The Inference of Regular LISP Programs from Examples

ALAN W. BIERMANN

Abstract—**A class of LISP programs that is analogous to the finite-state automata is defined, and an algorithm is given for constructing such programs from examples of their input–output behavior. It is shown that the algorithm has robust performance for a wide variety of inputs and that it converges to a solution on the basis of minimum input information.**

I. Introduction

THE DREAM of many a computer programmer has been a system that would accept a few randomly chosen examples of the desired behavior and that would immediately print out a general program for achieving the desired behavior in all possible situations. Thus the system user might type in the input–output pairs (6, 13), (2, 3), (7, 17), and (1, 2) and expect to have the system type out a program that reads an integer i (such as 6) and then prints the ith prime number (13 in this case). The system has the incredibly difficult task of discovering from this weak source of information

1) an algorithm for doing the desired computation, and
2) a correct implementation of the algorithm in some programming language.

It is, in fact, doubtful that any system could ever exist which could create a class of large and general programs from such minimal information within a practical amount of time.

There are, however, some very limited domains in which such performance is achievable, and one of those domains, the class of small "regular" LISP programs, is the subject of this paper. The synthesis of LISP programs from input–output behaviors is at least minimally tractable because these behaviors are given in terms of LISP S-expressions, and such expressions contain considerable structural information about how they were formed. For example, if a program is to be constructed such that input $X = ((A \cdot B) \cdot C)$ is to yield its reversal $Y = (C \cdot (B \cdot A))$, we know that Y can be computed from X using the primitive LISP functions CONS, CAR, and CDR (described below) as follows:

$$Y = (\text{CONS}(\text{CDR } X)(\text{CONS}(\text{CDR}(\text{CAR } X))(\text{CAR}(\text{CAR } X)))).$$

Manuscript received March 24, 1977; revised March 14, 1978. This work was supported in part by the National Science Foundation under Grant DCR 74-14445 and in part by the Health Resources Administration under Grant HS 01613.

The author is with the Department of Computer Science, Duke University, Durham, NC 27706.

This ability to directly decompose Y in terms of X is a huge step in the direction of creating a general program for reversing any S-expression and is the key reason that at least some LISP programs can be generated from just input–output information.

Having adjusted to the fact that we will probably never be able to generate very large and general classes of programs from such weak information, it is interesting to see how good a synthesis system can be built for a limited class. It is specifically desired that the synthesis system have the following properties (described below):

1) robust behavior for a variety of inputs,
2) convergence to a solution on the basis of minimum input information,
3) convergence to a known class of programs.

Concerning 1), it is desired that the program that is generated be relatively independent of whichever examples are presented. Thus it is desired that the program for reversing an S-expression should be generated from any one of the following input–output pairs or from any similar input–output pairs which might be presented by anyone.

Input	Output
$(A \cdot B)$	$(B \cdot A)$
$(A \cdot (B \cdot C))$	$((C \cdot B) \cdot A)$
$((G \cdot F) \cdot (H \cdot A))$	$((A \cdot H) \cdot (F \cdot G))$
$(D \cdot R)$	$(R \cdot D)$
$(((A \cdot B) \cdot (R \cdot Z)) \cdot X)$	$(X \cdot ((Z \cdot R) \cdot (B \cdot A)))$
$(A \cdot (B \cdot (R \cdot (M \cdot (V \cdot Q)))))$	$(((((Q \cdot V) \cdot M) \cdot R) \cdot B) \cdot A)$

Furthermore, the system should correctly generate the program if any subset of these examples is used, if all of them are presented, or if any similar set is chosen. The system should be similarly robust on every program within its range.

Concerning 2), suppose a user inputs a behavior example and is dissatisfied with the program that is generated. That is, he has found the created program to be incorrect either by testing it on some other examples or studying its code. Then he should be able to input a second example, a third example, and so forth, with the knowledge that the system will find the desired solution after only a finite number of examples. In fact, we would like our system to be such that not only will it find the solution, but it will have the property

0018-9472/78/0800-0585$00.75

that no other system could generate all programs on the basis of fewer examples than our current system. That is, we want our system to, in some sense, use input information optimally.

Concerning 3), we are interested in being able to predict exactly when a particular target program can be generated by the system and when it cannot. For our system the regular LISP programs are exactly the class of programs that can be converged to in the sense of 2). Membership in the class of regular LISP programs is decidable, and a method is given in Section V for showing that certain LISP computations cannot be done by any regular LISP program. Furthermore, the length of time required to generate a program grows exponentially with the size of the program, and because of this, our system is not able to generate programs that are much more then about five or six lines in length without using many minutes or hours of computation time. In summary, we can assure our user that the system will converge to a program to do the desired computation after a finite number of examples if there is a regular LISP program to do that computation. It will converge to the target program within a reasonable length of time if it has no more than about five or six lines of code.

These three properties are achieved by the system described in the paper by employing an enumerative synthesis algorithm. The search, of course, is highly pruned through the use of the structural information from the example input–output behaviors. General inference techniques using enumeration are described in [6]–[8], and the current work is built upon these results. Two critical theorems are restated and reproven in terms of the terminology of this paper in Section VI.

LISP program synthesis has been studied previously by Green *et al.* [9], Hardy [10], Shaw *et al.* [12], Siklóssy and Sykes [13], and Summers [14], [15]. The approach of this paper is particularly influenced by Summers, who created "fragments" much like our "computations traces" and who did "differencing" which corresponds to our node mergers. However, all of the above synthesis systems were "heuristic" in the sense that the class of problems that they could solve was unknown. The method described here is "algorithmic" in that it is capable of generating absolutely any program or its equivalent in the class of regular LISP programs (although it obviously may take an unacceptable amount of time to generate large ones). Also, none of these other efforts can make all of the claims 1), 2), and 3) above.

The reader interested in a general survey of the literature on automatic programming might wish to examine [3].

II. LISP and an Overview of the Synthesis Method

The only data structure in the LISP language is the *S*-expression which may be defined to be any string of symbols that can be generated by the grammar ⟨*S*-expression⟩ → ⟨atom⟩ | (⟨*S*-expression⟩ · ⟨*S*-expression⟩), where ⟨atom⟩ may be either an identifier or NIL. NIL is a special reserved symbol, a LISP language constant. Any *S*-expression which is defined by the rule ⟨*S*-expression⟩ → ⟨atom⟩ will be called an *atom*. An example *S*-expression is

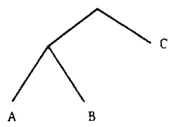

Fig. 1. Graphical representation of $((A \cdot B) \cdot C)$.

$X = ((A \cdot B) \cdot C)$, which can be represented graphically as the tree shown in Fig. 1.

A LISP convention that will be followed here is that a function f and its arguments $x_1, x_2, \cdots, x_n$ are often given in the list form $(f\ x_1\ x_2\ x_3 \cdots x_n)$. Using this notation, the definitions of several common LISP functions are

$$(\text{CAR } X) = \begin{cases} S_1, & \text{if } X = (S_1 \cdot S_2), \\ & \text{where } S_1 \text{ and } S_2 \text{ are } S\text{-expressions} \\ \text{undefined}, & \text{if } X \text{ is an atom} \end{cases}$$

$$(\text{CDR } X) = \begin{cases} S_2, & \text{if } X = (S_1 \cdot S_2), \\ & \text{where } S_1 \text{ and } S_2 \text{ are } S\text{-expressions} \\ \text{undefined}, & \text{if } X \text{ is an atom} \end{cases}$$

$$(\text{CONS } S_1 S_2) = (S_1 \cdot S_2), \quad \text{where } S_1 \text{ and } S_2 \text{ are } S\text{-expressions.}$$

We will also be using one predicate which can yield a value true T or false NIL:

$$(\text{ATOM } X) = \begin{cases} \text{T}, & \text{if } X \text{ is an atom} \\ \text{NIL}, & \text{otherwise.} \end{cases}$$

If $X = ((A \cdot B) \cdot C)$, then some example evaluations of these functions are

$$(\text{CAR } X) = (A \cdot B)$$

$$(\text{CDR } X) = C$$

$$(\text{CONS } XX) = (((A \cdot B) \cdot C) \cdot ((A \cdot B) \cdot C))$$

$$(\text{ATOM } X) = \text{NIL}.$$

The program synthesis method will function by assembling a program from a set of primitive functions of the form $(F_i X) =$ (LISP code), where (LISP code) could be theoretically almost any LISP program. In this paper only five different forms have been considered:

$$(F_i X) = \text{NIL}$$

$$(F_i X) = X$$

$$(F_i X) = (F_j(\text{CAR } X))$$

$$(F_i X) = (F_j(\text{CDR } X))$$

$$(F_i X) = (\text{CONS}(F_j X)(F_k X)).$$

Other primitive functions could have been used, such as $(F_i X) = (\text{CAR}(F_j X))$ or $(F_i X) = (\text{CONS}(\text{CAR}(\text{CDR } X))(F_j X))$, but modifications to the original set did not seem to result in a more interesting class of synthesized functions.

The program synthesis method will proceed approximately as follows. If it is desired to generate a LISP program which converts, for example, $X = ((A \cdot B) \cdot C)$ to $Y = (C \cdot (B \cdot A))$, the first step is to write Y in terms of X as was given in the previous section. Decomposing this expres-

sion into the primitive functions given above results in $Y = (F_1\, X)$, where F_1 is the top-level function in the decomposition and where F_1 and the lower level functions are defined as follows:

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

where

$$(F_1\, X) = (\text{CONS}(F_2\, X)(F_3\, X))$$
$$(F_2\, X) = (F_4(\text{CDR}\ X))$$
$$(F_3\, X) = (F_5(\text{CAR}\ X))$$
$$(F_4\, X) = X$$
$$(F_5\, X) = (\text{CONS}(F_6\, X)(F_7\, X))$$
$$(F_6\, X) = (F_8(\text{CDR}\ X))$$
$$(F_7\, X) = (F_9(\text{CAR}\ X))$$
$$(F_8\, X) = X$$
$$(F_9\, X) = X.$$

The synthesis procedure then searches for a way to merge these nine functions to obtain a program of minimum size. The only tool used is the insertion of McCarthy conditionals [11] with their associated predicate tests. In this example the functions are partitioned as $\{F_1, F_4, F_5, F_8, F_9\}$, $\{F_2, F_6\}$, and $\{F_3, F_7\}$ to obtain the program

$$(F_1\, X) = (\text{COND}((\text{ATOM}\ X)X)$$
$$(\text{T}(\text{CONS}(F_2\, X)(F_3\, X))))$$
$$(F_2\, X) = (F_1(\text{CDR}\ X))$$
$$(F_3\, X) = (F_1(\text{CAR}\ X)).$$

The COND function is defined in Section IV and means in this case, "If X is an atom yield X. Otherwise yield $(\text{CONS}(F_2\, X)(F_3\, X))$."

This whole process may be visualized graphically as shown in Fig. 2, where the composition of Y in terms of X with all intermediately defined functions is shown at the left and the synthesized program after merger is shown at right. The resemblance to finite-state automata is clear, giving rise to the name of the class of programs to be studied here: the "regular" LISP programs. We will, in fact, often represent LISP programs in this graphic fashion and refer to the parts of a program using graphic terminology. For example, we will say that F_1 has two transitions leading away, $((\text{ATOM}\ X)X)$ and $(\text{T}(\text{CONS}(F_2\, X)(F_3\, X)))$. The only significant deviation from the usual graph notions is that the transitions leading away from a program node will have an order, namely the order indicated in the McCarthy conditional representation.

Section III will study the problem of finding a unique decomposition of Y into primitive functions. Section IV will give a definition of regular programs, Section V will discuss one of their properties, and Section VI will give the synthesis method. The final two sections will give examples of system behavior and a discussion of results.

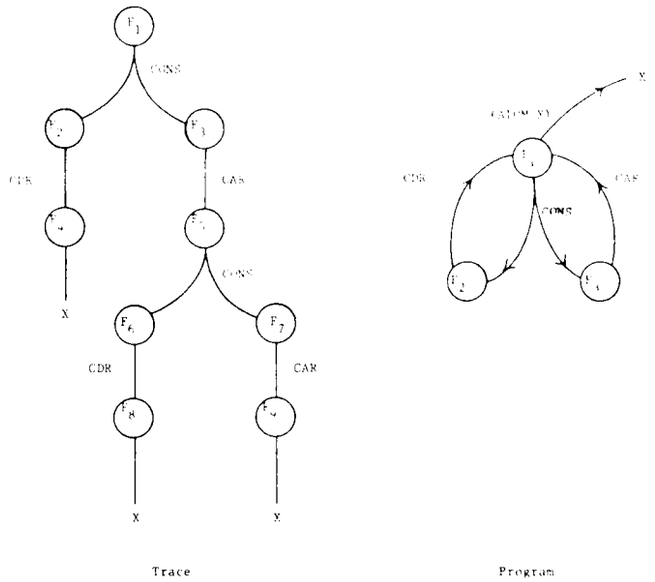

Fig. 2. Computation of $Y = (C \cdot (B \cdot A))$ from $X = ((A \cdot B) \cdot C)$ in terms of intermediate functions and synthesized program after merger.

III. The Generation of Computation Traces from Examples

We will be interested in finding a unique sequence of LISP operations for converting an input S-expression X into some output S-expression Y. In order to assure uniqueness, we will first require that all atoms in X be non-NIL and distinct. This means that it will be known that whenever an atom appears in Y it will have come from a uniquely defined position in X. Therefore, $((A \cdot B) \cdot C)$ will be an allowed argument X while $((A \cdot B) \cdot A)$ will not be. The philosophy is that the user of the automatic program synthesizer should want to express through his examples the desired computation in as direct a manner as possible and should not want to introduce unnecessary ambiguities. This assumption about argument X will be maintained throughout this paper.

The *composition* $c(X, Y)$ of Y in terms of X can thus be uniquely defined as

$$c(X, Y) = \begin{cases} \text{NIL}, & \text{if } Y \text{ is NIL} \\ Y \text{ written in terms of } X, & \\ \qquad \text{if } Y \text{ is equal to some atom of } X & \\ (\text{CONS}\ c(X, (\text{CAR}\ Y))c(X, (\text{CDR}\ Y))), & \text{otherwise.} \end{cases}$$

Thus for the example X and Y from Section II,

$$c(X, Y) = (\text{CONS}\ c(X, C)c(X, (B \cdot A)))$$
$$= (\text{CONS}(\text{CDR}\ X)(\text{CONS}\ c(X, B)c(X, A)))$$
$$= (\text{CONS}(\text{CDR}\ X)(\text{CONS}(\text{CDR}(\text{CAR}\ X))$$
$$(\text{CAR}(\text{CAR}\ X)))).$$

While $c(X, Y)$ specifies which operations must be applied to X to obtain Y, the *computation trace* $t(X, Y)$ will specify the exact order in which those operations are to be applied. That is, the composition $c(X, Y)$ does not uniquely define the sequence of computations in terms of the primitive functions. This can be verified by noting that the example Y given

above could also be computed by the following set of primitives:

$$(F_1 X) = (\text{CONS}(F_2 X)(F_3 X))$$
$$(F_2 X) = (F_4(\text{CDR } X))$$
$$(F_3 X) = (\text{CONS}(F_5 X)(F_6 X))$$
$$(F_4 X) = X$$
$$(F_5 X) = (F_7(\text{CAR } X))$$
$$(F_6 X) = (F_8(\text{CAR } X))$$
$$(F_7 X) = (F_9(\text{CDR } X))$$
$$(F_8 X) = (F_{10}(\text{CAR } X))$$
$$(F_9 X) = X$$
$$(F_{10} X) = X.$$

There may be many possible orderings for the primitive operations, and the efficiency of the program synthesis technique depends on discovering the correct one directly and without search. $t(X, Y)$ succeeds in doing this by giving CAR and CDR a higher precedence than CONS. Thus if there is ever a choice between applying a CAR or CDR operation to X or applying CONS to build output Y, the CAR or CDR operation is always executed first. Therefore, in the example X and Y given, $(F_3 X)$ could have been defined as $(F_5(\text{CAR } X))$, as shown in Section II, or $(\text{CONS}(F_5 X)(F_6 X))$, as shown here. $t(X, Y)$ resolves this ambiguity by choosing the former and computing (CAR X) first.

The idea of precedence for CAR and CDR can also be thought of in another way. If not all of input X is used in computing output Y, then that portion of X which is not used is thrown away immediately before any CONS operations are applied. In the above example, no portion of (CDR X) appears in the result of $(F_3 X)$; so (CDR X) is thrown away immediately, and (CAR X) is computed and passed on to the next stage of computation. This method of computation is called *direct*, as described in the next section.

The computation trace $t(X, Y)$ for computing Y from X is defined as

$$t(X, Y) = \begin{cases} (\bar{N}), & \text{if } c(X, Y) = \text{NIL} \\ (\bar{I}), & \text{if } c(X, Y) = X \\ (\bar{A}\ t((\text{CAR } X), Y)), & \text{if (CAR } X\text{) appears in } c(X, Y) \text{ and no (CDR } X\text{) or } X \text{ as an argument of CONS or } X \text{ alone appears in } c(X, Y) \\ (\bar{D}\ t((\text{CDR } X), Y)), & \text{if (CDR } X\text{) appears in } c(X, Y) \text{ and no (CAR } X\text{) or } X \text{ as an argument of CONS or } X \text{ alone appears in } c(X, Y) \\ (\bar{O}\ t(X, (\text{CAR } Y))\ t(X, (\text{CDR } Y))), & \text{otherwise.} \end{cases}$$

Intuitively, the codes $\bar{N}$, $\bar{I}$, $\bar{A}$, $\bar{D}$, and $\bar{O}$ represent the application of the operators nil, identity, CAR, CDR, and CONS, respectively. The nil operator always yields NIL as its result regardless of the input. The identity operator yields an output which is equal to its input. This notation ($\bar{N}$, $\bar{I}$, $\bar{A}$, $\bar{D}$, $\bar{O}$) is used instead of (NIL, identity, CAR, CDR, CONS) to prevent confusion between the composition $c(X, Y)$ and the trace $t(X, Y)$.

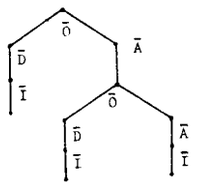

Fig. 3. Computation trace $t(X, Y)$ for example X and Y.

Computing $t(X, Y)$ for the example X and Y yields

$$\begin{aligned} t(X, Y) &= (\bar{O}\ t(X, C)\ t(X, (B \cdot A))) \\ &= (\bar{O}(\bar{D}\ t(C, C))(\bar{A}\ t((A \cdot B), (B \cdot A)))) \\ &= (\bar{O}(\bar{D}(\bar{I}))(\bar{A}(\bar{O}\ t((A \cdot B), B)\ t((A \cdot B), A)))) \\ &= (\bar{O}(\bar{D}(\bar{I}))(\bar{A}(\bar{O}(\bar{D}\ t(B, B))(\bar{A}\ t(A, A))))) \\ &= (\bar{O}(\bar{D}(\bar{I}))(\bar{A}(\bar{O}(\bar{D}(\bar{I}))(\bar{A}(\bar{I}))))). \end{aligned}$$

This expression can be represented graphically as shown in Fig. 3.

The interpretation of Fig. 3 is that Y is computed from X as follows. Y is equal to two things CONSed together. The first item is found by computing (CDR X) and returning the result. The second item is found by computing (CAR X) and passing the result $X' = (A \cdot B)$ to another calculation. In this calculation, two things are CONSed together, the results of computing (CDR X') and (CAR X'), respectively. Thus the total computation yields the value $Y = (C \cdot (B \cdot A))$.

The program synthesis method of this paper begins with a set of input–output pairs (X_i, Y_i) for the desired program and first computes the associated traces $t(X_i, Y_i)$ as described in this section. These traces then form the basis for the synthesis procedure which is presented in Section VI.

IV. Regular LISP Programs

This section will begin by defining the concept of a "semiregular" LISP program. Then the regular programs will be defined to be a certain well-behaved subset of the semiregular programs. Two theorems will be proven, one showing that the property of regularity is decidable and the other showing that the t operator of the previous section provides the correct trace for the synthesis of regular programs.

Many times the CAR and CDR functions are composed to a considerable depth; so space can be saved with an abbreviated form. Specifically, the middle letters of the composed functions can be concatenated between a C and R to represent the composition. Thus (CDR(CDR(CAR X))) will be written (CDDAR X). The set of such expressions can be written (C(A + D)*R X) with the understanding that (CR X) is just the identity function.

The conditional operator in LISP is written

$$(\text{COND}(p_1\ f_1)$$
$$(p_2\ f_2)$$
$$\vdots$$
$$(p_n\ f_n))$$

and is evaluated as follows. The predicates $p_i, i = 1, 2, \cdots, n$, are evaluated sequentially until one is found which yields true. In this study it will be assumed that p_n is the predicate T which is always true so that there will always be a first k such that p_k is true. Then the value returned by the COND function is the value of the function f_k.

A *chain of predicates* will be defined to be a sequence of predicates $p_1, p_2, \cdots, p_n$ such that

1) $p_i = (\text{ATOM}(\text{C}\ w_i\ \text{R}\ X))$, where $w_i \in (\text{A} + \text{D})^*$ for $i = 1, 2, \cdots, n-1$ and where w_i is a proper suffix of w_{i+1} for $i = 1, 2, \cdots, n-2$, and
2) $p_n = \text{T}$.

Predicates synthesized by the system described here will be in such chains. An example chain of predicates is

$$p_1 = (\text{ATOM}(\text{CAR}\ X))$$
$$p_2 = (\text{ATOM}(\text{CADAR}\ X))$$
$$p_3 = (\text{ATOM}(\text{CDDDADAR}\ X))$$
$$p_4 = \text{T}.$$

A *semiregular* LISP program P will be defined to be a finite nonempty set of component programs $F_i, i = 1, \cdots, m$, with one of them F_1 being designated as the *initial component*. The value of P operating on X, written (PX), is equal to $(F_1 X)$. A component program F_i of P is of the form

$$(F_i X) = (\text{COND}(p_{i1}\ f_{i1})$$
$$(p_{i2}\ f_{i2})$$
$$\vdots$$
$$(p_{in}\ f_{in}))$$

where $p_{i1}, p_{i2}, \cdots, p_{in}$ is a chain of predicates with arguments X and where each $f_{ij}, j = 1, \cdots, n$, is one of the following:

1) the nil function,
2) the identity function,
3) $(F_h(\text{CAR}\ X))$,
4) $(F_h(\text{CDR}\ X))$,
5) $(\text{CONS}(F_h X)(F_k X))$,

where F_h and F_k are component programs of P.

The predicates $p_{ij}, j = 1, 2, \cdots, n$, in the above definition are known as the predicates *associated* with component F_i. An argument X will be said to *satisfy* a particular predicate p_{ij} if

$$p_{ik}(X) = \text{false}, \qquad \text{for } k = 1, 2, \cdots, j-1$$

and

$$p_{ij}(X) = \text{true}.$$

The pairs $(p_{ij}\ f_{ij})$ are said to be *parts* or *transitions* of F_i. Finally, if $n = 1$ in the above definition, we will often write $(F_i X) = f_{i1}$ rather than $(F_i X) = (\text{COND}(p_{i1}\ f_{i1}))$, where $p_{i1} = \text{T}$. An example of a semiregular program appears in Section II.

An S-expression $\mathscr{U}$ is a *possible argument* for component F_i in P if there is an S-expression X such that evaluation of (PX) involves (depends on) evaluation of $(F_i\ \mathscr{U})$. For example, if P has initial component

$$(F_1 X) = (\text{COND}((\text{ATOM}(\text{CDAR}\ X))(F_2(\text{CAR}\ X)))$$
$$(\text{T}(F_3(\text{CDR}\ X)))),$$

then we know that possible arguments for F_2 include all S-expressions $\mathscr{U}$ such that $(\text{ATOM}(\text{CDR}\ \mathscr{U}))$ is true. Furthermore, F_3 can have absolutely any S-expression as a possible argument. An algorithm for determining the set of possible arguments for all of the components of a semiregular program is given in the Appendix.

The regular LISP programs will be differentiated from the semiregular programs by two properties, the first of which will be referred to as *directness*. Suppose $X = (A \cdot B)$ and $Y = (A \cdot A)$. Then we can write $Y = (F_1 X)$, where

$$(F_1 X) = (\text{CONS}(F_2 X)(F_3 X))$$
$$(F_2 X) = (F_4(\text{CAR}\ X))$$
$$(F_3 X) = (F_5(\text{CAR}\ X))$$
$$(F_4 X) = X$$
$$(F_5 X) = X.$$

However, Y can also be written as $(F_1 X)$, where

$$(F_1 X) = (F_2(\text{CAR}\ X))$$
$$(F_2 X) = (\text{CONS}(F_3 X)(F_4 X))$$
$$(F_3 X) = X$$
$$(F_4 X) = X.$$

Here CAR is to be applied to both of the arguments of the CONS operator so that the decision must be made whether to apply it after the CONS operator, as shown in the first case, or before, as shown in the second case. Regular programs will always use the second more efficient option and are differentiated from the semiregular programs by this characteristic.

The other major characteristic of regular programs concerns the absence of "CAR-NIL" or "CDR-NIL" instances. Suppose a program F_1 has the form

$$(F_1 X) = (F_2(\text{CAR}\ X))$$
$$(F_2 X) = \text{NIL}.$$

Then it is clear that the value of $(F_1 X)$ is NIL and that the computation of $(\text{CAR}\ X)$ is wasted. This is called a CAR-NIL

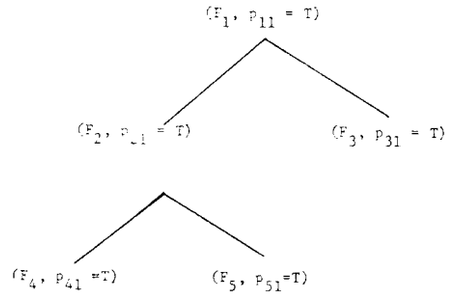

Fig. 4. Example CONS tree.

instance, and it can be avoided by defining F_1 as $(F_1 X) =$ NIL. The existence of a CAR-NIL or CDR-NIL instance is not always as obvious as in the example because one may have

$$\begin{aligned}(F_i X) = (\text{COND}&(p_{i1}\ f_{i1})\\ &(p_{i2}\ f_{i2})\\ &\vdots\\ &(p_{ij}\ \text{NIL})\\ &\vdots\\ &(p_{in}\ f_{in}))\end{aligned}$$

where $(F_i(\text{CAR}\ X))$ may occur somewhere else in the program. If it ever occurs that $(F_i(\text{CAR}\ X))$ is evaluated and p_{ij} is true, then a CAR-NIL instance will have occurred. Another example of a CAR-NIL instance occurs here:

$$\begin{aligned}(F_1 X) &= (F_2(\text{CAR}\ X))\\ (F_2 X) &= (\text{CONS}(F_3 X)(F_4 X))\\ (F_3 X) &= \text{NIL}\\ (F_4 X) &= \text{NIL}.\end{aligned}$$

In this case, F_1 will have a value (NIL · NIL), but one must trace down through F_2, F_3, and F_4 to discover that the CAR operation is wasted. These ideas are made more precise in the following paragraphs.

An *executable* CONS *tree* for a semiregular program P is a binary tree of component-associated predicate pairs (F, p) such that there is a possible argument X for the top component of the tree which satisfies every predicate in the tree and such that

1) if (F_i, p_{ik}) is a nonleaf node in the tree, then $(p_{ik}(\text{CONS}(F_{h_1} X)(F_{h_2} X)))$ is a part of F_i, and $(F_{h_1}, p_{h_1 k_1})$ and $(F_{h_2}, p_{h_2 k_2})$ are the immediate successor nodes to (F_i, p_{ik}) in the tree, and
2) if (F_i, p_{ik}) is a leaf node in the tree, then it is not true that $(p_{ik}(\text{CONS}\ Z_1\ Z_2))$ is a part of F_i for any Z_1 or Z_2.

Any given program P may have many such executable CONS trees. Suppose $(F_1 X) = (\text{CONS}(F_2 X)(F_3 X))$, $(F_2 X) = (\text{CONS}(F_4 X)(F_5 X))$, and $(F_j X) = X$ for $j = 3, 4$, and 5. Then the associated CONS tree appears in Fig. 4.

An executable CONS tree is *direct* if it is finite and if one of the following hold:

1) at least one leaf (F_i, p_{ij}) is such that $(p_{ij}(F_k(\text{CAR}\ X)))$ is a part of F_i, and at least one leaf (F_i, p_{ij}) is such that $(p_{ij}(F_k(\text{CDR}\ X)))$ is a part of F_i;
2) at least one leaf (F_i, p_{ij}) is such that $(p_{ij} X)$ is a part of F_i;
3) every leaf (F_i, p_{ij}) is such that $(p_{ij}\ \text{NIL})$ is a part of F_i.

For example, the CONS tree of Fig. 4 would not be direct if the F_i were defined as follows because none of 1), 2), or 3) would hold:

$$\begin{aligned}(F_1 X) &= (\text{COND}(P_{11}(\text{CONS}(F_2 X)(F_3 X))))\\ (F_2 X) &= (\text{COND}(p_{21}(\text{CONS}(F_4 X)(F_5 X))))\\ (F_3 X) &= (\text{COND}(p_{31}(F_6(\text{CAR}\ X))))\\ (F_4 X) &= (\text{COND}(p_{41}\ \text{NIL}))\\ (F_5 X) &= (\text{COND}(p_{51}\ \text{NIL})).\end{aligned}$$

However, if the definition of F_4 were changed to $(F_4 X) = (\text{COND}(p_{41}(F_6(\text{CDR}\ X))))$, then Fig. 4 would give a direct CONS tree because 1) would be satisfied. If F_4 were not changed but F_3 were modified to be $(F_3 X) = (\text{COND}(p_{31}\ X))$, then Fig. 4 would give a direct CONS tree by 2). The idea of directness is related to the idea of CAR-CDR precedence discussed in the previous section. If all unused information in argument X is thrown away by applying a series of CAR's and CDR's before using the CONS operator, then every CONS tree that has references to X will be followed by references to all of X, either 1) both $(\text{CAR}\ X)$ and $(\text{CDR}\ X)$ or 2) simply X.

An *executable* CAR-NIL *instance* for a semiregular program P is a quadruple $(F_h, p_{hi}, F_j, p_{jk})$ such that there is a possible argument X for F_h with the following properties: X satisfies p_{hi}; $(p_{hi}(F_j(\text{CAR}\ X)))$ is a part of F_h; $(\text{CAR}\ X)$ satisfies p_{jk}; and either

1) $(p_{jk}\ \text{NIL})$ is a part of F_j, or
2) (F_j, p_{jk}) is the top node of an executable CONS tree where every leaf (F_r, p_{rs}) is such that $(p_{rs}\ \text{NIL})$ is a part of F_r.

An *executable* CDR-NIL *instance* is similarly defined. An executable CAR-NIL or CDR-NIL instance represents wasted computation in that the CAR or CDR operation is performed on the argument even though it will never be used because NIL will be returned as the value. If $(F_1 X) = (F_2(\text{CAR}\ X))$ and $(F_2 X) = \text{NIL}$, then $(F_1, p_{11}, F_2, p_{21})$ is a CAR-NIL instance (where $p_{11} = \text{T}$ and $p_{21} = \text{T}$).

A semiregular program is *regular* if every executable CONS tree is direct and if there are no executable CAR-NIL or CDR-NIL instances. In other words, the regular programs are the well-behaved members of the semiregular class which always do direct computations without any wasted CAR or CDR operations.

There are two important results about regular programs that will be asserted. First, we will note that it is possible to test whether a given program is regular so that we will have a precise characterization of the class of programs which can be generated by the system. Second, we will observe that if P is a regular program, there is always an equivalent regular program P which performs its computations in the order given by $t(X, Y)$. Thus we can be sure that if we want to

generate P or an equivalent program, synthesis on the basis of traces $t(X, Y)$ is a correct approach to the problem.

Theorem 1: There exists an algorithm to determine whether semiregular program P is regular.

Proof: Construct all CONS trees for P and check whether they are direct. Use the procedure given in the Appendix to determine whether any of the nondirect CONS trees are executable. That is, determine whether there exists an argument X such that evaluation of (PX) involves execution of a nondirect CONS tree. Next construct all CAR-NIL and CDR-NIL instances for the given program, and similarly check whether any of them are executable. If every executable CONS tree is direct and there are no executable CAR-NIL or CDR-NIL instances, the program is regular.

A *computation trace t' of a semiregular program P* operating on an input X is a tree which traces the operations performed by P as it executes its calculation. Specifically, $t'(P, X) = t'(F_1, X)$, where F_1 is the initial component of P and $t'(F_i, X)$ is defined as follows. Suppose X satisfies p_{ij} and $(p_{ij}\, f_{ij})$ is a part of F_i; then

$$t'(F_i, X) = \begin{cases} (\bar{N}), & \text{if } f_{ij} = \text{NIL} \\ (\bar{I}), & \text{if } f_{ij} = X \\ (\bar{A}\ t'(F_k, (\text{CAR } X))), & \text{if } f_{ij} = (F_k(\text{CAR } X)) \\ (\bar{D}\ t'(F_k, (\text{CDR } X))), & \text{if } f_{ij} = (F_k(\text{CDR } X)) \\ (\bar{O}\ t'(F_k, X)\ t'(F_h, X)), & \text{if } f_{ij} = (\text{CONS}(F_k X)(F_h X)). \end{cases}$$

A trace t' is a tree like the one shown in Fig. 3, and all graph-associated concepts may be used. For example, the *parent* of any node is the node immediately above it in the tree.

Two programs P_i and P_j will be called *equivalent* (written $P_i = P_j$) if $(P_i X)$ is defined, if and only if $(P_j X)$ is defined, and if $(P_i X) = (P_j X)$ when both are defined.

Theorem 2: If P is a regular program, then there is an equivalent regular program P_0 such that $(P_0 X) = Y$ implies $t'(P_0, X) = t(X, Y)$.

Proof: In order to obtain the desired result that $(P_0 X) = Y$ implies $t'(P_0, X) = t(X, Y)$, we must have the guarantee that (call this Property A), whenever the identity function is executed in the computation of Y performed by P_0, it operates with an atom as an argument. In other words, because $t(X, Y)$ expresses Y in terms of the atoms of X, $t'(P_0, X) = t(X, Y)$ only if P_0 computes Y from the atoms of X. Consider as an example the case of computing the identity function. If $X = (A \cdot B)$ and $Y = (A \cdot B)$, then $t(X, Y) = (\bar{O}(\bar{A}(\bar{I}))(\bar{D}(\bar{I})))$, and an identity function P_0 must be found such that $(P_0 X) = Y$ and $t'(P_0, X) = (\bar{O}(\bar{A}(\bar{I}))(\bar{D}(\bar{I})))$. In fact, P_0 may be defined as follows:

$$(P_0 X) = (F_1 X) = (\text{COND}((\text{ATOM } X)X)\ (\text{T}(\text{CONS}(F_2 X)(F_3 X))))$$
$$(F_2 X) = (F_1(\text{CAR } X))$$
$$(F_3 X) = (F_1(\text{CDR } X)).$$

It is easy to construct P_0 from P in general such that Property A will hold. Suppose $(FX) = X$ is a component of P; then F may be replaced by F_1, F_2, and F_3 of the above paragraph without affecting the input–output behavior of P. Suppose (pX) is a part of some component of P, where p is not $(\text{ATOM } X)$. Then this component may be replaced by $(p(\text{CONS}(F_2 X)(F_3 X)))$ using the same F_1, F_2, and F_3 from the above paragraph. Here again, the input–output characteristics of P will be unchanged, and Property A will be made to hold. So the desired P_0 can be constructed from P and $P_0 = P$.

Assume that $(P_0 X) = (F_1 X)$, where F_1 is the initial component of P_0. We can show by induction on the length of the trace $t'(P_0, X)$ that for each F_i, if $(F_i X) = Y$, then $t'(F_i, X) = t(X, Y)$. Assume in each case that X satisfies p_{ij} and $(p_{ij}\, f_{ij})$ is a part of F_i.

Basis: Assume the length of the trace $t'(F_i, X)$ is one so that f_{ij} is either NIL or the identity function. Then in these two cases $t'(F_i, X)$ is either $(\bar{N})$ or $(\bar{I})$, respectively. Also $c(X, Y)$ is NIL or X (using Property A), meaning $t(X, Y)$ is $(\bar{N})$ or $(\bar{I})$, respectively. Therefore, $t'(F_i, X) = t(X, Y)$.

Induction: Assume the length of the trace $t'(F_i, X)$ is greater than one so that f_{ij} is $(F_k(\text{CAR } X))$, $(F_k(\text{CDR } X))$, or $(\text{CONS}(F_k X)(F_h X))$ for some F_k and F_h. Consider, for example, $f_{ij} = (F_k(\text{CAR } X))$. Assume $(F_k(\text{CAR } X)) = Y$, and consider $c(X, Y)$. Assume Y contains at least one reference to X. Then $c(X, Y)$ will contain a $(\text{CAR } X)$ and no reference to $(\text{CDR } X)$ or X alone because only $(\text{CAR } X)$ was available for computing Y. Therefore,

$$\begin{aligned} & t'(F_i, X) \\ & \quad = (\bar{A}\ t'(F_k, (\text{CAR } X))) && \text{by definition of } t' \\ & \quad = (\bar{A}\ t((\text{CAR } X), Y)) && \text{by the induction argument} \\ & \quad = t(X, Y) && \text{by definition of } t. \end{aligned}$$

The above assumption that Y contains at least one reference to X must hold by definition of the regularity of P. If Y contained no reference to X, then all atoms in Y would be NIL and $(F_i, p_{ij}, F_k, p_{km})$ would constitute a disallowed executable CAR-NIL instance in P. The induction argument for the case where $f_{ij} = (F_k(\text{CDR } X))$ is similar to the argument just given.

Suppose $f_{ij} = (\text{CONS}(F_k X)(F_h X))$ for some F_k and F_h. Then (F_i, p_{ij}) is the head of an executable CONS tree for this program. Then by the definition of regularity this CONS tree must be direct, meaning that either 1), 2), or 3) of the definition of CONS tree directness must hold. One can check that in all three cases

$$\begin{aligned} t(X, Y) &= (\bar{O}\ t(X, (\text{CAR } Y))\ t(X, (\text{CDR } Y))) \\ &\qquad \text{by definition of } t(X, Y) \\ &\qquad \text{and CONS tree directness} \\ &= (\bar{O}\ t'(F_k, X)\ t'(F_h, X)) \\ &\qquad \text{by the induction argument} \\ &= t'(F_i, X) \\ &\qquad \text{by definition of } t'(F_i, X). \end{aligned}$$

This completes the proof of Theorem 2.

V. On the Behavior of Regular LISP Programs

If someone has a LISP function in mind, he may wish to know whether there is a regular LISP program which can compute that function. It is shown in [1] that the set of executable paths (defined below) through a semiregular program is a finite-state language. If the computation of the desired function involves the execution of paths which do not constitute a finite-state language, the desired function cannot be computed by any semiregular program.

An *executable path* through a semiregular program P is a string $a_1 a_2 \cdots a_i$ such that $a_j \in \{\bar{N}, \bar{I}, \bar{A}, \bar{D}, \bar{O}\}$ for $j = 1, 2, \cdots, i$ and such that there is an S-expression X with the property that in tree $t'(P, X)$, a_j is a parent of a_{j+1} for $j = 1, 2, \cdots, i-1$.

Theorem 3: The set of executable paths through a semiregular program P is a finite-state language.

The proof of Theorem 3 appears in [1], and an example of its usage is as follows. Suppose it is desired to generate a program which converts $(A \cdot (B \cdot (C \cdot \text{NIL})))$ into $(A \cdot (B \cdot (B \cdot (C \cdot (C \cdot (C \cdot \text{NIL}))))))$, where in general, the ith item in the input appears i times in the output. But a study of the form of the traces $t(X, Y)$ for this computation reveals that they have paths of the form

$$\bar{O}\,\bar{D}\,\bar{O}\,\bar{O}\,\bar{D}\,\bar{O}\,\bar{O}\,\bar{O}\,\bar{D}\,\bar{O}\,\bar{O}\,\bar{O}\,\bar{O}\,\bar{D} \cdots \bar{D}\,\bar{O}^i\,\bar{O}\,w,$$

where w has length $i + 2$ or less. But this is not a regular set, proving that no regular LISP program can do the desired computation.

VI. Program Synthesis

A class of programs will be called *admissible* if it is enumerable and if the halting problem is solvable for each member of the class. Let $C = \{P_i | i = 1, 2, 3, \cdots\}$ be any admissible class of programs. A method for generating a program from a set S of example input–output pairs (X_i, Y_i) is to examine sequentially $P_1, P_2, P_3, \cdots$ until P_j is found such that $(P_j X_i) = Y_i$ for all (X_i, Y_i) in S. This technique has the advantages that it always finds a program which can do the given examples (soundness) and that it can always generate every program (or its equivalent) in the class C (completeness). Such a technique can also be shown to make optimal use of its input information (the input–output pairs) in the sense that no other synthesis technique can be found which will generate every program P_j on the basis of less input–output information than the technique given here.

An enumerative approach to program synthesis would be totally impractical if it were naively implemented, because most programs of any interest will have astronomically large indexes in the enumeration. However, in the case of regular LISP programs, the computation trace for obtaining each Y_i from its associated X_i is known (because of the regularity assumption). This means that the only programs which need be examined are those which can execute the given traces, and very powerful techniques exist for hurrying through this enumeration. This section will present a synthesis algorithm which is efficient enough to generate LISP programs of significant complexity and which has all of the desirable properties of a completely enumerative program.

Before beginning these discussions, it is important to note that the halting problem for the regular LISP programs is decidable.

Theorem 4: If P is a regular LISP program, then P will halt after a finite time on any argument.

Proof: If P fails to halt when operating on argument X, then $t'(P, X)$ must have an infinite path $w = a_1 a_2 a_3 \cdots$, $a_i \in \{\bar{N}, \bar{I}, \bar{A}, \bar{D}, \bar{O}\}$. w could not have an $\bar{N}$ or an $\bar{I}$ because this would terminate the path. It could not have an infinite number of $\bar{A}$'s or $\bar{D}$'s because this would imply an infinite number of CAR's or CDR's applied to finite X resulting in an undefined (thus halting) operation on an atom at some point. Thus $w = w_1 w_2$, where $w_1 \in \{\bar{A}, \bar{D}, \bar{O}\}^*$ has finite length and where w_2 is an infinite string of $\bar{O}$'s. But regular LISP programs are incapable of executing an infinite string of $\bar{O}$'s because this would imply an infinite executable CONS tree which contradicts the directness requirement included in the definition of regularity.

The general enumerative algorithm for program synthesis will now be given along with statements of some of its properties.

Algorithm A1:

Input: A finite set S of input–output pairs (X_i, Y_i) for the desired program.

Output: A program P_j from class C with the property that $(P_j X_i) = Y_i$ for each given (X_i, Y_i) in S.

1) $j \leftarrow 1$.
2) while there is $(X_i, Y_i) \in S$ such that $(P_j X_i) \neq Y_i$, increment j.
3) return with result P_j.

The result obtained if algorithm $A1$ enumerates class C and has input S will be denoted $A1(C, S)$.

A program synthesis algorithm A will be called *sound* if, whenever S represents the behavior of a program in C, $(A(C, S)X_i) = Y_i$ for each $(X_i, Y_i) \in S$.

Program P_j will be said to *cover* program P_i if the fact that $(P_i X)$ is defined implies that $(P_j X)$ is defined and that $(P_i X) = (P_j X)$. Algorithm A will be called *complete* over class C if for each $P \in C$ there is a finite set S of pairs (X, Y) such that $A(C, S)$ will halt, yielding P_j which covers P.

Algorithm A will be called *stable* if $(A(C, S)X) = Y$ for all $(X, Y) \in S'$ implies $A(C, S \cup S') = A(C, S)$. Thus if A chooses a program P on the basis of information S and if additional information S' is compatible with P, A will not make a different choice on the basis of $S \cup S'$.

Theorem 5: If C is admissible, then $A1$ enumerating class C is sound, complete, and stable.

Proof: The soundness and stability properties follow trivially from the construction of $A1$. The completeness property requires only a simple proof. Let P be an arbitrarily chosen program from C, and let P_j be the first program in the enumeration of C which covers P. For each $i = 1, 2, 3, \cdots, j-1$, choose an (X_i, Y_i) such that $(P_j X_i) = Y_i$ and $(P_i X_i)$ is undefined or $(P_i X_i) \neq Y_i$. Such an (X_i, Y_i) can be found for each $i = 1, 2, \cdots, j-1$ because the absence of such (X_i, Y_i) would imply P_i covers P_j and P for $i < j$, which contradicts

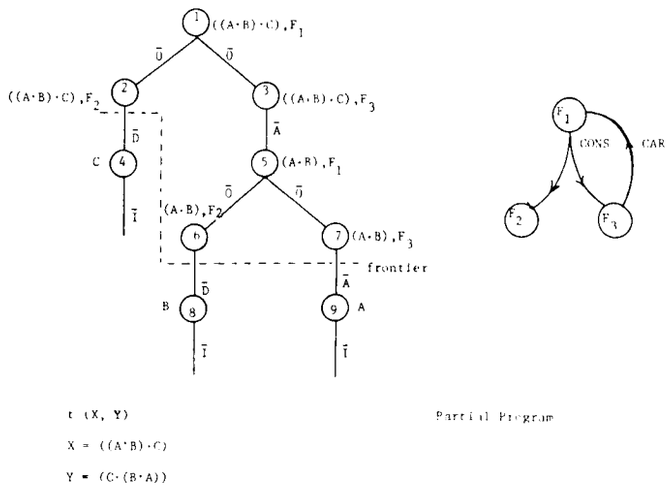

Fig. 5. Computation trace and partially synthesized program.

the definition of P_j. If $A1$ operates on finite set $S = \{(X_i, Y_i) | i = 1, 2, \cdots, j-1\}$, it will halt and return P_j as its answer. This completes the proof.

Suppose $A0$ is a program synthesis algorithm which is sound, complete, and stable over C. If for every S there is an associated $S' \subseteq S$ such that $A0(C, S') = A1(C, S)$ and if not all such S with associated S' are such that $A1(C, S') = A1(C, S)$, then $A0$ will be said to be *more input efficient* than $A1$.

Theorem 6: If a program synthesis algorithm $A0$ is sound, complete, and stable over C, then $A0$ is not more input efficient than $A1$.

Proof: Assume $A0$ is more input efficient than $A1$. Then there must be an S and an $S' \subseteq S$ such that $A1(C, S) = P_j$, $A0(C, S') = P_j$, and $A1(C, S') = P_i$ for $i < j$. Then there must be a subset $S'' \subseteq S'$ such that $A0(C, S' - S'') = P_i$ since $A0$ is more input efficient than $A1$, and $A0$ is complete. But for each $(X, Y) \in S''$, $(P_i\ X) = Y$ by the soundness of $A1$. So by the stability property of $A0$, $A0(C, (S' - S'') \cup S'') = P_i$. But this contradicts the fact that $A0(C, S') = P_j$ for $i < j$; so it must not be true that $A0$ is more input efficient than $A1$. This completes the proof.

Theorems 5 and 6 are adaptations of results by Gold [8] to the domain of this study.

Returning to the example of Section I, it is desired to generate a program that outputs $Y = (C \cdot (B \cdot A))$ from input $X = ((A \cdot B) \cdot C)$. If the desired program is regular, we know that the trace of the computation must be as shown in Fig. 5. A wise synthesis procedure is to carry out the enumeration $P_1, P_2, P_3, \cdots$ of the regular LISP programs, being sure to skip every program which cannot execute the "first part" of this trace $t(X, Y)$. "First part" in this case means all of the trace out to the trace *frontier*, which is an imaginary line across the trace showing how much of the trace has been processed at any given time. Fig. 5 shows one such frontier and a partial program which can execute the trace up to the indicated frontier. The synthesis procedure given here is thus strictly enumerative in the sense of algorithm $A1$, but the trace is used to drastically reduce the search.

Two S-expressions will be said to have *identical templates* if they become identical when all atoms are changed to NIL. Thus $((A \cdot B) \cdot C)$ and $((B \cdot \text{NIL}) \cdot D)$ have identical templates.

Notice that in the trace of Fig. 5 each node has an associated input S-expression. Thus node 3 has $((A \cdot B) \cdot C)$ which becomes $(A \cdot B)$ at node 5 after the CAR operation. Notice also that each trace node above the frontier has associated with it the component name from the partial program which executed that portion of the trace. At each point in the synthesis process, a set of *active* nodes will be chosen along the frontier for processing. The set of active nodes will have the property that they all have the same associated component names and identical associated input templates. Along the frontier shown, there are three possible (singleton) sets of active nodes to choose from: {2} with associated component F_2 and template ((NIL · NIL) · NIL), {6} with associated component F_2 and template (NIL · NIL) and, {7} with associated component F_3 and template (NIL · NIL). In general, a set of active nodes may be quite large so that many nodes in the trace can be processed in one algorithmic step. The component of the program which is associated with the active nodes is called the *active* component.

The synthesis procedure chooses a set of active nodes on the frontier and attempts to advance the frontier at those nodes by adding, if necessary, a transition to the active component of the program. Suppose that {2} is chosen as the set of active nodes. Then a transition in the program from active component F_2 on an input of the form ((NIL · NIL) · NIL) is to be added to the program. In terms of LISP, F_2 is about to be defined as $(F_2\ X) = (F_i(\text{CDR}\ X))$, where each $i = 1, 2, 3$, and 4 is tried. All possible values of i are tried because we are enumerating all possible programs which could execute the trace. In this case $i = 1$ succeeds, leaving the definition as $(F_2\ X) = (F_1(\text{CDR}\ X))$. The frontier is thus advanced from {2, 6, 7} to {4, 6, 7} and a new set of active nodes may be chosen.

If {6} is chosen as the next set of active nodes, we notice that the above definition of F_2 provides the proper CDR operation enabling the procedure to advance the frontier to {4, 8, 7} without additional modifications to the partial program. Here we say that the trace transition below {6} was *merged* with a transition from F_2. If node 6 had been followed by an $\bar{A}$ instead of $\bar{D}$, the merge would have failed, and the definition of F_2 would have been modified (if possible) to yield a CDR operation for inputs of type ((NIL · NIL) · NIL) and a CAR operation for inputs of type (NIL · NIL):

$$(F_2\ X) = ((\text{ATOM}(\text{CAR}\ X))(F_i(\text{CAR}\ X))$$
$$(\text{T}(F_1(\text{CDR}\ X)))$$

for some $i = 1, 2, 3, 4$, or 5.

The regular LISP program synthesizer will be called $A2$. It has three major routines: MAIN which sets the maximum allowed number of synthesizable transitions and then sends the system off looking for a solution, NODE PROCESSOR which

attempts to find a way to advance the frontier either by creating a new program transition or by merging trace transitions with a program transition as explained above, and CREATE NEW TRANSITION which creates a new transition in the program.

The program MAIN sets the maximum allowed number of transitions (MAXTRANS) to one, and then the complete space of one transition programs is searched for a program which can execute the given traces. If NODE PROCESSOR returns having found no solution, MAXTRANS is incremented and the search begins again. The program MAIN may be summarized as follows.

MAIN:
MAXTRANS = 1;
NO SOLUTION FOUND = TRUE;
WHILE NO SOLUTION FOUND;
  INITIALIZE SYNTHESIZED PROGRAM AS ONE COMPONENT AND NO TRANSITIONS;
  CALL **NODE PROCESSOR**
  MAXTRANS = MAXTRANS + 1;
END.

The program NODE PROCESSOR begins by updating the frontier on the traces and picking out a set of active nodes. If a contradiction is found among the active nodes, NODE PROCESSOR immediately fails and returns to the calling routine. Thus if two active nodes are both associated with component F_2 and have input template (NIL · NIL) but are followed by $\bar{A}$ and $\bar{D}$ transitions, respectively, no transition can be added to F_2 which can execute both portions of the trace. If no such contradictions are found, NODE PROCESSOR begins looking for a way to advance the frontier at the active nodes. If no transition leaves the active component, then CREATE NEW TRANSITION is called to create the first such transition. If there are transitions leaving the active component, an attempt is made to merge the trace transitions below the active nodes with the first transition leaving this active component. The merge will be successful if the operations associated with the transitions below the active nodes are identical to the operation associated with the proposed program transition and if the predicates of the active component can be revised to enable the program to correctly execute the traces up to the new frontier. If the merge is successful, NODE PROCESSOR is called recursively to update the frontier again, to choose a new set of active nodes, and to make further additions to the program being created. If the merge fails or if it succeeds but NODE PROCESSOR returns with failure, another transition from the active component is chosen for possible merger.

It may be that NODE PROCESSOR will not be able to merge the trace transitions with any existing transition from the active component. Then attempts are made to create a new transition leaving the active component. The new transition may be either before or after any existing transition, and every possibility must be tried. At first the new transition is proposed to precede the first existing transition. An attempt is made to build predicates for the active component so that this new transition and all others correctly execute the traces to the new frontier. If the attempt is successful, CREATE NEW TRANSITION is called to add this new transition at the appropriate position with the appropriate predicate. If the attempt fails or CREATE NEW TRANSITION returns with failure, a different location for the new transition is tried.

If all of the above attempts to account for the transitions leaving the active nodes fail, then NODE PROCESSOR returns with failure. There is no addition to the given partial program which will enable it to execute the given traces. Backup to a higher level routine will cause the partial program to be modified and the frontier to be raised to an earlier position. The NODE PROCESSOR routine may be summarized as follows.

NODE PROCESSOR:
IF NO TRANSITIONS HAVE BEEN CREATED, THE FRONTIER IS THE SET OF INITIAL NODES OF THE TRACES
ELSE THE FRONTIER IS ADVANCED TO ACCOUNT FOR THE TRANSITION PROCESSED IN THE CALLING ROUTINE;
IF THERE ARE UNPROCESSED NODES ON THE FRONTIER, CHOOSE A NEW SET OF ACTIVE NODES
ELSE PRINT THE SOLUTION PROGRAM;
  NO SOLUTIONS FOUND = FALSE;
  RETURN;
IF CONTRADICTIONS ARE FOUND AMONG THE ACTIVE NODES, RETURN;
IF NO TRANSITION LEAVES THE ACTIVE COMPONENT, CALL **CREATE NEW TRANSITION**; RETURN;
FOR EACH TRANSITION LEAVING THE ACTIVE COMPONENT, IF IT IS MERGABLE WITH THE TRANSITIONS LEAVING THE ACTIVE NODES, CALL **NODE PROCESSOR**;
FOR EACH LOCATION BEFORE AND AFTER AN EXISTING TRANSITION, IF PREDICATES CAN BE BUILT SO THAT A NEW TRANSITION CAN BE INSERTED AT THIS LOCATION, CALL **CREATE NEW TRANSITION**;
RETURN FROM **NODE PROCESSOR**.

The CREATE NEW TRANSITION routine first checks to see whether the existing number of transitions in the program being generated has reached the maximum MAXTRANS set by MAIN. If the maximum has been reached, CREATE NEW TRANSITION fails immediately. If the operation below the active nodes is $\bar{N}$ or $\bar{I}$, then the appropriate NIL or identity transition is added to the active component, and NODE PROCESSOR is called to advance the frontier and continue the processing. If the active nodes are followed by $\bar{A}$ or $\bar{D}$, the appropriate transition CAR or CDR is added to the active component. That is, part $(p(F_1(\text{C } u \text{ R } X)))$, $u \in \{\text{A, D}\}$, is added to the active component where p and the location of the addition were set by the calling routine NODE PROCESSOR. Then NODE PROCESSOR is called to advance the frontier, and so forth. If NODE PROCESSOR returns with failure, the parts $(p(F_2(\text{C } u \text{ R } X)))$, $(p(F_3(\text{C } u \text{ R } X)))$, etc., are tried sequentially until every possible destination has been attempted including a new component which was not previously created. If all such attempts fail, CREATE NEW TRANSITION returns with failure. If the active nodes are followed by $\bar{O}$, a new transition $(p(\text{CONS}(F_i X)(F_j X))$ is attempted, and every

possible pair of destinations F_i and F_j is tried. This routine may be summarized as follows.

```
CREATE NEW TRANSITION:
IF THE CURRENT NUMBER OF TRANSITIONS IN THE PROGRAM IS
   MAXTRANS, RETURN;
IF A NIL OR IDENTITY TRANSITION IS TO BE CREATED,
   CREATE THE TRANSITION;
   CALL NODE PROCESSOR;
   RETURN;
IF A CAR OR CDR TRANSITION IS TO BE CREATED,
   FOR EACH POSSIBLE DESTINATION OF THE TRANSITION,
      CREATE THE CORRECT TRANSITION TO THAT
      DESTINATION;
      CALL NODE PROCESSOR;
   RETURN;
IF A CONS TRANSITION IS TO BE CREATED,
   FOR EACH POSSIBLE PAIR OF DESTINATIONS,
      CREATE A TRANSITION TO THAT PAIR;
      CALL NODE PROCESSOR;
   RETURN;
END.
```

The reader interested in seeing this type of algorithm stated in much more precise form and accompanied by proofs of correctness should consult [5] or [2]. Even though these references are not concerned with LISP synthesis, they give two different methods by which the current algorithm could be formalized and proven correct.

In the following, A2 will be illustrated by working through a synthesis from the trace of Fig. 5. Many steps will be omitted where unsuccessful searches are performed.

```
ENTER MAIN
   MAXTRANS = 1 (FAILURE)
   MAXTRANS = 2 (FAILURE)
   MAXTRANS = 3 (FAILURE)
   MAXTRANS = 4
   INITIALIZE SYNTHESIZED PROGRAM AS ONE COMPONENT
      (Fig. 6(a))
ENTER NODE PROCESSOR
   NO TRANSITIONS HAVE BEEN CREATED: FRONTIER = {1}
   THERE ARE UNPROCESSED NODES: ACTIVE = {1}
   NO CONTRADICTIONS FOUND
   NO TRANSITION LEAVES THE ACTIVE COMPONENT F1
ENTER CREATE NEW TRANSITION
   A CONS TRANSITION IS TO BE CREATED
      TRY DESTINATIONS F1, F1 (FAILURE, VIOLATES
         REGULARITY)
      TRY DESTINATIONS F1, F2 (FAILURE, VIOLATES
         REGULARITY)
      TRY DESTINATIONS F2, F1 (FAILURE, VIOLATES
         REGULARITY)
      TRY DESTINATIONS F2, F2 (FAILURE REPORTED FROM
         NODE PROCESSOR)
      TRY DESTINATIONS F2, F3 (Fig. 6(b))
ENTER NODE PROCESSOR
   TRANSITIONS HAVE BEEN CREATED: FRONTIER = {2, 3}
   THERE ARE UNPROCESSED NODES: ACTIVE = {3}
```

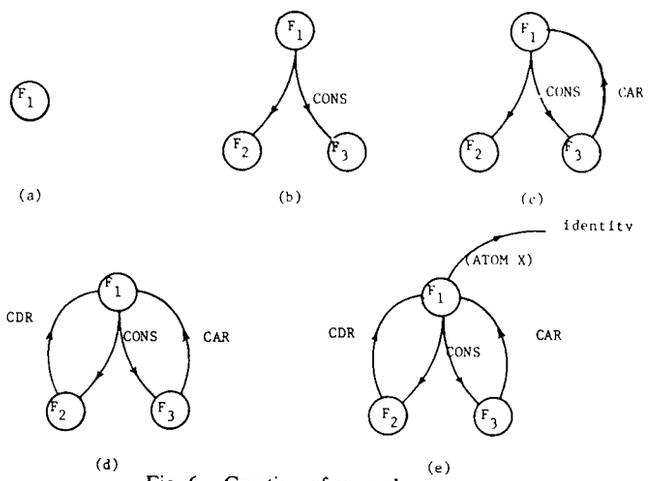

Fig. 6. Creation of example program.

```
   NO CONTRADICTIONS FOUND
   NO TRANSITION LEAVES THE ACTIVE COMPONENT F3
ENTER CREATE NEW TRANSITION
   A CAR TRANSITION IS TO BE CREATED
      TRY DESTINATION F1 (Fig. 6(c))
ENTER NODE PROCESSOR
   TRANSITIONS HAVE BEEN CREATED: FRONTIER = {2, 5}
   THERE ARE UNPROCESSED NODES: ACTIVE = {5}
   NO CONTRADICTIONS FOUND
   A TRANSITION LEAVES THE ACTIVE COMPONENT F1
      TRY MERGE OF TRANSITIONS FROM ACTIVE NODE AND
      ACTIVE COMPONENT (SUCCESS)
ENTER NODE PROCESSOR
   TRANSITIONS HAVE BEEN CREATED: FRONTIER = {2, 6, 7}
   THERE ARE UNPROCESSED NODES: ACTIVE = {2}
   NO CONTRADICTIONS FOUND
   NO TRANSITION LEAVES THE ACTIVE COMPONENT F2
ENTER CREATE NEW TRANSITION
   A CDR TRANSITION IS TO BE CREATED
      TRY DESTINATION F1 (Fig. 6(d))
ENTER NODE PROCESSOR
   TRANSITIONS HAVE BEEN CREATED: FRONTIER = {4, 6, 7}
   THERE ARE UNPROCESSED NODES: ACTIVE = {6}
   NO CONTRADICTIONS FOUND
   A TRANSITION LEAVES THE ACTIVE COMPONENT F2
      TRY MERGE OF TRANSITIONS FROM ACTIVE NODE AND
      ACTIVE COMPONENT (SUCCESS)
ENTER NODE PROCESSOR
   TRANSITIONS HAVE BEEN CREATED: FRONTIER = {4, 8, 7}
   THERE ARE UNPROCESSED NODES: ACTIVE = {7}
   NO CONTRADICTIONS FOUND
   A TRANSITION LEAVES THE ACTIVE COMPONENT F3
      TRY MERGE OF TRANSITIONS FROM ACTIVE NODE AND
      ACTIVE COMPONENT (SUCCESS)
ENTER NODE PROCESSOR
   TRANSITIONS HAVE BEEN CREATED: FRONTIER = {4, 8, 9}
   THERE ARE UNPROCESSED NODES: ACTIVE = {4, 8, 9}
   NO CONTRADICTIONS FOUND
   A TRANSITION LEAVES THE ACTIVE COMPONENT F1
      TRY MERGE OF TRANSITIONS FROM ACTIVE NODE AND
      ACTIVE COMPONENT (FAILURE)
```

TRY CREATING A PREDICATE SO THAT A NEW TRANSITION CAN BE INSERTED BEFORE THE EXISTING TRANSITION (SUCCESS: $p = (\text{ATOM } X)$)
ENTER **CREATE NEW TRANSITION**
AN IDENTITY TRANSITION IS TO BE CREATED (Fig. 6(e))
ENTER **NODE PROCESSOR**
TRANSITIONS HAVE BEEN CREATED: FRONTIER = { }
THERE ARE NO UNPROCESSED NODES
PRINT THE SOLUTION PROGRAM
NO SOLUTIONS FOUND = FALSE
POP TO THE TOP LEVEL OF THE NESTED RECURSIVE CALLS AND HALT.

Our implementation of $A2$ completes this computation in about half a second. It is interesting to note that even though for illustrative purposes this program was generated from the input-output pair $(((A \cdot B) \cdot C), (C \cdot (B \cdot A)))$, the same program could have been generated from $((A \cdot B), (B \cdot A))$ or any other example with a nonatomic input.

Before concluding this section, it may be helpful to briefly discuss the predicate generation process which is largely ignored above. The partial program must be able at each step of the synthesis process to execute all of the given traces down to their respective frontiers. Let p_{ij} be the jth predicate in component F_i. Let T_{ij} be the set of S-expressions taken from the traces which are inputs to F_i when the transition with predicate p_{ij} is taken. Then for each component F_i a chain of predicates $p_{i1}, p_{i2}, \cdots, p_{in_i}$ must be found such that for all $X \in T_{ij}$, for $j = 1, 2, \cdots, n_i - 1$, $p_{ij}(X) = \text{T}$ and such that for all $X \in T_{in_i}$, for all $j = 1, 2, \cdots, n_i - 1$, $p_{ij}(X) = \text{NIL}$. Each time that a program transition is created or a merge is made, the predicate generation routine must discover for some F_i whether such a chain of predicates can be found and must produce that chain if it exists. That such a predicate generation routine can be built can be verified easily by any programmer, and no details of its design need be included here.

VII. Some Synthesized Programs

Algorithm $A2$ was implemented on a PDP-11/45, and some of the generated programs are included here. Because $A2$ is enumerative, the time required to complete a synthesis rises exponentially with the size of the target program. With the current system, programs with three transitions or less usually could be created in a second or less. Four transitions often required several seconds, five transitions took most of a minute, and six or more transitions could require much more than a minute of time. No special search pruning techniques were employed to speed up $A2$ although many are known. For example, [4] explains a failure memory technique which was successful in speeding up a similar algorithm by two or more orders of magnitude for problems similar to the ones discussed here.

The construction of the trace $t(X, Y)$ was not automated; so the examples given below were generated by constructing the traces by hand and submitting them to $A2$. List notation is used in this section as is common in the LISP literature: $(x_1\ x_2\ x_3 \cdots x_n)$ is used to represent the S-expression $(x_1 \cdot (x_2 \cdot (x_3 \cdot (\cdots (x_n \cdot \text{NIL})) \cdots)$.

Example 1: Find the last element of a list: $(A\ B\ C)$ yields an output C. $(A\ B\ C)$ is defined as $(A \cdot (B \cdot (C \cdot \text{NIL})))$, which is changed to $(A \cdot (B \cdot (C \cdot D)))$ to accommodate the input specification of the program synthesizer. (All input atoms must be distinct and non-NIL.)

Example Input	Output
$(A \cdot (B \cdot (C \cdot D)))$	C

Program:

$$(F_1\ X) = (\text{COND}((\text{ATOM } X)X)$$
$$((\text{ATOM}(\text{CDR } X))(F_1(\text{CAR } X)))$$
$$(\text{T}(F_1(\text{CDR } X)))).$$

Time: $\frac{1}{2}$ s.

The same program would have been generated if any other input list of length two or more had been used. Similar behavioral robustness would be exhibited on any other example in this section.

Example 2: Find the third from last element of a list: $(A\ B\ C\ D\ E)$ yields an output C. The input list is converted for submission to the algorithm as in Example 1.

Example Input	Output
$(A \cdot (B \cdot (C \cdot (D \cdot (E \cdot F)))))$	C

Program:

$$(F_1\ X) = (\text{COND}((\text{ATOM } X)X)$$
$$((\text{ATOM}(\text{CDDDR } X))(F_1(\text{CAR } X)))$$
$$(\text{T}(F_1(\text{CDR } X)))).$$

Time: $\frac{1}{2}$ s.

Example 3: Suppose the user has submitted the behavior example of Example 2 with the goal in mind of producing a program to output the third element of a list. Then he tests the program on the input $(A\ B\ C\ D)$ and obtains B as an output. Clearly something is wrong; so he resynthesizes the program with two examples, $(A\ B\ C\ D\ E)$ yields C and $(A\ B\ C\ D)$ also yields C. This time he obtains the program he wanted. (List notation is converted as in Example 1.)

Example Input	Output
$(A \cdot (B \cdot (C \cdot (D \cdot (E \cdot F)))))$	C
$(A \cdot (B \cdot (C \cdot (D \cdot E))))$	C

Program:

$$(F_1\ X) = (\text{COND}((\text{ATOM } X)X)$$
$$(\text{T}(F_2(\text{CDR } X))))$$
$$(F_2\ X) = (F_3(\text{CDR } X))$$
$$(F_3\ X) = (F_1(\text{CAR } X)).$$

Time: 3 s.

In order to illustrate behavioral robustness, it should be mentioned that the same program would be generated if any two or more of the following input-output pairs had been used.

Input	Output
(*R S T A*)	*T*
(*X Y Z*)	*Z*
(*A B C D E*)	*C*
(*R O B U S T*)	*B*
(*R E G U L A R*)	*G*
(*A B C D E F G H*)	*C*

Example 4: If an *S*-expression is linear with an atom on at least one side of every CONSed pair, go to the bottom of the chain and give the atom furthest to the right.

Example Input	Output
$(A \cdot ((B \cdot C) \cdot D))$	*C*

Program:

$$(F_1\,X) = (\text{COND}((\text{ATOM}\ X)X)$$
$$((\text{ATOM}(\text{CAR}\ X))(F_1(\text{CDR}\ X)))$$
$$(\text{T}(F_1(\text{CAR}\ X)))).$$

Time: $1\frac{2}{3}$ s.

Example 5: Sequentially take CDR, CDR, CAR, CDR, CAR, CDR, and so forth until an atom is found. Return that atom as the result.

Example Inputs	Outputs
A	*A*
$(A \cdot B)$	*B*
$(A \cdot (B \cdot C))$	*C*
$(A \cdot (B \cdot (C \cdot D)))$	*C*
$(A \cdot (B \cdot ((C \cdot (D \cdot E)) \cdot F)))$	*D*

Program:

$$(F_1\,X) = (\text{COND}((\text{ATOM}\ X)X)$$
$$(\text{T}(F_2(\text{CDR}\ X))))$$
$$(F_2\,X) = (\text{COND}((\text{ATOM}\ X)X)$$
$$((\text{ATOM}(\text{CDR}\ X))(F_1(\text{CDR}\ X)))$$
$$(\text{T}(F_3(\text{CDR}\ X))))$$
$$(F_3\,X) = (F_2(\text{CAR}\ X)).$$

Time: 152 s.

Example 6: Find the atoms at the extreme left and right in an *S*-expression.

Example Input	Output
$((A \cdot B) \cdot (C \cdot D))$	$(A \cdot D)$

Program:

$$(F_1\,X) = (\text{CONS}(F_2\,X)(F_3\,X))$$
$$(F_2\,X) = (\text{COND}((\text{ATOM}\ X)X)$$
$$(\text{T}(F_2(\text{CAR}\ X))))$$
$$(F_3\,X) = (\text{COND}((\text{ATOM}\ X)X)$$
$$(\text{T}(F_3(\text{CDR}\ X)))).$$

Time: 24 s.

Example 7: Double all of the elements of a list: (*A B C*) yields (*A A B B C C*). First these lists are converted to *S*-expressions as described in Example 1.

Example Input	Output
$(A \cdot (B \cdot (C \cdot D)))$	$(A \cdot (A \cdot (B \cdot (B \cdot (C \cdot (C \cdot D))))))$

Program:

$$(F_1\,X) = (\text{COND}((\text{ATOM}\ X)X)$$
$$(\text{T}(\text{CONS}(F_2\,X)(F_3\,X))))$$
$$(F_2\,X) = (F_1(\text{CAR}\ X))$$
$$(F_3\,X) = (\text{CONS}(F_2\,X)(F_4\,X))$$
$$(F_4\,X) = (F_1(\text{CDR}\ X)).$$

Time: 10 s.

Example 8: In a list of lists, obtain the first element of each list: ((*A*) (*B*)) yields (*A B*). First these lists are converted to *S*-expressions as described in Example 1.

Example Input	Output
$((A \cdot D) \cdot ((B \cdot E) \cdot C))$	$(A \cdot (B \cdot C))$

Program:

$$(F_1\,X) = (\text{COND}((\text{ATOM}\ X)X)$$
$$((\text{ATOM}(\text{CAR}\ X))(F_1(\text{CAR}\ X)))$$
$$(\text{T}(\text{CONS}(F_2\,X)(F_3\,X))))$$
$$(F_2\,X) = (F_1(\text{CAR}\ X))$$
$$(F_3\,X) = (F_1(\text{CDR}\ X)).$$

Time: 18 s.

Example 9: In a list of atoms and lists, collect the first atoms of the lists: (*A* (*B*) *C* (*D*) (*E*) *F*) yields (*B D E*).

Example Input	Output
$(A \cdot ((B \cdot H) \cdot (C \cdot ((D \cdot I) \cdot ((E \cdot J) \cdot (F \cdot G))))))$	$(B \cdot (D \cdot (E \cdot G)))$

Program:

$$(F_1\,X) = (\text{COND}((\text{ATOM}\ X)X)$$
$$((\text{ATOM}(\text{CAR}\ X))(F_1(\text{CDR}\ X)))$$
$$(\text{T}(\text{CONS}(F_2\,X)(F_3\,X))))$$
$$(F_2\,X) = (\text{COND}((\text{ATOM}\ X)X)$$
$$(\text{T}(F_2(\text{CAR}\ X))))$$
$$(F_3\,X) = (F_1(\text{CDR}\ X)).$$

Time: About $\frac{1}{2}$ h.

Example 10: Collect the atoms in a list of atoms and lists: (*A* (*B*) *C* (*D*) (*E*) *F*) yields (*A C F*).

Example Input	Output
$(A \cdot ((B \cdot H) \cdot (C \cdot ((D \cdot I) \cdot ((E \cdot J) \cdot (F \cdot G))))))$	$(A \cdot (C \cdot (F \cdot G)))$

Program:

$$(F_1\,X) = (\text{COND}((\text{ATOM}\ X)X)$$
$$((\text{ATOM}(\text{CAR}\ X))(\text{CONS}(F_2\,X)(F_3\,X)))$$
$$(\text{T}(F_1(\text{CDR}\ X))))$$
$$(F_2\,X) = (F_1(\text{CAR}\ X))$$
$$(F_3\,X) = (F_1(\text{CDR}\ X)).$$

Time: 39 s.

Example 11: Interchange the atoms of the bottom pairs of a uniform S-expression.

Example Input	Output
$((A \cdot B) \cdot ((C \cdot D) \cdot (E \cdot F)))$	$((B \cdot A) \cdot ((D \cdot C) \cdot (F \cdot E)))$

Program:

$$(F_1\,X) = (\text{COND}((\text{ATOM}\ X)X)$$
$$(\text{T}(\text{CONS}(F_2\,X)(F_3\,X)))$$
$$(F_2\,X) = (\text{COND}((\text{ATOM}(\text{CAR}\ X))(F_1(\text{CDR}\ X)))$$
$$(\text{T}(F_1(\text{CAR}\ X))))$$
$$(F_3\,X) = (\text{COND}((\text{ATOM}(\text{CAR}\ X))(F_1(\text{CAR}\ X)))$$
$$(\text{T}(F_1(\text{CDR}\ X)))).$$

Time: 27 s.

Example 12: One might wonder what happens if random S-expressions are typed into the system. The answer is that the synthesizer will do the best it can.

Example Inputs	Output
A	A
$(A \cdot B)$	$(A \cdot \text{NIL})$
$((A \cdot B) \cdot C)$	A
$(A \cdot (((B \cdot C) \cdot D) \cdot E))$	B

Program:

$$(F_1\,X) = (\text{COND}((\text{ATOM}\ X)X)$$
$$((\text{ATOM}(\text{CDR}\ X))(F_2(\text{CAR}\ X)))$$
$$(\text{T}(F_2(\text{CDR}\ X))))$$
$$(F_2\,X) = (\text{COND}((\text{ATOM}\ X)(\text{CONS}(F_1\,X)(F_3\,X)))$$
$$(\text{T}(F_1(\text{CAR}\ X))))$$
$$(F_3\,X) = \text{NIL}.$$

Time: 69 s.

Example 13: Output a constant: $(\text{NIL} \cdot ((\text{NIL} \cdot \text{NIL}) \cdot \text{NIL}))$.

Example Input	Output
A	$(\text{NIL} \cdot ((\text{NIL} \cdot \text{NIL}) \cdot \text{NIL}))$

Program:

$$(F_1\,X) = (\text{CONS}(F_2\,X)(F_3\,X))$$
$$(F_2\,X) = \text{NIL}$$
$$(F_3\,X) = (\text{CONS}(F_4\,X)(F_2\,X))$$
$$(F_4\,X) = (\text{CONS}(F_2\,X)(F_2\,X)).$$

Time: $1\frac{1}{2}$ s.

Example 14: Distinguish between two complex S-expressions. This example simply tests the predicate-building mechanism.

Example Input	Output
$((A \cdot B) \cdot ((((C \cdot D) \cdot E) \cdot (((F \cdot G) \cdot H) \cdot I)) \cdot J))$	NIL
$((A \cdot B) \cdot ((((C \cdot D) \cdot E) \cdot (((F \cdot (L \cdot M)) \cdot H) \cdot I)) \cdot J))$	$(\text{NIL} \cdot \text{NIL})$

Program:

$$(F_1\,X) = (\text{COND}((\text{ATOM}(\text{CDAADADR}\ X))\text{NIL})$$
$$(\text{T}(\text{CONS}(F_2\,X)(F_2\,X)))$$
$$(F_2\,X) = \text{NIL}.$$

Time: $\frac{1}{2}$ s.

VIII. Discussion

The class of regular LISP programs probably can compute most of the LISP functions that one might think of which do not require the EQUAL predicate or the use of additional variables. One might consider modifications of the definition and examine what kind of synthesis performance might be achieved. Broadening the definition some might add interesting programs to the class, but it would also increase the time required to generate programs. Too great an increase in the class size would also cause the halting problem to become unsolvable, in which case even enumerative methods run into difficulties. The system described here is able to discard quickly huge classes of candidate programs because of the regularity assumption (which implies the solvability of the halting problem). If the halting problem is not solvable, then the astronomical number of programs which are not known to halt on the given examples must be saved as possible solutions to the synthesis problem until one is found which can do the examples.

One could also narrow the class and greatly speed up the system. For example, in one experiment the enumeration was limited to linear recursions, and the program of Example 9 was generated in less than 1 min, indicating a speedup of about 30. A narrow enough definition makes it possible to discard enumerative techniques and to generate programs in polynomial time. The author will report on such a system at a later time, or the reader might wish to study some of the other efforts in LISP synthesis [9], [10], [12]–[15].

Continued research in this area will look for a narrowed class of control structures which will make it possible to search more deeply in the set of all possible programs and still be able to generate many programs of interest. It is also important to try to eliminate the limitations listed above,

inability to use the EQUAL predicate or additional variables. If some success can be achieved in these areas and if this method can be used to sequentially generate a hierarchy of programs, truly large and useful programs will be created automatically.

Appendix
Discovering the Set of all Possible Arguments for Each Component Function

It is necessary to begin by developing some notation. We will be using predicates p of the following form:

$$(\text{ATOM}(\text{C } w \text{ R } X)) \text{ or } (G(\text{ATOM}(\text{C } w \text{ R } X)))$$

where $w \in (\text{A} + \text{D})^*$. The length of such a predicate will be defined to be the length of its associated w. The second form may be understood intuitively to mean "the S-expression (C w R X) is greater than an atom." Specifically,

$$(G\ p) = \begin{cases} \text{T}, & \text{if } p \text{ is defined and } p = \text{NIL} \\ \text{NIL}, & \text{otherwise.} \end{cases}$$

For example, if $X = (A \cdot (B \cdot C))$, then

$$(G(\text{ATOM}(\text{CDR } X))) = \text{T}$$

$$(G(\text{ATOM}(\text{CDDR } X))) = \text{NIL}$$

$$(G(\text{ATOM}(\text{CDDDR } X))) = \text{NIL}.$$

If $R = \{p_1, p_2, \cdots, p_n\}$ is a set of predicates, define

$$S_R = \{X \mid X \text{ is an } S\text{-expression, } p(x) = \text{T for all } p \in R\}.$$

By convention, if $\varnothing$ is the empty set, $S_\varnothing$ will be defined to be the set of all possible S-expressions. Suppose $R = \{(\text{ATOM}(\text{CAAR } X)), (G(\text{ATOM}(\text{CDR } X)))\}$. Then S_R is the set of all S-expressions with an atom in the CAAR location and a nonatomic S-expression at the CDR location; e.g., $((A \cdot B) \cdot (C \cdot D))$ is in S_R.

If S is a set of S-expressions, define

$$(\text{CAR } S) = \{Y \mid X \in S \text{ and } Y = (\text{CAR } X)\}$$

$$(\text{CDR } S) = \{Y \mid X \in S \text{ and } Y = (\text{CDR } X)\}$$

One can check that, if R is a set of predicates, then another set R' of predicates can be found such that $S_{R'} = (\text{CAR } S_R)$. R' is constructed by 1) deleting from R all predicates with innermost function (ATOM X) or (CDR X) and 2) replacing (CAR X) by X in all remaining predicates. Thus, using the above example R, we can construct $R' = \{(\text{ATOM}(\text{CAR } X))\}$. This leads to the definition of a new function car on sets of predicates. car $(R) = R'$ if $S_{R'} = (\text{CAR } S_R)$. The function cdr is similarly defined.

The general method for finding the set of all possible arguments for each component function is as follows. If S_R is a set of possible arguments to F_i and F_i has, for example, the form

$$(F_i X) = (\text{COND}(p_{i1}(\text{CONS}(F_h X)(F_k X)))$$
$$(p_{i2}(F_j(\text{CAR } X)))$$
$$(p_{i3}(F_m(\text{CAR } X)))),$$

some sets of possible arguments for other component functions can be found. Thus F_h and F_k have $S_{R \cup \{p_{i1}\}}$ as possible arguments. F_j has (CAR X) for all X in $S_{R \cup \{p_{i2}\}}$ as possible arguments, which is written $S_{\text{car}(R \cup \{p_{i2}\})}$. Since $p_{i3} = \text{T}$, F_m will have as possible arguments (CAR X) for all X in $S_{R \cup \{(Gp_{i2})\}}$, which is written $S_{\text{car}(R \cup \{(Gp_{i2})\})}$.

The goal of this computation is to find for each F_i the set Q_i of all sets S_R of possible arguments. The Q_i will be computed in stages $Q_i^{(0)}, Q_i^{(1)}, Q_i^{(2)}, \cdots$ such that $Q_i^{(0)} \subseteq Q_i^{(1)} \subseteq \cdots$, and for some finite k, $Q_i^{(k)} = Q_i$. Initially, $Q_1^{(0)} = \{S_\varnothing\}$ since F_1 can have any S-expression as an argument, and $Q_i^{(0)} = \varnothing$ for $i > 1$. Each $Q_i^{(j+1)}$ is computed by adding to $Q_i^{(j)}$ all new possible argument sets $S_{R'}$ which result from the fact that $S_R \in Q_h^{(j)}$ for some component F_h and that F_h evaluates by calling F_i. The algorithm is as follows.

1. Initialize the $Q_i^{(1)}$ as described above, and set $k = 1$.
2. While $k = 1$ or there is a j such that $Q_j^{(k-1)} \neq Q_j^{(k)}$, do:
   For each i, compute $Q_i^{(k+1)}$ from $Q_1^{(k)}, Q_2^{(k)}, \cdots, Q_n^{(k)}$ as follows, and then increment k:
   Initialize $Q_i^{(k+1)}$ to be equal to $Q_i^{(k)}$.
   If there is an h such that S_R is in $Q_h^{(k)}$, then
   1) if $(F_h X) = (F_i(\text{CAR } X))$, add $S_{\text{car}(R)}$ to $Q_i^{(k+1)}$, if F_h has part $(p_{im}(F_i(\text{CAR } X)))$, $m > 1$, $p_{im} = \text{T}$, add $S_{\text{car}(R \cup \{(Gp_{i(m-1)})\})}$ to $Q_i^{(k+1)}$, if F_h has part $(p_{im}(F_i(\text{CAR } X)))$, $m > 1$, $p_{im} \neq \text{T}$, add $S_{\text{car}(R \cup \{p_{im}\})}$ to $Q_i^{(k+1)}$,
   2) add set S_R, similar to those in 1) to $Q_i^{(k+1)}$ if F_h calls F_i with argument (CDR X),

   and

   3) if $(F_h X) = (\text{CONS}(F_{r_1} X)(F_{r_2} X))$, where either $r_1 = i$ or $r_2 = i$, add S_R to $Q_i^{(k+1)}$, if F_h has part $(p_{im}(\text{CONS}(F_{r_1} X)(F_{r_2} X)))$, $m > 1$, $p_{im} = \text{T}$, add $S_{R \cup \{(Gp_{i(m-1)})\}}$ to $Q_i^{(k+1)}$, if F_n has part $(p_{im}(\text{CONS}(F_{r_1} X)(F_{r_2} X)))$, $m > 1$, $p_{im} \neq \text{T}$, add $S_{R \cup \{p_{im}\}}$ to $Q_i^{(k+1)}$.
3. For each i, $Q_i = Q_i^{(k)}$.

Because the predicates are bounded in length by the longest one found among the components F_i, there are only a finite number of them. So there are only a finite number of sets of such predicates. And because $Q_i^{(j)} \subseteq Q_i^{(j+1)}$, there must be a k such that $Q_i^{(k-1)} = Q_i^{(k)}$ for all i. So the algorithm must halt.

Acknowledgment

The author is extremely grateful to Dr. P. Summers, whose writings and discussions have led to many of the insights described here. D. Smith is also involved in the LISP synthesis effort and has contributed to this work. One of the referees provided helpful suggestions which have been incorporated into the final version of the paper.

References

[1] A. W. Biermann, "Regular LISP programs and their automatic synthesis from examples," Dep. Computer Science, Duke Univ., Durham, NC, Rep. CS-1976-12, June 1976.
[2] ——, "Automatic indexing in program synthesis processes," Dep.

Computer Science, Duke Univ., Durham, NC, Rep. CS-1976-4, June 1975.

[3] ——, "Approaches to automatic programming," in *Advances in Computers*, vol. 15, M. Rubinoff and M. Yovits, Eds. New York: Academic, 1976, pp. 1–63.

[4] A. W. Biermann, R. I. Baum, and F. E. Petry, "Speeding up the synthesis of programs from traces," *IEEE Trans. Comput.*, vol. C-24, no. 2, pp. 122–136, 1975.

[5] A. W. Biermann and R. Krishnaswamy, "Constructing programs from example computations," *IEEE Trans. Software Eng.*, vol. SE-2, Sept. 1976.

[6] L. Blum and M. Blum, "Toward a mathematical theory of inductive inference," *Inform. Contr.*, vol. 28, pp. 125–155, 1975.

[7] J. A. Feldman, "Some decidability results on grammatical inference and complexity," *Inform. Contr.*, vol. 20, no. 3, pp. 244–262, 1972.

[8] M. Gold, "Language identification in the limit," *Inform. Contr.*, vol. 10, no. 5, pp. 447–474, 1967.

[9] C. C. Green *et al.*, "Progress report on program understanding systems," Stanford Artificial Intelligence Laboratory, Stanford, CA, Memo AIM-240, 1974.

[10] S. Hardy, "Synthesis of LISP functions from examples," in *Advance Papers 4th Int. Joint Conf. Artificial Intelligence*, Tbilisi, Georgia, USSR, Sept. 1975, pp. 240–245.

[11] J. McCarthy, P. W. Abrahms, D. J. Edwards, T. P. Hart, and M. I. Levin, *LISP 1.5 Programmer's Manual.* Cambridge, MA: M.I.T., 1962.

[12] D. Shaw, W. Swartout, and C. Green, "Inferring LISP programs from examples," in *Advance Papers 4th Int. Joint Conf. Artificial Intelligence*, Tbilisi, Georgia, USSR, Sept. 1975, pp. 260–267.

[13] L. Siklóssy and D. A. Sykes, "Automatic program synthesis from example problems," in *Advance Papers 4th Int. Joint Conf. Artificial Intelligence*, Tbilisi, Georgia, USSR, Sept. 1975, pp. 268–273.

[14] P. D. Summers, "Program construction from examples," Ph.D. dissertation, Yale Univ., New Haven, CT, 1975.

[15] ——, "A methodology for LISP program construction from examples," *J. Ass. Comput. Mach.*, vol. 24, pp. 161–175, 1977.