IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS, VOL. SMC-9, NO. 5, MAY 1979

A Production Rule Mechanism for Generating LISP Code

ALAN W. BIERMANN, MEMBER, IEEE, AND DOUGLAS R. SMITH

Abstract—**Production rule schemas are given which hold the basic information necessary for coding recursive loops and branches in LISP. Information from the user concerning the desired program is used to instantiate the schemas to yield production rules, and then these rules generate executable code in a strictly syntactic fashion. Emphasis is placed on decomposing the synthesis problem into a hierarchy of tasks which can each be solved by application of a schema. The method is demonstrated by showing how programs can be synthesized from examples of their input–output behaviors.**

I. Introduction

THE COMPLEXITY of an automatic program synthesis system tends to explode as more facilities are added, and the abilities of the designer are heavily taxed in building, debugging, and modifying the system so that it can function reliably. The innumerable special rules and procedures needed to build hierarchies of nested and combined loops and branches with various kinds of variable handling, condition checking, and so forth are enough to confuse the best minds, and methods are needed to systematize and properly modularize this knowledge so that it can be dealt with in an orderly way. This paper will suggest a production rule mechanism for converting an intermediate-level specification of a program into executable code.

Production rule systems have been found to be useful in a number of artificial intelligence projects (see below). The following definition is adapted from Davis and King [11]. A *production rule system* (PRS) is composed of

1) a set of rules of the form $\alpha \Rightarrow \beta$, where α and β are strings of symbols,
2) a data string of symbols,
3) an interpreter or control program.

The interpreter scans the data string and whenever a substring α is found which matches the left-hand side of a rule, let us say $\alpha \Rightarrow \beta$, then the substring α is replaced by β in the data string. The interpreter repeats this process until no more rules are applicable to the data string. The left-hand side of a rule is often called the *nonterminal* and the right-hand side is called the *generated string*.

One variation of this definition of a PRS is to allow rules with parameters called *rule schemas*. These production rule schemas must be instantiated (thus becoming an ordinary rule) before they can be used. In this paper we define production rule schemas which hold the basic information about various programming constructs such as loops and branches. Our approach to program synthesis uses this PRS as follows.

1) The user supplies information about the desired program.
2) This information is transformed into an intermediate-level specification of a program which is used to instantiate some rule schemas.
3) The control program applies the rules to an initial string, and the final string is the requested program.

In a production rule system the knowledge that the synthesis system has about programming is encoded in the rules making the system modular, precise, and easy to understand. Note that a separation of the tasks of code generation and control of the synthesis process is allowed by a PRS. Production rule systems have been used and described in Waterman [28], Davis *et al.* [10], Feigenbaum [12], Lenat [17], and Barstow [2].

Although it would seem that this approach is applicable in many domains, the particular problem area considered here is the generation of LISP programs from examples of their behavior. Thus we will assume that an example input, such as $(A\ B\ C\ D\ E)$, will be given with the output which the desired program is to return, perhaps $(A\ C\ E)$ in this case. The intermediate-level specification here is that a loop is desired which will scan through the input $m = 2$ steps at a time and generate the output along the way. The first step in the generation of the goal program utilizes production rule schemas which have the form

$$\alpha(m, Z) \Rightarrow \beta(m, Z)$$

where m and Z represent information which must be given by the intermediate level specification. Suppose in this example that $m = 2$ and $Z = Z_1$. Then the final code is generated from a known initial string in the usual syntactic way:

$$\text{initial string} \underset{\text{rule 0}}{\Rightarrow} \text{intermediate string} \underset{\text{rule 1}}{\Rightarrow} \text{final code.}$$

Of course, in general, many such rules may be derived from the schemas, and code generation may involve many steps. While this paper will assume that the user specifies the program by giving examples, the approach would seem to be equally valid if some other form of specification were used as long as a translator to the intermediate level is available.

The approach discussed here emphasizes the use of

Manuscript received June 16, 1977; revised December 19, 1978. This work was supported by the National Science Foundation under Grant MCS 74-14445-A01.

The authors are with the Department of Computer Science, Duke University, Durham, NC 27706.

hierarchy to deal with complexity. In the synthesis of programs from examples, a problem reduction technique as described by Nilsson [22] is used to generate the code at the lowest level of the hierarchy, then the next level, and so forth until the task is complete. If specifications from the user were available in a top-down form, then the rules could be used to generate code top down.

The range of programs that the current methodology can handle consists of programs that are composed of arbitrarily deep nestings of loops, sequences of loops, and branches. Also, the programs perform a purely structural mapping from the input to the output so that any semantic information present in the input atoms is ignored. Production rule schemas are introduced for the following programming constructs:

1) teardown loops	(in Section III)
2) straight-line code	(in Section III)
3) buildup loops	(in Section V)
4) if-then	(in Section VII)
5) if-then-else	(in Section VII).

We see this as a step towards a general program synthesis system which would include a more extensive set of rule schemas. Many examples are presented in this paper which show the flexibility of the schemas and their ability to handle deep nestings of loops. Of the schemas presented, schemas 1, 2, and 3 above have been implemented on our experimental system. This system accepts input–output pairs from the user and automatically produces an intermediate-level specification. From the specification, the system instantiates the appropriate schemas and finally produces LISP code. Synthesis time for all examples in this paper is under 2 s. Section VIII gives several examples of the use of the system.

A general survey of automatic programming literature appears in Biermann [4]. One approach to automatic programming that has been explored by Manna and Waldinger [18] is the use of formal logic. A program is specified by a statement in first-order logic, and its code is produced as a byproduct of the proof of the validity of the specification. In more recent work by the same authors [19], the program specification is repeatedly decomposed into subgoals until these subgoals can be satisfied by some program construct or statement. An important aspect of this work is the possibility of building a correctness proof of the program during synthesis and also a proof of termination. Their approach uses rules also, but the rules are designed to be transformations of segments of logical program specifications (i.e., subgoals) into a simpler, more refined form. Their rules encode knowledge about integers, the target language, meanings of constructs in the specification, and target language, besides programming constructs. Our rules, of course, deal only with the latter. Another difference is that our current system is driven by the user's input–output examples rather than by logical statements.

Another major approach to automatic programming involves natural language dialogue between user and machine concerning the program specifications. Recent work in this area has been done by Green [14], Balzer *et al.* [1], Martin *et al.* [20], and Heidorn [16]. These systems tend to be knowledge based, needing not only knowledge about programming and the target language but also a significant amount of knowledge about how to meaningfully accept and output natural language statements. We consider the relatively weak information about program behavior exhibited by examples to be sufficient for our purposes. By avoiding the large problem of natural language dialogue, we can concentrate more on the program synthesis task.

Biermann [3] has studied the synthesis of programs from example computations or traces. A user sits at a computer CRT display and steps the system through a hand calculation. From this information the synthesis system can then find the smallest program in some class which mimics the users trace. This method, as with some of the others described above, suffers from the search time required for building programs of reasonable size. Our current approach attempts to improve the efficiency of such techniques by restricting the search to a few tightly constrained control structures.

The generation of LISP programs from examples has been studied by a number of researchers: Biermann [5], Biermann and Smith [6], Biggerstaff [8], Biggerstaff and Johnson [9], Green *et al.* [13], Hardy [15], Shaw *et al.* [23], Siklóssy and Sykes [24], Smith [25], and Summers [26], [27]. The major point differentiating the current work from previous papers is the use of production rules which make it possible to hierarchically generate nestings and compositions of loops and branches to a high degree. Also a production rule system allows for a separation of the synthesis control structure from the domain knowledge encoded in the rules.

After introducing notation in Section II, Sections III and IV show how production rules can be used to generate looping code nested to any degree. Section V shows how to modify the mechanism to deal with buildup variables, and Section VI gives rules for branching code. Section VII attempts to illustrate how production rules might be used as part of a total program synthesis system for the creation of a wide variety of programs. Section VIII briefly discusses variations in the method that were studied and two programs that were written and run in the development of these results. Many examples of generated programs are given throughout the paper.

II. LISP and Some Notation

The basic LISP data structure used in this paper is the *list* which is written $(X_1 X_2 X_3 \cdots X_n)$ where the X_i's may be LISP atoms or lists. Functions in LISP are written as lists of the form $(f\ X_1\ X_2 \cdots X_n)$ where f is the name of the function and the X_i's are the arguments. The primary LISP functions used in this paper are CONS, CAR, CDR, and ATOM as defined in the LISP literature (McCarthy *et al.* [21]). For example, if X is the list $(A\ B\ C)$, then

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

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

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

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

It will sometimes be necessary to nest the CDR function to a considerable depth. In this case we will write $(\mathrm{CD}^k\mathrm{R}\ X)$ which means

$$(\mathrm{CDR}\ (\mathrm{CDR}\ (\mathrm{CDR}\ \cdots\ (\mathrm{CDR}\ X)\ \cdots)))$$

where there are k different CDR's. $(\mathrm{CD}^0\mathrm{R}\ X)$ will be defined to be simply X.

III. Generating Scanning Functions

This paper will give a number of production rules which can be used for the generation of LISP code. The first rule that we will examine is a rule for generating a lowest level routine. While most of the routines discussed in this paper will do their work by calling other routines, the lowest level routines are the ones which actually construct the output.

Rule 0 (Lowest Level Routine):

$$\underbrace{[P^0_w,\ (X_0\,XL),\ \mathrm{next}]}_{\text{nonterminal}} \Rightarrow \underbrace{(P^0_w\,X_0\,XL) = (\mathrm{CONS}\ (\mathrm{CAR}\ X_0)\ \mathrm{next})}_{\text{generated string}}$$

Instantiated for each use of this rule:

w	identifier for each instantiation,
XL	an unparenthesized list of arguments from higher level routines,
next	the S-expression to which the current result is to be appended.[1]

This rule has two parts, the nonterminal symbol $[P^0_w,\ (X_0\,XL),\ \mathrm{next}]$ and the generated string $(P^0_w\,X_0\,XL) = (\mathrm{CONS}\ (\mathrm{CAR}\ X_0)\ \mathrm{next})$. The nonterminal symbol $[P^0_w,\ (X_0\,XL),\ \mathrm{next}]$ can be interpreted to say essentially that program P^0_w is to be defined with arguments $(X_0\,XL)$ and that the result of its computation is to be appended onto "next." The arrow $\Rightarrow$ means "generates" so that this rule can be understood to say that the nonterminal symbol $[P^0_w,\ (X_0\,XL),\ \mathrm{next}]$ with w, XL, and next properly instantiated will generate the code given. Thus if $[P^0_{23},\ (X_0\,X_4\,X_6),\ \mathrm{NIL}]$ ever appears during a code generation computation, then the following code will be generated:

$$(P^0_{23}\,X_0\,X_4\,X_6) = (\mathrm{CONS}\ (\mathrm{CAR}\ X_0)\ \mathrm{NIL})$$

where $w = 23$, $XL = X_4\,X_6$, and next $=$ NIL. Thus a function P^0_{23} has been defined with parameters X_0, X_4, and X_6, and the value is computed as shown.

Besides using production rules as described above, we will also be using *production rule schemas* which can be partially instantiated to produce a production rule and then further instantiated in the generation of code. An example production rule schema is Schema 1 for generating code which scans an input list while generating the output. Schema 1 will generate code which recursively implements a loop as follows. First, a function P^i_w is defined which checks a loop entry condition and exits if the loop cannot be executed. If the loop is to be executed, then a series of routines $P^{k_1}_{w1}, P^{k_2}_{w2}, P^{k_3}_{w3}, \cdots, P^{k_r}_{wr}$ will be executed sequentially, and then control will return to P^i_w to complete the loop. This is illustrated in Fig. 1.

It will be assumed that each $P^{k_j}_{wj}$ will be called with

[1] Often "next" will be expressed as a call to a LISP function.

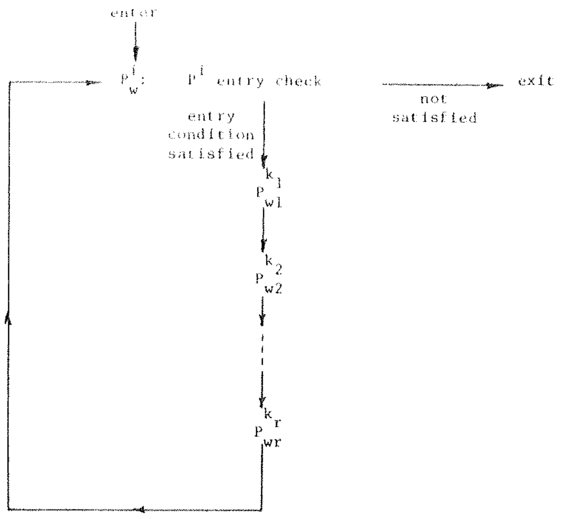

Fig. 1. Flow of control for Schema 1.

argument $(\mathrm{CD}^{h_j}\mathrm{R}\ X)$ as illustrated below. This means that h_j elements will be removed from the front of list argument X before $P^{k_j}_{wj}$ is executed. The usefulness of this capability will be apparent in some of the examples. Finally, when the loop body has been executed, P^i_w will be called with its argument decremented by m units: $(P^i_w(\mathrm{CD}^m\mathrm{R}\ X))$. The *loop decrement* m must be greater than zero in order to avoid an infinite loop.

Schema 1 thus has the following form:

$[P^i_w,\ (X_i\,XL),\ \mathrm{next}] \Rightarrow$

definition of P^i_w which either exits the loop or passes control to $P^{k_1}_{w1}$

nonterminal for generating $P^{k_1}_{w1}$ and then passing control to $P^{k_2}_{w2}$

nonterminal for generating $P^{k_2}_{w2}$ and then passing control to $P^{k_3}_{w3}$

$\vdots$

nonterminal for generating $P^{k_r}_{wr}$ and then returning control to P^i_w.

Hopefully this introduction is enough to make Schema 1 understandable.

Schema 1 (for Generating Scanning Code):

$$\underbrace{[P^i_w,\ (X_i\,XL),\ \mathrm{next}]}_{\text{nonterminal}} \Rightarrow$$

$$\left.\begin{array}{l}
(P^i_w\,X_i\,XL) = \\
\quad (\mathrm{COND}\ ((P^i\ \text{entry check})\ \mathrm{next}) \\
\qquad (T(P^{k_1}_{w1}(\mathrm{CD}^{h_1}\mathrm{R}\ X_i)X_i\,XL))) \\
\quad [P^{k_1}_{w1},\ (X_{k_1}\,X_i\,XL), \\
\qquad (P^{k_2}_{w2}(\mathrm{CD}^{h_2}\mathrm{R}\ X_i)X_i\,XL)] \\
\quad [P^{k_2}_{w2},\ (X_{k_2}\,X_i\,XL), \\
\qquad (P^{k_3}_{w3}(\mathrm{CD}^{h_3}\mathrm{R}\ X_i)X_i\,XL)] \\
\qquad \cdots \\
\quad [P^{k_r}_{wr},\ (X_{k_r}\,X_i\,XL), \\
\qquad (P^i_w(\mathrm{CD}^m\mathrm{R}\ X_i)XL)]
\end{array}\right\}\ \text{generated string.}$$

Instantiated to produce a production rule:

i = rule designation,

(P^i entry check) = a predicate which yields true for the exit condition,

$(P^{k_j}_{wj}, h_j)$, $j = 1, 2, 3, \cdots, r$ is a sequence of calls of subroutines $P^{k_j}_{wj}$ with argument decrements h_j,

m = loop decrement.

Instantiated for each use of the resulting rule:

w	identifier for each instantiation,
XL	an unparenthesized list of arguments supplied by higher level routines,
next	the S-expression to which the current result is to be appended.

Rules of this form generate two things:

1) a LISP definition which uses the conditional COND and
2) a series of nonterminals of the form $[P^{k_j}_{wj}, (X_{k_j} X_i XL), (\cdots)]$ which generates additional code.

The COND function is defined as follows:

$$(\text{COND } (p\ S_1)(T\ S_2)) = \begin{cases} S_1, & \text{if predicate } p \text{ is true} \\ S_2, & \text{otherwise.} \end{cases}$$

Notice that when a function $P^{k_j}_{wj}$ is defined, it is given parameters $X_{k_j} X_i XL$, but when it is called, the substituted arguments are $X_i X_i XL$. This means that the variable X_{k_j} associated with $P^{k_j}_{wj}$ is loaded with the contents of X_i at the time $P^{k_j}_{wj}$ is called. The following example will illustrate this and a number of other points.

Suppose it is desired to generate a program which reads a list $X = (A\ B\ C\ D\ E\ F\ G)$ and outputs $Y = (B\ C\ A\ D\ E\ C\ F\ G\ E)$. This might seem like very complicated behavior, but a clear pattern emerges if the output is graphed as shown in Fig. 2(a). In fact, three points placed in a kind of triangle with respect to each other appear repeatedly in a sloping path to the right. The next section will discuss how to discover this pattern automatically, but for the current purpose it is only necessary to realize this pattern may be analyzed as shown in Fig. 3. Thus the three points are distances $h_1 = 1$, $h_2 = 2$, and $h_3 = 0$ from the highest level of the pattern, and the highest level of the next pattern occurrence begins $m = 2$ units lower than the first pattern.

The synthesis of the desired program proceeds by first building a set of production rules for generating code and then by expanding the rules to obtain the code. The first such production rule will be Rule 0 for generating lowest level routines P^0_w. This rule, which is given the designation $i = 0$, will generate the LISP code which produces the output of Fig. 2(a). The program synthesis problem thus becomes modified to that shown in Fig. 2(b). That is, a lowest level routine P^0_w is to be called with an argument $(B\ C\ D\ E\ F\ G)$, then a P^0_w is to be called with argument $(C\ D\ E\ F\ G)$, then one will be called with argument $(A\ B\ C\ D\ E\ F\ G)$, and so forth. In other words, a point P^0 at a level corresponding to, let us say, an input C means call P^0_w with argument $(C\ D\ E\ F\ G)$.

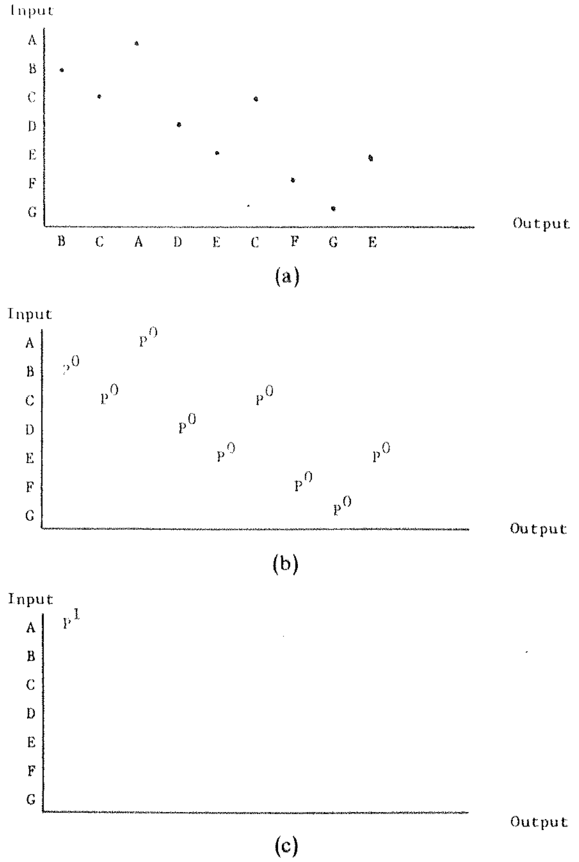

Fig. 2. Problem reduction of first example. (a) Graphing original problem. (b) Sequence of lowest level function calls after creating one production rule. (c) Sequence of function calls required after creating second production rule.

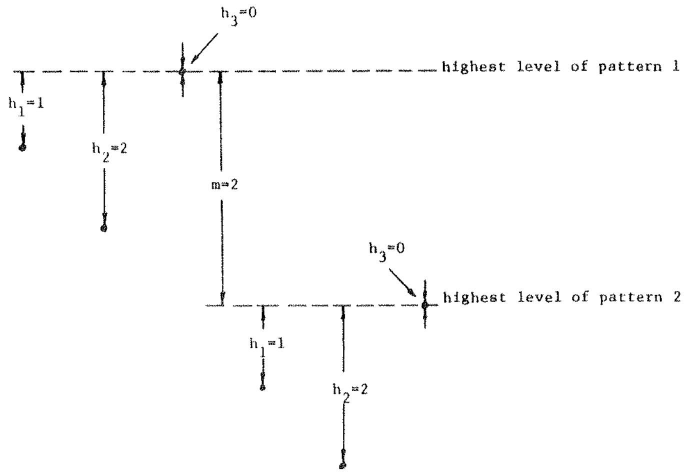

Fig. 3. Analyzing pattern of Fig. 2(a).

A second production rule, designated $i = 1$, will be needed to produce the LISP routine P^1_w to generate the sequence of calls in Fig. 2(b). This rule will be obtained by properly instantiating Schema 1 as indicated above. For example, it is necessary to find the sequence of calls which are to be executed in one pass through the loop. In this case, it is $(P^0_{w1}(\text{CD}^1\text{R } X))$, $(P^0_{w2}(\text{CD}^2\text{R } X))$, and $(P^0_{w3}(\text{CD}^0\text{R } X))$, which will be written for short $(P^0_{w1}, 1)$, $(P^0_{w2}, 2)$, and $(P^0_{w3}, 0)$. Thus enough CDR's are applied to X to account for the decrements h_1, h_2, and h_3 shown in Fig. 3 before P^0_w is executed in each case. Also we note that the next pattern of three points in Fig. 2(b) is $m = 2$ units below the first pattern, and this is the loop decrement mentioned above. Thus the loop will be completed by returning control to the looping routine P^1_w with the argument decremented by 2: $(P^1_w(\text{CD}^2\text{R } X))$. Finally, a predicate called (P^1 entry check) must be constructed which will yield true and cause an exit if any of the above calls would yield an error. That is, if $(P^0_w(\text{CD}^1\text{R } X))$ is executed when X is an atom or if $(P^0_w(\text{CD}^2\text{R } X))$ is executed when X or (CDR X) is an atom, then an error will result. Therefore, (P^1 entry check) is ((ATOM X_1) OR (ATOM (CDR X_1))), which will be written in the paper as (ATOM $(\text{CD}^{(0:1)}\text{R } X_1)$). In general, ($P^i$ entry check) will be (ATOM $(\text{CD}^{(0:j)}\text{R } X_i)$), where $j = \max(h_1, h_2, \cdots, h_r, m) - 1$ and where the notation $(0:j)$ designates the indices 0, 1, 2, $\cdots$, j. The composite of all of these substitutions into Schema 1 results in the following code production rule:

$$[P^1_w, (X_1\, XL), \text{next}] \Rightarrow$$
$$(P^1_w X_1\, XL) = (\text{COND } ((\text{ATOM } (\text{CD}^{(0:1)}\text{R } X_1)) \text{ next})$$
$$(T(P^0_{w1}(\text{CD}^1\text{R } X_1)X_1\, XL)))$$
$$[P^0_{w1}, (X_0 X_1\, XL), (P^0_{w2}(\text{CD}^2\text{R } X_1)X_1\, XL)]$$
$$[P^0_{w2}, (X_0 X_1\, XL), (P^0_{w3}(\text{CD}^0\text{R } X_1)X_1\, XL)]$$
$$[P^0_{w3}, (X_0 X_1\, XL), (P^1_w(\text{CD}^2\text{R } X_1)XL)]$$

Substitutions made to produce this rule:

$$i = 1$$
$$(P^i \text{ entry check}) = (\text{ATOM } (\text{CD}^{(0:1)}\text{R } X_1)).$$

Sequence of subroutine calls for the loop body:

$$(P^{k_1}_{w1}, h_1) = (P^0_{w1}, 1), (P^{k_2}_{w2}, h_2) = (P^0_{w2}, 2),$$

and

$$(P^{k_3}_{w3}, h_3) = (P^0_{w3}, 0).$$
$$m = 2$$

Necessary substitutions for each use of this rule:

w	identifier for each instantiation,
XL	unparenthesized list of arguments from higher level routines,
next	S-expression to which the current result is to be appended.

The creation of this second code generation rule reduces the original problem to that shown in Fig. 2(c). That is, the desired computation can be obtained by calling P^1_w with argument $(A\ B\ C\ D\ E\ F\ G)$. When the problem can be reduced to one function call P^i_w as has been illustrated he e, the desired LISP code can be generated by expanding to termination the nonterminal $[P^i, (X_i), \text{NIL}]$. In the current problem, we expand $[P^1, (X_1), \text{NIL}]$ by making the substitution w = string of length zero, XL = null list, and next = NIL into the above production rule:

$$[P^1, (X_1), \text{NIL}] \Rightarrow$$
$$(P^1 X_1) = (\text{COND } ((\text{ATOM } (\text{CD}^{(0:1)}\text{R } X_1)) \text{ NIL})$$
$$(T(P^0_1(\text{CDR } X_1)X_1)))$$
$$[(P^0_1, (X_0 X_1), (P^0_2(\text{CD}^2\text{R } X_1)X_1)]$$
$$[P^0_2, (X_0 X_1), (P^0_3(\text{CD}^0\text{R } X_1)X_1)]$$
$$[P^0_3, (X_0 X_1), (P^1(\text{CD}^2\text{R } X_1))].$$

This generation has produced three nonterminals which can be expanded using Rule 0:

$$[P^0_1, (X_0 X_1), (P^0_2(\text{CD}^2\text{R } X_1)X_1)] \Rightarrow$$
$$(P^0_1 X_0 X_1) = (\text{CONS } (\text{CAR } X_0)(P^0_2(\text{CD}^2\text{R } X_1)X_1))$$
$$[P^0_2, (X_0 X_1), (P^0_3(\text{CD}^0\text{R } X_1)X_1)] \Rightarrow$$
$$(P^0_2 X_0 X_1) = (\text{CONS } (\text{CAR } X_0)(P^0_3(\text{CD}^0\text{R } X_1)X_1))$$
$$[P^0_3, (X_0 X_1), (P^1(\text{CD}^2\text{R } X_1))] \Rightarrow$$
$$(P^0_3 X_0 X_1) = (\text{CONS } (\text{CAR } X_0)(P^1(\text{CD}^2\text{R } X_1)))$$

The solution to the original problem is the collection of the above code:

$$(P^1 X_1) = (\text{COND } (((\text{ATOM } X_1) \text{ OR } (\text{ATOM } (\text{CDR } X_1))) \text{ NIL})$$
$$(T(P^0_1(\text{CDR } X_1)X_1)))$$
$$(P^0_1 X_0 X_1) = (\text{CONS } (\text{CAR } X_0)(P^0_2(\text{CDDR } X_1)X_1))$$
$$(P^0_2 X_0 X_1) = (\text{CONS } (\text{CAR } X_0)(P^0_3 X_1 X_1))$$
$$(P^0_3 X_0 X_1) = (\text{CONS } (\text{CAR } X_0)(P^1(\text{CDDR } X_1))).$$

The important point to be noted about Schema 1 is not that it can generate code for a loop but that it can produce arbitrary nestings of loops in widely varying situations. The additional examples of this paper should partially illustrate its power.

In order to demonstrate the operation of Schema 1 in a deeper hierarchy, let us generate a program which inputs a list such as $X = (A\ B\ C\ D)$ and outputs the result of two nested scans $Y = (A\ B\ C\ D\ B\ C\ D\ C\ D\ D)$. Here the desired function is graphed as shown in Fig. 4(a), and Rule 0 (designated $i = 0$) is introduced to reduce the problem to that shown in Fig. 4(b). Here Schema 1 can be instantiated to produce production rule $i = 1$ which accounts for the four sequential scans of the list and which reduces the problem as indicated in Fig. 4(c):

$$[P^1_w, (X_1\, XL), \text{next}] \Rightarrow$$
$$(P^1_w X_1\, XL) = (\text{COND } ((\text{ATOM } X_1) \text{ next})$$
$$(T(P^0_{w1}(\text{CD}^0\text{R } X_1)X_1\, XL)))$$
$$[P^0_{w1}, (X_0 X_1\, XL), (P^1_w(\text{CD}^1\text{R } X_1)XL)].$$

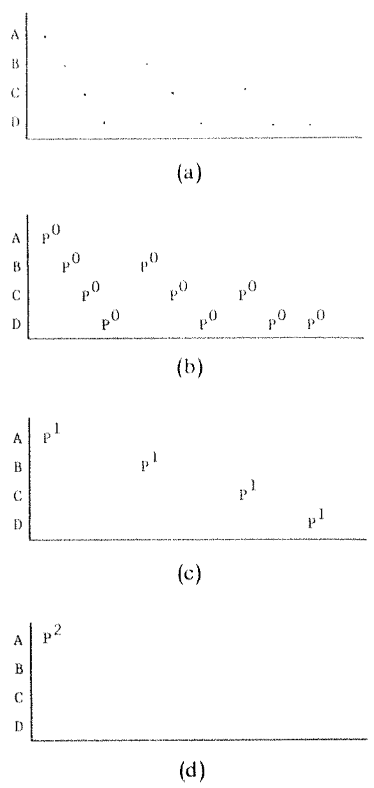

Fig. 4. Reducing problem with three levels of hierarchy. (a) Original problem. (b) After rule $i = 0$ is introduced. (c) After rules $i = 0$ and $i = 1$ are introduced. (d) After rules $i = 0$, $i = 1$, and $i = 2$ are introduced.

Instantiated to produce this rule:

$$i = 1$$

$$(P^i \text{ entry check}) = (\text{ATOM } X_1)$$

$$\text{Sequence of calls: } (P^{k_1}_{w1}, h_1) = (P^0_{w1}, 0)$$

$$m = 1.$$

To be instantiated for each use of this rule: w, XL, and next as described above. Then Schema 1 can be instantiated again to produce another production rule $i = 2$ which reduces the problem to the single call given in Fig. 4(d):

$$[P^2_w, (X_2 XL), \text{next}] \Rightarrow$$
$$(P^2_w X_2 XL) = (\text{COND } ((\text{ATOM } X_2) \text{ next})$$
$$(T(P^1_{w1}(\text{CD}^0\text{R } X_2)X_2 XL)))$$
$$[P^1_{w1}, (X_1 X_2 XL), (P^2_w(\text{CDR } X_2)XL)].$$

Instantiated to produce this rule:

$$i = 2$$

$$(P^i \text{ entry check}) = (\text{ATOM } X_2)$$

$$\text{Sequence of calls: } (P^{k_1}_{w1}, h_1) = (P^1_{w1}, 0)$$

$$m = 1.$$

To be instantiated for each use of this rule: w, XL, and next as described above. Introduction of these three rules has reduced the problem to the single call P^2 so that the code may be produced by expanding $[P^2, (X_2), \text{NIL}]$. Using rule $i = 2$, we substitute w = string of length zero, XL = null list, and next = NIL to obtain

$$[P^2, (X_2), \text{NIL}] \Rightarrow$$
$$(P^2 X_2) = (\text{COND } ((\text{ATOM } X_2) \text{ NIL})(T(P^1_1 X_2 X_2)))$$
$$[P^1_1, (X_1 X_2), (P^2(\text{CDR } X_2))].$$

This introduces one nonterminal which can be expanded using rule $i = 1$. The necessary substitution is $w = 1$, $XL = X_2$, and next $= (P^2(\text{CDR } X_2))$:

$$[P^1_1, (X_1 X_2), (P^2(\text{CDR } X_2))] \Rightarrow$$
$$(P^1_1 X_1 X_2) = (\text{COND } ((\text{ATOM } X_1)(P^2(\text{CDR } X_2)))$$
$$(T(P^0_{11} X_1 X_1 X_2)))$$
$$[P^0_{11}, (X_0 X_1 X_2), (P^1_1(\text{CDR } X_1)X_2)].$$

Another nonterminal has been introduced which can be expanded using rule $i = 0$ with substitutions $w = 11$, $XL = X_1 X_2$ and next $= (P^1_1(\text{CDR } X_1)X_2)$:

$$[P^0_{11}, (X_0 X_1 X_2), (P^1_1(\text{CDR } X_1)X_2)] \Rightarrow$$
$$(P^0_{11} X_0 X_1 X_2) = (\text{CONS } (\text{CAR } X_0)(P^1_1(\text{CDR } X_1)X_2)).$$

The final code thus becomes

$$(P^2 X_2) = (\text{COND } ((\text{ATOM } X_2) \text{ NIL})$$
$$(T(P^1_1 X_2 X_2)))$$
$$(P^1_1 X_1 X_2) = (\text{COND } ((\text{ATOM } X_1)(P^2(\text{CDR } X_2)))$$
$$(T(P^0_{11} X_1 X_1 X_2)))$$
$$(P^0_{11} X_0 X_1 X_2) = (\text{CONS } (\text{CAR } X_0)(P^1_1(\text{CDR } X_1)X_2)).$$

This section has introduced two schemas (Rule 0 and Schema 1) for program creation and has illustrated their use in the hierarchical generation of two programs. The following section will give an algorithm which finds patterns of the type shown in the above figures and then utilizes these schemas to construct programs.

Before concluding this section, another schema will be introduced which will be used for generating straight line code. This is only a slight modification of Schema 1.

Schema 2 (for Generating Straight Line Code):

$$[P^i_w, (X_i XL), \text{next}] \Rightarrow$$
$$(P^i_w X_i XL) = (\text{COND } ((P^i \text{ entry check}) \text{ next})$$
$$(T(P^{k_1}_{w1}(\text{CD}^{h_1}\text{R } X_i)X_i XL)))$$
$$[P^{k_1}_{w1}, (X_{k_1} X_i XL), (P^{k_2}_{w2}(\text{CD}^{h_2}\text{R } X_i)X_i XL)]$$
$$[P^{k_2}_{w2}, (X_{k_2} X_i XL), (P^{k_3}_{w3}(\text{CD}^{h_3}\text{R } X_i)X_i XL)]$$
$$\vdots$$
$$[P^{k_r}_{wr}, (X_{k_r} X_i XL), \text{next}].$$

Instantiated to produce a production rule:

i = rule designation

(P^i entry check) = a predicate which yields true for the exit condition

$(P^{k_j}, h_j), j = 1, 2, \cdots, r$ is a sequence of calls of subroutines P^{k_j} with argument decrements h_j

m = loop decrement.

Instantiated for each use of the resulting rule: w, XL, and next as described above.

IV. A Synthesis Algorithm

Referring to Fig. 2(b), the triangular pattern of routine calls is seen to appear three times, each one being $m = 2$ units below its neighbor to the left and the same distance above its neighbor to the right. The question arises as to how many repetitions of a pattern must occur before one can infer that a loop should be constructed. We will assume that either the system designer or the system user has set a parameter M which specifies the required number of repetitions and will construct our synthesis algorithm to build a loop whenever M or more repetitions of a pattern appear. It will always be assumed that M is greater than one.

A set of j sequential routine calls as in Fig. 2(b) will be called an *M segment group* if $j = r \cdot M$ and the sequence of calls has the following form for some X:

$$(P^{k_1}(\mathrm{CD}^{h_1}\mathrm{R}\ X)), (P^{k_2}(\mathrm{CD}^{h_2}\mathrm{R}\ X), \cdots, (P^{k_r}(\mathrm{CD}^{h_r}\mathrm{R}\ X)),$$
$$(P^{k_1}(\mathrm{CD}^{h_1+m}\mathrm{R}\ X)), (P^{k_2}(\mathrm{CD}^{h_2+m}\mathrm{R}\ X)),$$
$$\cdots, (P^{k_r}(\mathrm{CD}^{h_r+m}\mathrm{R}\ X)),$$
$$(P^{k_1}(\mathrm{CD}^{h_1+2m}\mathrm{R}\ X)), (P^{k_2}(\mathrm{CD}^{h_2+2m}\mathrm{R}\ X)),$$
$$\cdots, (P^{k_r}(\mathrm{CD}^{h_r+2m}\mathrm{R}\ X)),$$
$$\vdots$$
$$(P^{k_1}(\mathrm{CD}^{h_1+(M-1)m}\mathrm{R}\ X)), \cdots, (P^{k_r}(\mathrm{CD}^{h_r+(M-1)m}\mathrm{R}\ X)).$$

Thus for $M = 3$, the M segment group in Fig. 1(b) would be

$$(P^0(\mathrm{CD}^1\mathrm{R}\ X)), (P^0(\mathrm{CD}^2\mathrm{R}\ X)), (P^0(\mathrm{CD}^0\mathrm{R}\ X)),$$
$$(P^0(\mathrm{CD}^3\mathrm{R}\ X)), (P^0(\mathrm{CD}^4\mathrm{R}\ X)), (P^0(\mathrm{CD}^2\mathrm{R}\ X)),$$
$$(P^0(\mathrm{CD}^5\mathrm{R}\ X)), (P^0(\mathrm{CD}^6\mathrm{R}\ X)), (P^0(\mathrm{CD}^4\mathrm{R}\ X)).$$

After an M segment group is discovered, the synthesis mechanism performs a *loop reduction* as is illustrated by the transition from Fig. 2(b) to Fig. 2(c). Specifically, a production rule is generated using Schema 1 unless one already exists to do the job, and all of the subroutine calls accounted for by this rule are removed and replaced by the associated routine name.

There are other kinds of reduction. Thus a *lowest level reduction* will refer to the conversion of the original problem into a sequence of calls of lowest level routines. A *straight line reduction* will refer to the application of Schema 2 to reduce a sequence of calls to just one call.

Before giving Algorithm 1 for program synthesis, it may be helpful to illustrate the method for finding repeated patterns in a string. Suppose that $M = 3$, so that we are looking for three repetitions of some pattern in some example string *ABCBCBCD*, for example. The method will be to advance an index j across the string examining the last three, six, or nine, etc., symbols looking for three sequential and identical substrings. If $j = 3$, the last three symbols are *ABC*, which is not what is being searched for. As j increases, additional strings are examined: $j = 4$, *BCB*; $j = 5$, *CBC*; $j = 6$, *BCB* and *ABCBCB*. Finally at $j = 7$, the last three symbols are *CBC*, and the last six are *BCBCBC*, which satisfy the requirement of three sequential and identical substrings. This is the method that Algorithm 1 uses to find M segment groups.

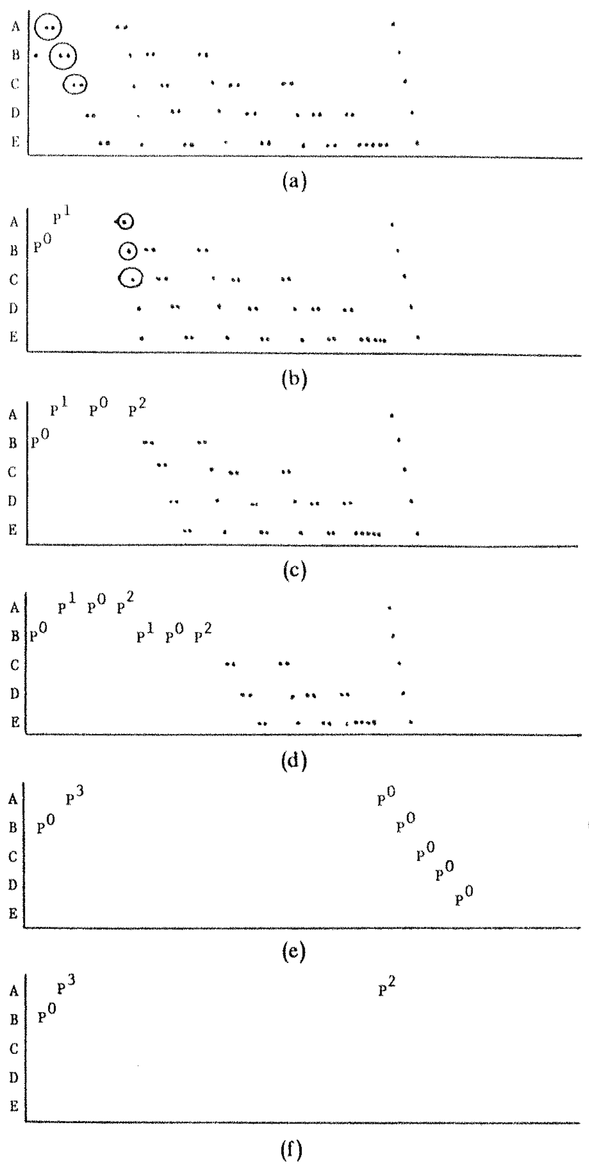

Fig. 5. Sequence of problem reductions. (a) Problem. (b) After P^1 reduction. (c) After P^2 reduction. (d) After two more reductions. (e) After P^3 reduction. (f) Discovering another P^2 reduction.

Algorithm 1:

1) Do all possible lowest level reductions to produce a sequence of subroutine calls using Rule 0.

2) For $j = M$ to the length of the sequence of subroutine calls:
   For $k = M$ to j incrementing by M each time:
      If (starting from the jth call and moving backwards toward the beginning) the last k calls form an M segment group, then do the associated loop reduction using Schema 1 and go to 2.
3) If the current sequence of routine calls has length greater than one, do a straight line reduction using Schema 2 to reduce the sequence to one call.
4) Assuming P^i is the last remaining subroutine call, generate the code by expanding $[P^i, (X_i), \text{NIL}]$.

Assuming that $M = 3$, we will examine the performance of Algorithm 1 on the problem of Fig. 5(a). Step 1 changes all the graph points to P^0's as in previous examples, but for simplicity this was not graphed. Step 2 proceeds to increment j looking for an M segment group when at $j = 7, k = 6$ it finds one as indicated by the circled points. So Schema 1 is employed to create a rule which accounts for ten points and reduces the problem as shown in Fig. 5(b). Step 2 begins again, and this time at $j = 6$, $k = 3$ it finds another loop reduction as shown in Fig. 5(c). The next two reductions found by step 2 (Fig. 5(d)) can be accounted for by rules already created, so no new rules have been generated. A later entry of step 2 finds the repeated pattern P^1, P^0, P^2 and produces the reduction of Fig. 5(e). Notice that P^3 accounts for all but the last five points in the graph. Finally, step 2 accounts for the last five points with P^2 yielding the problem of Fig. 5(f) which can be reduced to one point P^5 by a straight line reduction (step 3). Thus the final code can be generated by expanding the nonterminal $[P^5, (X_5), \text{NIL}]$, and the result is

$$(P^4 X_4) = (\text{COND }((\text{ATOM } X_4)\text{ NIL})$$
$$(T(P^0_1(\text{CDR } X_4)X_4)))$$
$$(P^0_1 X_0 X_4) = (\text{CONS }(\text{CAR } X_0)(P^3_2 X_4 X_4))$$
$$(P^3_2 X_3 X_4) = (\text{COND }((\text{ATOM } X_3)(P^2_3 X_4 X_4))$$
$$(T(P^1_{21} X_3 X_3 X_4)))$$
$$(P^2_3 X_2 X_4) = (\text{COND }((\text{ATOM } X_2)\text{ NIL})$$
$$(T(P^0_{31} X_2 X_2 X_4)))$$
$$(P^1_{21} X_1 X_3 X_4) = (\text{COND }((\text{ATOM } X_1)(P^0_{22} X_3 X_3 X_4))$$
$$(T(P^0_{211} X_1 X_1 X_3 X_4)))$$
$$(P^0_{22} X_0 X_3 X_4) = (\text{CONS }(\text{CAR } X_0)(P^2_{23} X_3 X_3 X_4))$$
$$(P^2_{23} X_2 X_3 X_4) = (\text{COND }((\text{ATOM } X_2)$$
$$(P^3_2(\text{CDR } X_3)X_4))$$
$$(T(P^0_{231} X_2 X_2 X_3 X_4)))$$
$$(P^0_{31} X_0 X_2 X_4) = (\text{CONS }(\text{CAR } X_0)(P^2_3(\text{CDR } X_2)X_4))$$
$$(P^0_{211} X_0 X_1 X_3 X_4) = (\text{CONS }(\text{CAR } X_0)(P^0_{212} X_1 X_1 X_3 X_4))$$
$$(P^0_{212} X_0 X_1 X_3 X_4) = (\text{CONS }(\text{CAR } X_0)(P^1_{21}(\text{CDR } X_1)X_3 X_4))$$
$$(P^0_{231} X_0 X_2 X_3 X_4) = (\text{CONS }(\text{CAR } X_0)(P^2_{23}(\text{CDR } X_2)X_3 X_4))$$

The execution time for a synthesis algorithm is sometimes so great as to severely limit its usefulness. For Algorithm 1 most of the running time occurs in step 2. Step 1 is actually included only as a conceptual aid to the reader of this paper and need never be executed on a real system. Its only effect is to generate rule $i = 0$, which is known *a priori*, and to label the points on the graph as calls to P^0. Steps 3 and 4 will have running times approximately proportional to the length of the target program. Step 2 involves two major costs: the indexing and testing in the inner loop and the cost of the reduction for each M segment group found. It is shown in the Appendix that the number of comparisons required to generate a complete program is proportional to $NL^3(M + 1)^2$, where N is the number of looping subroutines and L is the length of the largest one. The time required for the reduction will be primarily absorbed by the running of the generated loop routine to see what points it accounts for. In summary, it appears that the time required to create a program from an example input–output pair will be approximately proportional to $NL^3(M + 1)^2$ plus the time required to execute the target program to generate the example output.

V. Buildup Variables

While the previous sections show how to generate code to repeatedly scan an input list in generating an output list, it may be desired to build up a list on a *buildup variable* and return its contents as the results. For example, one can scan an input list and create its reversal by appending each item to the front of a buildup list as it is encountered. A schema for generating this kind of code can be obtained by slightly modifying Schema 1 as follows.

Schema 3 (for Generating Buildup Code):

$$[P^i_w, (X_i XL), \text{next}] \Rightarrow$$
$$(P^i_w X_i XL\ Y) = (\text{COND }((P^i \text{ entry check})Y)$$
$$(T(P^i_w(\text{CD}^m\text{R } X_i)XL-$$
$$(P^{k_1}_{w1}(\text{CD}^{h_1}\text{R } X_i)X_i XL\ Y))))$$
$$[P^{k_1}_{w1}, (X_{k_1} X_i XL\ Y), (P^{k_2}_{w2}(\text{CD}^{h_2}\text{R } X_i)X_i XL\ Y)]$$
$$[P^{k_2}_{w2}, (X_{k_2} X_i XL\ Y), (P^{k_3}_{w3}(\text{CD}^{h_3}\text{R } X_i)X_i XL\ Y)]$$
$$\cdots$$
$$[P^{k_r}_{wr}, (X_{k_r} X_i XL\ Y), Y].$$

Instantiated to produce a production rule:

i = rule designation

(P^i entry check) = a predicate which yields true for the exit condition

$(P^{k_j}_{wj}, h_j), j = 1, 2, \cdots, r$ is a sequence of calls of subroutines $P^{k_j}_{wj}$ with argument decrements h_j

m = loop decrement.

Instantiated for each use of the resulting rule: w, XL, and next as defined above. Y means include the buildup variable Y here if this routine is not nested below some other buildup

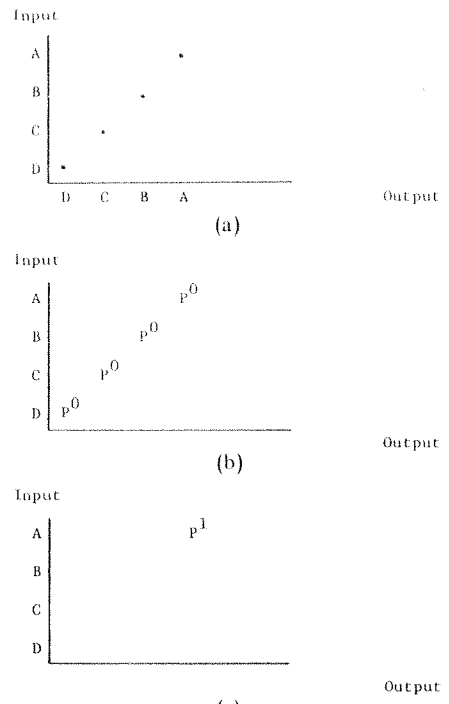

Fig. 6. Problem reduction for list reversal program. (a) Input–output information. (b) After lowest level reduction. (c) After loop reduction.

routine which has already introduced a buildup variable, and $XL-$ means remove any buildup variable in XL.

Fig. 6 shows a utilization of Schema 3 to generate the reverse routine. Fig. 6(a) gives the example input–output pair and Fig. 6(b) and (c) shows the reduced problem after utilizing Rule 0 and Schema 3. The proper instantiation of Schema 3 yields the following rule:

$$[P^1_w, (X_1 XL), \text{next}] \Rightarrow$$
$$(P^1_w X_1 XLY) = (\text{COND}\ ((\text{ATOM}\ X_1)Y)$$
$$(T(P^1_w(\text{CDR}\ X_1)XL-$$
$$(P^0_{w1} X_1 X_1 XLY))))$$
$$[P^0_{w1}, (X_0 X_1 XLY), Y].$$

Substitutions made to produce this rule:

$i = 1$

$(P^i \text{ entry check}) = (\text{ATOM}\ X_1)$

Sequence of subroutine calls: $(P^0_{w1}, 0)$

$m = 1.$

The program is generated from an expansion of the nonterminal $[P^1, (X_1), \text{NIL}]$:

$$(P^1 X_1 Y) = (\text{COND}\ ((\text{ATOM}\ X_1)Y)$$
$$(T(P^1(\text{CDR}\ X_1)(P^0_1 X_1 X_1 Y))))$$
$$(P^0_1 X_0 X_1 Y) = (\text{CONS}\ (\text{CAR}\ X_0)Y).$$

Then if it is desired to reverse list Z, one uses P^1 with Y initialized to NIL: $(P^1 Z\ \text{NIL})$ will yield the reversal of Z.

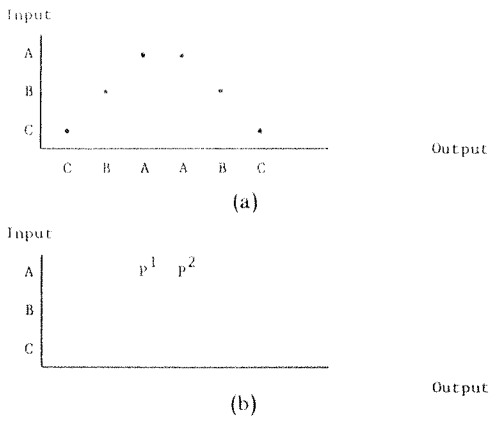

Fig. 7. Problem involving both buildup and teardown reductions. (a) Problem. (b) After buildup and teardown reductions.

It turns out that all of the above Schemas 1, 2, and 3 have a shortcoming which we will now examine and then repair. The problem appears when *teardown* routines of the type generated by Schema 1 are mixed with buildup routines. This will be illustrated by attempting the synthesis of Fig. 7(a) which includes both buildup and teardown segments in the output. Application of Schema 3 to reduce the first half of the output yields the rule given above. Application of Schema 1 reduces the second half and yields the following rule:

$$[P^2_w, (X_2 XL), \text{next}] \Rightarrow$$
$$(P^2_w X_2 XL) = (\text{COND}\ ((\text{ATOM}\ X_2)\ \text{next})$$
$$(T(P^0_{w1} X_2 X_2 XL)))$$
$$[P^0_{w1}, (X_0 X_2 XL), (P^2_w(\text{CDR}\ X_2)XL)]$$

Substitutions to produce this rule:

$i = 2$

$(P^i \text{ entry check}) = (\text{ATOM}\ X_2)$

Sequence of subroutine calls: $(P^0_{w1}, 0)$

$m = 1.$

These reductions yield the problem of Fig. 7(b), where Schema 2 is now applicable to generate this rule:

$$[P^3_w, (X_3 XL), \text{next}] \Rightarrow$$
$$(P^3_w X_3 XL) = (\text{COND}\ ((\text{ATOM}\ X_3)\ \text{next})$$
$$(T(P^1_{w1} X_3 X_3 XL)))$$
$$[P^1_{w1}, (X_1 X_3 XL), (P^2_{w2} X_3 X_3 XL)]$$
$$[P^2_{w2}, (X_2 X_3 XL), \text{next}].$$

This reduces the problem to one point on the graph, which means the correct code should be generated from $[P^3, (X_3), \text{NIL}]$. Examining the resulting code, one finds that P^3 properly calls P^1_1 but that P^2_2 is never entered. A mechanism must be included to add a result to the right-hand end of a buildup segment, and the proper way is to put that result on

P_1^1's buildup variable before P_1^1 begins its computation. The desired code is

$$(P^3 X_3) = (\text{COND } ((\text{ATOM } X_3) \text{ NIL})$$
$$(T(P_1^1 X_3 X_3 (P_2^2 X_3 X_3))))$$
$$(P_1^1 X_1 X_3 Y) = (\text{COND } ((\text{ATOM } X_1) Y)$$
$$(T(P_1^1(\text{CDR } X_1) X_3 (P_{11}^0 X_1 X_1 X_3 Y))))$$
$$(P_{11}^0 X_0 X_1 X_3 Y) = (\text{CONS } (\text{CAR } X_0) Y)$$
$$(P_2^2 X_2 X_3) = (\text{COND } ((\text{ATOM } X_2) \text{ NIL})$$
$$(T(P_{21}^0 X_2 X_2 X_3)))$$
$$(P_{21}^0 X_0 X_2 X_3) = (\text{CONS } (\text{CAR } X_0)(P_2^2(\text{CDR } X_2) X_3)).$$

Notice that neither the buildup routine P_1^1 nor its subroutine P_{11}^0 ever transfers control to the teardown routine P_2^2. However, when P_1^1 is called in the second line of P^3, its buildup variable Y has been initialized with the result of P_2^2.

In order to obtain this behavior, all subroutine calls in Schemas 1, 2, and 3 of the form $(P_{wj}^{k_j}(\text{CD}^{h_j}\text{R } X_i) X_i XL)$ are now written as (call $(P_{wj}^{k_j}, h_j), \cdots, (P_{wr}^{k_r}, h_r)$; next), which means

1) $(P_{wj}^{k_j}(\text{CD}^{h_j}\text{R } X_i) X_i XL)$, if $P_{wj}^{k_j}$ is a teardown routine,
2) $(P_{wj}^{k_j}(\text{CD}^{h_j}\text{R } X_i) X_i XL\text{--}\ (P_{w(j+1)}^{k_{(j+1)}} X_i X_i XL))$, if $P_{wj}^{k_j}$ is buildup and $P_{w(j+1)}^{k_{(j+1)}}$ is teardown,
3) $(P_{wj}^{k_j}(\text{CD}^{h_j}\text{R } X_i) X_i XL\text{--}\ (P_{w(j+1)}^{k_{(j+1)}}(\text{CD}^{h_{(j+1)}}\text{R } X_i) X_i XL\text{--}$ $(P_{w(j+2)}^{k_{(j+2)}}(\text{CD}^{h_{(j+2)}}\text{R } X_i) X_i XL)))$, if $P_{wj}^{k_j}$ and $P_{w(j+1)}^{k_{(j+1)}}$ are buildup and $P_{w(j+2)}^{k_{(j+2)}}$ is teardown,

and so forth. A more complete definition of the (call $\cdots$) notation appears in Appendix B along with revised definitions for the three schemas. The modified schemas will be denoted 1A, 2A, and 3A, and will be used in all future examples instead of the ones given above.

These new schemas can now be used together to build arbitrarily complicated concatenations and nestings of buildup and teardown code. This can be partially illustrated by solving the problem of Fig. 8(a) which requires a buildup routine nested within a teardown routine which is nested within a buildup routine. After the P^0 reduction, the lowest level looping characteristic is buildup so P^1 will be generated by Schema 3A:

$$[P_w^1, (X_1 XL), \text{next}] \Rightarrow$$
$$(P_w^1 X_1 XLY) = (\text{COND } ((\text{ATOM } X_1) Y)$$
$$(T(P_w^1(\text{CDR } X_1) XL\text{--}$$
$$(P_{w1}^0 X_1 X_1 XLY))))$$
$$[P_{w1}^0, (X_0 X_1 XLY), Y].$$

Substitutions made to produce this rule:

$i = 1$

(P^i entry check) = (ATOM X_1)

Sequence of subroutines calls: $(P_{w1}^0, 0)$

$m = 1.$

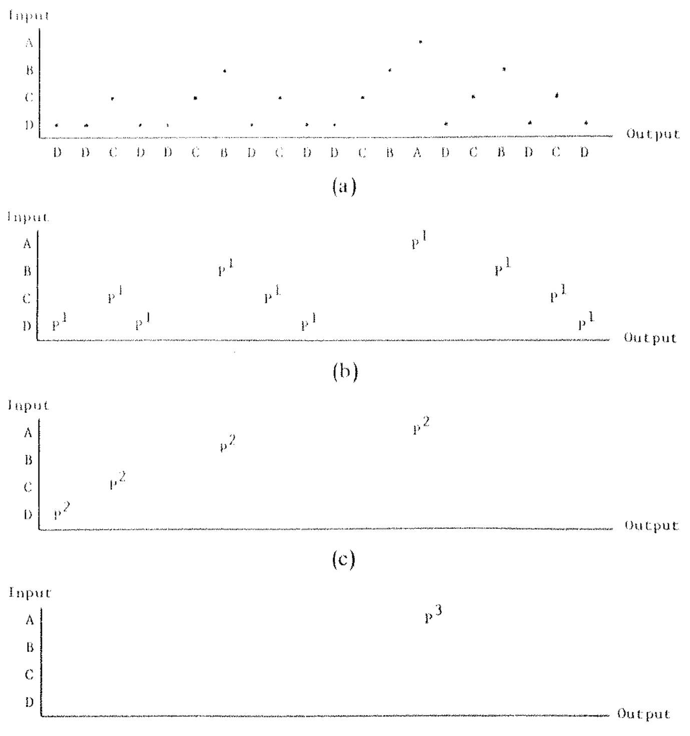

Fig. 8. Generating program which uses both buildup and teardown code. (a) Problem which requires both Schemas 1 and 3. (b) After P^0 and buildup reduction P^1. (c) After teardown reduction P^2. (d) After buildup reduction P^3.

This reduction converts the problem to that shown in Fig. 8(b), where Schema 1A is applicable. In fact, the generated rule is

$$[P_w^2, (X_2 XL), \text{next}] \Rightarrow$$
$$(P_w^2 X_2 XL) = (\text{COND } ((\text{ATOM } X_2) \text{ next})$$
$$(T(P_{w1}^1 X_2 X_2 XL\text{--}$$
$$(P_w^2(\text{CDR } X_2) XL))))$$
$$[P_{w1}^1, (X_1 X_2 XL), (P_w^2(\text{CDR } X_2) XL)].$$

Substitutions to produce this rule:

$i = 2$

(P^i entry check) = (ATOM X_2)

Sequence of subroutine calls: $(P_{w1}^1, 0)$

$m = 1.$

The remaining problem is shown in Fig. 8(c), where Schema 3A is again applicable to generate a rule:

$$[P_w^3, (X_3 XL), \text{next}] \Rightarrow$$
$$(P_w^3 X_3 XLY) = (\text{COND } ((\text{ATOM } X_3) Y)$$
$$(T(P_w^3(\text{CDR } X_3) XL\text{--}$$
$$(P_{w1}^2 X_3 X_3 XLY))))$$
$$[P_{w1}^2, (X_2 X_3 XLY), Y].$$

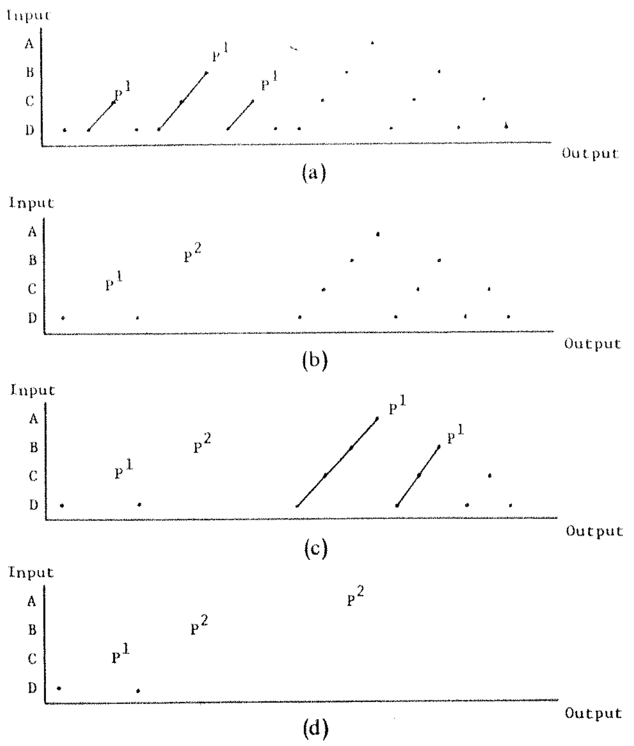

Fig. 9. Sequential reductions discovered by Algorithm 1.

Instantiated to produce this production rule:

$i = 3$

(P^i entry check) $=$ (ATOM X_3)

Sequence of subroutine calls: $(P^2_{w1}, 0)$

$m = 1.$

This reduces the problem to one point so the code may be generated by expanding the nonterminal $[P^3, (X_3), \text{NIL}]$ to obtain the following:

$$(P^3 X_3 Y) = (\text{COND } ((\text{ATOM } X_3) Y)$$
$$(T(P^3(\text{CDR } X_3)(P^2_1 X_3 X_3 Y))))$$
$$(P^2_1 X_2 X_3 Y) = (\text{COND } ((\text{ATOM } X_2) Y)$$
$$(T(P^1_{11} X_2 X_2 X_3 (P^2_1 (\text{CDR } X_2) X_3 Y))))$$
$$(P^1_{11} X_1 X_2 X_3 Y) = (\text{COND } ((\text{ATOM } X_1) Y)$$
$$(T(P^1_{11}(\text{CDR } X_1) X_2 X_3 (P^0_{111} X_1 X_1 X_2 X_3 Y))))$$
$$(P^0_{111} X_0 X_1 X_2 X_3 Y) = (\text{CONS } (\text{CAR } X_0) Y).$$

Algorithm 1 is applicable for discovering both buildup and teardown reductions as long as one understands that the loop reductions can be of either kind. That is, if the M-segment group found by Algorithm 1 proceeds in the forward direction relative to the example input, then Schema 1A for teardown code is selected. If the pattern is in the reverse direction, then Schema 3A for buildup loops is selected. In order to illustrate its operation, the problem of Fig. 8 will be analyzed again concentrating on the operation of Algorithm 1 in finding the appropriate reductions. The total synthesis process turns out to be slightly more complicated than was indicated in the previous section.

Assuming $M = 2$, step 2 of the algorithm begins scanning from left to right looking for two segment groups separated from each other by $m > 0$ vertical units. Step 2 finds two groups of length one and calls for a P^1 reduction as shown near the left end of Fig. 9(a). Then step 2 is reentered two more times giving the additional P^1 reductions in the middle of Fig. 9(a). The next entrance to step 2 discovers the M segment group made up of the middle two P^1 points and calls for a P^2 reduction as shown in Fig. 9(b). Then in Fig. 9(c), two more P^1 reductions are discovered, then the P^2 reduction of 9(d), and finally a P^3 reduction based on the two P^2's that have been found. The P^3 reduction accounts for all of the rest of the points and completes the synthesis.

VI. Rules for Branching Code

The generation of a program which converts $(A\ B\ C(D)(E)F)$ to $(A\ B\ C\ F)$ is quite simple if rules are available for generating conditional branches. This section will introduce two schemas, one for an IF–THEN construction and one for an IF–THEN–ELSE construction, and will give examples to illustrate their use.

Schema 4 (IF–THEN *Code*):

$$[P^i_w, (X_i XL), \text{next}] \Rightarrow$$
$$(P^i_w X_i XL) = (\text{COND } ((P^i \text{ condition check})$$
$$(P^{k_1}_{w1}(\text{CD}^{h_1}\text{R } X_i) X_i XL))$$
$$(T \text{ next}))$$
$$[P^{k_1}_{w1}, (X_{k_1} X_i XL), \text{next}].$$

Instantiated to produce a production rule:

$i =$ rule designation

(P^i condition check)

$(P^{k_1}_{w1}, h_1)$, the routine to be conditionally executed.

Instantiated for each use of the resulting rule:

w, XL, and next as defined above.

Schema 5 (IF–THEN–ELSE *Code*):

$$[P^i_w, (X_i XL), \text{next}] \Rightarrow$$
$$(P^i_w X_i XL) = (\text{COND } ((P^i \text{ condition check})(P^{k_1}_{w1}(\text{CD}^{h_1}\text{R } X_i) X_i XL))$$
$$(T(P^{k_2}_{w2}(\text{CD}^{h_2}\text{R } X_i) X_i XL)))$$
$$[P^{k_1}_{w1}, (X_{k_1} X_i XL), \text{next}]$$
$$[P^{k_2}_{w2}, (X_{k_2} X_i XL), \text{next}].$$

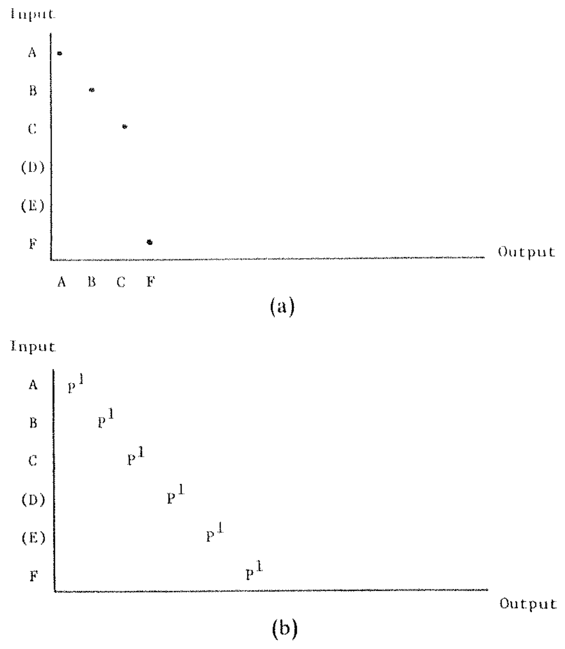

Fig. 10. Synthesis including IF–THEN code. (a) Problem. (b) Extrapolation of loop after P^1 reduction.

Instantiated to produce a production rule:

i = rule designation

(P^i_{w1} condition check)

($P^{k_1}_{w1}$, h_1), the routine to be conditionally executed

($P^{k_2}_{w2}$, h_2), the ELSE routine to be executed if the condition fails.

Instantiated for each use of the resulting rule:

w, XL, and next as defined above.

Suppose the problem of Fig. 10(a) is to be solved. With $M = 2$ or $M = 3$, the loop-finding mechanism of Algorithm 1 will discover a loop, but the execution of that loop will result in an error at the fourth element of the input. Here we suspect that conditional code is appropriate and call a predicate formation mechanism to see whether it can find a difference between the fourth input item and the first three. Predicate formation will not be discussed in this paper since the reader can find a number of reasonable methods described in Biermann [5], Biggerstaff and Johnson [9], Smith [25], Summers [26], and others. In this case, it is clear that an item is to be added to the output if (ATOM (CAR X)) is true. Introduction of a conditional rule P^1 using Schema 4 makes it possible to extrapolate the loop as shown in Fig. 10(b) and to complete the synthesis:

$$(P^2 X_2) = (\text{COND } ((\text{ATOM } X_2) \text{ NIL}) \quad (T(P^1_1 X_2 X_2)))$$

$$(P^1_1 X_1 X_2) = (\text{COND } ((\text{ATOM } (\text{CAR } X_1))(P^0_{11} X_1 X_1 X_2)) \quad (T(P^2(\text{CDR } X_2))))$$

$$(P^0_{11} X_0 X_1 X_2) = (\text{CONS } (\text{CAR } X_0)(P^2(\text{CDR } X_2))).$$

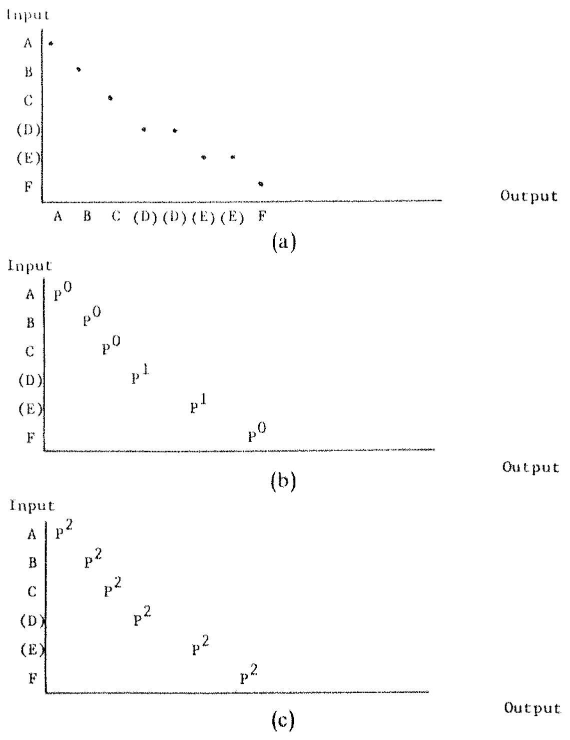

Fig. 11. Synthesis including IF–THEN–ELSE code. (a) Problem. (b) Straight-line code P^1 reduces two pairs of points. (c) IF–THEN–ELSE code prepares for introduction of loop.

If the original problem had been to generate an output $(F\ C\ B\ A)$ from the above input, the synthesis would have been essentially the same except that buildup code would be generated. Clearly because of the generality of the schemas, much more complicated nestings and compositions of code can be generated, as has been demonstrated in earlier sections. The loop-finding mechanism when conditionals are allowed is a much more complicated problem than has been indicated here and deserves additional study.

Suppose a program is to be generated which converts $(A\ B\ C(D)(E)F)$ to $(A\ B\ C(D)(D)(E)(E)F)$. This time a complete IF–THEN–ELSE rule is needed as can be produced from Schema 5, and the synthesis is outlined by Fig. 11. The straight-line schema is used to reduce two pairs of points to single routine calls P^1. Then Schema 5 is used to give the loop generator a regular sequence of calls, and the code is produced in the usual way:

$$(P^3 X_3) = (\text{COND } ((\text{ATOM } X_3) \text{ NIL}) \quad (T(P^2_1 X_3 X_3)))$$

$$(P^2_1 X_2 X_3) = (\text{COND } ((\text{ATOM } (\text{CAR } X_2))(P^0_{11} X_2 X_2 X_3)) \quad (T(P^1_{12} X_2 X_2 X_3)))$$

$$(P^0_{11} X_0 X_2 X_3) = (\text{CONS } (\text{CAR } X_0)(P^3(\text{CDR } X_3)))$$

$$(P^1_{12} X_1 X_2 X_3) = (\text{COND } ((\text{ATOM } X_1)(P^3(\text{CDR } X_3))) \quad (T(P^0_{121} X_1 X_1 X_2 X_3)))$$

$$(P^0_{121} X_0 X_1 X_2 X_3) = (\text{CONS } (\text{CAR } X_0)(P^0_{122} X_1 X_1 X_2 X_3))$$

$$(P^0_{122} X_0 X_1 X_2 X_3) = (\text{CONS } (\text{CAR } X_0)(P^3(\text{CDR } X_3))).$$

VII. PRODUCTION RULES AS A PART OF A LARGER SYSTEM

Of course, the production rules given in this paper do not alone constitute a program synthesis system, but they do form a good set of building blocks. Specifically, they are able to form the loops and branches of a program, but most programs of interest need specially adapted lowest level routines and occasionally some special variable handling. The additional problem-dependent coding must come either directly from the user or from higher level routines in the program synthesis system. Techniques for automatically generating special-purpose lowest level routines are a matter for further research, but a few ideas follow. The purpose of the lowest level routine is to retrieve information from a variable and use it for constructing the output. For all the examples above we have used

$$(\text{CONS } (\text{CAR } X) \text{ OUTPUT}).$$

An automatic mechanism might be constructed which appropriately modifies this routine using a data base of replacement functions.

Suppose, for example, it is desired to add a list of numbers. Then the lowest level production rule should be

$$[P^0_w, (X_0 XL), \text{next}] \Rightarrow (P^0_w X_0 XL) = (\text{PLUS } (\text{CAR } X_0) \text{ next}).$$

Here we have replaced CONS with PLUS. The standard teardown routine will provide the looping behavior. The program is generated from the nonterminal $[P^1, (X_1), 0]$:

$$(P^1 X_1) = (\text{COND } ((\text{ATOM } X_1)\ 0)$$
$$(T(P^0_1 X_1 X_1)))$$
$$(P^0_1 X_0 X_1) = (\text{PLUS } (\text{CAR } X_0)(P^1(\text{CDR } X_1))).$$

Another modification to the lowest level routines would be a more complex retrieval from the variable. For example, suppose it is desired to find all of the values associated with Z in a list. That is, input list $((Z\ B1)(A\ B2)(Z\ B3)(Z\ B4)(B\ B5))$ is to yield $(B1\ B3\ B4)$. Here the lowest level rule should be modified to pick out the second item of each pair that is chosen. It is a small matter to automatically construct CADAR to retrieve $B1$, resulting in the rule:

$$[P^0_w, (X_0 XL), \text{next}] \Rightarrow$$
$$(P^0_w X_0 XL) = (\text{CONS } (\text{CADAR } X_0) \text{ next}).$$

A conditional routine at the next higher level checks whether the first item in a pair is Z:

$$[P^1_w, (X_1 XL), \text{next}] \Rightarrow$$
$$(P^1_w X_1 XL) = (\text{COND } ((\text{EQUAL } (\text{CAAR } X_1)(\text{QUOTE } Z))$$
$$(P^0_{w1} X_1 X_1 XL))$$
$$(T \text{ next}))$$
$$[P^0_{w1}, (X_0 X_1 XL), \text{next}].$$

The next higher routine P^2 is the usual teardown routine, and the code is generated normally. It is possible that only one pair will appear with Z as the first item, and the user might want only that one associated value. Then the lowest level rule would be

$$[P^0_w, (X_0 XL), \text{next}] \Rightarrow$$
$$(P^0_w X_0 XL) = (\text{CADAR } X_0).$$

Suppose a data base holds information of the following kind, where Z means "supports":

(((TYPE CUBE)(NAME $B1$)(Z $B2$)(Z $B5$))
((TYPE BLOCK)(NAME $B2$)(Z $B7$)(COLOR GREEN))
((TYPE CUBE)(NAME $B3$)(Z $B9$))
((TYPE BALL)(NAME $B7$)(COLOR RED))).

That is, object $B1$ is a cube which supports $B2$ and $B5$ and so forth. Suppose it is desired to find all of the objects supported by cubes so that the above input should yield $(B2\ B5\ B9)$. Here the three lowest level production rules would be identical to those of the previous paragraph, and the fourth rule P^3 would be an IF-THEN form that checks for the condition (EQUAL (CADAAR X_3)(QUOTE CUBE)). The highest level routine P^4 would be teardown, and the generated code would be as follows:

$$(P^4 X_4) = (\text{COND } ((\text{ATOM } X_4) \text{ NIL})$$
$$(T(P^3_1 X_4 X_4)))$$
$$(P^3_1 X_3 X_4) = (\text{COND } ((\text{EQUAL } (\text{CADAAR } X_3)(\text{QUOTE CUBE}))$$
$$(P^2_{11}(\text{CAR } X_3) X_3 X_4))$$
$$(T(P^4(\text{CDR } X_4))))$$
$$(P^2_{11} X_2 X_3 X_4) = (\text{COND } ((\text{ATOM } X_2)(P^4(\text{CDR } X_4)))$$
$$(T(P^1_{111} X_2 X_2 X_3 X_4)))$$
$$(P^1_{111} X_1 X_2 X_3 X_4) = (\text{COND } ((\text{EQUAL } (\text{CAAR } X_1)(\text{QUOTE } Z))$$
$$(P^0_{1111} X_1 X_1 X_2 X_3 X_4))$$
$$(T(P^2_{11}(\text{CDR } X_2) X_3 X_4)))$$
$$(P^0_{1111} X_0 X_1 X_2 X_3 X_4) = (\text{CONS } (\text{CADAR } X_0)$$
$$(P^2_{11}(\text{CDR } X_2) X_3 X_4)).$$

Further examples may be found in Biermann and Smith [7].

VIII. VARIATIONS ON THE METHOD

It should be made clear that the mechanism described here is the result of studies over about a two-year period and that a great deal of information has been omitted for the sake of clarity. Many variations were tried on the form of the algorithm and the details of the generated programs, but a lengthy description here does not seem necessary.

For example, a program was written to test variable addition mechanisms in nested loops. This particular system was more economical in its generation of additional variables than the system described in this paper, but inclusion of those economies would have unpleasantly increased the complexity of our notation. This system generated executable code for our local LISP processor, and many programs were created from examples such as the following.

Example A1: This involves three nested buildup routines:

Input: $(A\ B\ C\ D)$.

Output: (*D D D C D D C D C B D D C D C B D C B A*).

Example A2: This involves two nested teardown routines on top of a buildup routine:

Input: (*A B C D*).
Output: (*D C B A D C B D C D D C B D C D D C D D*).

Example A3: This involves a nesting from top to bottom of six loops (teardown, buildup, buildup, teardown, teardown, and buildup):

Input: (*A B C D*).
Output: (*D D D C D D D D C D D D C B D C D D C D 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 C B D C D D C D D D D D C D D D*).

The actual code generated by our system is given for the next example.

Example A4: This involves a teardown routine on top of two nested buildup routines:

Input: (*A B C D*).
Output: (*D D C D C B D C B A D D C D C B D D C D*).

```
(F (LAMBDA (x)(R6 x)))
(R6 (LAMBDA (a)
    (COND
    (T(R5 a)))))
(R5 (LAMBDA (a)
    (COND
    ((ATOM a) NIL)
    (T(R3 a)))))
(R3 (LAMBDA (a)
    (COND
    (T(R4 a(R5 (CDR a)))))))
(R4 (LAMBDA (a v)
    (COND
    ((ATOM a)v)
    (T(R4 (CDR a)(R1 a v))))))
(R1 (LAMBDA (a v)
    (COND
    (T(R2 a v)))))
(R2 (LAMBDA (a v)
    (COND
    ((ATOM a)v)
    (T(R2 (CDR a)(R0 a v))))))
(R0 (LAMBDA (a v)
    (COND
    (T(CONS (CAR a)v))))).
```

The code for each example in this section was generated in less than a second. These and other programs generated by the system may be found in Biermann and Smith [7].

In another series of studies, attempts were made to separate the task segmentation computation from the routine generation portion of the synthesis. That is, instead of the algorithm given in this paper, one might wish to have one program which decomposes the output into parts and a separate program which generates the routines to account for those parts. Many techniques were tried, but none was as successful as the method given here which combines both functions. That is, after step 2 finds an M segment and creates a routine, it runs that routine to see which points it has accounted for. Then it looks for another M segment and generates another routine.

Some examples of the behavior of a system which separated the problem-segmentation and routine-generation phases are given below. The first phase scanned the example output looking for rises and falls which could be accounted for by buildup and teardown routines. The second phase generated the routines, and then control would return to the first phase to look for a new set of segments. Note that in the code that appears here a slightly more complicated conditional was used than described above. Unfortunately, the number of different subroutines tended to be unnecessarily large, and a post-processor would be useful to clean up the code.

Example A5:

Input: (*A B C D E*).
Output: (*E D C B A B C D E D C B A*).

Example A6:

Input: (*A B C D*).
Output: (*A A A B B C C D D D D C B A*).

Example A7:

Input: (*A B C D E F G*).
Output: (*A B C D E F G A C E G B C D E F G B D F C D E F G C E G D E F G D F E F G E G F G F G G*).

The code generated by our system for Example A7 follows:

```
(DEF FLY (LAMBDA (x)(FLY5 x)))
(DEF FLY5 (LAMBDA (a)
    (COND
    (t(FLY4 a)))))
(DEF FLY4 (LAMBDA (a)
    (COND
    ((ATOM (CDDR a))(FLY6 a a))
    (t(FLY1 a a)))))
(DEF FLY2 (LAMBDA (a)
    (COND
    (t(FLY0 a a)))))
(DEF FLY8 (LAMBDA (a)
    (COND
    (t(CONS (CAR a) NIL)))))
(DEF FLY7 (LAMBDA (a b)
    (COND
    (t(CONS (CAR a)(FLY8 (CDR b)))))))
(DEF FLY0 (LAMBDA (a b)
    (COND
    (t(CONS (CAR a)(FLY7 (CDR b)b))))))
(DEF FLY6 (LAMBDA (a b)
    (COND
    ((ATOM a)(FLY2 b))
    (t(FLY9 a b)))))
```

```
(DEF FLY9 (LAMBDA (ab)
    (COND
    (t(CONS (CAR a)(FLY6 (CDR a)b))))))
(DEF FLY3 (LAMBDA (a b)
    (COND
    ((ATOM a)(FLY4 (CDR b)))
    ((ATOM (CDR a))(FLY11 a b))
    (t(FLY10 a b)))))
(DEF FLY11 (LAMBDA (a b)
    (COND
    (t(CONS (CAR a)(FLY4 (CDR b)))))))
(DEF FLY10 (LAMBDA (a b)
    (COND
    (t(CONS (CAR a)(FLY3 (CDDR a)b))))))
(DEF FLY1 (LAMBDA (a b)
    (COND
    ((ATOM a)(FLY3 b b))
    (t(FLY12 a b)))))
(DEF FLY12 (LAMBDA (a b)
    (COND
    (t(CONS (CAR a)(FLY1 (CDR a)b)))))).
```

It appears that other modifications such as increased predicate-building facilities and the ability to handle multiple-argument functions can be built into the system described here without changing its basic structure.

IX. Conclusion

This paper has presented a study of production rules as a systematic mechanism for generating code in automatic programming systems. Production rule systems allow a clearer separation of the synthesis control structure from the code generation and programming knowledge than in other approaches. By localizing knowledge into rule schemas we feel that a program synthesis system will be easier to understand, debug, and extend. With the few schemas we have used here there is no problem of choosing the appropriate schema at a given point. With a larger rule set, the problem of rule choice may indeed become significant. During the first year of this research, code generation rules were written as long lists of English language statements such as, "Any routine that is hierarchically one level below a buildup routine appends its result to the buildup variable." The production rule mechanism has made it possible to reduce these lists to a much simpler and uniform representation which is easier to understand and to implement. All conventions having to do with hierarchy are coded into the mechanism of the nonterminals and the subscripting technique. The variable addition rules are handled by the argument mechanism, and the flow of control is imbedded in the nonterminal conventions and the substitutions for the parameter "next."

No attempt has been made to include enough rules here to comprise a complete program synthesis system. It is hoped, however, that enough rules have been given to illustrate the flexibility and the power of the method. It is also clear that the rules that have been given could be modified in many ways to generate a wider variety of programs and to generate, in some cases, more efficient code. For the sake of readability of this paper, most of these added complexities have been omitted. By constraining the range of allowed programming constructs to a select few we have greatly reduced the amount of search, which is a basic limitation of many approaches to program synthesis. Thus this approach is very efficient over the class of programs that it is designed to handle. The use of production rule systems is a current topic in artificial intelligence projects, and it is not yet possible to say what limits exist on the extendibility or applicability of this technique.

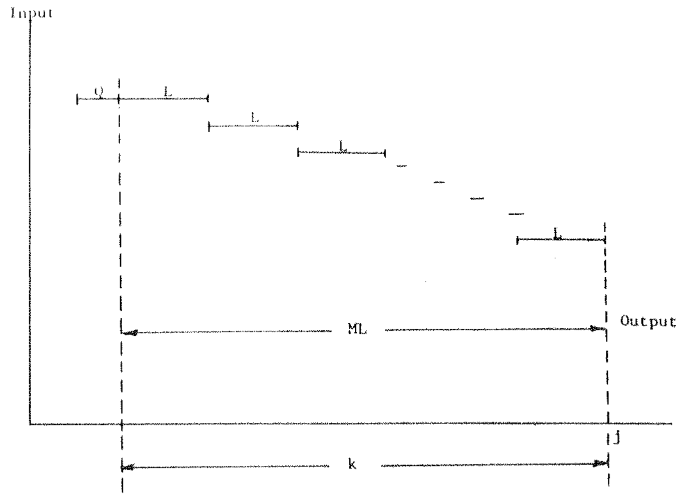

Fig. 12. Algorithm 1 discovering loop.

Appendix A
The Number of Comparisons Required to Find an M-Segment Group

Consider the operation of Algorithm 1 in discovering the longest routine in a particular program. We understand routine *length* L in this discussion to be the number of subroutine calls which are made by the routine. The discovery of the routine comes from finding an M-segment group as shown in Fig. 12. Algorithm 1 finds this group in step 2 by advancing j to the point shown and incrementing k until it equals LM. Q is less than L because we are assuming the M-segment group is associated with the longest routine. So the outside loop of step 2 which advances j is entered approximately $Q + LM$ times. We approximate this with $(M + 1)L > Q + LM$, since $Q < L$. The inner loop is entered L times with $k = M, 2M, \cdots, LM$. Within the inner loop, approximately k comparisons are made each time to check whether an M-segment group has been found. Assume that approximately LM comparisons are made each time which is an upper bound to k. Then the total number of comparisons made is the product of the number of entries into the outer loop times the number of entries into the inner loop times the number of comparisons:

$$(M + 1)L * L * LM < (M + 1)^2L^3.$$

If N routines must be found, the total number of comparisons is less than $N(M + 1)^2L^3$.

Appendix B
Revised Schemas 1, 2, and 3

First, the call notation must be defined. Let $C =$ (call $(P^{k_1}_{w1}, h_1), (P^{k_2}_{w2}, h_2), \cdots, (P^{k_r}_{wr}, h_r)$; next1) where

$$\text{next1} = \begin{cases} (P^i_w, m), & \text{for teardown routines} \\ Y, & \text{for buildup routines} \\ \text{next}, & \text{for straight-line routines.} \end{cases}$$

If $P^{k_1}_{w1}$ is not a buildup routine, then C is defined to be

$$C = (P^{k_1}_{w1}(\text{CD}^{h_1}\text{R } X_i)X_i XL\ Y).$$

If $P^{k_j}_{wj}$ is the leftmost nonbuildup routine for $1 < j \le r$, then C is defined to be

$$\begin{aligned} C = (&P^{k_1}_{w1}(\text{CD}^{h_1}\text{R } X_i)X_i XL- \\ &(P^{k_2}_{w2}(\text{CD}^{h_2}\text{R } X_i)X_i XL- \\ &\quad\vdots \\ &(P^{k_j}_{w(j-1)}(\text{CD}^{h_j-1}\text{R } X_i)X_i XL- \\ &\quad(P^{k_j}_{wj}(\text{CD}^{h_j}\text{R } X_i)X_i XL\ Y))\cdots)). \end{aligned}$$

If all $P^{k_1}_{w1}, P^{k_2}_{w2}, \cdots, P^{k_r}_{wr}$ are buildup, then C is defined to be

$$\begin{aligned} C = (&P^{k_1}_{w1}(\text{CD}^{h_1}\text{R } X_i)X_i XL- \\ &(P^{k_2}_{w2}(\text{CD}^{h_2}\text{R } X_i)X_i XL- \\ &\quad\vdots \\ &(P^{k_r}_{wr}(\text{CD}^{h_r}\text{R } X_i)X_i XL-\ \text{next1})\cdots)). \end{aligned}$$

Finally, if there are no routine calls so that $C =$ (call; next1), the C is defined to equal next1.

Using this notation, it is now possible to give revised definitions of Schemas 1, 2, and 3 which are general enough to allow arbitrary nestings and compositions of routines.

Schema 1A (for Generating Scanning Code)

$$\begin{aligned} &[P^i_w, (X_i XL), \text{next}] \Rightarrow \\ &\quad (P^i_w X_i XL) = (\text{COND } ((P^i \text{ entry check}) \text{ next}) \\ &\qquad\qquad (T(\text{call } (P^{k_1}_{w1}, h_1), \cdots, (P^{k_r}_{wr}, h_r); (P^i_w, m)))) \\ &[P^{k_1}_{w1}, (X_{k_1} X_i XL), (\text{call } (P^{k_2}_{w2}, h_2), \cdots, (P^{k_r}_{wr}, h_r); (P^i_w, m))] \\ &[P^{k_2}_{w2}, (X_{k_2} X_i XL), (\text{call } (P^{k_3}_{w3}, h_3), \cdots, (P^{k_r}_{wr}, h_r); (P^i_w, m))] \\ &\qquad\vdots \\ &[P^{k_r}_{wr}, (X_{k_r} X_i XL), (\text{call}; (P^i_w, m))]. \end{aligned}$$

Schema 2A (for Generating Straight-Line Code)

$$\begin{aligned} &[P^i_w, (X_i XL), \text{next}] \Rightarrow \\ &\quad (P^i_w X_i XL) = (\text{COND } ((P^i \text{ entry check}) \text{ next}) \\ &\qquad\qquad (T(\text{call } (P^{k_1}_{w1}, h_1), \cdots, (P^{k_r}_{wr}, h_r); \text{next}))) \\ &[P^{k_1}_{w1}, (X_{k_1} X_i XL), (\text{call } (P^{k_2}_{w2}, h_2), \cdots, (P^{k_r}_{wr}, h_r); \text{next})] \\ &[P^{k_2}_{w2}, (X_{k_2} X_i XL), (\text{call } (P^{k_3}_{w3}, h_3), \cdots, (P^{k_r}_{wr}, h_r); \text{next})] \\ &\qquad\vdots \\ &[P^{k_r}_{wr}, (X_{k_r} X_i XL), (\text{call}; \text{next})]. \end{aligned}$$

Schema 3A (for Generating Buildup Code)

$$\begin{aligned} &[P^i_w, (X_i XL), \text{next}] \Rightarrow \\ &\quad (P^i_w X_i XL\ Y) = (\text{COND } ((P^i \text{ entry check}) Y) \\ &\qquad\qquad (T(P^i_w(\text{CD}^m\text{R } X_i)XL- \\ &\qquad\qquad\quad (\text{call } (P^{k_1}_{w1}, h_1), \cdots, (P^{k_r}_{wr}, h_r); Y)))) \\ &[P^{k_1}_{w1}, (X_{k_1} X_i XL), (\text{call } (P^{k_2}_{w2}, h_2), \cdots, (P^{k_r}_{wr}, h_r); Y)] \\ &[P^{k_2}_{w2}, (X_{k_2} X_i XL), (\text{call } (P^{k_3}_{w3}, h_2), \cdots, (P^{k_r}_{wr}, h_r); Y)] \\ &\qquad\vdots \\ &[P^{k_r}_{wr}, (X_{k_r} X_i XL), (\text{call}; Y)]. \end{aligned}$$

References

[1] R. Balzer, N. Goldman, and D. Wile, "Informality in program specifications," *Proc. 5th Int. Joint Conf. Artificial Intelligence*, Cambridge, MA, Aug. 1977, pp. 389–397.

[2] D. R. Barstow, "Automatic construction of algorithms and data structures using a knowledge base of programming rules," Ph.D. dissertation, Stanford Univ., 1977.

[3] A. W. Biermann and R. Krishnaswamy, "Constructing programs from example computations," *IEEE Trans. Software Eng.*, vol. SE-2, no. 3, Sept. 1976.

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

[5] ——, "Regular LISP programs and their automatic synthesis from examples," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-8, no. 8, pp. 585–600, Aug. 1978.

[6] A. W. Biermann and D. R. Smith, "Hierarchical synthesis of LISP scanning functions," in *Information Processing 1977*, B. Gilchrist, Ed. Amsterdam: North Holland, 1977, pp. 41–45.

[7] ——, "A production rule mechanism for generating LISP code," Report CS-1977-6, Dep. Computer Science, Duke Univ., Durham, NC, June 1977.

[8] T. J. Biggerstaff, "C2: A 'super compiler' approach to automatic programming," Ph.D. dissertation, Dep. Computer Science, Univ. Washington, Seattle, WA, Jan. 1976.

[9] T. J. Biggerstaff and D. L. Johnson, "Design directed program synthesis," Rep. 77-02-01, Dep. Computer Science, Univ. Washington, Seattle, WA, Feb. 1977.

[10] R. Davis, B. Buchanan, and E. Shortliffe, "Production rules as a representation for a knowledge-based consultation program," *Artificial Intelligence*, vol. 8, pp. 15–45, Feb. 1977.

[11] R. Davis and J. King, "An overview of production rule systems," *Machine Intelligence 8*, pp. 300–332, 1977.

[12] E. Feigenbaum, "The art of artificial intelligence," in *Proc. 5th Int. Joint Conf. Artificial Intelligence*, Cambridge, MA, Aug. 1977, pp. 1014–1029.

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

[14] C. Green, "The design of the psi program synthesis system," *Proc. 2nd Int. Conf. Software Engineering*, San Francisco, CA, Oct. 1976, pp. 4–18.

[15] 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.

[16] G. Heidorn, "Supporting a computer directed natural language dialogue for automatic business programming," tech. rep. RC-6041 (#26157) IBM T. J. Watson Research Center, Yorktown, NY, 1976.

[17] D. Lenat, "Automated theory formation in mathematics," in *Proc. 5th Int. Joint Conf. Artificial Intelligence* Cambridge, MA, Aug. 1977, pp. 833–842.

[18] Z. Manna and R. Waldinger, "Toward automatic program synthesis," *Communications of the ACM*, vol. 14, no. 3, Mar. 1971, pp. 151–165.

[19] ——, "Synthesis: Dreams ⇒ programs," tech. rep., Stanford Research Institute, Menlo Park, CA, 1977.

[20] W. Martin *et al.*, Internal memos, Automatic Programming Group, M.I.T., Cambridge, MA, 1974.

[21] J. McCarthy, P. Abraham, D. Edwards, T. Hart, and M. Levin, *LISP 1.5 Programmers Manual*. Cambridge, MA: M.I.T. Press, 1965.

[22] N. J. Nilsson, *Problem Solving Methods in Artificial Intelligence*. New York: McGraw-Hill, 1971.

[23] 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.

[24] 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.

[25] D. R. Smith, "A class of synthesizeable LISP programs," M.Sc. thesi Dep. Computer Science, Duke Univ., Durham, NC, June 1977.

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

[27] ——, "A methodology for LISP program construction fro examples," in *Proc. 3rd ACM Symp. Principles of Programming La guages*, Atlanta, GA, Jan. 1976, pp. 68–76.

[28] D. A. Waterman, "Generalization learning techniques for automat ing the learning of heuristics," *Artificial Intelligence*, vol. 1, 197 pp. 121–170.