every other article..." —*Artificial Intelligence Reporter*

"The *Encyclopedia* is a first-class piece of work that will be an indispensable part of any AI library." —*Computing Reviews*

"...A tour de force...a truly fantastic encyclopedia which no one in the field of artificial intelligence can afford to be without."
—*Systems Research & Information*

WILEY-INTERSCIENCE
John Wiley & Sons, Inc.
Professional, Reference and Trade Group
605 Third Avenue, New York, N.Y. 10158-0012
New York • Chichester • Brisbane • Toronto • Singapore
ISBN 0 471-50307-X (Two-Volume Set)
ISBN 0 471-50305-3 (Vol. 1)
ISBN 0 471-50306-1 (Vol. 2)

ISBN 0-471-50305-3

9 780471 503057 90000