INFORMATION PROCESSING 77, B. GILCHRIST, EDITOR

THE HIERARCHICAL SYNTHESIS OF LISP SCANNING PROGRAMS*

ALAN W. BIERMANN and DOUGLAS R. SMITH
Department of Computer Science, Duke University
Durham, North Carolina

Enumerative techniques for program synthesis have the desirable property that they can reliably find the minimum program which satisfies any given decidable requirements. They have the disadvantage, however, that the synthesis time is usually exponential on the size of the target program. A method for attempting to retain the advantages of the approach while scaling the synthesis time down to manageable proportions is to hierarchically decompose the problem into parts which can be individually solved enumeratively. This paper shows how this method may be used to generate at low cost LISP "scanning" programs of reasonable size and complexity from examples of their input-output behavior.

1. INTRODUCTION

An attractive method from a theoretical viewpoint for automatically sythesizing a computer program is to enumerate from the set of all candidate programs until an acceptable one is found. (We assume acceptability is a decidable property.) If the enumeration scans the whole set of possible programs in order of their size, this technique will be guaranteed to find the minimum program which satisfies the given requirements. The major barrier to practically using this scheme is the high cost of the enumeration which can become astronomical even for problems of moderate size. For example, in [2], we describe a system which uses this approach to synthesize "regular LISP programs" from examples of their input-output behavior. This system generates programs of three lines or less in a fraction of a second, programs of length four or five in a few seconds, and programs of length six or more in minutes or even hours. Clearly the growth of synthesis time is exponential as a function of the target program size which leads to a discouraging appraisal of the technique for problems of reasonable size.

However, if the synthesis task can be effectively decomposed into a hierarchy of tasks, the exponential growth rate can be reduced to linear growth making possible the generation of large programs at reasonable cost. That is, if a program is known to be decomposable into routines of size n_1 and n_2, the cost of generating the complete program would be approximately proportional to $C^{n_1+n_2}$ using an enumeration scheme but would be proportional to $C^{n_1}+C^{n_2}$ if the routines could be generated separately. This paper shows how this effect may be used to generate large programs at reasonable cost if task modularization can be accomplished. In order to achieve easy task decomposability, this paper will consider the "LISP scanning programs" which produce an output list by linearly (and repeatedly) scanning an input list in the process of generating that output. The results of each scan are clearly discernible in the output of such programs so that the subroutine which produced any individual scan can be separately generated.

The synthesis technique thus begins with one or several examples of the desired input-output behavior. It decomposes the desired output into segments which can be individually generated by small routines and then uses an enumerative procedure to create those routines. The problem has been transformed into a new problem, synthesis from input-output behavior where the output is a sequence of calls to the newly created small routines. Next this sequence of calls is broken down into segments which can be individually generated by small routines and those routines are generated. The process thus proceeds hierarchically until the top level routine is generated and the task is complete.

This approach has the advantage that the very powerful enumerative procedure is still used for program generation so that no restriction is placed on the type of program which can be produced. Yet the hierarchical break down of the synthesis task speeds up the process tremendously. The central problem is the creation of the task decomposition mechanism. In this paper, we restrict out study to the LISP scanning programs where task decomposition can be easily accomplished.

The reader interested in other approaches to automatic programming might wish to examine [1]. The LISP synthesis problem has been studied by Green et al, [4], Hardy, [5], Shaw et al, [6], Siklóssy and Sykes, [7], and Summers, [8,9]. Our own work differs from most of the other LISP efforts in that we tend to use uniform methods and avoid heuristic techniques. Thus we attempt to produce a system which is absolutely reliable over a definable class of problems even if that class is small.

2. THE CREATION OF A LISP SCANNING PROGRAM

Suppose it is desired to generate a program which yields an output (A C E G I B E H C E G I D G J E G I F I G I H I J) from an input of (A B C D E F G H I J). The complexity of the target program is beyond the range of our enumerative synthesis system (assuming realistic time constraints) and so a hierarchical synthesis must be attempted. If one graphs the desired output versus the input, the simplicity of the task decomposition problem for LISP scanning functions becomes apparent. The output, in fact, divides naturally into eight sequential segments each one of which results from a scan of the input: (A C E G I), (B E H), (C E G I), (D G J), (E G I), (F I), (G I), and (H I J). The synthesis method involves finding for each segment a routine which can generate that segment. Then a higher level routine will be created which can call all of the lower level routines in order. If necessary, many levels of hierarchy can be solved sequentially in the generation of a program.

If an input list has the form $(A_1\ A_2\ A_3\ \ldots A_k)$ we will use the notation A_i- to denote the portion of the list

*Supported by National Science Foundation Grant No. DCR 74-14445

$(A_i\ A_{i+1}\ \ldots A_k)$. Thus in the current example F- means (F G H I J). The first step of the synthesis is to generate the lowest level routine R_0 which actually places each atom on the output list. We define $(R_0\ X_0)$= (CONS (CAR X_0) $EXIT_0$) where $EXIT_0$ is a quantity to be later determined.

The next step is to create routines to generate the eight segments given using R_0 as a lowest level routine. The first such segment is (A C E G I) and a routine R_1 will do this task by performing the following calls: $(R_0$ A-), $(R_0$ C-), $(R_0$ E-), $(R_0$ G-) and $(R_0$ I-). The enumerative synthesizer, in fact, generates the following code for R_1:

$(R_1\ X) = (F_1\ X)$

$(F_1\ X_1) = (F_0\ X_1\ X_1)$

$(F_0\ X_0\ X_1) = (CONS\ (CAR\ X_0)\ (F_1'\ X_1))$

$(F_1'\ X_1) = (COND\ ((ATOM\ (CDDR\ X_1))\ EXIT_1)$

$\qquad\qquad (T\quad (F_1\ (CDDR\ X_1))))$

R_0 is implemented in this code by the function F_0 and R_1 is implemented by F_1 and F_1'. This code contains a single loop which calls R_0 repeatedly as the argument X is broken down sequentially by the CDDR function. The loop is exited when the predicate (ATOM (CDDR X_1)) yields true; the quantity $EXIT_1$ will be determined later.

Next the second segment of the output must be accounted for and R_1 is run on this segment to see if it is satisfactory. It fails, so a new routine R_2 is generated.

$(R_2\ X) = (F_2\ X)$

$(F_2\ X_2) = (F_0\ X_2\ X_2)$

$(F_0\ X_0\ X_2) = (CONS\ (CAR\ X_0)\ (F_2'\ X_2))$

$(F_2'\ X_2) = (COND\ ((ATOM\ (CDDDR\ X_2))\ EXIT_2)$

$\qquad\qquad (T\quad (F_2\ (CDDDR\ X_2))\ ))$

Next the third segment is examined and it is found that R_1 can execute it. R_1 can also execute the fifth and seventh segments, and the fourth and sixth segments can be executed by R_2 if its exit condition is modified to be (ATOM (CDR X_2)) OR (ATOM (CDDR X_2)) OR (ATOM (CDDDR X_2)). All of these individual routines are well within the capabilities of enumerative synthesis systems, [3]. Finally R_3 is generated to account for the eighth segment.

$(R_3\ X) = (F_3\ X)$

$(F_3\ X_3) = (F_0\ X_3\ X_3)$

$(F_0\ X_0\ X_3) = (CONS\ (CAR\ X_0)\ (F_3'\ X_3))$

$(F_3'\ X_3) = (COND\ (\ (ATOM\ (CDR\ X_3))\ \ EXIT_3)$

$\qquad\qquad (T\quad (F_3\ (CDR\ X_3))\ ))$

This leaves the problem to be solved hierarchically decomposed into the following problem: For the given input (A B C D E F G H I J), construct a program which executes the following sequence of calls: $(R_1$ A-), $(R_2$ B-), $(R_1$ C-) $(R_2$ D-), $(R_1$ E-), $(R_2$ F-), $(R_1$ G-), and $(R_3$ H-). This again is a simple construction and the resultant routine R_4 has this form:

$(R_4\ X) = (F_4\ X)$

$(F_4\ X_4) =$ call $(R_1\ X_4)$ exiting to $(F_4'\ X_4)$

$(F'_4\ X_4) = (COND\ ((ATOM(CDDDDR\ X_4))$

$\qquad\qquad (F_4^{IV}\ (CDR\ X_4)))$

$\qquad\qquad (T\quad (F_4''\ (CDR\ X_4)))\ ))$

$(F_4''\ X_4) =$ call $(R_2\ X_4)$ exiting to $(F_4'''\ X_4)$

$(F_4'''\ X_4) = (F_4\ (CDR\ X_4))$

$(F_4^{IV}\ X_4) =$ call $(R_3\ X_4)$ exiting to $(F_4^{V}\ X_4)$

$(F_4^{V}\ X_4) = NIL$

This code has one loop which calls R_1, R_2, R_1, R_2 repeatedly until the loop is exited after a call to R_1. Then R_3 is executed and the last evaluation F_4^V returns NIL which is the atom used to terminate the output list.

The composite of all of these routines is the solution to the original problem and is given below. The internal variable handling follows these rules:

(1) Each routine R_i modifies and tests only its own variable X_i.

(2) Each function F in routine R_i has as arguments its own variable X_i and variables X_k of routines hierarchically above it.

(3) When a routine R_i calls a routine R_j, the assignment $X_j \leftarrow X_i$ is made.

Following these rules we obtain the code for the desired program P.

$(P\ X) = (F_4\ X)$

$(F_4\ X_4) = (F_1\ X_4\ X_4)$

$(F_1\ X_1\ X_4) = (F_0\ X_1\ X_1\ X_4)$

$(F_0\ X_0\ X_1\ X_4) = (CONS\ (CAR\ X_0)\ (F_1'\ X_1\ X_4))$

$(F_1'\ X_1\ X_4) = (COND$

$\qquad ((ATOM\ (CDDR\ X_1))\qquad (F_4'\ X_4)\ \ )$

$\qquad (T\quad (F_1\quad (CDDR\ X_1)\quad X_4)\qquad))$

$(F_4'\ X_4) = (COND$

$\qquad (\ (ATOM\ (CDDDDR\ X_4))\ (F_4^{IV}\quad (CDR\ X_4)))$

$\qquad (T\quad (F_4''\ \ (CDR\ \ X_4))\qquad))$

$(F_4''\ X_4) = (F_2\ X_4\ X_4)$

$(F_2\ X_2\ X_4) = (F_0'\ X_2\ X_2\ X_4)$

$(F_0'\ X_0\ X_2\ X_4) = (CONS\ (CAR\ X_0)\ (F_2'\ X_2\ X_4))$

$(F_2'\ X_2\ X_4) = (COND$

$\qquad ((ATOM\ (CDR\ X_2))\qquad (F_4'''\ \ X_4)\qquad)$

$\qquad ((ATOM\ (CDDR\ X_2))\quad (F_4'''\ X_4)\qquad)$

$\qquad ((ATOM\ (CDDDR\ X_2))\quad (F_4'''\ X_4)\qquad)$

$\qquad (T\qquad (F_2\ (CDDDR\ X_2)\ X_4)\qquad))$

$(F_4'''\ \ X_4) = (F_4\ (CDR\ X_4))$

$(F_4^{IV}\quad X_4) = (F_3\ X_4\ X_4)$

$(F_3\ X_3\ X_4) = (F_0''\ X_3\ X_3\ X_4)$

$(F_0''\ X_0\ X_3\ X_4) = (CONS\ (CAR\ X_0)\ (F_3'\ X_3\ X_4))$

$(F_3'\ X_3\ X_4) = (COND$

$\qquad ((ATOM\quad (CDR\ \ X_3))\qquad (F_4^{V}\ \ X_4)\quad)$

$\qquad (T\qquad (F_3\ (CDR\ X_3)\ X_4)\qquad))$

$(F_4^{V}\quad X_4) = NIL$

This code may be collapsed into more readable form but is left as primitive code so that its generation

can be more easily understood. The generation of programs of this complexity on our new prototype system requires less than one half second.

3. REVERSE SCANNING

If the output in the example solved in the previous section had been reversed to become (J I H I .... G E C A), the problem would have been no harder. The output segmentation problem can be performed as before but now routines must be generated which place the atoms on the output list in ascending order. This is done with "build-up" routines which perform their work by collecting the output on "build-up" variables. (Thus all variables and routines in the previous section become known as "teardown" variables and "teardown" routines.) Build-up routines are nearly identical to the tear-down routines except that:

(1) In a nesting of build-up routines there is (in addition to the usual tear-down variables) one build-up variable upon which the output is constructed.
(2) A build-up routine appends the results of its computation onto its build-up variable and returns that variable as its value upon exit.

Rule (2) is illustrated by the function

```
(F X Y) = (COND ( (ATOM X)    Y)
                (T   (F (CDR X)
                        (CONS (CAR X) Y))))
```

where on the second line F calls itself appending its result onto build-up variable Y and on the first line F eventually returns Y as its value.

(3) Any routine which is hierarchically one level below a build-up routine appends its result to the build-up variable, returns that variable as its value, and then places its result on the build-up variable in the higher routine.

Suppose as an illustration of (3) that $(F_1\ X_1\ Y)$ is part of a build-up routine which is defined in terms of $(F_2\ X_1\ X_2\ Y)$. Further suppose that the above defined routine F is hierarchically one level below F_1. Then F satisfies the first part of (3) because it appends its result to the build-up variable Y and returns Y as its result. Concerning the second part of (3), its result is left on the build-up variable of the higher level routine F_1 if F_1 is defined as $(F_1\ X_1\ Y) = (F_2\ X_1\ X_2\ (F\ X_1\ Y))$. The meaning of these rules will become clearer if the example is completed.

The lowest level routine R_0 may be created as before except that rule (3) for build-up variables applies. R_0 appends its result to the build-up variable Y as follows: $(R_0\ X_0\ Y) = (CONS\ (CAR\ X_0)\ Y)$. The value that R_0 returns is placed on Y by the build-up routine immediately above R_0. Such a routine is R_1 which is required to generate the segment (I G E C A). This routine is created as in the previous section except that the rules for build-up routines are followed:

$(R_1\ X\ Y) = (F_1\ X\ Y)$

$(F_1\ X_1\ Y) = (F_1'\ X_1\ (F_0\ X_1\ X_1\ Y))$
$(F_0\ X_0\ X_1\ Y) = (CONS\ (CAR\ X_0)\ Y)$
$(F_1'\ X_1\ Y) = (COND$
((ATOM (CDDR X_1)) Y)
(T (F_1 (CDDR X_1) Y)))

Routine R_1 is again a one loop program with control being passed back and forth between F_1 and F_1'. Each time around the loop, F_0 is called and its result is placed on build-up variable Y.

Routines R_2, R_3, and R_4 can be generated in a similar manner and assembled to yield the following code:

$(P\ X) = (F_4\ X\ NIL)$
$(F_4\ X_4\ Y) = (F_4'\ X_4\ (F_1\ X_4\ X_4\ Y))$
$(F_1\ X_1\ X_4\ Y) = (F_1'\ X_1\ X_4\ (F_0\ X_1\ X_1\ X_4\ Y))$
$(F_0\ X_0\ X_1\ X_4\ Y) = (CONS\ (CAR\ X_0)\ Y)$
$(F_1'\ X_1\ X_4\ Y) = (COND$
((ATOM (CDDR X_1)) Y)
(T (F_1 (CDDR X_1) X_4 Y)))
$(F_4'\ X_4\ Y) = (COND$
((ATOM (CDDDDR X_4))
(F_4^{IV} (CDR X_4) Y))
(T (F_4'' (CDR X_4) Y)))
$(F_4''\ X_4\ Y) = (F_4'''\ X_4\ (F_2\ X_4\ X_4\ Y))$
$(F_2\ X_2\ X_4\ Y) = (F_2'\ X_2\ X_4\ (F_0'\ X_2\ X_2\ X_4\ Y))$
$(F_0'\ X_0\ X_2\ X_4\ Y) = (CONS\ (CAR\ X_0)\ Y)$
$(F_2'\ X_2\ X_4\ Y) = (COND$
((ATOM (CDR X_2)) Y)
((ATOM (CDDR X_2)) Y)
((ATOM (CDDDR X_2)) Y)
(T (F_2 (CDDDR X_2) X_4 Y)))
$(F_4'''\ X_4\ Y) = (F_4\ (CDR\ X_4)\ Y)$
$(F_4^{IV}\ X_4\ Y) = (F_4^{V}\ X_4\ (F_3\ X_4\ X_4\ Y))$
$(F_3\ X_3\ X_4\ Y) = (F_3'\ X_3\ X_4\ (F_0''\ X_3\ X_3\ X_4\ Y))$
$(F_0''\ X_0\ X_3\ X_4\ Y) = (CONS\ (CAR\ X_0)\ Y)$
$(F_3'\ X_3\ X_4\ Y) = (COND$
((ATOM (CDR X_3)) Y)
(T (F_3 (CDR X_3) X_4 Y)))

$(F_4^{V}\ X_4\ Y) = Y$

4. COMPOSITIONS OF BUILD-UP AND TEAR-DOWN ROUTINES

If the outputs of the previous two sections were concatenated, the difficulty of the synthesis would still not increase greatly. That is, let W represent the desired output for the program of section 2, let W^r represent the desired output for the program of section 3, and let (W,W^r) represent the concatenation of the two lists. Then the current goal is to generate programs P_1 and P_2 which will respectively convert the input list (A B ... J) to (W,W^r) and (W^r,W). This is done by carrying out the steps of sections 2 and 3 as before generating the routines R_4 and R_4' and then producing a highest level routine R_5 to call them sequentially. The construction of R_5 must follow these concatenation rules:

1) The output of a tear-down routine is concatenated to the left of any list by appending the routine's result to that list as the routine is exited.

2) The output of a build-up routine is concatenated to the left of any list by

initializing its build-up variable with that list before the computation begins.

Then in the generation of P_1 which is to output (W,W^r), routine R_5 is to call R_4 (a tear-down routine) followed by R_4' (a build-up routine). Following rule 1) above, R_4 would be created as in section 2 except that the NIL at exit would be replaced by a call to R_4'. R_4' would be the routine generated in section 3 as given with its build-up variable initialized to NIL.

In the generation of P_2 which is to output (W^r,W), routine R_5 is to call R_4' (a build-up routine) followed by R_4 (a tear down routine). Following rule 2) above, R_4' would be constructed as in section 3 except that Y would not initialized with NIL as shown. Instead, it would be initialized with the result of routine R_4 as given in section 2.

We now have enough tools to tackle truly difficult problems. For example, if the input is (A B C D) and the output is (D D D C D D D D C D D D C B D C D D C D D D C D D D C B D C D D C D D D C B A D C B D C D D C B D C D D C D D D D D C D D D D C D D D D C B D C D D C D D D D D C D D D) for the desired program, the system must be able to properly discover and construct nested loops to a depth of six. But the program is constructed trivially using the method given above.

The lowest level routine R_0 will be called by a build-up routine and thus will have this form:

$(R_0\ X\ Y_1) = (F_0\ X\ Y_1)$
$(F_0\ X_0\ Y_1) = (\text{CONS}\ (\text{CAR}\ X_0)\ Y_1)$

This decomposes the original problem to generating the buildup routine R_1 which can make the following series of calls: $(R_0$ D-), $(R_0$ D-), $(R_0$ D-), $(R_0$ C-), $(R_0$ D-), ... R_1 is generated as follows:

$(R_1\ X\ Y) = (F_1\ X\ Y)$
$(F_1\ X_1\ Y_1) = (F_1'\ X_1\ (F_0\ X_1\ X_1\ Y_1))$
$(F_0\ X_0\ X_1\ Y_1) = (\text{CONS}\ (\text{CAR}\ X_0)\ Y_1)$
$(F_1'\ X_1\ Y_1) = (\text{COND}$
$\quad((\text{ATOM}\ (\text{CDR}\ X_1))\quad Y_1)$
$\quad(\text{T}\ \ (F_1\ (\text{CDR}\ X_1)\ \ Y_1)))$

This reduces the problem to generating the following (shorter) sequence of calls: $(R_1$ D-), $(R_1$ D-), $(R_1$ C-), $(R_1$ D-), ... Examining this sequence one can find subsequences of the form $(R_1$ A-), $(R_1$ B-), $(R_1$ C-), $(R_1$ D-) indicating that the next level of hierarchy requires a tear-down routine. So R_2 is generated which repeatedly calls R_1 as desired:

$(R_2\ X) = (F_2\ X)$
$(F_2\ X_2) = (F_1\ X_2\ X_2\ (F_2'\ X_2))$
$(F_1\ X_1\ X_2\ Y_1) = (F_1'\ X_1\ X_2\ (F_0\ X_1\ X_1\ X_2\ Y_1))$
$(F_0\ X_0\ X_1\ X_2\ Y_1) = (\text{CONS}\ (\text{CAR}\ X_0)\ Y_1)$
$(F_1'\ X_1\ X_2\ Y_1) = (\text{COND}$
$\quad((\text{ATOM}\ (\text{CDR}\ X_1))\quad Y_1)$
$\quad(\text{T}\ \ (F_1\ (\text{CDR}\ X_1)\ X_2\ Y_1)\ ))$
$(F_2'\ X_2) = (\text{COND}$
$\quad((\text{ATOM}\ (\text{CDR}\ X_2))\ \ \text{EXIT}_2)$
$\quad(\text{T}\ (F_2\ (\text{CDR}\ X_2))\quad\ ))$

As in the other cases, R_2 is a one loop program with control moving back and forth between F_2 and F_2'. Notice that in the definition of F_2, F_1's build-up variable Y_1 is initialized with the segment which is to follow F_1's result, namely $(F_2'\ X_2)$. This is in accordance with rule 2) for compositions.

This synthesis process may continue by generating R_3 (tear-down), R_4 (build-up), R_5 (build-up), and finally the highest level routine R_6 (tear-down). Following rule 1) for build-up routines, the highest level build-up routine in a nesting of such routines introduces a build-up variable which is used by all members of the nesting. Thus routines R_5 and R_1 introduce build-up variables Y_5 and Y_1 which are used in the two separate nestings of the build-up routines. The final program is:

$(P\ X) = (F_6\ X)$
$(F_6\ X_6) = (F_5\ X_6\ X_6\ (F_6'\ X_6))$
$(F_5\ X_5\ X_6\ Y_5) = (F_5'\ X_5\ X_6\ (F_4\ X_5\ X_5\ X_6\ Y_5))$
$(F_4\ X_4\ X_5\ X_6\ Y_5) = (F_4'\ X_4\ X_5\ X_6$
$\qquad(F_3\ X_4\ X_4\ X_5\ X_6\ Y_5))$
$(F_3\ X_3\ X_4\ X_5\ X_6\ Y_5) = (F_2\ X_3\ X_3\ X_4\ X_5\ X_6\ Y_5)$
$(F_2\ X_2\ X_3\ X_4\ X_5\ X_6\ Y_5) =$
$\quad(F_1\ X_2\ X_2\ X_3\ X_4\ X_5\ X_6$
$\qquad(F_2'\ X_2\ X_3\ X_4\ X_5\ X_6\ Y_5)\ Y_5)$
$(F_1\ X_1\ X_2\ X_3\ X_4\ X_5\ X_6\ Y_1\ Y_5) =$
$\quad(F_1'\ X_1\ X_2\ X_3\ X_4\ X_5\ X_6$
$\qquad(F_0\ X_1\ X_1\ X_2\ X_3\ X_4\ X_5\ X_6\ Y_1\ Y_5)\ Y_5)$
$(F_0\ X_0\ X_1\ X_2\ X_3\ X_4\ X_5\ X_6\ Y_1\ Y_5) =$
$\quad(\text{CONS}\ (\text{CAR}\ X_0)\ Y_1)$
$(F_1'\ X_1\ X_2\ X_3\ X_4\ X_5\ X_6\ Y_1\ Y_5) = (\text{COND}$
$\quad((\text{ATOM}\ (\text{CDR}\ X_1))\quad Y_1)$
$\quad(\text{T}\ (F_1\ (\text{CDR}\ X_1)\ X_2\ X_3\ X_4\ X_5\ X_6\ Y_1\ Y_5)\ ))$
$(F_2'\ X_2\ X_3\ X_4\ X_5\ X_6\ Y_5) = (\text{COND}$
$\quad((\text{ATOM}\ (\text{CDR}\ X_2))\ (F_3'\ X_3\ X_4\ X_5\ X_6\ Y_5))$
$\quad(\text{T}\ (F_2\ (\text{CDR}\ X_2)\ X_3\ X_4\ X_5\ X_6\ Y_5)))$
$(F_3'\ X_3\ X_4\ X_5\ X_6\ Y_5) = (\text{COND}$
$\quad((\text{ATOM}\ (\text{CDR}\ X_3))\quad Y_5)$
$\quad(\text{T}\ (F_3\ (\text{CDR}\ X_3)\ X_4\ X_5\ X_6\ Y_5)))$
$(F_4'\ X_4\ X_5\ X_6\ Y_5) = (\text{COND}$
$\quad((\text{ATOM}\ (\text{CDR}\ X_4))\quad Y_5)$
$\quad(\text{T}\ (F_4\ (\text{CDR}\ X_4)\ X_5\ X_6\ Y_5)))$
$(F_5'\ X_5\ X_6\ Y_5) = (\text{COND}$
$\quad((\text{ATOM}\ (\text{CDR}\ X_5))\quad Y_5)$
$\quad(\text{T}\ (F_5\ \ (\text{CDR}\ X_5)\ X_6\ Y_5))$
$(F_6'\ X_6) = (\text{COND}$
$\quad(\ (\text{ATOM}\ (\text{CDR}\ X_6))\quad \text{NIL})$
$\quad(\text{T}\ (F_6\ (\text{CDR}\ X_6))\ ))$

Our implementation of this synthesis procedure produces nicer code than what is shown here after a

total computation time of about one half second.

5. DISCUSSION

This paper shows how relatively large and complicated programs can be generated economically from examples of input-output behavior if the synthesis task is decomposable. The class of scanning programs that can be created using this technique could be defined precisely (as was done for the "regular" LISP programs in [2]) although their general nature should be clear from the discussions given. It is needless to say that all of the programs appearing in this paper and most of the programs in the other literature on LISP synthesis [2,4,5,6,7,8,9] are scanning programs.

The LISP code which appears in this paper is more ugly and primitive than it need be but is left in that form so that the method of generation is clear. We are currently implementing a system that generates more readable code and will report on it at a later time.

REFERENCES

[1] A. W. Biermann, Approaches to automatic programming, in Advances in Computers, vol. 15, (Eds. M. Rubinoff and M. Yovits), Academic Press, 1976, 1-63.

[2] A. W. Biermann, Regular LISP programs and their automatic synthesis from examples, Report CS-1976-12, Computer Science Department, Duke University, Durham, North Carolina, 1976.

[3] A. W. Biermann, R. I. Baum, and F. E. Petry, Speeding up the synthesis of programs from traces, IEEE Transactions on Computers, vol. C-24, no. 2, 1975, 122-136.

[4] C. C. Green, R. J. Waldinger, D. R. Barstow, R. Elschlager, D. B. Lenat, B. P. McCune, D. E. Shaw, and L. I. Steinberg, Progress report on program understanding systems, Memo AIM-240, Stanford Artificial Intelligence Laboratory, Stanford, California, 1974.

[5] S. Hardy, Synthesis of LISP functions from examples, Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, Sept. 1975, 240-245.

[6] D. Shaw, W. Swartout, C. Green, Inferring LISP programs from examples, Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, Sept. 1975, 260-267.

[7] L. Siklóssy and D. A. Sykes, Automatic program synthesis from example problems, Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, Sept. 1975, 268-273.

[8] P. D. Summers, Program construction from examples, Doctoral Dissertation, Yale University, New Haven, Conn., 1975.

[9] P. D. Summers, A methodology for LISP program construction from examples, Proceedings of the Third ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, Jan. 1976, 68-76. Also in Journal of the ACM, vol. 24, no. 1, 1977, 161-175.