STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-240

STAN-CS-74-444

PROGRESS REPORT ON PROGRAM-UNDERSTANDING SYSTEMS

BY

C. CORDELL GREEN, R. J. WALDINGER, DAVID R. BARSTOW,
ROBERT ELSCHLAGER, DOUGLAS B. LENAT, BRIAN P. McCUNE,
DAVID E. SHAW, AND LOUIS I. STEINBERG

SUPPORTED BY

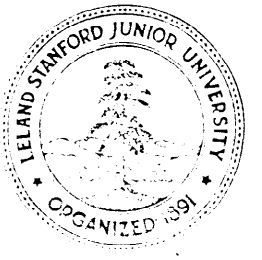
ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 2494

AUGUST 1974

COMPUTER SCIENCE DEPARTMENT

School of Humanities and Sciences

STANFORD UNIVERSITY



STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-240

AUGUST 1974

COMPUTER SCIENCE DEPARTMENT
REPORT STAN-CS-74-444

PROGRESS REPORT ON PROGRAM-UNDERSTANDING SYSTEMS

by

C. Cordell Green, Richard J. Waldinger,

David R. Barstow, Robert Elschlager, Douglas B. Lenat,

Brian P. McCune, David E. Shaw, and Louis I. Steinberg

Abstract This progress report covers the first year and one half of work by our automatic-programming research group at the Stanford Artificial Intelligence Laboratory. Major emphasis has been placed on methods of program specification, codification of programming knowledge, and implementation of pilot systems for program writing and understanding. List processing has been used as the general problem domain for this work.

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract DAHC 15-73-C-0435, in part by the National Science Foundation through an NSF Graduate Fellowship, and in part by the State of California through a California State Fellowship. Richard J. Waldinger was affiliated with the Artificial Intelligence Center of the Stanford Research Institute during the period of this research.

The views and conclusions in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the US Government.

Reproduced in the USA. Available from the National Technical Information Service, Springfield, Virginia 22151.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the helpful criticisms of drafts of this report given by D. Bruce Anderson, Avra J. Cohn, and C. A. R. Hoare. Computer time for much of the research reported herein was made available by the Artificial Intelligence Center of the Stanford Research Institute and the Information Sciences Institute of the University of Southern California.

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Goals

The object of this research is to pursue the question of whether it is possible to develop an intelligent computer system that both understands and writes programs. The research includes high-level methods of specifying programs, codification of programming knowledge, and implementation of working program-writing systems. The domain of programming knowledge ranges from the fundamentals of programming through list processing to simple searching, sorting, and inductive-inference programs. Much of this knowledge is more-or-less pure programming knowledge, along with such domain-dependent knowledge as is necessary. A major emphasis is the codification of the considerable body of list-processing and fundamental programming knowledge. In the implementation aspect of our research, an eventual target system is expected to have a deep understanding of programming as demonstrated by its program-writing ability, its line of reasoning in creating a program, and its own discussion of why it made each choice and what factors were involved.

1.2 Progress

One of our earliest efforts was an exploration of more "human" methods of program specification, such as example input-output pairs, program traces, and generic examples. In the area of codification of programming knowledge, we have developed sets of rules for program synthesis that cover low-level list and register operations, several types of generate and process paradigms, and simple searching and sorting programs. We have implemented 7 different programs that do all or part of the job of program

synthesis. The more recent programs have been moderately successful. In particular, they can (1) write list-transformation programs, given example input-output pairs; (2) write low-level list- and register-manipulation programs; (3) write 3 sorting and permutation programs; and (4) write a concept-formation program.

1.3 Organization of the Report

The reader should note that material in this progress report is presented roughly chronologically and that some of our false starts have been included for historical completeness. Consequently, our later (and hopefully more successful) work is presented towards the end of the paper. Some readers might wish to scan the first parts and focus on Sections 4.4 through 4.7.

Section 2 of this report represents our initial explorations into declarative methods for specifying procedures. These methods include both individual and generic examples of input-output pairs to which the program being specified must conform; traces of the input(s), output, and perhaps intermediate values throughout the execution of the program; high-level programming operations and concepts expressed in English words and phrases; and combinations of these. As a result of the conciseness of such program descriptions, they are often incomplete or ambiguous. Some of the methods of Section 2 are utilized in the running systems discussed in Section 4.

Section 3 is a brief discussion of what we view as one of the most important aspects of research in automatic programming: the codification of programming knowledge so that it can be used by a system which understands and writes programs. Concrete examples of such knowledge are given in Section 4 for some of the systems currently implemented.

Section 4 embodies the history of actual program-understanding systems which have been implemented by our group over the past year and one half. These systems span a wide range of input-specification types, built-in programming and task-domain knowledge, and target-program complexity, but they all have the programming domain of list processing in common. Although the systems are discussed in chronological order for the sake of continuity, the reader should note that our most recent and continuing efforts involve the final 3 systems [see Sections 4.5 - 4.7].

2. METHODS OF PROGRAM SPECIFICATION

One of our goals is to find better ways for people to specify programs. A central question is whether or not there exist any methods or languages that are better than those that currently exist. It is possible that, say, ALGOL is the best language for specifying a particular algorithm. However, it seems that for certain programs we can find new descriptions that are easier for people to use. Certainly, very good special-purpose languages can be designed for particular application areas.

We will present a few methods for specifying list-processing programs. It is not yet clear which methods are suited to which classes of programs. Some methods considered so far are examined below. Most of these have evolved from discussions within our group. McCune has contributed the most recent efforts at analyzing and cataloging them.

In general, our target user is a person familiar with programming and the subject domain of the desired program, but not necessarily with the details of that program or the language in which it is to be implemented.

2.1 Example Input-output Pairs

Grammatical inference and the inference of automata from ordered pairs representing example input-output behavior have been investigated [3,4,7,12,17]. Example input-output pairs can similarly be used to describe low-level list-transformation algorithms [1,14].

Consider the program that "flattens" a list. An example of its behavior is as follows:

input		output
(A (B C (D) E))	⟶	(A B C D E)

This example pair is quite simple to write and to most people specifies the desired effect of the program, but not the detailed operation. Note that if we add the phrase "remove inner parentheses" to the input-output pair description, the intent is even clearer. Of course, we still don't know whether to create a new list or modify the input list unless this, too, is specified.

Another list transformation that is easily specified by example is

input		output
(A B C D)	⟶	((A B)(A C)(A D)(B C)(B D)(C D))

which describes the generation of all 2-element combinations from a list.

A simple observation is that several I/O pairs may be required to specify a program (actually a class of equivalent programs) unambiguously. One disadvantage of this method is that the program inferred by the system may not be the intended program. Also, examples have to be carefully chosen. Hopefully the program-writing program will have some model of human preferences and will not infer, say, the function having the constant output (A B C D E) from the flatten example given above. In cases where it is difficult to disambiguate the intended program using only examples, other information sources could be used. These include programming context and simple descriptors like "a recursive function, not merely table look-up". The program-writing program should verify with the user that its choice of program is what the user intended. One way to do this is to automatically generate for the user a new I/O pair that disambiguates among the major candidates.

In any program-specification method that requires some inference on the part of the computer, there will be a chance that the computer will synthesize the wrong program. This lack of control is especially upsetting to good programmers. However, high-level specifications are invariably inexact, which leads to the need for inference to fill in details. More research on this problem area is required, but any solution would seem to require a high degree of 2-way dialog between the user and the system.

2.2 Program Traces

Some work has been done on the inference of programs from traces [2]. This method is more complete than example I/O pairs in that it tends to describe the algorithm used to compute the output, as well as the input-output relation. Thus the pair

input		output
(3 1 4 2)	⟶	(1 2 3 4)

specifies a sort. But the trace of the input and output

	input	output
initially:	(3 1 4 2)	()
next:	(1 4 2)	(3)
next:	(4 2)	(1 3)
next:	(2)	(1 3 4)
finally:	()	(1 2 3 4)

implies an insertion-sort algorithm, with details omitted.

We would like to emphasize a new aspect of program inference from traces, namely, the utilization of several knowledge sources to write the program. These sources include the subject domain for which the program is written, a knowledge of what the common operations are, and other

specifications that are given for the program. The user could supply further information by annotating the traces to provide disambiguation or further specification, as is discussed in Section 2.9. An example of additional specification of the sort program above might be the word "recursive".

2.3 Generic Examples

Generic examples lie somewhere between example I/O pairs and formal predicate-calculus I/O specifications [13,32] in explicitness. The ellipsis notation is used to specify an indefinite number of elements. For example, the specification

input		output
$(x_1\ x_2\ x_3\ \cdots\ x_n)$	$\longrightarrow$	$(x_n\ x_{n-1}\ x_{n-2}\ \cdots\ x_1)$

gives the reverse function. The alternate function may be specified by

input		output
$(x_1\ x_2\ x_3\ x_4\ x_5\ \ldots)$	$\longrightarrow$	$(x_1\ x_3\ x_5\ \ldots)$

This notation is, of course, ambiguous, and a verification phase would have to confirm the hypothesized program.

2.4 Generic Traces

Similarly, the ellipsis notation can be used in a trace. As an example, here is a generic trace which specifies the combinations of elements of a set taken 2 at a time:

car(input)	cdr(input)	output
x_1	$(x_2\ x_3\ x_4\ \ldots)$	$((x_1\ x_2)(x_1\ x_3)(x_1\ x_4)\ldots)$
x_2	$(x_3\ x_4\ \ldots)$	$((x_1\ x_2)(x_1\ x_3)(x_1\ x_4)\ldots(x_2\ x_3)(x_2\ x_4)\ldots)$
.	.	.
.	.	.
.	.	.

2.5 Graphical Descriptions

Pictures of input and output are obviously well suited for depicting simple list transformations in which the structures are difficult to describe in linear strings, yet easy to describe in 2 dimensions [19]. We have not investigated any of these methods.

2.6 Conceptual Descriptions

High-level program description is, of course, the most convenient specification technique if the right high-level primitives are available. In the extreme case we would just give the name (or number) of the desired program. More interesting cases for automatic-programming studies are those in which there is some distance from the primitives to the program description.

For the domain being considered (list transformations), nice conceptual descriptors (primitives) include "element conserving", "order preserving", "represents a set", "represents a tree", "represents a graph", "permutation", "table look-up", etc. These can be embedded in either inherently ambiguous or unambiguous languages (ranging from versions of English to unambiguous high-level, but conventional, programming languages) and can either partially or completely specify the program. We would like to emphasize partial

descriptions, ambiguous languages, and primitives that are not quite high-level enough to make the task too easy for the system. By combining several ambiguous partial descriptions with knowledge of the programming domain, a system may be able to decipher descriptions that humans can easily produce. (Conventional programming languages that are completely descriptive and unambiguous, but lacking primitives of a high enough level, are still of interest.)

2.7 Natural-language Descriptions

As used by documentors and describers of algorithms [19], natural (English) language mixed with mathematical and programming jargon can be an effective method for communicating an algorithm. Good English-like program descriptions can be easily understood by humans, although again it's not clear under what circumstances they are the easiest descriptions to generate. English descriptions can, of course, describe input-output relations or algorithms, be partial or complete, high-level or low-level, interactive or not, etc. Here is an example of a partial algorithm specification [20]:

> An exchange sort. If two items are found to be out of order, they are interchanged. This process is repeated until no more exchanges are necessary.

We intend to examine the issues of when English is a useful adjunct in program description and how a programming system might deal with it. Elschlager is studying natural-language descriptions of programs in order to develop an appropriate internal representation for them. From this has come a representation which is primarily relational, but also has

qualification and quantification primitives. Possible inputs into this system might be either a limited subset of English or a more rigidly structured "parenthesized" English. Future work includes relating this internal representation of a program to the programming concepts and data structures it will use.

2.8 User-machine Dialog

A simple and useful vehicle for embedding the various methods of program specification is conversational interaction or dialog between the user and the automatic-programming system. Such a style was used by Floyd [10] in his suggestion for a system aimed at interactive program verification and synthesis. With such a system there is a need for a large amount of programming knowledge of both a domain-specific and general nature. The system would draw on this knowledge in order to understand the user's remarks and pose meaningful questions.

One could imagine a conversation beginning with the user's specification of the general program desired and proceeding in a basically top-down manner. As the system needed more precise information, it would ask for it. Presumably the user could indicate a preference for a particular programming decision (e.g., choice of data representation) or possibly a lack of preference, in which case the system would make its own reasonable choice. A variety of methods of specification (e.g., examples, traces, formal specifications, etc.) might play a role in such a conversation. This yields a paradigm similar to that of structured programming [6], but one in which the computer fills in the details.

A typical low-level dialog might proceed as follows (for readability the dialog is mostly English):

USER: Please write me a program which extracts information from pairs in an input list.

MACHINE: What will the input be?

USER: The input will consist of 2 parts: an atom and a list of pairs.

MACHINE: What should the output be?

USER: A list of the other half of the pairs which contain the input atom.

MACHINE: What do you mean by "the other half of the pairs"?

USER: If the first element of one of the pairs is the input atom, I want the second element; if the second element is the input atom, I want the first element.

MACHINE: What shall be done if one of the pairs is (x x) , where x is the input atom?

USER: Then the output list should include x .

MACHINE: OK, I think that's enough.

[pause]

MACHINE: Wait a minute. Does the order of the elements in the output list matter?

USER: No, not at all.

MACHINE: Then I will simply use the order in which the pairs occur in the input list, since that is the easiest. In this case I could overwrite the input list. Should I do that, or would you rather that the list be preserved?

USER: Preserve it.

[pause]

MACHINE: Do you want this written as a function of the 2 input arguments?

USER: Sure, call the function f .

MACHINE: How's this:

```
f(x, y) ← if null(y) then nil else
          if x = caar(y) then cons(cadar(y), f(x, cdr(y))) else
          if x = cadar(y) then cons(caar(y), f(x, cdr(y))) else
          f(x, cdr(y));
```

USER: Looks good to me.

The above dialog is at such a low level that it is completely dependent upon the fact that LISP lists are being dealt with, thus requiring the user to have some familiarity with LISP. Higher-level dialogs of domain-specific programs can be less representation dependent and can be carried on in the vocabulary of the particular problem domain.

2.9 Information Necessary to Complete the Specification of a Program

In completing the specification of a program, we can imagine a "checklist" that a program-writing system might have for each type of program it can handle. It might work on completing its checklist by inference from partial specifications, interactions with the user, context, and default conditions. Such a checklist might include terminating conditions, auxiliary functions, restrictions on input (e.g., whether a list has constant or variable length), what data representations are available, etc. Certainly a program-understanding system needs to ask many questions about the target program. (But not, "What's the first instruction? Now, what's the second? ...")

2.10 A Comparative Example

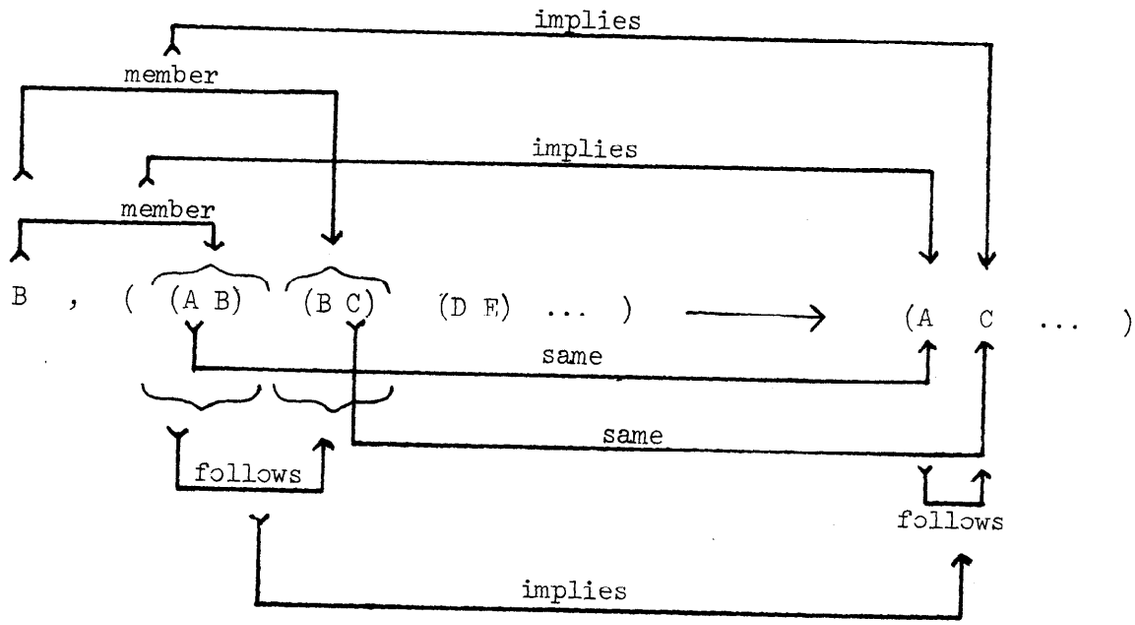
Let's consider the specification of a simple program as a vehicle for discussion of the merits of various methods of description. Consider the following example of the association search synthesized in Section 2.8:

input 1	input 2	output
B	((A B)(B C)(D E)...)	⟶ (A C ...)

Note that we've incorporated the ellipsis notation of generic examples into an example input-output pair. Subjectively, this specification seems not as thorough as we might wish. Can input 1 be non-atomic? What if (B B) occurs in input 2? What if an element of input 2 is atomic? Etc.

As the complexity of the transformation increases, example input-output pairs begin to require more inference to determine the intended transformations. One way out is to clarify the intended function by describing more elementary relations between input and output elements, namely, "The letters A and C are in the output because they occur in the second input paired with B (the first input)". If we allow a higher-level concept, it is even easier to describe: "a commutative LISP assoc operation". This phrase describes the function fairly clearly (to a LISP programmer). The added description, "order preserving", explains why C follows A in the output, but a reasonable program should assume (and test) order preservation in the absence of other information. Obviously the conceptual descriptions alone, without the example, do not clearly determine the intended program. Together they do a reasonable job.

As another more explicit technique, McCune and Lenat have suggested describing the lower-level relations for the above example graphically as, say,



This scheme clarifies why each element of the output is where it is and from where in the input it came.

Of course, a partial or even complete, but precise description can be given in predicate calculus [13,32]. Here is one possibility:

$$(\forall\, v, w, x, y, z)\ [\mathrm{input}(x, y) \wedge \mathrm{output}(z) \wedge \mathrm{atom}(x) \wedge \mathrm{list}(y) \wedge \mathrm{list}(z) \wedge \mathrm{sublist}(w, y) \wedge \mathrm{length}(w, 2) \wedge \mathrm{member}(x, w) \wedge \mathrm{member}(v, w) \wedge (x \neq v \vee \forall u[\mathrm{member}(u, w) \supset u = v])] \equiv \mathrm{member}(v, z)$$

$$(\forall\, t, u, v, w, x, y, z)\ [\mathrm{input}(x, y) \wedge \mathrm{output}(z) \wedge \mathrm{list}(y) \wedge \mathrm{list}(z) \wedge \mathrm{member}(v, z) \wedge \mathrm{member}(w, z) \wedge \mathrm{sublist}(t, y) \wedge \mathrm{sublist}(u, y) \wedge \mathrm{member}(v, t) \wedge \mathrm{member}(w, u) \wedge \mathrm{before}(t, u, y)] \equiv \mathrm{before}(v, w, z)$$

(where before(t, u, y) means element t occurs before element u in list y).

At this low level the above formal description, which may or may not be correct, appears to be at least as difficult to write correctly as the program itself. The program (in an imaginary version of Meta-LISP) is merely

```
f(x, y) ← if null(y) then nil else
          if member(x, car(y)) then
              append(delete(x, car(y)), f(x, cdr(y))) else
          f(x, cdr(y));
```

The low-level LISP program (which doesn't make use of the functions member , append , and delete) is just

```
f(x, y) ← if null(y) then nil else
          if x = caar(y) then cons(cadar(y), f(x, cdr(y))) else
          if x = cadar(y) then cons(caar(y), f(x, cdr(y))) else
          f(x, cdr(y));
```

As another alternative, a program trace is a fair way to describe the program:

input 1	car(input 2)	output
B	(A B)	(A)
B	(B C)	(A C)
B	(D E)	(A C)
.	.	.
.	.	.
.	.	.

3. CODIFICATION OF PROGRAMMING KNOWLEDGE

The easy part of codifying programming knowledge is the now more-or-less conventional formal specification of the semantics of each operation in one's programming language [9, 15, 23]. The more interesting aspect is the concrete specification of high-level programming constructs (e.g., a loop with an exit), and those programming methods that are used in the process of designing a program, but never appear explicitly in the program. An example is the detailed specification of sufficient methods for performing a generate-and-test operation on an implicit representation of a set. Newell [24] has presented a fairly high-level (non-programmable) description of 5 common artificial-intelligence problem-solving methods, including generate and test, heuristic search, hill climbing, match, and induction. Much of the work in structured programming [6] has been aimed at explicating such programming methodology, but has generally been at too high a level for implementation, being aimed at human programmers. We have begun to codify and embed this type of knowledge in 2 of our systems [see Sections 4.6 and 4.7].

How big a body of knowledge are we interested in, and how much detail is needed? Our crude preliminary estimate is that something like a few thousand "facts" (any convenient chunks of knowledge, such as production rules, axioms, or goal statements) could enable a program to understand simple list-processing programs. We have generated a proposed set of facts necessary for a program-understanding system to understand very simple insertion- and selection-sort programs. 100 to 200 facts seem adequate, without counting either the semantics of LISP or any efficiency or optimization knowledge. Including these other knowledge sources would

bring us to several hundred. Manna and Waldinger's experience [22] with the domain of pattern matching indicates that about 75 facts are sufficient to enable the construction of a unification algorithm (leaving out efficiency, programming-language semantics, and high-level program-construction concepts).

Such estimates, crude as they are, give us an idea of how smart a program-understanding system might become in the next few years; that is, we can expect a system to deeply understand a very small set of programs.

Our plans are to finish the characterization of simple sorting and then to consider simple tree searching, table look-up, and set operations. At the same time we will increase our emphasis on the automatic selection of representations. These areas all involve more-or-less "general" programming knowledge and are not too domain specific. Our first more domain-specific area under attack is that of concept-formation programs [18, 34], a class of inductive-inference programs that encompasses enough general programming knowledge to be interesting for that reason. We are currently defining a set of increasingly complex concept-formation programs to pace our efforts. PUP5 [see Section 4.6] indicates that there are about 75 units of knowledge necessary to write a concept-formation program, where each unit contains about a dozen facts.

It would be nice to know the size of the body that constitutes the "core" of programming knowledge. As yet, we can only guess. Finding the knowledge is still a more-or-less linear process; that is, to add a new capability to an understanding system requires about as much time and effort as it took to add the previous capability. We are beginning to find some commonality in the utilization of previously codified knowledge,

but it's too early yet to make any claims of great insight. However, we do have a fair degree of faith that there is a subject-independent core that we will slowly extract and refine.

4. IMPLEMENTATION OF PROGRAM-UNDERSTANDING SYSTEMS

For the sake of historical completeness, we will discuss 3 early implementations that are of limited significance before discussing our later, more successful systems. Perhaps the main conclusion to be drawn from these is that small efforts seem inadequate for serious progress in program-understanding systems. Good programming systems will be very large and complex and will take many man-years of work.

4.1 Schema Instantiation to Fit Example Input-output Pairs

The first running system in our group was Lenat's PW1, which was implemented in MLISP [30]. It takes as input several example input-output list pairs and produces as output LISP programs. The idea is simple: most elementary programs in the class of interest have 1 or 2 termination conditions followed by a recursive call. The structure of such a program can be given by a few high-level schemata.

The system infers the number and type of arguments by examining the example input-output pairs. From the number of arguments either the 1-input schema or the 2-input schema is selected. The 1-input schema is

$$
\begin{aligned}
&f(x) \leftarrow \\
&\qquad \underline{\text{if}}\ f_1(x) = c_1\ \underline{\text{then}}\ f_2(x)\ \underline{\text{else}} && \text{[line 1]} \\
&\qquad \underline{\text{if}}\ f_3(x) = c_2\ \underline{\text{then}}\ f_4(x)\ \underline{\text{else}} && \text{[line 2]} \\
&\qquad f_5(f_6(f_7(x)),\ f_8(f_9(x))); && \text{[line 3]}
\end{aligned}
$$

where f_1 through f_9 are functions and c_1 and c_2 are constants, all to be determined later. Lines 1 and 2 correspond to termination conditions, and line 3 corresponds to a recursive call.

The user is asked if the function is recursive. (If it is not, line 3 is not used.) The default condition is to assume a recursive function, but no attempt is made to guess that the function is recursive. The automatic program writer next determines, again by asking, whether there are 1 or 2 terminating conditions (i.e., line 1 only or both lines 1 and 2) and whether the user wants to suggest either the test or the value for lines 1 or 2.

Whatever pieces are not supplied by the user are filled in by a constrained search process that also fills in the functions in line 3. The search proceeds as follows. First, an ordered set of candidates is formed for each subfunction and constant. The user can give advice in the form of suggested subfunctions that are likely to occur. A second information source is the type (atom, list, or number) of each argument. These factors are combined, using a rating table containing the probability of each known function appearing in a particular schema position, to yield a final ordering. Then the candidate instances of the schema are generated one by one, in accordance with the orderings of the subfunctions.

Several tricks prune the search space. A function is not applied to the wrong number or type of arguments. To check this the instantiated schema is run on the examples, and checking occurs at every step of execution. Infinite recursions are detected and prevented. "Infinity" is a parameter set in advance, usually to a number between 17 and 100. The function being defined may only occur in line 3, the recursion step, and its arguments in the recursive call cannot be the same arguments it receives in the original call. Some check should be made that the arguments are somehow moving toward the termination form, but actually any

perceived change is allowed. Several special subfunctions, such as the identity function and a projection (or selection) function, are provided to enable the desired program to be forced into one of the 2 given procrustean beds.

The program is known to have generated at least 8 correct programs, but run out of time on most other attempts. Among the programs PW1 wrote are

function name	function operation
sub2	subtract 2 from the (numeric) argument [from 2 examples: 2 → 0 and 7 → 5]
last	produce a 1-element list containing only the last element of the input list [from 2 examples: (A B) ⟶ (B) and (A B C D E) ⟶ (E)]
reverse	reverse a list [from 1 example: (A B C D E) ⟶ (E D C B A)]
Fibonacci	the obvious [from 3 examples: 1 → 1 , 6 → 8 , and 7 → 13]
factorial	the obvious [from 2 examples: 1 → 1 and 4 → 24]
insert	insert a number into its proper place in an ordered list of numbers [from 3 examples: 2, (1 3 8) ⟶ (1 2 3 8) ; 2, (8) ⟶ (2 8) ; and 7, (1 5) ⟶ (1 5 7)]
sort	sort a list of numbers, given insert as a primitive function [from 4 examples: (2 3) ⟶ (2 3) , (3 2) ⟶ (2 3) , (1 7 6 4) ⟶ (1 4 6 7) , and (8 1 2 5 3 9) ⟶ (1 2 3 5 8 9)]
flatten	change a tree into a single-level list of the atoms in the tree [from 1 example: (A (B C (D E)) F) ⟶ (A B C D E F)]

This approach appeared to have limited potential, so no controlled experiments were run. The main disadvantage was that the program had a limited model of its task and little programming knowledge, so it consequently engaged in large searches.

4.2 Sequence-extrapolator Writer

This was an INTERLISP [31] program by Lenat. The question was whether it is possible to write a highly specialized program writer that produces programs for a given sub-area of inductive inference, in this case sequence extrapolation [25, 29]. Other specialized program-writing programs, like compilers and compiler-compilers, have been around for a while. This new task turned out to be easy.

The program begins with a schema for a generalized sequence-extrapolation program consisting of 5 subparts. The user describes, via a dialog directed by a decision tree, which capabilities are to be included for each subpart. (Not all choices are independent, however.) The system then includes the appropriate pieces of program or data that meet this description. For example, for the subpart of known sequences, the user indicates which sequences should be immediately recognizable by exact match.

Not much was learned, except that it is possible to write a highly specialized program writer for this domain. We can guess that it would be easy to turn out specialist program writers for other simple, well-structured domains. The system had little of the character of what we call an understanding system.

4.3 Ellipsis Translator

This was a small study and INTERLISP program by Shaw designed to translate a class of ambiguous generic examples into a list of candidate unambiguous internal representations. For example, the program translates $(x_2 + x_4 + \ldots + x_n)$ into the 2 unambiguous interpretations

$$\sum_{1 \le i \le n/2} x_{2i} \qquad \text{and} \qquad \sum_{1 \le i \le \log_2 n} x_{2^i}$$

(although the 2 interpretations are not represented internally in a form isomorphic to the above). The experimental program was not pushed, so it never left the nearly debugged stage. However, there are a few comments and observations we can make.

The notation seems to be useful, and the intent of the user is often easy to guess by straightforward techniques. First, observe that finding an interpretation reduces to sequence-extrapolation on the indices of the variables. Sequence-extrapolation techniques [25, 29], including successive differences, successive quotients, and tests for common sequences, have allowed the construction of relatively powerful sequence extrapolators that behave well and usually produce the desired interpretation, although a non-cooperative user can often evoke a false interpretation. A more serious problem is that of communicating to a cooperative user the algorithm used to interpret the ellipsis notation and either verifying that the first candidate is the intended interpretation or else finding it by some interactive procedure.

The internal representation of the meaning does not appear to be a problem, and good ones should fall out naturally when an ellipsis-translating mechanism is incorporated into a larger program-understanding system.

An ideal system should, of course, be forgiving. For example, it should produce the same interpretation for the following 4 styles:

$(x_1 + x_2 + x_3 + \ldots + x_n)$

$(x_1 + x_2 + x_3 + \ldots \quad x_n)$

$(x_1 + x_2 + x_3 \quad \ldots + x_n)$

$(x_1 + x_2 + x_3 \quad \ldots \quad x_n)$

If the user provides a meaningfully subscripted last element, that information should be used. For example, in $(x_2\ x_4\ \ldots\ x_{2n})$ the last element should resolve the ambiguity in the sequence beginning 2, 4, ... Our ideal system should also handle interleaved sequences (say, from different sources), such as $(x_1\ y_2\ x_3\ y_4\ \ldots)$; specified intermediate elements, such as $(x_1\ x_3\ \ldots\ x_{2i+1}\ \ldots)$; deleted elements, perhaps represented as $(x_1\ x_2\ \ldots\ \neg x_i\ \ldots\ x_n)$ or in other ways; and various operators, such as + , - , etc.

Waldinger has suggested that a more powerful induction mechanism be used to allow "formula extrapolation", e.g., to handle examples such as (A, B, AA, AB, BA, BB, ...) . Such a mechanism could be of use in specifying more complex, but frequently used, enumeration algorithms. Fusaoka [11] has implemented an embryonic formula extrapolator.

4.4 Our Simplest Program-understanding Program

The next program showed some rudimentary program-understanding behavior. It dealt with simple list manipulation, assignment operations, and arithmetic. The 2 versions of the program were Lenat's PUP1 and a revised version, PUP2, by Steinberg. Both versions of PUP were written in QLISP [26] (the successor to QA4 [27]) and INTERLISP.

The specification of the program to be written is basically a formal input-output relation. The program is structured around QLISP goal statements, which specify both the desired state and an "apply" list of subprograms that may be able to achieve that state. A subprogram may achieve the goal state directly or may decompose the goal into subgoals and use goal statements to achieve these. We'll describe several of the tasks PUP accomplished, along with a description of the stored facts used in each case.

4.4.1 Interchange of Elements This is a simple problem, similar to one solved by Simon's Heuristic Compiler [28]. The problem statement is

initial state	final state
contents(x) = a	contents(x) = b
contents(y) = b	contents(y) = a

The initial state is assumed and the final state taken as the goal. One of the programs on the apply list decomposes goals of the form $\alpha \wedge \beta$ into the separate conjuncts and uses goal statements to attain first one, then the other, in a more-or-less depth-first manner.

The program that handles the subgoal contents(x) = b sees that contents(y) = b is true and so adds x ← y to the program being written. It also adds a comment " x previously contained a " at that point in the program and updates the world model to say that contents(x) = b now holds. Next, this same program is given the subgoal contents(y) = a and finds that a no longer exists, so it looks back in the program to find where a was destroyed. It finds the comment " x previously contained a " and so

patches the program to save a in a temporary variable before it is destroyed. The program now looks like

```
begin
temp ← x;
x ← y; comment  x  previously contained  a ;
```

Now a exists in temp ; so the program can achieve contents(y) = a by

```
y ← temp; comment  y  previously contained  b ;
end;
```

The interesting issue here is whether to look ahead when a is destroyed and predict that it will be needed again, or to go back and patch if the need is discovered. In this case patching was much easier than predicting, largely because a comment was made in order to facilitate any needed patching. (Far better programmers than PUP use many comments for just that purpose.)

4.4.2 3-element Sort This problem, sorting the contents of 3 cells without using recursion or iteration, is non-trivial even for humans. Experienced programmers can take several minutes and often come up with incorrect programs. Formally, the problem is

initial state	final state
contents(x) = a	contents(x) $\leq$ contents(y)
contents(y) = b	contents(y) $\leq$ contents(z)
contents(z) = c	contents of x , y , and z are, in some order, a , b , and c

No information is given about the ordering of a , b , and c . The third conjunct of the goal is presently handled by a kludge: nothing

PUP knows how to do in achieving the rest of the goal changes this condition. Thus the goal PUP gets is actually just

$\text{contents}(x) \leq \text{contents}(y) \wedge \text{contents}(y) \leq \text{contents}(z)$.

The basic method is to use case analysis, which is adequate (although a more clever approach is possible). The AND handler begins by decomposing the main goal into its 2 subgoals. To achieve $\text{contents}(x) \leq \text{contents}(y)$ PUP knows to try 2 things:

(1) Is $\text{contents}(x) \leq \text{contents}(y)$ already true? PUP can prove that it is true if it has been explicitly stated or, since PUP knows that $\leq$ is transitive, if there is a simple transitivity chain such that $\text{contents}(x) = \alpha \leq \beta \leq \ldots \leq \gamma = \text{contents}(y)$. In either case, if $\text{contents}(x) \leq \text{contents}(y)$ is already true, PUP is done.

(2) Is $\text{contents}(y) \leq \text{contents}(x)$? PUP can know this too by having it explicitly stated or from a transitivity chain. PUP also knows that $\neg(\alpha \leq \beta) \supset \beta \leq \alpha$, so that if it knows $\neg(\text{contents}(x) \leq \text{contents}(y))$, then it can deduce $\text{contents}(y) \leq \text{contents}(x)$. In any case, if it decides $\text{contents}(y) \leq \text{contents}(x)$ is true, PUP interchanges x and y . To do this PUP calls itself recursively, giving itself the interchange problem discussed above in Section 4.4.1. (Some future version of PUP should probably save some information about each problem it solves, so that when it is given another similar problem it has an easier time. At present, however, PUP completely redoes the interchange.) After the interchange, PUP interchanges everything it knows about x and y that depends on their contents. That is, every fact that refers to the contents of x is modified to refer to the contents of y and vice versa.

Unfortunately, from the initial state none of the relevant ordering information is known, so the goal of $\text{contents}(x) \leq \text{contents}(y)$ fails to be achieved and the AND handler fails. (A smarter program might have first noticed that no ordering information was given about a , b , and c , and not attempted either of the above steps.)

Failure of the AND handler causes the goal-statement mechanism to try further programs on the apply list. One of these is a case-analysis handler. This program picks one of the subgoals, say contents(x) $\leq$ contents(y) , and constructs a program of the form

$$\underline{\text{if}}\ x \leq y\ \underline{\text{then}}\ \text{subprogram}_1\ \underline{\text{else}}\ \text{subprogram}_2;$$

We note that the implicit assumption here that the $\leq$ predicate is computable should be made explicit. A smarter system might recognize this program as a sort program and go on to produce a nice algorithm.

To find subprogram_1 , contents(x) $\leq$ contents(y) is assumed and the entire goal retried. Again the AND handler fails. (Although the first subgoal succeeds since it is assumed, the second subgoal, contents(y) $\leq$ contents(z) , fails.) Again we enter the case-analysis handler. This time since the first subgoal is true (by assumption), it will not be picked; so the second subgoal is picked. By now, the first part of the program being constructed looks like

```
if x ≤ y then
    begin
    if y ≤ z then
```

The entire goal is again retried. Since both subgoals are assumed, the AND handler succeeds this time, and this case is done.

A point to note is that as each subgoal of the AND goal is achieved, it is added to a list of "protected" facts. After each operation this list is checked to see that none of the facts on it has been altered. If any have, an immediate attempt is made to restore them. This can, of course, lead to infinite loops in which restoring one alters another,

restoring that alters the first, ad infinitum. To prevent this, at some arbitrary level of restoring within restoring, a cutoff is made and failure reported. The importance of the process of restoring protected facts will be shown shortly.

Now we do the else part of the innermost if. To do this the assumption contents(y) $\leq$ contents(z) is removed, and the assumption $\neg$(contents(y) $\leq$ contents(z)) is made. Then the whole goal is retried. The first subgoal, still assumed, succeeds and is added to the protected list. The second subgoal is tried, and since contents(z) $\leq$ contents(y) now holds, y and z are interchanged. A side effect of this interchange is to modify the fact contents(x) $\leq$ contents(y) to be contents(x) $\leq$ contents(z) .

After the interchange the protection list is checked, and because of the interchange PUP no longer has the fact contents(x) $\leq$ contents(y) . So an attempt is made to restore that condition. As before, direct methods fail, and the case-analysis handler is invoked. As before, a conditional statement is added to the program, and the true and false branches are written by assuming the truth and falsehood, respectively, of the condition. The true case results in the null program, and the false case results in an interchange. The attempt to restore contents(x) $\leq$ contents(y) succeeds, so the else part of the innermost if succeeds and thus the whole innermost if does too. The program now looks like this (without comments):

```
if x ≤ y then
     begin
     if y ≤ z then else
          begin
          temp1 ← y;
          y ← z;
          z ← temp1;
          if x ≤ y then else
               begin
               temp2 ← x;
               x ← y;
               y ← temp2
               end
          end
     end
     else
     subprogram2;
```

Finally $subprogram_2$ is written. All assumptions and deductions specific to the process of writing $subprogram_1$ are removed, and $\neg(contents(x) \leq contents(y))$ is assumed. An interchange is needed to establish the first subgoal, but otherwise the process is similar to that of writing $subprogram_1$. The final program is

```
if x ≤ y then
    begin
    if y ≤ z then else
        begin
        temp1 ← y;
        y ← z;
        z ← temp1;
        if x ≤ y then else
            begin
            temp2 ← x;
            x ← y;
            y ← temp2
            end
        end
    end
    else
    begin
    temp3 ← x;
    x ← y;
    y ← temp3;
    if y ≤ z then else
        begin
        temp4 ← y;
        y ← z;
        z ← temp4;
        if x ≤ y then else
            begin
            temp5 ← x;
            x ← y;
            y ← temp5
            end
        end
end;
```

4.4.3 Integer Square Root In this example the desired program should find $\lfloor\sqrt{x}\rfloor$, the floor of the square root of input x. This task was chosen to coincide with Manna's tutorial on automatic programming [21], which compared the abilities of existing systems to synthesize or verify such a program. PUP's performance was gained by

sacrificing formal methods -- and the associated formal guarantees.

PUP has just the right knowledge about numeric functions, number systems, ordering, maxima and minima, searching, and the real square-root function to make the problem interesting yet doable. For example, PUP does not know any program which directly computes the square root of x. However, it does know how to test if an input is equal to the square root of x, by comparing the square of the input to x. And PUP does have a program to compute the square of a number: multiply it by itself.

Let us investigate the dialog now. The user asks for the integer square root of some number, say isqrt(82). Since PUP doesn't recognize the function isqrt, it assumes the user either made a typographical error or wants PUP to write a new function. The user settles that question in favor of the latter alternative, and PUP notices that there is 1 numeric argument. The knowledge of numeric functions is sufficient to realize that the domain and range of the function should be pinpointed if possible. The user indicates that both domain and range are the natural numbers. PUP now picks names for the input and output variables, say x and y, respectively, and asks the user to describe the function in terms of these variables. The user replies with

$$\text{isqrt}(x) \leftarrow \max y \text{ such that } y \leq \text{square\_root}(x);$$

PUP first considers whether or not the condition $y \leq \text{square\_root}(x)$ is directly testable given x and y, i.e., whether PUP already has a program which can do it. Knowledge of the $\leq$ relation says that the test can be done if and only if each side is computable. We trivially have the left side, given x and y. But PUP doesn't have an algorithm to compute square_root(x), so we must look deeper for the right side.

Knowledge of inequalities says to fix this up by finding an inverse function of square_root , say i , and by replacing the old inequality by $i(y) \leq x$. A warning note says that such an inverse must be computable (and in addition both the inverse and the original function must be monotone); otherwise, we're no better off than before. The main fact about square_root is that its inverse is achieved by squaring. Both the square_root and square functions have tags indicating monotonicity. Also, square is known to be computable, so the problem statement is now reformulated as

$$\text{isqrt}(x) \leftarrow \max y \text{ such that } \text{square}(y) \leq x;$$

The second problem is whether an algorithm is already known which computes the maximum element in the range of a given predicate. Knowledge about max includes only 1 algorithm: start by choosing the upper bound of the range and then iterate, decrementing the candidate each time, until the predicate is satisfied. Knowledge of the natural numbers says that an upper bound does not exist, so this straightforward method won't work. Fortunately, max knows a transformation of itself when the predicate is monotone and the range is a segment of the integers: max y such that p(y) becomes min y such that $\neg p(y + 1)$. Both the conditions are verified in our case, so the change is tentatively made, and the problem statement becomes

$$\text{isqrt}(x) \leftarrow \min y \text{ such that } \neg(\text{square}(y + 1) \leq x);$$

(Notice that PUP implicitly assumes that the negation of a computable predicate is computable. This should probably be made explicit.) Knowledge

of negation allows the replacement of $\neg\leq$ by $>$ at this point, and we get

$$\text{isqrt}(x) \leftarrow \min y \text{ such that } \text{square}(y + 1) > x;$$

Now algorithms for computing min are examined. The only one says to start at the lower bound of the range and repeatedly increment until the predicate is satisfied. Knowledge of natural numbers informs us that a lower bound is 0 . PUP converts this to the final code:

$$\text{isqrt}(x) \leftarrow \text{isqrt}_1(0, x);$$
$$\text{isqrt}_1(y, x) \leftarrow \underline{\text{if}}\ \text{square}(y + 1) > x\ \underline{\text{then}}\ y\ \underline{\text{else}}\ \text{isqrt}_1(y + 1, x);$$

PUP enters the program in its records, recalls the original request for isqrt(82) , and runs the new program on it.

Notice the flavor of PUP's operation: locating relevant information, which either provides some of the final code or points to more information which is needed. It is the structuring of this knowledge which beats the combinatorial explosion of searching for relevant facts.

4.5 Examples Program

This program, called EXAMPLE, infers recursive LISP functions from single example input-output pairs. The program was written in INTERLISP by Shaw and later revised by William Swartout. The inductive inference of functions from example I/O pairs has also been explored by J. C. R. Licklider [1] and Hardy [14].

As a typical problem solved by EXAMPLE, given the example I/O pair

input		output
(A B C D)	⟶	(D D C C B B A A)

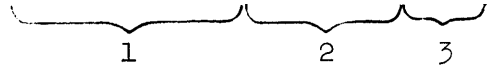
it synthesizes the "reverse and double" function

```
f(x) ← if null(x) then nil else
       append(f(cdr(x)), list(car(x), car(x)));
```

EXAMPLE can infer a class of functions which can be approximately characterized as simple list-to-list transformations. A somewhat more precise characterization of the class is that each function recurs along an input list (or lists) and produces some part of the output (possibly empty) for each step of the recursion. These pieces of the output are assembled into the output list without any reordering (with the possible exception of completely reversing the output). At each step of the recursion, a similar recursive subfunction can be used to produce that step's portion of the output. There can be several input arguments, and the function written can be recursive in any number of arguments.

As an example, consider the I/O pair

input		output
(A B C D)	⟶	((A B)(A C)(A D)(B C)(B D)(C D))



The output is produced in 3 steps as indicated. A recursive subfunction produces the sublists (1, 2, and 3 shown above) in successive steps, and

the main function appends them together. EXAMPLE can synthesize this function and variations, such as having the output reversed or the same output but with each sublist reversed.

The program works as follows. Consider the synthesis of the function discussed above. Call it f . First EXAMPLE decides how much of the output is produced in the first step of the recursion (referred to as the recursive head). Thus, in the example above, it decides that the first sublist (A B)(A C)(A D) is produced in the first step and is the recursive head. (The heuristic by which it decides this is interesting and is discussed later.) Next it sets up the subproblem of synthesizing the code that produces the head. This can be thought of as specifying a subfunction, although in-line code may be used if no recursion is necessary. In our example a recursive subfunction, call it f_1 , is required. First the arguments of f_1 are selected. In this case EXAMPLE chooses 2 arguments for f_1 , car of the input, A , and cdr of the input, (B C D) . Obviously f_1 just lists car of the input with each of the elements of the cdr . After the inputs are set up, the subfunction is written in the same manner as the main function, by a recursive call to EXAMPLE. Returning to the synthesis of the main function, there are 3 remaining steps: (1) the terminating conditions are selected; (2) the results from each recursive step are joined properly, using either cons or append ; and (3) the recursive call of the main function is formed. The recursive call can be on the cdr , cddr , cdddr , etc. For example, in (A B C D E F) $\longrightarrow$ (A C E) the recursive call is on the cddr of the input.

The program written for (A B C D) ⟶ ((A B)(A C)(A D)(B C)(B D)(C D)) is

```
f(x) ← if null(x) then nil else
       if null(cdr(x)) then nil else
       append(f1(car(x), cdr(x)), f(cdr(x)));

f1(y, z) ← if null(z) then nil else
           cons(list(y, car(z)), f1(y, cdr(z)));
```

EXAMPLE is fairly complex, but we will describe one interesting part, namely the heuristic that decides where to break the output list into the recursive head and the rest. The output list is scanned left to right (and possibly right to left if necessary), looking for a simple progression. When a large change is encountered, this point is proposed as the break. In our example, (A B C D) ⟶ ((A B)(A C)(A D)(B C)(B D)(C D)) , the pattern (A next_input) , where next_input signifies the successive elements in the input past A (i.e., B , C , and D), is discovered to match the first 3 elements of the output but not (B C) , so the break occurs before (B C) . This heuristic, along with many others, such as determining when to write a subfunction and the number of arguments for a subfunction, works fairly well.

The following examples are ones for which a reasonable program was automatically generated. Some 1-input examples are

input		output
(A B C D)	⟶	(D C B A)
(A B C)	⟶	(A A B B C C)
(A B C D)	⟶	(D D C C B B A A)
(A B C D E F)	⟶	(A C E)
(A B C D E F)	⟶	(E C A)
(A B C D E F)	⟶	(B D F)
(A B C D)	⟶	((A)(B)(C)(D))
(A B C D)	⟶	((A B)(A C)(A D)(B C)(B D)(C D))
(A B C D)	⟶	(A B C D B C D C D D)
(A B C D)	⟶	(D C B A D C B D C D)
(A B C D E F)	⟶	(B A D C F E)

Some 2-input examples are

input 1	input 2		output
FN	(A B C D)	⟶	((FN A)(FN B)(FN C)(FN D))
(A B C)	(D E F)	⟶	(A D B E C F)
(A B C)	(D E)	⟶	(A D B D C D A E B E C E)
(A B C)	(D E)	⟶	((A D)(A E)(B D)(B E)(C D)(C E))
(A B C)	(D E F)	⟶	((A D)(B E)(C F))

The limitations of the system are

(1) Only the position of an element, and not its identity, is considered in deciding what to do with it. Thus a reverse program can be written, but a sort cannot.

(2) On the input, only top-level list recursions, as opposed to tree recursions, are attempted. Thus the flatten function [e.g., (A B (C (D E) F) G) ⟶ (A B C D E F G)] is not possible.

(3) The organization of the program makes extension into new areas reasonably difficult. We plan to reorganize the program and to add cleverer, domain-specific facts to increase its power.

4.6 Synthesis of Large Inductive-inference Programs

Our next system, PUP5 by Lenat, represents an attempt at the synthesis of larger, more domain-specific programs. The system was designed to write concept-formation programs, a class of programs which inductively infer the definition of a concept from a number of instances of that concept [18]. The original target program to be synthesized semi-automatically was SPOT, a small version of Winston's concept-formation program [34] without its fancy graph-matching algorithm, written by Peter Gadwa at Stanford University. SPOT was specifically designed to be a simple (5-page), yet still interesting program. During the course

of the design of PUP5, the target program evolved into a somewhat different program.

PUP5 is still only an experimental vehicle, but it has proved moderately successful. It has indeed written a concept-formation program similar to the intended one, although augmented by self-documentation. PUP5 is being revised to write a wider class of inductive-inference programs. The next target program is a simple grammatical-inference program, upon which work should be completed shortly.

Although the system is written entirely in INTERLISP, many popular AI-language features [5] (e.g., pattern matching, assertions, goal direction, apply teams, backtracking, special data types, demons, etc.) were hand coded expressly for this system. The entire 100 pages of code is organized as an interacting community of small units, called beings. Although complex, the structure of each being is the same: a set of answers to about 30 fixed questions. These questions, called the being parts, represent "everything you always wanted to know about a small program". Neither the exact set chosen nor the number 30 is very important; the approximate size of the set is relevant to automatic programming, however. Each being part is itself a little program which knows what the 30 questions are and which may ask any being any question it wants to. Since some beings must write target code, we choose to have each being x write all code similar to x . For example, the sort being contains a costly "big switch" hooked to various sorting algorithms, but the code it writes in any specific instance will be a tailor-written implementation of a particular sort algorithm.

Although PUP5 insists on doing structured programming (hence uses something like macro expansion), its control structure employs feed forward,

feedback, backtracking, and a contextual assertion base. One bit of inherent philosophy is that the system should defer making all decisions as long as possible. We hope that by this deferral, along with careful record keeping, we can eliminate most of the carelessness "bugs" that typically arise in humans as a result of brain-hardware limitations. This is in contrast to earlier versions of PUP [see Section 4.4], which viewed debugging as the predominant part of programming. Thus, PUP5 rarely believes it is finished if in fact it has overlooked some details.

We now present (most of) the current parts of a being:

name	description
identity	how the being is referenced in English sentences
arguments	which arguments are required and which are optional
argument_check	predicate which examines each argument for suitability
evaluate_arguments	which arguments of the being and in the code generated by the being should be evaluated
what	brief summary of what the being does
why	justification for the being's existence: why it is called
how	summary of the method(s) used by the being to do its thing
effects	postconditions which will be true after calling the being
when	factors and weights telling how apropos the being is right now
meta_code	body of the code, but with uninstantiated subparts
comments	aid to filling in the meta_code
requisites	what must be actively satisfied just before (prerequisites), during (corequisites), and just after (postrequisites) the being is executed
demons	which demons should be enabled during the being's execution
affects	which other beings might be called by this being

name	description
complexity	vector describing such features as recursiveness, overall cost, chance of failing, transparency to user, etc.
specializations	what must be known to write a streamlined version of this being
alternatives	equivalent beings in case this one doesn't work
generalizations	more general beings in case none of the alternative beings works
predicate	what type of values the being returns
data_structure	if being is a data structure, how it is initialized and accessed, how elements are inserted and deleted
encodable	description of the flow of control in writing a specialized new being
inhibit_current_demons	enable/inhibit mechanism for demons
form_changing	where in the being tree this being can directly return to

Although each being has about 30 answers, each of which might contain several facts, only about 10 facts from any given being are actually employed during the course of the program-writing dialog. A typical programming being is obtain_usable_information . Its when being part says that calling this being is generally undesirable, but may be the only reasonable course to follow if there exists new information which is not directly usable. Its how being part says to choose (creating a non-deterministic backtrack point) from among these: translate, get totally new raw information, extract a small subset of existing raw information to concentrate upon, or analyze the implications of a small set of existing raw information. A typical domain-specific being is partition_a_domain . It specializations being part says to find out whether the partition is partial or total, whether it is weak or strong, and whether it is built by repeatedly accepting ⟨element, class name⟩

pairs and/or accepting an element (then guessing and verifying its class name) and/or accepting a class name (then guessing and verifying its element(s)).

The dialog involved in a PUP5 run is carried on in a miniscule subset of English. Since it encompasses precisely the sentences which the user wants to say, the dialog gives the illusion of being unconstrained. However, the term "the user" is not generic as there has only been 1 user so far. The interaction system works by each being recognizing and processing phrases referring to it. The dialog for synthesizing the concept-formation program takes several hours of console time. Much of the interaction is unnecessary: PUP5 asks the user to name things which are never referenced again. This annoyance is being worked on.

A promising sign of programming-knowledge convergence is that out of 67 programming beings 50 are used by PUP5 during the course of writing both of the target programs (concept formation and grammatical inference). Future plans for PUP5 work include studying the various types of knowledge needed for programming, inductive inference, and specific target programs. This will (hopefully) be done by extending PUP5 to handle more and bigger tasks.

4.7 Sorting

During the past year, Green and Barstow have attempted to isolate and codify those "facts" of programming knowledge which are necessary for a system which can understand and write simple iterative sorting programs. To keep the working domain small, such techniques as recursion and exchange

sorting (e.g., bubble sort) and such fast algorithms as quicksort [16] and heapsort [8, 33] were explicitly excluded from consideration. In the course of this attempt, it became apparent that many concepts were involved and needed to be analyzed. The present set of facts is a list of 100 rules which deal with sorting and permutations, generators for explicitly given sets, set constructors, and several types of generate-and-test methods. The rules allow for either array or list representations of sets. There are at present no rules regarding efficiency considerations or formal verification of correctness. This we consider a shortcoming, and Elaine Kant has recently begun studying the addition of rules for optimization.

One interesting aspect of our list of rules is that it covers a wide range of levels. As an example of the range covered, there are rules dealing with the choice between selection and insertion sorts, with state-saving schemata for generators, with the choice of variable names, and with the addition of elements to the front of a list. One initial goal of our work was to have each rule be relatively simple and explicit; we feel that we have been moderately successful in this regard. Thus, these rules provide a knowledge base for a program-writing system, and it is the interaction of these rules which provides the foundation for the system's "understanding" of sort programs.

The rules have been organized in a goal/subgoal fashion, with the capabilities of disjunctive and sequential subgoals and subgoaling by cases. A preliminary implementation of a system based upon these rules has been completed. Each rule has been written as an INTERLISP function. The control system consists of several other functions which describe the efforts of the system as it writes a program, ask for choices at

OR-rule junctures, and provide limited additional explanatory information on request (e.g., a why function to explain the purpose of a section of the final program). The traces tend to be overly verbose, but confirm our belief that the rules can form the basis of an understanding system.

It should be emphasized that this system was primarily a "quick and dirty" effort, intended as a device for testing and refining rules, rather than as a program-writing system. One test of the rules is, of course, adequacy, and the system has successfully written 3 substantially different programs: a reverse program, a selection sort, and an insertion sort. Although not all of the variations have been completed to date, we expect that with perhaps 20 additional rules our system should be capable of generating a few dozen distinct (although in many cases similar) programs. The programs produced are generally about 1 page in length (using the INTERLISP prettyprint function as a standard of measurement).

We feel that this line of research has been fruitful and plan to continue it in the future. It is our expectation that such a structuring of knowledge will make possible the incremental addition of rules for other aspects of low-level programs and that any additional rules will use many of the present rules as subgoals.

BIBLIOGRAPHY

[1] "Automatic Composition of Functions from Modules", Project MAC Progress Report X: July 1972 - July 1973, Section III.E.1, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts, pages 151-156.

[2] Biermann, A. W., Baum, R., Krishnaswamy, R., and Petry, F. E., Automatic Program Synthesis Reports, OSU-CISRC-TR-73-6, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, October 1973.

[3] Biermann, A. W., and Feldman, J. A., "On the Synthesis of Finite-state Machines from Samples of Their Behavior", IEEE Transactions on Computers, Volume C-21, Number 6, June 1972, pages 592-597 (also On the Synthesis of Finite-state Acceptors, Memo AIM-114, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, April 1970).

[4] Blum, L., and Blum, M., "Inductive Inference: A Recursion Theoretic Approach", Information and Control, to appear (also Memorandum ERL-M386, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, California, 13 March 1973).

[5] Bobrow, Daniel G., and Raphael, Bertram, "New Programming Languages for AI Research", Computing Surveys, Volume 6, Number 3, September 1974 (invited tutorial lecture, Third International Joint Conference on Artificial Intelligence, Stanford University, Stanford, California, 20-23 August 1973; also Report CSL-73-2, Xerox Palo Alto Research Center, Palo Alto, California, 20 August 1973; also Technical Note 82, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California, August 1973).

[6] Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press, Inc., New York, New York, 1972.

[7] Feldman, J. A., and Shields, P. C., Total Complexity and the Inference of Best Programs, Memo AIM-159, Report STAN-CS-72-253, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, April 1972.

[8] Floyd, Robert W., "Algorithm 245: TREESORT3", Communications of the ACM, Volume 7, Number 12, December 1964, page 701.

[9] Floyd, Robert W., "Assigning Meanings to Programs", in Schwartz, J. T., editor, Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics, Volume 19, American Mathematical Society, Providence, Rhode Island, 1967, pages 19-32.

[10] Floyd, Robert W., "Toward Interactive Design of Correct Programs", in Freiman, C. V., editor, Foundations and Systems, Information Processing 71: Proceedings of IFIP Congress 71, Volume 1, North-Holland Publishing Company, Amsterdam, The Netherlands, 1972, pages 7-10 (also Memo AIM-150, Report STAN-CS-71-235, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, September 1971).

[11] Fusaoka, Akira, and Waldinger, Richard, "Program Writing using Sequences", Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California, January 1974.

[12] Gold, E. Mark, "Language Identification in the Limit", Information and Control, Volume 10, Number 5, May 1967, pages 447-474.

[13] Green, Claude Cordell, The Application of Theorem Proving to Question-answering Systems, Ph.D. thesis, Electrical Engineering Department, Memo AIM-96, Report STAN-CS-69-138, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, June 1969.

[14] Hardy, Steven, "Automatic Induction of LISP Functions", AISB Summer Conference, University of Sussex, Brighton, England, July 1974, pages 50-62

[15] Hoare, C. A. R., "An Axiomatic Basis for Computer Programming", Communications of the ACM, Volume 12, Number 10, October 1969, pages 576-580, 583.

[16] Hoare, C. A. R., "Quicksort", The Computer Journal, Volume 5, 1962, pages 10-15.

[17] Horning, James Jay, A Study of Grammatical Inference, Ph.D. thesis, Memo AIM-98, Report STAN-CS-69-139, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, August 1969.

[18] Hunt, Earl B., Concept Learning: An Information Processing Problem, John Wiley and Sons, Inc., New York, New York, 1962.

[19] Knuth, Donald E., The Art of Computer Programming, Volumes 1-3, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1973, 1969, 1973.

[20] Knuth, Donald E., Sorting and Searching, The Art of Computer Programming, Volume 3, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1973, page 73.

[21] Manna, Z., "Automatic Programming", invited tutorial lecture, Third International Joint Conference on Artificial Intelligence, Stanford University, Stanford, California, 20-23 August 1973.

[22] Manna, Zohar, and Waldinger, Richard, "Knowledge and Reasoning in Program Synthesis", in preparation.

[23] McCarthy, J., "Towards a Mathematical Science of Computation", in Popplewell, Cicely M., editor, Information Processing 1962: Proceedings of IFIP Congress 62, North-Holland Publishing Company, Amsterdam, The Netherlands, 1963, pages 21-28.

[24] Newell, Allen, "Heuristic Programming: Ill-structured Problems", in Aronofsky, Julius S., editor, Relationship between Operations Research and the Computer, Progress in Operations Research, Volume 3, John Wiley and Sons, Inc., New York, New York, 1969, pages 361-414.

[25] Persson, Staffan, Some Sequence Extrapolating Programs: A Study of Representation and Modeling in Inquiring Systems, Ph.D. thesis, School of Business Administration, University of California, Berkeley, California, Memo AIM-46, Report STAN-CS-66-50, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, 26 September 1966.

[26] Reboh, Rene, and Sacerdoti, Earl, A Preliminary QLISP Manual, Technical Note 81, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California, August 1973.

[27] Rulifson, Johns F., Derksen, Jan A., and Waldinger, Richard J., QA4: A Procedural Calculus for Intuitive Reasoning, Technical Note 73, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California, November 1972.

[28] Simon, Herbert A., "Experiments with a Heuristic Compiler", Journal of the Association for Computing Machinery, Volume 10, Number 4, October 1963, pages 493-506.

[29] Simon, Herbert A., and Kotovsky, Kenneth, "Human Acquisition of Concepts for Sequential Patterns", Psychological Review, Volume 70, Number 6, November 1963, pages 534-546.

[30] Smith, David Canfield, MLISP, Memo AIM-135, Report STAN-CS-70-179, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, October 1970.

[31] Teitelman, Warren, INTERLISP Reference Manual, Xerox Palo Alto Research Center, Palo Alto, California, 1974.

[32] Waldinger, Richard J., Constructing Programs Automatically using Theorem Proving, Ph.D. thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1969.

[33] Williams, J. W. J., "Algorithm 232: HEAPSORT", Communications of the ACM, Volume 7, Number 6, June 1964, pages 347-348.

[34] Winston, Patrick H., Learning Structural Descriptions from Examples, Ph.D. thesis, Department of Electrical Engineering, TR-76, Project MAC, TR-231, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1970.