Stanford Artificial Intelligence Laboratory
Memo AIM-258

JANUARY 1975

Computer Science Department
Report No. STAN-CS-75-476

A Hypothetical Dialogue Exhibiting a Knowledge Base For a Program-Understanding System

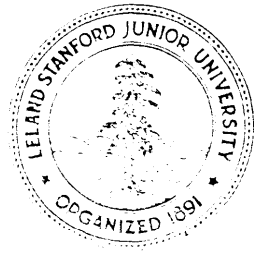
by

Cordell Green
David Barstow

Research sponsored by

Advanced Research Projects Agency
ARPA Order No. 2494

COMPUTER SCIENCE DEPARTMENT
Stanford University



Stanford Artificial Intelligence Laboratory
Memo AIM-258

Computer Science Department
Report No. STAN-CS-75-476

JANUARY 1975

A Hypothetical Dialogue Exhibiting a Knowledge Base For a Program-Understanding System

by

Cordell Green
David Barstow

ABSTRACT

A hypothetical dialogue with a fictitious program-understanding system is presented. In the interactive dialogue the computer carries out a detailed synthesis of a simple insertion sort program for linked lists. The content, length and complexity of the dialogue reflect the underlying programming knowledge which would be required for a system to accomplish this task, The nature of the knowledge is discussed and the codification of such programming knowledge is suggested as a major research area in the development of program-understanding systems.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC 15-73-C-0435 . The views and conchsions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, ARPA, or the U.S. Government.

Reproduced in the U.S.A. Available from the National Technical Information Service, Springfield, Virginia 2215 I.

A HYPOTHETICAL DIALOGUE
EXHIBITING A KNOWLEDGE BASE FOR A PROGRAM-UNDERSTANDING SYSTEM

Green and Barstow

January, 1975

TABLE OF CONTENTS

I. INTRODUCTION

(a) SUMMARY

The overall objective of our research is to gain more insight into the programming process as a necessary step toward building *program-understanding systems.* Our approach has been to examine the process of synthesizing very simple programs in the domain of sorting. We hope that by beginning with this simple domain and developing and implementing a reasonably comprehensive theory, we can then gauge what is required to create more powerful and general program-understanding systems.

Toward this end, we are working on first isolating and codifying the knowledge appropriate for the synthesis and understanding of programs in this class and then embedding this knowledge as a set of rules in a computer program. Along the way, we have developed some preliminary views about what a program-understanding system should know.

Our goal in this particular paper is to present a *dialogue* with a *hypothetical* program-understanding system. A dialogue was chosen as a method of presentation that would exemplify, in an easily understood fashion, what such a system should know. The subject of the dialogue is the synthesis of a simple insertion sort program. Each step in the dialogue corresponds to the utilization of one or more pieces of suggested programming knowledge. Most of this knowledge is stated explicitly in each step, The dialogue presented here is a highly fictional one, although some port ions of the reasoning shown in the dialogue have been tested in an experimental system.

We are now in the process of formulating the necessary programming knowledge as a set of synthesis rules. However, the scope of this paper does not include the presentation of the current state of our rules. So far some 110 rules have been developed and are being refined in a rule-testing system. The synthesis tasks on which these rules are being debugged include two insertion sorts, one selection sort, and a list reversal. We hope to present in a later paper a description of the set of rules.

As will become apparent in the dialogue, one of our conjectures is that a program-understanding system will need very large amounts of many different kinds of knowledge. This seems to be the key to the flexibility necessary to synthesize, analyze, modify, and debug a large class of programs. In addition to the usual types of programming knowledge, such as the semantics of programming languages or techniques of local optimization, many other types are needed. These include, at least, high-level programming constructs, strategy or planning information, domain-specific and general programming knowledge, and global optimization techniques. In Section III we discuss this further and show where these kinds of knowledge occur in the dialogue.

(b) DOMAIN OF DISCOURSE

Topics mentioned in the dialogue include data structures, low-level operations, and -high-level programming constructs. The main data structures mentioned in our dialogue are ordered sets represented by lists. The-low level operations mentioned include assignment, pointer manipulation, list insertion, etc. Some of the higher-level (in some sense) notions or constructs we consider are permutation, ordering (by various

criteria), set enumeration, generate and test, generate and process, proof by induction, conservation of elements during a transfer, and methods of temporary marking (or place-saving) of positions and elements. Time and space requirements for various methods are not discussed.

The target language is LISP, in particular the INTERLISP language [10]. However, in the dialogue we represent the programs in a fictitious meta-LISP.

IL. A DIALOGUE

(a) INTRODUCTION

In this section we wish to exhibit what we consider to be a reasonable level of understanding on the part of a program-understanding system. It is not obvious how best to present this in a way that is easy for the reader to follow, since the synthesis process is rather complex. Wc hope that an English language dialogue is adequate. We have added to the English several "snapshots" of the developing program that help to indicate where the system is in the programming process. These diagrams are similar to the stepwise refinements used in structured programming [1]. Our dialogue may be considered as a continuation of the technique of presentation used by Floyd for a program verifier-synthesizer [2], although our more hypothetical system has been -allowed to know more about program synthesis for its domain of discourse.

In certain ways we feel that the dialogue is *not* representative of how a program-understanding -system would appear to the user during the synthesis process

(although such a low-level dialogue would at times have its place). For expository purposes the dialogue has certain choices and inferences made by the machine and others made by the user. Depending upon the application, these decisions and inferences could reasonably be re-distributed between user and machine, with some made by other automated "experts", such as an efficiency expert, and other decisions forced by the cant ext in which the program is written. For example, the data structures for this insertion sort might be determined in advance if the sort routine were part of a larger system, and all choices made- by the user could be made instead by rules calculating the efficiency of each branch. A more typical user-machine dialogue would usually be conducted at a much higher level and in the words used to talk about the application area for the program. But to present a hypothetical dialogue in which the program merely said, "OK, I'm all finished," or, "I wrote a sort program as part of your operating system," wouldn't explicate very much programming knowledge.

The style of the dialogue is not intended to be a strong suggestion about what the process of *specifying* a program should be. In fact, we feel that program specification is a difficult problem in automatic programming research [4], and we touch on only a few aspects of it here. However, some of the techniques used in the dialogue will probably find some, as yet undetermined, place in program-understanding systems. The obvious techniques include interactive specification, natural language communication, and the periodic presentation of summaries of the structure and content of the evolving program.

We note that we have endowed our hypothetical system with various abilities,

including an inference capability, a program simplification capability, and the ability to illustrate and summarize where it is in the programming process. Further, the dialogue sometimes takes on the character of a decision-tree, sometimes of goal-oriented problem-solving, and sometimes of step-wise refinement, These all seem appropriate to the task.

Various decisions, such as how large to make a step, when to suppress or explicate an inference, and when to present a diagram, are somewhat arbitrary, but were made primarily on the basis of what seemed to be the clearest exposition.

Before the start of the dialogue, we wish to remind the user that the focus is not on the particular mechanisms which might be employed to produce the hypothesized behavior, but rather on the reasoning itself.

(b) PART 1: Setting Up the Main Tasks

1. User: Please write me a *sort* program. Use *linked lists* to represent the input and output, which are ordered sots. The output set should be ordered so that each element is *numerically greater than* the preceding element. The input set will be called INPUT and the output set will be called OUTPUT.

2. Machine: As I understand it, a sort program produces as output a permutation (of the input) that satisfies a test. In this case, the test is that the elements be ordered on the basis of the above-mentioned ordering relation, "numerically greater than". Please choose one of the two known methods† for finding a

† The limitation to only two "known" choices is arbitrary, but it is reasonable to expect a system to have a limited number of choices.

permutation that satisfies a test:

(a) Generate and test permutations until the desired one is found.

(b) Carefully construct the desired pcrmut at ion.

3. User: Carefully construct the desired permutation.

4. Machine: Should the output set be constructed in the same list cells as the input set, or should a new list be created?

5. User: A new list should be created.1

6. Machine: Do you want an iterative or recursive program?

7. User: Iterative. [We wish to remind the reader that the distinctions between "user" and "machine" are somewhat arbitrary. Yet a decision must be made, so we have chosen to let our hypothetical "user" make it. Furthermore there is no reason to expect the user to have a preference for an iterative or recursive program. We have our user choose an iterative structure over what might be a more natural recursive structure, because there are fundamental aspects of the algorithm which are more explicit in its iterative form.]

8. Machine: Then the program will be in the form of a *transfer program* which must be *total*ˆ. The program will transfer all elements from the input to positions in the output, such that the final output is a permutation (of the input) which satisfies the above ordering relation, namely increasing by numerical value.

---

† This does not preclude overwriting the input set during the sort. Overwriting might be done in order to indicate which elements have been selected from the input.

ˆ By "tot al", we mean that the transfer loop must be repeated until all elements have been transferred.

The transfer paradigm††

```
transfer program:
   LOOP UNTIL all elements of input have been processed:
       selector:
              select the next element from input;
       constructor:
              add this element to output:
   REPEAT;
```

9. Machine: In order to write such a transfer program, I must achieve several goals:

(1) Write a *selector.* On each **call, the selector will find that element of the input which comes next in the chosen transfer order. The selector consists of two parts:**

(a) selector initializer

(b) selector **body**

(2) Write a *constructor.* **On each call, the constructor will produce a new output (or modify the old output); so that the new element is in the appropriate position but that otherwise the order of elements in the output is unchanged. The construct or consists of two parts:**

(a) constructor initializer

---

† **As illustrated, the transfer** program has a **"selector" part and a "constructor" part, The selector** produces the elements one at a time, and **the constructor** builds the new **set. Either the selector or the constructor may re-order the set, i.e., the selector can produce the element** s in an order different from the stored order, or the construct or **can build a set with an ordering different from the order of receipt of the elements,**

†† **We will generally present in our diagrams only those parts which seem relevant to the current cont ext.**

(b) constructor body

Before I can write these parts, I must:

(3) Select a *transfer order*--the order in which the variables will be transferred from the input to the output.

(4) Select a *transfer termination method*--a test that ail elements have been transfered, and a position in the program appropriate to the type of test selected.

(5) Select a name for the *transfer variable*--a variable which holds as its value the element being transferred.

After writing the parts, I must:

(6) Combine the selector, constructor and transfer termination test into a *generate and process* loop. This completes the final program.

The basic structure of the program (including test position)†

```
transfer initializer:
   selector initializer:                                    (1) (a)
   constructor initialize'r:                                (2) (a)
transfer body:
   LOOP UNTIL DONE:
        transfer termination test:
            IF all elements have been transferred           (4)
                THEN DONE:
        selector body:
            select the next element from input;             (1) (b)
        constructor body:
            add the element to output;                      (2) (b)
   REPEAT;
   DONE: HALT.
```

10. Machine: Should the transfer order be *linear* with respect to the initial order or the final order?‡

11. User: Linear with respect to the initial order.

12. Machine: Should the transfer order be identical to the initial order or the converse of the initial order?

13. User: Identical.

---

† Our syntax for iterative statements and exits is that proposed by Zahn [11] and supported by Knuth [6]. Loops are exited upon encountering an *event*, such as DONE in the above example. The program exits to the occurrence of the event label after the body of the loop. Events are declared after LOOP UNTIL. REPEAT denotes the end of the loop.

‡ We define the "initial" and "final" orders to be the conventional list-representation ordering of the input set and of the desired output set, respectively. By "linear with respect to an order" we mean either identical to that order or the converse of that order; The user's choice of "linear with respect to the initial order" amounts to choosing an insertion -sort algorithm, rather than a selection sort algorithm. The full effect of this choice will be seen when choosing the "construction re-ordering" during the synthesis of the constructor.

14. Machine: Now let's select a transfer termination method. This will consist of a test and a position for the test. Since the transfer will be tot al, the transfer termination test must test whether ail elements have been transferred. This is equivalent to testing whether no elements remain to be selected. This is the same test which is used by the select or to stop its generation of elements. I will use this test as the transfer termination test. While writing the selector I will thus have to write a third part: the selector termination test. Since this test must precede a call to the selector body, I will put the test at the beginning of the loop.1

15. Machine: I will use ELEMENT as the transfer variable.†

---

t For ease in presentation the preceding diagram showed the result of this choice (i.e., the test appeared in the proper location), although the choice had not yet been made. Considering ail of the tests which might have been used and ail of the places where such a test might have been put, one could imagine a great deal of planning going into making this-rather difficult decision.

† To choose meaningful variable names requires knowledge about communication with the user and is a difficult problem.

(c) PART 2: Synthesizing the Selector

We are about to work on the indicated? parts of the selector

```
transfer initializer:
    selector initializer:                                    ☜
    constructor initial izer:
transfer body:
    LOOP UNTIL DONE:
        transfer termination test:
            IF all elements havr been transferred+
                all elements have been selected              ☜
                THEN DONE:
        selector body:
            select the next element from input;              ☜
        constructor body:
            add the element to output:
    REPEAT;
    DONE: HALT,
```

16. **Machine:** *I* **will now choose a** *selection re-ordering. I* **will use the term "selection** re-ordering" to refer **to the relation between** (a) **the order** of the elements in the input (positional order) and (b) **the order in which** the elements are selected by the selector (temporal **order). Since the transfer order is** *linear* and *forward*

---

t Throughout **this dialogue, we will use special characters and type fonts to** indicate different **sect ions of** these **diagrams. We present the meaning of the fonts here in case** confusion arises, **but the meaning should be clear from the context,**

areas not under cons ideration
[wi th . . . indicating areas omitted]
areas currently under consideration
PROGRAM PARTS JUST COMPLETED
RESERVED WORDS
-BASIC LISP FUNCTIONS
USER- OR MACHINE-DEFINED LABELS
➔ the right arrow indicates a step in the refinement process

(that is, identical to the input order), the selection re-ordering must also be linear and forward.

17. **Machine:** A selector is a type of *generator*, a "total" generator that produces each element of the input once and only once. Corresponding to the three parts of a selector, this type of generator also has three parts:†

(1) a body which actually produces the elements

(2) an initializer which can set up the generator before it is called the first time

(3) a termination test (to be used as the transfer termination test) which tests whether all elements have been generated

18. **Machine:** I must now select the *state-saving scheme* by which the generator will save its state after each call. In order to save its state, the generator must remember how the input set is divided into two sets;

(1) those elements which have already been generated

(2) those elements which are yet to be generated

Recall that the generator is moving linearly through the input list from first to last to find each new element. Therefore all elements before its current position in the list have already been generated, and all elements after its current position are yet to be generated. To save the state of the generator, I need

- -

† Technically, before these parts can be written, the production re-ordering for the generator must be chosen. In this case, it is obviously the same as the selection re-ordering, so we have omitted it from the dialogue in the interests of clarity. The initializer is written after the producer, since in general the form of an initializer is dependent on the body of the loop to be initialized.

only save a marker for the generator's current position in the input list. One pointer will be sdequat e for this purpose. Let us call this pointer REST.

We are about to work on the indicated parts of the selector body

```
transfer initializer:
   . . .
transfer body:
   LOOP UNTIL DONE:
   . . .
        selector body:
            select the next element from input*
                generate the next element from input⇒
                    produce element;                    (2)
                    increment state of generator;       (1)
   . . .
   REPEAT;
   DONE: HALT.
```

19. **Machine:** Now I will synthesize the body of the generator. I need code for two pieces:

(1) increment the state of the generator

(2) produce the next element

20. Machine: To increment the state of the generator, I will advance the position pointer by one. Since the representation is a linked list, the code for this is merely:

```
REST ← CDR(REST)†
```

21. Machine: Now I will write the producer. Since the state-saving scheme uses a pointer, REST, to separate those elements generated from those not yet generated, and since the next element to be generated is the first element of the non-generated set, then the process of saving the state will also effectively mark the next element to be generated.1 To produce the next element, then, I need merely take the first element of the non-generated set, which is pointed to by REST.‡ Since the represent at ion is linked-lists, the subprogram for this is:

```
ELEMENT ← CAR(REST)
```

22. Machine: t now put the pieces of the generator body together into a block:

```
ELEMENT ← CAR(REST);
REST ← CDR(REST)
```

- -

t Throughout, we will use the standard LISP terminology:
CAR(L) gives the first element of L
CDR(L) gives the remaining elements of L
CONS(X L) creates a new list with first element X and with the rest of the elements being those of the list L
NIL is the empty list (and list terminator)

† In a selection sort, the code to find the next element to be generated would be considerably more complex since it would need to find the largest of the remaining elements.

‡ There are situations in which the value of the element or a pointer into the list might be returned, rather than the element itself.

We have just finished the indicated parts

```
transfer initializer:
   • • •
transfer body:
   LOOP UNTIL DONE:
   • • •
        selector body:
            select the next element from input*
                generate the next element from input⇒
                    produce element⇒
                        ELEMENT ← CAR(REST);
                    increment state of generator⇒
                        REST ← CDR(REST);
   • • •
   REPEAT;
   DONE: HALT.
```

23. Machine: I will now write the generator initializer and the generator termination test. Since I am using the pointer **REST** to mark the elements not generated, and since initially no elements have been generated, I must initialize it to the entire input set:

REST ← INPUT

24. **Machine:** Now let's write the generator termination test {which will be used as the transfer termination test). To test whether all elements have been generated, I will test whether there are no elements remaining. Since the elements are being generated in a forward linear order, I simply test for being at the end of the input list. Since the representation is a linked list, this is simply a test for emptiness:

REST = NIL?

We have just finished the indicated parts

```
transfer initial izer:
   selector initializer:*
       generator initializer:*
           REST ← INPUT;                                    ☜
   . • .
transfer body:
   LOOP UNTIL DONE:
       transfer tcrminat ion test:
           IF all elements have been transferred*
               all elements have been selected⇒
                   all elements have been generated*
                       no element 9 remain⇒
                           REST = NIL                       ☜
           THEN DONE:
       se l ec t or body:
           select the next element from input=+
               generate the next element from input*
                   produce element⇒
                       ELEMENT ← CAR(REST);
                   increment state of generator-
                       REST ← CDR(REST);
   . . •
   REPEAT;
   DONE: HALT,
```

25. Machine: I have finally finished the generator which the selector needed. In fact, that is all there is to the entire selector, so the selector is finished.

(d) PART 3: Synthesizing the Constructor

We are now working on the indicated parts of the constructor

```
transfer initializer:
   selector initializer:
   constructor initializer:                                    ☜1
transfer body:
   LOOP UNTIL DONE:
      transfer termination test:
         IF all elements have been transferred=>
               all elements have been selected
               THEN DONE;
      selector body:
         select the next element from input;
      constructor body:
         add the element to output;                            ☜1
   REPEAT;
   DONE: HALT,
```

26. Machine: One way to create an output set that is a permutation of the elements received by the constructor is *to* write an *inductive* constructor that satisfies the following constraints:

(1) The initial output set must be empty.

(2) Each construction operation must be such that the new output set consists of the old set plus one new element.1

(3) After each construction operation, the new set must satisfy our ordering relation.

27. Machine: To write such a constructor, two decisions must first be made:

(1) Choose a construction re-ordering.

---

1 It is sometimes convenient to think of the constructor as producing a sequence of output sets, and sometimes as successively modifying one output set.

(2) Choose a construction met hod for building a new set, given the old set, the new element, and the position where the new element should occur in the set.

After these choices are made, we will produce code for the constructor body and then the constructor initializer.1

28. Machine: I will now choose a *construction re-ordering*. I will use the term "construction re-ordering" to refer to the relation between (a) the order in which the elements are received by the constructor (temporal order) and (b) the order of the elements in the output (positional order). The re-orderings may be linear or complex, where linear means same as or converse of. Since the order in which the elements are received (the transfer order) is the same as the input order, and the desired output order is not necessarily the same as (nor the converse of) the input order, the construction re-ordering must be complex (non-linear).† This re-ordering will be based on the "numerically greater than" ordering relation. This means that the correct position of each element in the output list will 'be such that all preceding elements are less than or equal to the new element and all following elements are greater than the new element.8

---

† Note that we have implicitly decided that the construction consists of two steps: (1) finding the position; (2) adding the element to the set. One might imagine schemes in which finding a posit ion and modifying the set are interleaved operations,

† This choice of a complex construction re-ordering, along with the choice of a linear selection re-ordering, completes the specification of an insertion sort, all of which follow from the user's choice of a transfer order "linear with respect to the initial order".

‡ For clarity, we will refer throughout to an element being "less than" or "greater than"

29. Machine: Now we must select a *construction method* for building a new set from the old one. The choices are:

(a) List insertion: Modify the structure of the old list by inserting a new cell containing the new element.

(b) Shifting; Allocate a new space at the beginning (or end) of the set and shift all elements before (after) the new element by one position, Then put the new element into the vacant place.1

(c) Copying: Copy the old set into a new space, one cell larger than the old space, with the new element taking up the extra position.

Which shall I use?

30. **User:** List insertion.

31. Machine: Now I will write the body of the constructor. The body consists of two parts:

(1) a *position finder*, which finds the position where the new element belongs

(2) *a list inserter*, which inserts the element at this position

32. Machine: When a position has been found, there are several ways to indicate that position. First we must select one of these representations for a position,

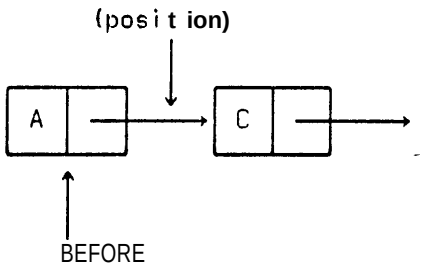
---

another element and mean that the first element precedes or follows the second element according to the chosen ordering relation.

† Perhaps, since the "shifting" operation is such an unlikely choice when lists are being used, it shouldn't be presented to the user, It could be called forth only upon questioning or in unusual programming contexts.

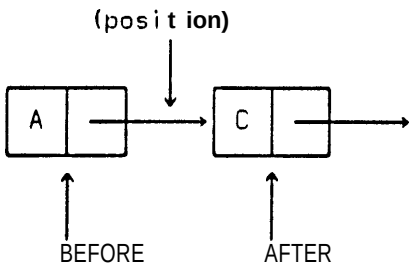
We are considering positions to be between two elements in a list. Since the construct ion met hod involves inserting a new cell into the list, we must choose a representat ion appropriate to insertions into linked lists. I know about two types of such represent ations:

(a) Use one pointer as illustrated:



In using this representation, a special unique value for BEFORE must be used if the correct position is at the front of the list. "FRONT" will be used for this purpose.

(b) Use two pointers as illustrated:



In using this representation; a special unique value for BEFORE must be used if the correct position is at the front of the list. "FRONT" will be used for this purpose. If the correct position is at the end of the list, AFTER will point to the empty list.

Which shall I use?

33. **User:** Please use two pointers. [This choice might be based either on looking ahead or on knowing at this point that the BEFORE pointer is useful for inserting

into the list and the AFTER pointer is useful for incrementing the state of the position finder and for testing the correctness of a position.]

84. **Machine:** I need variable names for the two pointers. I will use **BEFORE and AFTER.**

We arc about to work on the position finder as indicated

```
transfer initializer:
   • • •
transfer body:
   LOOP UNTIL DONE:
        • • •
        constructor body:
             add the element to output*
                  find the position for the new element;
                  insert the new element into the position;
   REPEAT;
   DONE: HALT.
```

36. Machine: I am now writing the program that finds the position. First we will select a *search strategy* for searching the output list to find the correct position. The only strategy which I know about -for linked lists is a *forward linear scan, so* I will use that strategy.

36. Machine: Since the search strategy is a forward linear scan of a finite list, the search strategy is total;- that is, all possible positions will be considered. Since the old output set is well-ordered, and since the ordering relation applies to all elements involved, one of these possible positions must be correct. Therefore, the search for a correct position must succeed, and no exit will be required for the case of not finding a correct position. So I will use a loop which consists only

of a test and exit on the correctness of the position and an incrementer. The incrementer will increment the state of the search according to the search strategy and specify the next position to be considered. I will also need to initialize the loop.

37. Machine: Since the search strategy is forward linear, the correctness test and the incrementer are independent. This means that the next position specified by the incrementer does not depend on the results of the correctness test, This allows a simple non-branching incrementer.†

We are about **to work on the indicated parts** of the position finder

```
transfer initializer:
   • • •
transfer body:
   LOOP UNTIL DONE:
      • • •
      constructor body:
         add the element to output⇒
            find the position for the new element⇒
               initialize position finder loop;                    ☜ (2)
               LOOP UNTIL CORRECT POSITION FOUND;
                  IF position is correct                           ☜ (3)
                     THEN CORRECT POSITION FOUND;
                  increment state and specify position;            ☜ (1)
               REPEAT;
               CORRECT POSITION FOUND:
            insert the new element into the position;
   REPEAT;
   DONE: HALT.
```

38. Machine:' Now let's design the incrementer, First we must select a scheme for

---

† If the search strategy had been to use a binary chop, for example, then the next position to be tested would be based on the results of the test.

representing the state of the search. The incrementer will increment this state and will also specify the next position to be considered,

39. Machine: Since the search strategy is a forward linear scan of the linked list, and since the two-pointer position representation has been selected, the two pointers (BEFORE and AFTER), which are already being used to represent the posit ion, are also sufficient to remember the st ate.1

40. Machine: An incrementer for a forward linear scan using the two-pointer position representation requires two pieces of code: one to reset BEFORE to the value of AFTER, and one to increment AFTER. This is adequate both for the general case in the interior of a linked list, and in the special case at the front of a list, where BEFORE originally has the special value "FRONT" and is reset to point to the first cell in the list. The pieces will be put together in that order,

41. Machine: I am now writing the code to reset BEFORE to the value of AFTER, The code to do that is:

```
BEFORE ← AFTER
```

42. Machine: I am now writing the code to increment AFTER. Since the representation is a linked list, the code to do that is:

```
AFTER ← CDR(AFTER)
```

---

† The position memory and the state memory need not always be the same. For example, if only BEFORE were used to remember the position, a good choice of state-memory would probably include a pointer playing the same role as AFTER.

43. Machine: Putting these two pieces together, we have the following increment er for the position finder:

```
BEFORE ← AFTER;
AFTER ← CDR(AFTER)
```

44. Machine: I will also need a position initializer which initializes both the state of the position finder and the position representation to the first position according to the search strategy.

45. Machine: Since the position representation is also used to represent the state, we need only initialize the position representation.

46.. Machine: Since the search strategy is forward linear scan, we must initialize the posit ion represent ation to the position at the front of the list. The convention for representing the position at the front of the list is that BEFORE has the unique value **"FRONT"** and that **AFTER** points to the entire list.

47. Maohinet The code to set BEFORE to "FRONT" is:

```
BEFORE ← "FRONT"
```

48. Machine: The code to set AFTER to the entire list is:

```
AFTER ← OUTPUT
```

40. Mauhinet Putting these two pieces together, the initializer for the position finder is:

```
BEFORE ← "FRONT";
AFTER ← OUTPUT
```

We havo just finished t ho indicated parts of the position finder

```
transfer initial izer:
   • • •
transfer body:
   LOOP UNTIL DONE:
       • • •
       constructor body:
           add the element to output*
               find the position for the new element-
                   initial ize position finder loop=+
                       BEFORE ← "FRONT";
                       AFTER ←OUTPUT;
                   LOOP UNTIL CORRECT POSITION FOUND;
                       IF position is correct
                           THEN CORRECT POSITION FOUND;
                       increment state and specify posi t ion=>
                           BEFORE ← AFTER;
                           AFTER ←CDR(AFTER);
                   REPEAT;
                   CORRECT POSITION FOUND:
               insert the new element into the position:
   REPEAT;
   DONE: ) IALT.
```

50. Machine: I will now work on the exit test for the correct position. I will first try to infer the necessary pieces of this test.

51. Machine: Since the set is ordered, it follows from transitivity that the new element need not be compared against all of the elements of the set, but merely against the elements immediately adjacent to the position being considered,

62. Machine: Since the search strategy is forward linear scan, at any point in the search, all of the elements occurring before the current position in the list have already been tested and found to be less than the new element. This holds for all positions including the initial position in which no elements occur before the position in the list.

63. Machine: Thus, the test for correctness of position need only include a test that all elements occurring after the current position in the list are greater than the new element.

64. Machine: To test that all elements occurring after the current posit ion are greater than the new element, the program must first test whether there are no elements occurring after the current position. Then, if there are such elements, it must test whether all such elements are greater than the new element.

65. Machine: A test that there are no elements occurring after the current position is simply a test whet her the list pointed to by the second pointer (AFTER) is empty:

```
AFTER = NIL ?
```

66. Machine: As noted previously, the correctness test need only consider the immediately adjacent elements, so that a test that all elements occurring after the current position are greater than the new element is simply a test that the first element occurring after the current position is greater than the new element. To write such a test I need code to extract the first element from the list pointed to by AFTER and a test whether this element is greater than the new element.

67. Machine: The code to extract the first element from a list is:

```
CAR(AFTER)
```

68. Machine: The code to test whether this element is greater than the new element is:

```
CAR(AFTER)> ELEMENT ?
```

60. Machine: Given the two partial tests, the complete correctness test is an ordered OR† of the two partial tests:

[AFTER . NIL] OR [CAR(AFTER)> ELEMENT] ?

The indicated parts complete the position finder

```
transfer initializer:
   • • •
transfer body:
   LOOP UNTIL DONE:
       • • •
       constructor body:
           add the element to output-
               find the position for the new element-
                   initial ize posit ion f incler loop*
                       BEFORE ← "FRONT";
                       AFTER ← OUTPUT;
                   LOOP UNTIL CORRECT POSITION FOUND;
                       IF position is correct⇒
                           no elements occur after position*
                               AFTER = NIL                    ☜
                           OR next element is greater-
                               CAR(AFTER)> ELEMENT            ☜
                           THEN CORRECT POSITION FOUND:
                       increment state and speci fy posi t ion⇒
                           BEFORE ← AFTER;
                           AFTER ←CDR(AFTER);
                   REPEAT;
                   CORRECT POSITION FOUND:
               insert the new element into the position;
   REPEAT;
   DONE: HALT,
```

60. Machine: Having completed the position finder, we must now write the list

---

† By "ordered OR" we mean that the second argument is evaluated only if the first argument is false.

inserter which takes a new element, a position represented by the two-pointer schema, and the output set, and produces a new output set with the new element in the indicated position. The method of inserting an element into a list depends on whether or not the indicated position is at the front of the list. I will need to write a test to separate the "front of the list" case from the "anywhere else" case, and use a different method for each case.

We are about to work on the indicated parts of the list inserter

```
transfer initializer:
   . . .
transfer body:
   LOOP UNTIL DONE:
      . . .
      constructor body:
         add the element to output-
            find the position for the new element:
            insert the new element into the position-
               IF position is at front                    ☜
                  THEN add element on to front            ☜
                  ELSE insert into interior               ☜
   REPEAT;
   DONE: HALT.
```

61. Machine: A test for being at the front of the list is a test whether the value of the first pointer is "FRONT". The code for this is;

```
BEFORE = "FRONT" ?
```

62. Machine: The addition of the new element in the "front of the list" case entails no replacement into the list, but instead an addition to the front of the list. The code to do this is:

```
OUTPUT ←CONS(ELEMENT OUTPUT)
```

63. Machine: The addition of the new element anywhere else in the list requires a replacement into a cell in the list. The code to do this is:

CDR(BEFORE) ← CONS(ELEMENT AFTER)†

The marked parts complete the element adder

```
transfer initializer:
   • • •
transfer body:
   LOOP UNTIL DONE:
        • • •
        constructor body:
            add the element to output;
                find the position for the new element:
                insert the new element into the position*
                    IF position is at front⇒
                        BEFORE = "FRONT"                          ☜
                        THEN add element onto front*
                            OUTPUT ← CONS(ELEMENT OUTPUT)         ☜
                        ELSE insert into interior*
                            CDR(BEFORE) ← CONS(ELEMENT AFTER);    ☜
   REPEAT;
   DONE: HALT.
```

64. Machine: Having completed the list inserter (and thus the constructor body), we must now write the constructor initializer. Since we are writing an inductive construct or that must begin with an empty output list, we simply set the out put list to the empty list:

OUTPUT ← NIL

---

† LISP programmers will recognize this as RPLACD.

The initializer completes the constructor

```
transfer initial izer:
   • • •
   constructor initializer*
      OUTPUT ← NIL;
transfer body:
   LOOP UNTIL DONE;
      • • •
      cons truc tot- body:
         add the element to output*
            find the position for the new element⇒
               initialize position finder loop⇒
                  BEFORE ← "FRONT";
                  AFTER ← OUTPUT;
               LOOP UNTIL CORRECT POSITION FOUND;
                  IF position is correct*
                     no e l ements occur after posi t ion⇒
                        AFTER = NIL
                     OR n e x t element is greater⇒
                        CAR(AFTER)> ELEMENT
                     THEN CORRECT POSITION FOUND;
                  incremen t state and specif y pos i t i on⇒
                     BEFORE ← AFTER;
                     AFTER ← CDR(AFTER);
               REPEAT;
               CORRECT POSITION FOUND:
            insert the new element into the posi tion⇒
               IF position is at f r o n t*
                  BEFORE = "FRONT"
                  THEN add element onto front-
                     OUTPUT ← CONS(ELEMENT OUTPUT)
                  ELSE insert i n t o interior⇒
                     CDR(BEFORE) ← CONS(ELEMENT AFTER);
   REPEAT;
   DONE: HALT,
```

65. -Machine: We have now finished the constructor.

(e) PART 4: Completing the Program

The final program

```
transfer initializer:
    selector initializer:*
        generator initializer:*
            REST ← INPUT;
    constructor initializer*
        OUTPUT ← NIL;
transfer body:
    LOOP UNTIL DONE:
        transfer termination test:
            IF all elements have been transferred*
                all elements have been selected⇒
                    all elements have been generated*
                        no elements remain⇒
                            REST = NIL
                THEN DONE;
        selector body:
            select the next element from input=+
                generate the next element from input⇒
                    produce element*
                        ELEMENT ← CAR(REST);
                    increment state of generator*
                        REST ← CDR(REST);
        constructor body:
            add the element to output*
                find the position for the new element*
                    initialize position finder loop*
                        BEFORE ← "FRONT";
                        AFTER ← OUTPUT;
                    LOOP UNTIL CORRECT POSITION FOUND:
                        IF position is correct=+
                            no elements occur after position*
                                AFTER = NIL
                            OR next element is greater⇒
                                CAR(AFTER)> ELEMENT
                          . THEN CORRECT POSITION FOUND;
                        increment state and specify position⇒
                            BEFORE ← AFTER;
                            AFTER ← CDR(AFTER);
                    REPEAT;
                    CORRECT POSITION FOUND:
                insert the new element in to the position⇒
                    IF position is at front⇒
                        BEFORE = "FRONT"
                        THEN add element onto front⇒
                            OUTPUT ← CONS(ELEMENT OUTPUT)
                        ELSE insert into interior*
                            CDR(BEFORE) ← CONS(ELEMENT AFTER);
    REPEAT;
    DONE: HALT,
```

66. Machine: Having completed the pieces of the Selector and the pieces of the Constructor, I can assemble the pieces into the final program using a **Generate** and Process paradigm with the termination test before the generator.

67. Machine: The final program, without any **meta-code** or comments, and its translation into LISP are given below:

```
REST ← INPUT;
OUTPUT ← NIL;
LOOP UNTIL DONE;
   IF
        REST = NIL
     THEN DONE;
   ELEMENT ← CAR(REST);
   REST ← CDR(REST);
   BEFORE ← "FRONT";
   AFTER ← OUTPUT;
   LOOP UNTIL CORRECT POSITION FOUND;
     IF
          AFTER = NIL
          OR CAR(AFTER) > ELEMENT
        THEN CORRECT POSITION FOUND;
     BEFORE ← AFTER;
     AFTER ← CDR(AFTER);
   REPEAT;
   CORRECT POSITION FOUND:
   IF
       BEFORE = "FRONT"
     THEN OUTPUT ←
          CONS(ELEMENT OUTPUT)
     ELSE
          CDR(BEFORE) ←
            CONS(ELEMENT AFTER);
REPEAT;
DONE: HALT,
```

```
[LAMBDA NIL
  (PROG NIL
        (SETQ OUTPUT NIL)
        (SETQ REST INPUT)
   L1
        (COND
          ((NULL REST)
            GO L2)))
        (SETQ ELEMENT (CAR REST))
        (SETQ REST (CDR REST))
        (SETQ BEFORE "FRONT")
        (SETQ AFTER OUTPUT)
   L3
        (COND
          ((OR (NULL AFTER)
               (GREATERP (CAR AFTER) ELEMENT))
            GO L4)))
        (SETQ BEFORE AFTER)
        (SETQ AFTER (CDR AFTER))
        (GO L3)
   L4
        (COND
          ((EQUAL BEFORE "FRONT")
            (SETQ OUTPUT
                  (CONS ELEMENT OUTPUT)))
          (T
            (RPLACD BEFORE
                    (CONS ELEMENT AFTER))))
        (GO L1)
   L2 (RETURN NIL]
```

III. TYPES OF PROGRAMMING KNOWLEDGE

On reviewing the dialogue, we can see that there are several types of knowledge involved. We first note that there is significant use of a kind of strategy or planning knowledge. On one level, we see this in steps 9 and 14, where the system discusses what must be done to write a transfer program. In step 9 for example, the sub-steps 3 and 4, where the transfer order and the transfer termination method are chosen, are really a kind of strategy for determining the form that the basic algorithm will take. On a different level, we see a kind of global optimization in steps 21 and 39, where the system decides that information structures designed for one purpose are sufficient for another. In step 21, for example, the pointer originally chosen to save the state of the selector (by marking the dividing point between those elements generated and those not yet generated) is found to be adequate for the purpose of indicating the next element to be generated. One could imagine, as an alternative to this type of planning, the use of more conventional local optimization such as post-synthesis removal or combination of redundant portions.

We also see that the system makes considerable use of inference and simplification knowledge. **Inference plays a role in the global optimization** planning mentioned above, and also **appears** in steps 16 and 28, where the selection and construct ion re-orderings are determined. Simplification and inference are both apparent in steps 50 through 56, where the test for the correctness of the posit ion was reduced to a simple test on the variable **AFTER.** Simplification and inference are also needed in step 36 where the system decides that an error exit (for the case of no position being found) is unnecessary.

Additionally, there are types of knowledge which are spread throughout the dialogue. Relatively domain-specific knowledge (in this case, about sorting) is particularly necessary in the earlier stages. Language-specific knowledge (in this case, about LISP) is necessary when the final code is being generated. General programming knowledge, such as knowledge about set enumeration and linked lists, is necessary throughout the synthesis process. Further, one could imagine significant use of efficiency information, although it is not present in our particular dialogue.

The variety of types and amounts of knowledge used in the dialogue would tend to indicate that much more information is required for automatic synthesis of sorting programs than appeared in earlier, computer-implemented, systems for writing sort programs [3, 7, 11]. Ruth has developed a formulation of the knowledge involved in interchange and bubble sort programs [9]. His formulation is aimed primarily at the analysis of simple student programs in an instructional environment and the analysis task as defined does not seem to require the same depth and generality of knowledge suggested by our dialogue. Our intuition is t-hat a significantly greater depth of programming knowledge would be required to extend his formulation to a larger class of programs. It is also interesting to compare the information involved in our dialogue to that found in non-implemcntcd (and not intended for machine implementation) human-oriented guides for sort-algorithm selection and in text books on sorting. Martin [8] gives methods for selecting a good algorithm for a particular sorting problem. Those algorithms are much more powerful than those we deal with and their derivation would require considerably more information. We note that at the level of algorithm

description presented, little explicit information is available to allow pieces of algorithms to be fitted together or to allow slight modification of existing algorithms, A sorting textbook such as [5], gives several orders of magnitude more information on sorting than is required for our dialogue.

Can we measure or estimate in some way how much knowledge is necessary for program-understanding systems? The fact that the dialogue describing the synthesis took some seventy steps (with some of the steps rather complex) is an indication that considerable information is involved. From our experiments, we estimate that about one or two hundred explicitly stated "facts" or rules would get a synthesis system through the underlying steps of this dialogue. Furthermore, it is our guess that at least this much knowledge density will be required for other similar tasks, in order to have the flexibility necessary for the many aspects of program understanding. Although we are suggesting that such information must be effectively available in some form to a system, we are not in a position to estimate how much of this information should be stated explicitly (as, say, rules), how much should be derivable (from, say, meta-rules), how much should be learned from experience, or available in any other fashion.

XV. SUMMARY AND CONCLUSIONS

In this paper we have tried to exemplify and specify the knowledge appropriate for a-program-understanding system which can synthesize small programs, by presenting a dialogue between a hypothetical version of such a system and a user. Our conjecture

is that unless a system is capable of exceeding the reasoning power, and even some of the communication abilities, exemplified by the dialogue, the system will not effectively "understand" what it is doing well enough to synthesize, analyze, modify, and debug programs. It appears that a system which attempts to meet this standard must have large amounts of many different kinds of knowledge. Most such programming knowledge remains to be codified into some form of machine implementable theory. In fact, the codification of such knowledge is one of the main research problems in program-understanding systems.

As for our own work, in the near future we expect to refine our experimental system until it approaches (as closely as seems useful and possible) the standard suggested by our dialogue (but without the actual language interface). We hope then to extend the system to deal with several different types of sorting programs. Perhaps then we will be in a better position to estimate the requirements of larger program-understanding systems.

V. ACKNOWLEDGEMENTS

The authors gratefully acknowledge the helpful suggestions given by Avra J. Cohn, Brian P. McCune, Richard J. Waldinger, and Elaine Kant after numerous readings of earlier drafts of this paper. Computer time for our preliminary tests was made available by the Artificial Intelligence Center of the Stanford Research Institute.

VI. REFERENCES

[1] Dahl, O.J., Dijkstra, E.W., and Hoare, C.A.R.,Structured Programming (New York: Academic Press, Inc., 1972).

[2] Floyd, Robert W., Toward Interactive Design of Correct Programs, Computer Science report STAN-CS-71-235, (Stanford University, September, 1971).

[3] Green, C. Cordell, The Application of Theorem Proving to Question-Answering Systems, Computer Science report STAN-CS-69-138, AIM-96, AD696394,(Stanford University, August, 1969).

[4] Green, CC., Waldinger, R.J., Barstow, D.R., Elschlager, R., Lenat,D.B,McCune, B.P., Shaw, D.E., and Steinberg, L.I., Progress Report on Program-Undcrst anding Systems, Computer Scienc report STAN-CS-74-444, (Stanford University, August, 1974).

[5] Knuth, Donald E., Sorting and Searching, *The Art of Computer Programming, vol. 3* (Reading, Mass.: Addison-Wesley, 1973).

[6] Knuth, Donald E., Structured Programming with *GOTO* Statements, *Compting Surveys, vol.6* (December, 1974), 26 l-301.

[7] Kowalski, R.A., Predicate logic as programming language, Memo no, 70, Dept. of Computational Logic, (University of Edinburgh, November, 1 974).

[8] Mart in, William A, Sorting, *Computing Surveys, vol. 3* (December, 1971),147-174.

[9] Ruth, Gregory R., Analysis of Algorithm Implementations, Project MAC report MAC TR- 130, (Massachusetts Institute of Technology, May, 1974).

[10] Teitelman, Warren, *et al*, INTERLISP Reference Manual, (Xerox Palo Alto Research Cent cr and Bolt Beranek & Newman, 1974).

[11] van Emden, M.H., First-order predicate logic as a high-level program language, Report MIP-R-106, Dept. of Machine Intelligence, (University of Edinburgh, May, 1974).

[123 Zahn, Charles T., A Control Statement for Natural Top-down Structured Programming, presented at Symposium on Programming Languages, Paris, (1974).