SOME RULES FOR THE AUTOMATIC SYNTHESIS OP PROGRAMS

Cordell Green and David Bar stow
Stanford Artificial Intelligence Laboratory
Stanford, California USA

ABSTRACT

A set of rules (or facts) about program synthesis is presented. The rules are about the process of programming, and are sutficient for the synthesis of an insertion sort program. The use of the rules to write a short LISP program is described. Taken together, the rules are an embodiment of a detailed theory which explains one small part of the programming process. The size of the set of rules suggests the complexity of the process of writing programs and indicates that much work will be required to codify significant amounts of programming knowledge as a step toward the development of program-understanding systems.

INTRODUCTION

This paper may be viewed as a sequel to an earlier paper [1], in which we tried to exhibit the knowledge required by a computer system in order to synthesize a simple insertion sort program. In this paper, we present the results of an attempt to codify that knowledge in the form of an *explicit system of rules.* Although there are other ways in which this knowledge might be available to a computer system (e.g., derivable through some kind of inference applied to more general knowledge), the rules presented here seem to express the information used in the process of writing one simple program. As such, the rules constitute one detailed part of a *theory of programming.* A more complete theory could provide the basis for a knowledge-based program-understanding system.

The rules presented here are about the *process of programming,* rather than about program semantics. Thus one of the rules states that one method of generating all elements in a stored set is to first select a method for saving the state of the generator, then write the body, then the initializer, and so on. Additional rules elaborate how to achieve each subgoal The subjects of this set of rules include the synthesis of *transfer programs* (in which the elements of one ordered set are transferred and possibly re-ordered into a new ordered set), generators of elements and positions in ordered sets, constructors of sets, search strategies, tests for correctness of a position, etc. Also included are the lower-level programming rules (for the LISP language) that are necessary to actually code the program. The complete set of rules has in fact been used to synthesize the intended program in an implementation which is basically a rule-testing system. This implementation is described at the end of the paper.

It may seem strange that so many rules are used for one simple program. The number and complexity of these rules has resulted from a desire to adequately reflect the amount of knowledge needed for this particular synthesis and from an effort to minimize the amount of remodeling required to add more knowledge

Despite our desires and intentions, however, there are several shortcomings of the rules and we would like to make some disclaimers. The most obvious limitation is the amount of higher-level programming knowledge which is *not* included,,e.g., more complex search strategies (such as binary chop), other state-saving schemes (such as the overwriting of elements), and other knowledge necessary for more efficient sort programs. The reader will usually find only one method explicated when many more come to rnind. We hope to correct these omissions by expanding this rule set in the future. A real likelihood with this "skeletal" approach is that the framework provided by this initial set of rules may well need remodeling to accommodate further programming knowledge. The only overall framework presented in this paper, that of an iterative transfer program, will perhaps not be the reader's (or the eventual user's) favorite way to view the synthesis of a sort program Othor weaknesses are that the rules do not allow enough "planning" activity (resulting in a rather rigid sequence that must be followed), and that a few rules probably have too much specialized knowledge. It is also important to note that the rules are currently designed for *synthesis* and would require modification (or else a different set of rules) in order to *analyse* or *modify* existing programs. Our omission of conventional inference ability is intentional, since we feel that the state of the art in the area is well ahead of the state of any theory about the process of programming. However, it is clear that in a more general program-under stanoing system inference plays a part in selecting applicable rules and in putting the pieces of a program together in any non-preconceived order.

f

OVERVIEW

To provide a context within which the rules can be easily understood, we present here a short desctiption of the target insertion sort program. The complete program produced by our rules appears in the section giving an example application of the rules, and the reader may wish to refer to it now.

```
transfer program:
  initializer:
    initialize the generator and the output list;
  body: repeat until finished;
    termination test:
      if there are no elements left in
       the input list then finished;
    selector:
      generate the next element;
      remember the state of the generator;
    constructor:
      position-finder:
        find the position in the output list
         for the generated element;
      element-adder:
        add the generated element in the
         position found;
    repeat;
```

The program is basically a loop with three parts: a termination test which tests whether all of the elements in the input list have been generated; a selector which generates the next element in a first to last order and saves its computational state; and a constructor which finds the position in the output list for the element and then adds it at that position. Thus, the program takes each of the elements from the input list (in first to last order) and builds up the output list by inserting these (one at a time) into their appropriate places.

Note that the program is basically iterative rather than recursive. This is primarily a reflection of our feeling that significant aspects of the algorithm are hidden by a recursive call. In many real applications, programmers must deal with the notion of generating elements sequentially, and of saving the state of a computation, so we have chosen to look at these explicitly.

We will now give an overview ot how these rules would be used to synthesize this program,

0: As stated in the introduction, we will assume that the system has already decided to use a *transfer program* as the basic paradigm. A transfer program consists of two parts, the *selector,* which enumerates elements from the input set, and the *constructor,* which places these elements in the output set. This transfer program is an instance of a *generate and process* paradigm, in which the selector is the generator and the constructor is the process

I: The system decides to write a *insertion sort* rather than an s*election so*rl program. (An insertion sort removes the elements from the input in first to last order and inserts each into its correct place in the output such that after each step the output set is ordered; by contrast, a selection sort removes the largest element from the input and adds it onto the front of the output; the rules presented here only consider the case of insertion sort programs.)

2: The selector for the transfer program is written. First a pointer into the input list is chosen as the method for saving the state of the selector between colls to it. Then the selecter *body* (consisting of code to generate the element and code to increment the pointer) is synthesized Finally the *selector initializer* and the selecto*r pre -test* (a test which the transfer program can apply to determine whether there are any more elements to be generated) are written.

3: Work on the output constructor is begun First it is decided that the new set will be built up by *list insertion* Then, noticing that the elements arrive not necessarily in increasing (or decreasing) order, the system decides that it needs a somewhat complex constructor that must search for the position to insert each element It chooses a pair of pointers into the list to represent that position

4: Now the *position finder* is synthesized. This requires choosing a *search strategy* for finding the correct position, and a method for representing the state of the search. Once these have been chosen, the loop which finds the correct position is written: this includes writing *an initializer,* a *position tester,* and an *mcrenenfer*

5: Now the element inserter is written It requires a special case for inserting at the front of the list The position finder and element inserter together constitute the *constructor body.*

6: To complete the output constructor, the system writes the *constructor initializer* which initializes the output set to the empty list.

THE RULE SYSTEMS

As an aid in understanding the rules, let us point out that they are grouped into structured systems of rules, each system dealing with a different aspect of the process of writing a transfer program. As will become apparent, there is a simple hierarchical structure to the rule systems. This, we feel, is a reflection of the nature of the task: some goals require other goals to be satisfied, hence some rules call others as sub-rules. However, a strict hierarchy is certainly unnecessaryfor example, in the process of writing a non-linear generator, the generator rule is called again, in order to write the code to generate the elements linearly for use in the comparison.

We wish to point out that, in the rules presented here, wherever a choice is indicated, only one alternative is actually provided. When we have extended the rules to enable the synthesis of a large class of programs, most of these rules will contain several options. Choices between them could be made either by using additional rules dealing with efficiency or by leaving such choices up to the user. But the important point here is that the notion of "understanding" seems to carry with it an awareness of possible alternatives

The Rule system for Transfer Program

There is one transfer program rule:

TI: tn order to write a transfer program, the following lour sub-tasks must be achieved:

(1) select a *transfer rt-oulning*

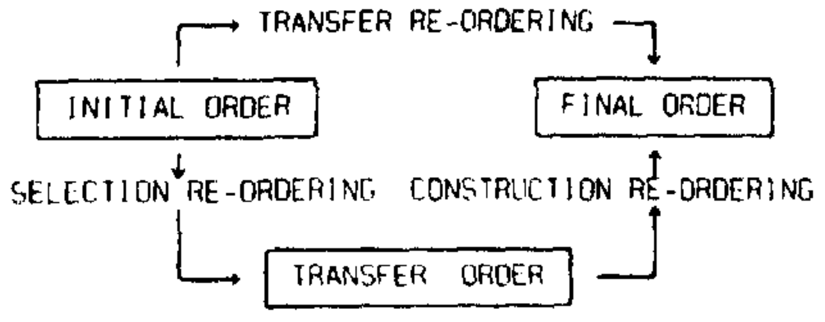
(2) write a *selector,* which will include a body, initializer, and pre-test;

(3) write a *constructor,* which will include a body, and an initializer

(4) put the selector and constructor together into a *grvaatr and process* paradigm with the selector pre-test used as the termination test and applied just before the call to the selector body

The meanings of most parts of this rule are relatively obvious, and we shall delay the detailed explanation of each of its sub-tasks until we discuss the rules for achieving those sub-task

The first choice which must be made in the process of writing a transfer program is to choose the pattern of re-ordering to be used by the program. The input to the transfer program we wilt call the sou and the output we will call the *target.* As part of the transfer process, there will be some overall re-ordering from the positional ordering of the source *(initial order)* to the positional ordering of the target *(final order).* There is also an intermediate ordering: the temporal ordering of the elements as they are transferred *(transfer order).* Corresponding to these three orderings, there are relationships between them We shall discuss these relationships as functions from one order to the next order, i.e. *u-otlinings* or permutations. The orderings and re-orderings are shown in the diagram.



The *transfer reordering* is the main re-ordering accomplished by the transfer program. This re-ordering is part of the top-level specification of the program, presumably furnished by the user or a higher-level program An example of a transfer re-ordering specification for a "reverse" program is "final order is converse of initial order". In our paradigm, we factor this re-ordering into two re-ordenngs, the *selection reordering* and the c*onstruction re ordering.* Accordingly the transfer program is also factored into two sub-programs, the *selector* and the *constructor* which perform these re-orderings.

The rules consider four possible re-ordering operations:

1. SAME; makes a copy with the identical ordering.

2. CONVERSE; reverses an ordered set.

3. BASED ON COMPARISON PREDICATE: the re-ordered set is such that one element precedes another if and only if the comparison predicate (supplied by the user) applied to these elements is true. The order of the re-ordered set is independent of the initial order (except in the case where two elements have equal keys)

4 BASED ON EVALUATION FUNCTION: the re-ordered set is such that one element precedes another if and only if the value of the first element is less than the the value of the second, where the values are the results of applying the evaluation function (supplied by the user). The order of the *re-ordered* set is independent of the initial order (Although this is actually a special case of 3, we include it since people seem to use it frequently.)

It is convenient to classify the first two re-orderings as *linear* since the programs to accomplish these are simple and are treated as special cases

In addition to the transfer program rule given above, there is a set of transfer re-ordering rules which guide the choice of selection re-orderings and construction re-orderings. Their effect is to constrain the choices for the two re-orderings to be such that their composition will yield the transfer re-ordering An example of such a rule is; if we are writing a "reverse" program and the selection re-ordering is SAME, then the construction re-ordering must be CONVERSE The normal case in a sort program is that either the selector or the constructor will do the bulk of the work of the total re-ordering, and that the other will perform only a simple (linear) re-ordering (either SAME or CONVERSE). The transfer re-ordering rules are straightforward but lengthy, so we shall not present them here.

The Rule system for Selector

In considering the rule system for a *selector,* we first observe that a selector is a special kind of *generator.* That is, a selector is required to generate exactly the elements of the source set, once and only once, and in a particular order (the order discussed earlier as the transfer order). Perhaps the most important aspect of any generator is that, relative to the program which uses it, it is essentially a process or co-routine; that is, between calls to it, it must somehow "remember" which elements have been generated and which have not, in order to guarantee generating all elements one time each This "remembering" must be part of what is performed by the body of the generator, in addition to "generating" the next element

Also, there must be some way of determining when the generator is finished, that is, when each element in the source set has been generated and none remain. In the generators for which our rules are intended, this will be done by using a *pretest* which the calling program guarantees to apply before each call to the generator (for example, a test whether a pointer points to the empty list.) This is the test which the transfer program will use to decide when to stop transferring. Finally, there must be some way of initializing the generator prior to its first call. From this discussion, we can see that the rule system for the selector (generator) must produce three separate sections of code: the main body, the pre-test, and the initializer

It is also worth noting that the generator must know the manner in which the generated element should be given to the caller. In the case of our rules, we consider only the case of setting some global variable to a particular value, either the element itself or a pointer to the sublist beginning with that element

Si: In order to write a selector, one of the following sub-tasks must be achieved:
(\) write a *generator* with the *production Older* the same as the selection re-ordering, and constrain the generator to be TOTAL

S2: In order to write a generator, the following four sub-tasks must be achieved.
(!) select a *state-saving scheme* for the generator
(2) write the *generator body,* based on the state-saving scheme
(3) write the *generator initializer,* based on the state-saving scheme
(4) write the *generator pretest,* based on the state-saving scheme

Note that the first sub-task of this rule is the selection of a state-saving scheme for the generator This state-saving scheme will be essentially a specification of some plan about how to do the "remembering" discussed above. There are many possible schemes which will perform adequately (eg, bit strings, property list marks, hash table entries), but there are certain common characteristics. Each scheme includes some way of looking at a particular description of the state of the generator and knowing (or deducing) which elements of the set have been generated. Additionally, each scheme includes some way of incrementing the state after the appropriate element has been produced. Also implicit in any such scheme must be some way of determining (from a state description) whether any elements remain to be generated.

The choice of state-saving scheme depends upon the representation of the set, the order of enumeration, and whether destructive operations are allowed. For the case we are considering, the enumeration of the elements in a list requires only one variable, which points into the list. In this standard enumeration scheme, the pointer always points to the sublist beginning with the next element to be generated and is bumped by one more list cell each time a new element is required. We will refer to this scheme as POINTER INTO SOURCE and to the pointer as POINTER. The rule may be written as;

GSI: In order to select a state-saving scheme for a generator:
(I) if a non-destructive state-saving scheme is desired, the source-set is represented as a linked list, and the production re-ordertng is SAME, then use POINTER INTO SOURCE

For a simple linked-list, forward scanning generator, the part of the program that finds the next element and the part of the program that increments the state communicate very simply through one pointer, the pointer used for the state representation With more complex state-saving schemes, additional variables may be required to communicate between these two subprograms. For example, with o destructive, element-deleting scheme, the state is represented by one pointer to the front of the list, but in addition another pointer (indicating which element is to be deleted) must be passed from the element finder (which finds the element) to the state incrementer (which deletes that element) The following rule is thus a minor simplification

S3: In order to write the generator body, the following three sub-tasks must be achieved;
(1) write the part of the body which produces the element to be generated
(2) write the part of the body which increments the state of the generator
(3) put these two parts together into a sequential block

Two ways in which a generator can report an element to its caller are (1) by a pointer to the sublist beginning with that element and (2) by a pointer to the element itself. These two *element tepresanations* we call SUBLIST and ITEM respectively (Actually, the transfer program rule, TI, needed to specify the ITEM representation, so that the selector and constructor could communicate.)

S4: In order to write the part of the body which produces the element to be generated, the following two sub-tasks must be achieved;
(1) write an expression which has as its value the element to be generated
(2) write a statement which assigns this value to the variable which should hold the element to be generated

S5: In order to write an expression which has as its value the element to be generated;
(1) if the generator state-saving scheme is POINTER INTO SOURCE, then write an expression which has as its value the clement representation of the first element in the list pointed to by POINTER

S6: In order to write an expression which has as its value the element representation of the first element in a list:
(I) if the element representation is ITEM, then write an expression which has as its value the first element of the list pointed to by POINTER

Note that ss considers which element to produce and S6 considers how to represent it.

S7: In order to write the part of the body which increments the state of the generator;
()) if the generator state-saving scheme is POINTER INTO SOURCE, then write a statement which increments POINTER

S8: In order to write the generator initializer:
(1) if the generator is constrained to be TOTAL, then write code which initializes the state to the entire source

s9: In order to write code which initializes the state of the generator to the entire source;
(1) if the generator state-saving scheme is POINTER INTO SOURCE, then write a statement which assigns the source to POINTER

S10: In order to write the generator pre-test:
(I) if the generator is constrained to be TOTAL, then write a statement which tests whether all of the elements have been generated

SII: In order to write a statement which tests whether all of the elements have been generated;
(1) if the generator state-saving scheme is POINTER INTO SOURCE, then write a statement which tests whether POINTER points to the end of a list

The Rule system for Constructor

A *constructor* builds a new ordered set in accordance with the construction re-ordering. The constructor rules given here synthesize a constructor which operates inductively; that is, the new set is initially empty and after each element has been added, the resulting subset satisfies the re-ordering relation. The constructor has a *constructor initializer*, which makes the new set initially empty, and a *constructor body*. The main body first finds the correct position for the new element and then inserts the element.

CI: In order to write a constructor, one of the following sub-tasks must be achieved:
(1) write an *inductive constructor* with the *construction order* the same as the construction re-ordering; the target should be INITIALLY EMPTY

C2: In order to write an inductive constructor, the following three sub-tasks must be achieved:
(1) select a *construction method* for the constructor
(2) write the *constructor body*, based on the construction method
(3) write the *constructor initializer*, based on the construction method

The rule for selecting a construction method is very simple:

CMI: In order to select a construction method:
(1) if the set being constructed is represented as a linked list, then use LIST INSERTION

Although there are several ways to do this operation, our use of the term "list insertion" refers to the commonly used method in which the "cdr" of the list cell preceding the desired position is destructively modified to point to a newly created cell which is in turn linked back to the rest of the list. Note that this technique requires saving a pointer to the list cell preceding the position and it also requires special treatment at the front of the list.

We will classify list insertion as being *nuhpnuii-nt,* meaning that the code which finds the position where the new element belongs and the code which adds the new element at that position are independent and may be written as separate pieces of code. [If the construction method were, say, array shifting, then this independence would no longer hold]

C3: In order to write the constructor body;
(1) if the construction method has the property "independent", then achieve all of the following sub-tasks:
<1.1) select a method for representing the desired position
0.2) write the part of the body which finds the position in which the new element belongs
(1.3) write the part of the body which adds the new element at this position
(1.4) put these two parts together into a sequential block

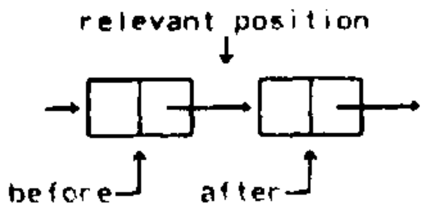
The position finder and the element adder both require that a method be specified for representing the position at which the new element will be added For example, in an array, the index of the correct position is an adequate representation. For lists, one or two pointers into the list are convenient position representations.

The rule we shall use for selecting a position representation is:

PRI; In order to select a position representation;

(1) if the set being constructed is represented as a linked list and the construction method is LIST INSERTION, then use TWO POINTERS (which will point to the list cells before and after the point where the new cell will be added)

Pictorially, this position representation looks like this;



This representation has a special case for the front of a list: the "before" pointer will have as its value a unique string constant, say, "FRONT". For the end of the list, the "after" pointer will point to NIL. We will refer to the two variables involved as BEFORE and AFTER

Note that this position representation is more than adequate for the list insertion method used, since it maintains not only a pointer to the previous cell in the list, but also a pointer to the succeeding cell in the list. The second pointer will be used by the correctness test. This representation leads to a relatively simple algorithm.

The selection of this rule and the corresponding construction method is pretty obviously a cheat, since the rules should first do some planning, i.e., notice that later parts of the program would have profited from the two-pointer representation An alternative to a planning technique would have been some later simplification or optimization of the program.

C4: In order to write the part of the body which finds the position in which the new element belongs:

(1) if the construction order is not "linear", then write a *position finder* which guarantees success (discussed later before rule P1); it must find the desired position according to the construction order, and specify it accoiding to the position representation

In a similar manner to the selector, a linear construction re-ordering would mean that the position is already implicit in the state, say an index moving linearly through an array, and need not be recomputed. In the remaining rules, rather than saying "an ordered set represented as a linked list", we will use the term "list'.

C5: In order to write the part of the body which adds the element at the position:

(1) if the position representation has a special case for the front of the list and the construction method has a special case for the front of the list, then achieve all of the following sub-tasks:

(1.1) write a statement which tests whether the position (which has already been found) occurs at the front of the list

(1.2) write a statement which adds the element onto the front of the list

(1.3) write a statement which inserts the element into the list at the position specified (assuming the position is not at the front)

(1.4) put these three pieces together into a conditional expression

C6: In order to write a statement which tests whether the position is at the front of the list:

(1) if the position representation is TWO POINTERS, then write a statement which tests whether BEFORE is equal to the special string constant "FRONT"

C7: In order to write a statement which adds the element onto the front of the list;

(1) if the construction method is LIST INSERTION, then write a statement which destructively adds the element onto the front of the list

C8: In order to write a statement which inserts the element into the middle of the list:

(J) if the construction method is LIST INSERTION, then write a statement which destructively inserts the element into the middle of the list

C9: In order to write a statement which destructively inserts the element into the middle of the list;

(1) if the position representation is TWO POINTERS, then write a statement which destructively inserts the clement between two pointers

C10: In order to write the constructor initializer:

(1) if the target is specified to be INITIALLY EMPTY, then write code which initializes the target to an empty list

C11: In order to write code which initializes the target to an empty list:

(1) if the construction method is LIST INSERTION, then write a statement which assigns the empty list to the target

The Rule system for Position Finder

The *postion finder* looks through the available positions in the set being constructed and finds the correct position at which to insert the new element. The position finder can be thought of as a total generator that generates each position and tests whether the position is correct. A *search strategy* is required to determine the order of enumeration of positions A *search-state saving scheme* is used to remember the state of the search. A *conectness test* is synthesized to test whether the proposed position results in an ordered set. Each position must be represented according to the earlier specified position representation, so some code may be required to translate from the search state representation into the position representation.

The reader may wonder why the position finder rules do not call the high level generator rules discussed in the selector section Instead a slightly different type of generator is effectively entailed by these position finder rules In other versions we have the position finder call the existing generator rules. For clarity we present separate rules here for the position finder, although we feel that it is conceptually better to combine the two into a unifying paradigm

The position finder was constrained by the constructor to be one that guarantees success, i.e., it must find a correct position or else the inductive constructor won't work Structurally, this means there is no failure branch for the position finder

Pi: In order to write a position finder that guarantees success, the following five sub-tasks must be achieved:

(1) select a *search strategy* that guarantees success

(2) select a *search-state saving scheme* for the position finder, based on the search strategy

(3) write the *position finder body,* based on the search strategy and the search-state saving scheme

(4) write the *position finder initializer,* based on the search strategy and the search-state saving scheme

(5) put the last two together into a sequential block

The selection of a search strategy that guarantees success consists of lots of checking followed by the selection of a simple forward linear scan.

SSI: To choose a search strategy that guarantees success, one technique is:
(1) check that a correct position exists
(2) select a total search strategy

SS2: To check that a correct position exists, one technique is to check these conditions:
(1) if the correctness criteria is based on a comparison predicate (an ordering relation);
(1.1) the ordering relation is transitive
(1.2) the set being scanned is ordered

SS3: To select a total search strategy;
(1) if the search list is represented as a linked list, then use FORWARD LINEAR SCAN for the search strategy

We will speak of this search strategy as being *tndepenent ,* meaning that the part which finds the position and the part which tests it for correctness are independent of each other [as opposed to the case with a binary chop].

The search for a position requires a state-saving mechanism, just as did the earlier kind of generator. The state-saving scheme given by our rule will be TWO POINTERS, pointing respectively before and after the current position. Recall that this is the same as the position representation. The rule first checks some conditions to see if a two pointer scheme is adequate Next it notes that since the position representation and the search state representation are the same, they will be combined and the state representation will be said to *subsume* the position representation. Finally, the variables used for the position representation become the variables used for the search state representation

SSS1: To select a position finder search-state saving scheme:
(1) if a non-destructive scheme is all right, a "linear" strategy is being used, and the search list is represented as a linked list, then;
(1.1) a two pointer state representation will work
(1.2) note that the position representation is subsumed by the search state representation
(1.3) if the position representation includes a "before" pointer, then that is superceded by the state's "before" pointer
(1.4) if the position representation includes an "after" pointer, then that is superceded by the state's "after" pointer

We note that this rule is something of a cheat in that, .again, some planning should be required to select such a good and optimizing state representation. This rule is a good example of trying to embody knowledge in a form too specific to a particular state or position representation. More reasonable would be a general rule giving a preference for using one variable instead of two when they would perform similar functions,

P2: In order to write the position finder body:
(1) if the search strategy is "independent", then achieve ail of the following sub-tasks;
(1.1) write the part of the body which translates from the representation of the state of the search into the representation of the current position
(1.2) write the part of the body which tests whether the determined position is correct according to the criteria for acceptance
(1.3) write the part of the body which increments the state of the search
(1.4) write the part of the body which tests whether there are no positions left, thus indicating failure to find the desired position
(1.5) put the pieces together into a loop

The fact that the position representation is subsumed by the search state means that at any point in the search, the next position to be tested is completely specified by the search-state representation, i.e., absolutely no work must be done in order to specify the position to be tested, given the state of the search This is reflected in the next rule

P3: In order to write the part of the body which translates from the representation of the state of the search into the current position:
(1) if the position representation is subsumed by the search-state scheme, then there is nothing left to do to determine the position to be tested

The next five rules, *PA* through P8, represent a chain of simplifications of the test that the position is correct, i.e., the new set is ordered. The chain proceeds from testing the new element against each other element to checking against only one. This situation may occur frequently enough that one special case rule should be used.

P4: In order to write the part of the body which tests whether the determined position is correct according to the criteria for acceptance:
(1) if the criteria is based upon a comparison predicate, then achieve all of the following sub-tasks:
(1.1) write a statement which tests the new element against all of the elements preceding the determined position
(1.2) write a statement which tests the new element against all of the elements following the determined position
(1.3) if either statement is vacuous, then return only the other; otherwise, combine the two tests into a conjunction

P5: In order to write a statement which tests an element against all of the elements preceding the determined position:
(1) if the comparison predicate is transitive, and the search strategy is FORWARD LINEAR SCAN, then there is nothing left to do to test the element against all of the elements preceding the determined position

P6: In order to write a statement which tests an element against all of the elements following the determined position, the following three sub-tasks must be achieved;
(1) write a statement which tests whether there are no more elements after the determined position
(2) write a statement which tests the new element against all of the elements following the determined position, assuming there is at least *one* such element
(3) combine these two tests into an ordered disjunction

P7; In order to write a statement which tests whether there are no more elements after a position:
(1) if the position representation is TWO POINTERS, then write a statement which tests whether AFTER points to the end of a list

P8: In order to write a statement which tests an element against all of the elements following a position, assuming there is at least one:
(1) if the comparison predicate is transitive and the search list is ordered, then write a statement which tests the element against the immediately following element

The rest of the rules are more or less self-explanatory,

P9: In order to write a statement which tests an element against the immediately following element, the following two sub-tasks must be achieved:
(1) write an expression which has as its value the first element after the position
(2) write a statement which applies the comparison predicate to the new element and the expression just written

P10: In order to write a statement which has as its value the first element after a position;
(1) if the position representation is TWO POINTERS, then write a statement which has as its value the first element of the list pointed to by AFTER

P11: In order to write the part of the body which increments the state of the search:
0) if the search strategy is FORWARD LINEAR SCAN, then write a statement which increments the search state to the next following position

P12: In order to write a statement which increments the search state to the next following position;
(1) if the search-state saving scheme is TWO POINTERS, then achieve all of the following sub-tasks;
(1.1) write a statement which assigns to BEFORE the value of AFTER
(1.2) write a statement which increments AFTER
(1.3) combine these two into a sequential block

Note that this incrementing technique works both for the special case where BEFORE has the special value "FRONT" and for the general case where BEFORE points into the output list,

Pi3: In order to write the part of the body which tests whether there are no positions left:
(1) if the position finder guarantees success, then this part is unnecessary

PI4: In order to write the position finder initializer:
(1) if the search strategy is FORWARD LINEAR SCAN, then write a statement which initializes the search state to the first position in the search list

PI5: In order to write a statement which initializes the search state to the first position in a list:
(1) if the search representation is TWO POINTERS, then achieve all of the following sub-tasks:
(1.1) write a statement which assigns to BEFORE the string constant "FRONT"
(1.2) write a statement which assigns to AFTER the search list
(1.3) combine these into a sequential block

The Rules for LISP Statements

The rules given here are included only for completeness. They embody the LISP knowledge necessary for the insertion sort program.

LI: In order to write an expression which has as its value the first element of a list, return the list:
(CAR Hist name))

L2: In order to write a statement which assigns a value to a variable, return the list:
(SETQ [variable name) [value))

13: In order to write a statement which increments a pointer, return the list:
(SETQ [pointer name) (CDR [pointer name)))

L4: In order to write a statement which tests whether a pointer points to the end of a list, return the list:
(NULL (pointer name])

15: In order to write a statement which tests whether a variable is equal to a string constant, return the list:
(EQUAL [variable name) [string constant))

L6: In order to write a statement which destructively adds an element onto the front of a list, return the list:
(SETQ [list name)
(CONS (element name] [list name]))

17: [n order to write a statement which destructively inserts an element between two pointers, return the list:
(RFLACP ["before" pointer name)
(CONS [element name] ("after" pointer name]))

18: In order to write a statement which assigns the empty list to a variable, return the list:
(SFTQ [variable name] NIL)

L9: In order to combine two tests into an ordered disjunction, return the list:
(OR [first test) Isecond test])

LIO: In order to apply a function to a list of arguments, return the list:
([function name) ... arguments ...)

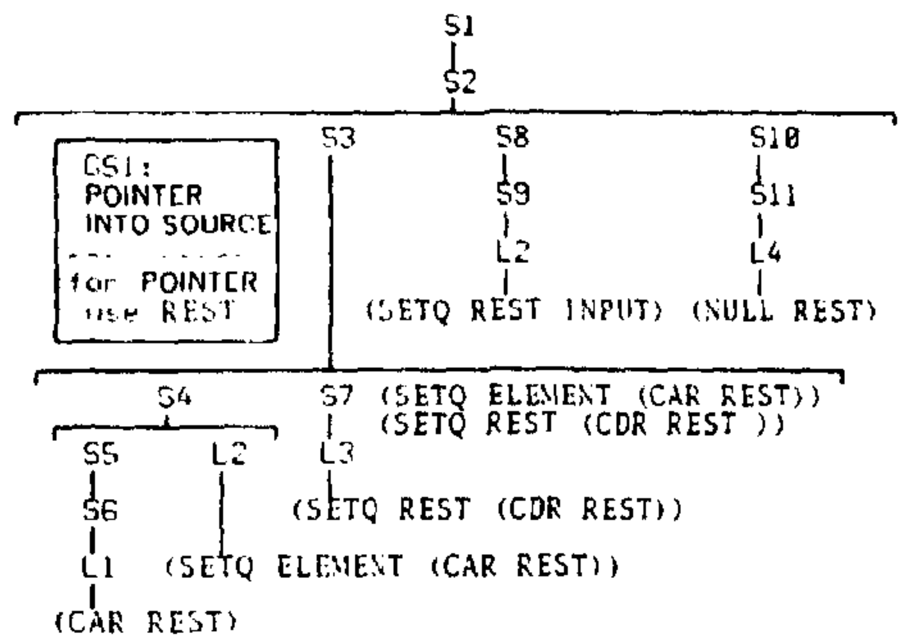
AN EXAMPLE

Wo now present an example of the use of the rules We will look in detail at steps I and 2 of the overview presented earlier

Let INPUT be the name of the source list and let OUTPUT be the name of the target list. Let ELEMENT be the variable name chosen to do the communicating between the selector and constructor. We will suppose that the final order is BASED ON COMPARISON PREDICATE, with GREATERP as the comparison predicate We assume that the system has already decided to write a transfer program. That is, rule TI has been invoked

Step 1 involved the choice of an insertion sort rather than a selection sort, if we consider what the possible legitimate combinations of selection re-ordenng and construction re-ordering are, we note that using SAME for the selection te-ordering implies that the constructor must do the bulk of the work (i.e., an insertion sort). In fact, the construction re-ordering must then be the same as the final order: BASED ON COMPARISON PREDICATE.

Step 2 involved writing the selector for the transfer program That is, rule SI is invoked by TI This will result in each of the other selector rules being invoked at the appropriate times. The accompanying chart gives a structural diagram of the various rule invocations.



The final result of invoking rule S1 consists of the following three sections of code:

```
(SETQ ELEMENT (CAR REST))    ! selector body
(SETQ REST (CDR REST))
(SETQ REST INPUT)            ! selector initializer
(NULL REST)                  ! selector pre-test
```

In the interests of brevity, we do not go into any further detail for the other steps of the overview, except to note that steps 3 through 6 are essentially the results of using the constructor rule system, with 4 being an invocation of the position finder rules. After all of the rules have been invoked, the parts of the generate and process paradigm have been filled in. The result is the following program:

```
(PROG NIL
   (SETQ REST INPUT)          | selector initializer
   (SETQ OUTPUT NIL)          | constructor initializer
L3 (COND                      | selector pre-test
    ((NULL REST) (GO L4)))
   (SETQ ELEMENT (CAR REST))| selector body
   (SETQ REST (CDR REST))
   (SETQ BEFORE "FRONT")      | constructor body
   (SETQ AFTER OUTPUT)
L1 (COND
    ((OR (NULL AFTER)
         (GREATERP ELEMENT (CAR AFTER)))
      (GO L2)))
   (SETQ BEFORE AFTER)
   (SETQ AFTER (CDR AFTER))
   (GO L1)
L2 (COND
    ((EQUAL BEFORE "FRONT")
      (SETQ OUTPUT (CONS ELEMENT OUTPUT)))
    (T (RPLACD BEFORE (CONS ELEMENT AFTER))))
   (GO L3)
L4 (RETURN NIL))
```

Note that REST, BEFORE and AFTER were chosen as new variable names; nothing in our rules deals with the selection of "reasonable" variable names. Our rules also have no knowledge about the scope of variables.

AN IMPLEMENTATION

The rules given in this paper are an abstraction of some of the rules now operational in a system being developed. We have endeavored to separate out those aspects of the rules which are implementation independent from those aspects which are the results of idiosyncrasies in the implementation Thus, the rules presented here are not precisely in correspondence to our implementation We include here a short discussion of our implementation for those who may be curious about it.

The basic mode of operation is to invoke rules by name. Each rule is responsible for determining which rules to invoke to accomplish its sub-goals. The user may be queried for a preference when alternative sub-goals are possible. The invocation by name has two major implications for the system. First, addition of knowledge is not strictly incremental, although the addition of "systems" of rules is generally fairly simple. Second, the system has no sophisticated abilities above and beyond those stemming from the knowledge represented by the rules. In particular, the system has no inference capabilities apart from those implicit in the rules. As the system is primarily a rule-testing device, rather than a sophisticated user-oriented system, these defects do not seem critical.

REMARKS

Having looked at these rules, it is natural to wonder what they can do for us. In regard to their "understanding" ability, it seenic that they can quite adequately *r>plain* the program that they produce One need merely look at the rule path which produced any part of the final program to understand the telationship of that part to the entire program. On the other hand, the rules are clearly unable to *analyre* or *modrfy* the program. It seems plausible that they could be extended to do some rudimentary analysis through 'jorne kind of parsing technique. Since the rules presented here are incomplete, in the sense that they do not present various alternatives, they cannot be expected to allow modification When they have been extended to a broader class of programs, different specifications will yield different progi arns, and modification would seem to require only slight extensions This is, however, mere speculation; further experimentation is required before anything definite can be determined

One of our original intentions was to determine a set of rules with a wide applicability We feel that we have done this. We are now in the process of extending these rules into a larger set, capable of synthesizing an entire class of sort programs If we are successful, then we will have an indication that these rules do indeed have some generality. But more than that, we expect the rules to be useful in many other programming tasks. Set operations (such as intci section and union) seem to involve many of the concepts with which our rules already deal (e.g., generating elements from an ordered set) Another area of potential usefulness seems to be in various kinds of searching and table look-up operations. Obviously the entire set of rules will not be applicable in any one situation, but it seems that for each situation some subset of these rules will be useful

In regard to such generality, it seems to us that some of the rules may be overly specific to the one particular target program, in the sense that such rules (or equivalent knowledge) could be derived or inferred from more general principles during synthesis of the program. The choice between these two modes of operation seems to be largely a computational issue: some kind of trade-off between space and time considerations.

At this point it is far too early to prefer one mode to any other, but having explicated the knowledge in some form, we may gauge the requirements of future knowledgo-based program-understanding systems. We have seen that about 50 "rules" are needed for a system to "understand" what is involved in writing even one simple sort pi ogram. If we are to design *knowledge-based pypgiam unlit-minuting systems,* with the abilities of synthesis, analysis, and modification, applicable to many different kinds of programs, then the body of programming knowledge which will be required is quite large. Our current work indicates that well over a hundred rules will be required to enable the synthesis of a large class of sort programs. It is still too early to estimate the size of the body of programming knowledge which will be necessary for the establishment of reasonable knowledge-based program-understanding systems, but we can at least see that the task of codifying this knowledge is likely to be a long one.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the diligence with which Avra Conn read the drafts of this paper, and the generosity with which the Artificial Intelligence Center of the Stanford Research Institute provided machine time for the research upon which this paper is based.

REFERENCES

[1] Green, Cordell, and Barstow, David. A Hypothetical Dialogue Exhibiting a Knowledge Base for a Program-Understanding System, presented at the NATO Advanced Study Institute on Machine Representations of Knowledge, Santa Cruz, California, July 1975.