in Automatic Program Construction Techniques,
Eds. Biermann, Guiho, and Kodratoff,
MacMillan Pub. Co., New York, pp 307-324, 1984.

CHAPTER 15

The Synthesis of LISP Programs from Examples: A Survey*

Douglas R. Smith
Naval Postgraduate School
Monterey, California

A. Introduction

For some kinds of programs at least, a few well chosen examples of input and output behavior can convey quite clearly to a human what program is intended. The automatic construction of programs from the information contained in a small set of input/output pairs has received much attention recently, especially in the LISP language. The user of such an automatic programming system supplies a sequence of input-output (I/O) pairs $<x_1,y_1>$, $<x_2,y_2>$, ..., $<x_n,y_n>$. The system tries to obtain enough information from the examples to infer the target programs behavior on the full domain of inputs. For example, if a user inputs the sequence <nil,nil>, <(a),(a)>, <(a b),(b a)>, <(a b c),(c b a)> then the system should return a program such as

* The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

```
(F x) = (G x nil)
(G x z) = (cond ((atom x) z)
   (T (G (cdr x) (cons (car x) z)))).
```

If the system is unable to synthesize a program or needs more examples to verify a hypothesized program then the machine may request more examples. This paper presents an overview of the basic work on program construction from examples and recent approaches to this problem in the domain of the LISP language. It has not been possible to avoid glossing over many details and interesting mechanisms in the synthesis techniques discussed here. Our discussion is a simplified treatment of these techniques aimed at presenting what is essential and novel about them. Further detail may be found by consulting the original papers.

In general the classes of programs which have been studied for synthesis purposes are constructed from the LISP primitives car, cdr, cons, the predicates atom or null, the McCarthy conditional and recursive function procedures. An arbitrary composition of car and cdr functions will be called a *basic* function. e.g., car^cdr is a basic function (usually abbreviated to cadr) where ^ is the composition operator. A *cons-structure* is a function which can be described recursively as follows: i) the special atom nil is a cons-structure, ii) a basic function or a call to a program is a cons-structure, and iii) a function of the form (cons F_1 F_2) is a cons-structure if F_1 and F_2 are cons-structures. For example, (cons (car x) (F (cdr x) z)) is a cons-structure with arguments x and z. The programs to be synthesized usually have the following general form:

```
(F x z) = (cond ((p1 x) (f1 x z))
                ((p2 x) (f2 x z))
                   ...
                ((pk x) (fk x z))
                (T (H x (F (b x) (G x z)))))
```

where H and G are themselves programs of this type, $p_1, p_2, \ldots, p_k$ are predicates, $f_1, f_2, \ldots, f_k$ are cons-structures, and b is a basic function. For simplicity the arguments are restricted above to x and z; in general more or less will be required. If there is no variable z and no function G then F will be called a *forward-loop program.* If there is no H function then the last line of the above scheme has the form (T (F (b x) (G x z))))), and F is called a *reverse-loop program.*

B. Basic Work

Work on inductive inference properly includes programming by examples because the synthesis of a program generally involves the inference of an extended pattern of program behavior from the patterns discovered in the examples. In the most general setting of computability theory the problem of inductive inference of functions from input/output pairs has been explored in [Gold 1967, Blum & Blum 1975, Barzdin 1977]. The basic result from this work is that a function can be inferred from I/O pairs if it belongs to a class of enumerable functions with a decidable halting problem. Given I/O pairs $\{<x_1, y_1>, <x_2, y_2>, \ldots <x_n, y_n>\}$ our inference mechanism enumerates functions one at a time until one is found which generates y_i given x_i for each $i = 1,2,\ldots n$. If we then extend our examples set and the chosen function does not satisfy some new I/O pair then our mechanism continues its enumeration until a new function is found which does satisfy the examples. Of course such a mechanism is useful only for theoretical purposes since the process of enumerating over a whole class of functions is too expensive for practical use. Nonetheless, this work shows that it is possible to infer large useful classes of programs simply from examples of input/output behavior.

Biermann and his co-workers [Biermann et al. 1975, Biermann and Krishnaswamy 1976] have looked at methods for speeding up this enumerative search process based on the use of a program trace. Given an example input and a desired output, a *semitrace* is a functional expression which correctly computes the example output from the input but which may allow several different orders of evaluation. A *trace* is a semitrace which only has one possible order of evaluation. E.g. a semitrace for a computation of 3! is (times 3 2 1). Because of

the associativity of multiplication we obtain two distinct traces from this semitrace: (times 3 (times 2 1)) and (times (times 3 2) 1). Given this kind of information Biermann has shown how to speed up the enumerative process enormously by enumerating over only those programs whose trace on an example input corresponds to some initial subsequence of the given trace. This task is accomplished by partitioning the trace such that each block corresponds to a unique instruction in the desired program. The process of finding such a partition is controlled by a straightforward backtrack search. Pattern matching, generation of predicates and looping control structures are involved in determining whether a given trace instruction belongs to certain block (a process Biermann calls instruction merging). See [Bauer 1979] for recent work on program synthesis from traces.

Despite the great gain which can be realized by using trace information in program synthesis, Biermann's mechanism is again an enumerative method and thus can take large amounts of time in order to synthesize all but the simplest programs. The special attraction of LISP programs is the possibility that a semitrace may be easily constructed from example I/O pairs and furthermore certain restricted classes of LISP programs can be synthesized from the resulting semitraces without the need for enumerative search over the class. Some of the early work on LISP program synthesis [Shaw et al. 1975, Hardy 1975, Siklossy and Sykes 1975] could generate interesting programs but relied on heuristic techniques and provided no characterization of the class of target programs. Summers [Summers 1976, 1978] was the first to put the possibility of LISP program synthesis from examples on a firm theoretical foundation.

All of the methods to be described in this paper derive from Summers' insight that under certain conditions a semitrace of a computation can be constructed from input/output examples in the domain of LISP. Once such a semitrace has been generated a synthesis from traces method can be applied to construct the desired program.

It will be useful to abstract the key elements of a synthesis from traces method in order to facilitate the description and comparison of the LISP synthesis methods. These elements are described below and illustrated by Summers' method.

1. Data Types and Operators

A certain set of data types and operators must be available for use in traces and target programs. Summers used the S-expression data type and the primitive LISP operators cons, car, cdr, atom, nil and occasionally quote. Note that the predicate eq is not used. This means that any predicates constructed in target programs will be concerned with structure rather than semantics of the input S-expression.

2. Control Structures

A set of control structures must be available for constructing target programs. The McCarthy conditional (cond) and recursive function procedures are used in Summers' method.

3. Program Schemas

All of the synthesis methods described here make use of program schemas in order to constrain the way in which the control structures and data operators are used. The basic schema used by Summers' system has the form

$$\begin{aligned} F[x] \leftarrow [\ & p_1[x] \rightarrow f_1[x]; \\ & \ldots \\ & p_k[x] \rightarrow f_k[x]; \\ & T \rightarrow A[\ F[b[x]];\ x]\ ] \end{aligned}$$

where $f_1,\ldots,f_k$ are cons-structures, and A[w;x] is a cons-structure in which w occurs exactly once. The predicates $p_i[x]$ have the form $atom[b_i[x]]$ where b_i is a basic function.

4. Method of Obtaining a Semitrace

The obvious method for obtaining a semitrace is to ask the user to supply one. As previously mentioned there are other possibilities. In particular there are computational domains in which a semitrace may be obtained from input/output examples. Also one might obtain a semitrace of a computation by simulating an inefficient program in the hope of synthesizing a more efficient version.

In Summers' method the user is required to supply a small set of I/O pairs $<x_1,y_1>,\ldots,<x_k, y_k>$. For each pair $<x_i, y_i>$ a semitrace, called a program fragment, is created based on the following assumptions:

a. the atoms in the input S-expression x_i are unique; i.e., no atoms appear twice in x_i,

b. all atoms in y_i appear also in x_i (except possibly the special atom nil).

An example I/O pair will be called *self-contained* if the input and output satisfy these properties. The semitrace produced from the pair $<x_i,y_i>$ is a cons-structure formed recursively as follows:

$$ST(x_i,y_i) = \begin{cases} (p\ x) & \text{if } y_i \neq \text{nil and } y_i = p(x) \text{ where p is a basic function} \\ \text{nil} & \text{if } y_i = \text{nil} \\ (\text{cons } ST(x_i, \text{car}(y_i))\ ST(x_i, \text{cdr}(y_i))) & \text{otherwise} \end{cases}$$

Note that the output of ST is an unEVALed expression. For example ST((a b),(b a)) = (cons (cadr x) (cons (car x) nil)).

5. Detection methods for each control structure

For each control structure allowed in target programs there must be a method for detecting its presence in a trace or semitrace.

If a conditional statement is allowed, as in Summers' method, then we must have a mechanism for determining the number and ordering of cases, the predicates and their corresponding actions in each case. The first step in the synthesis of a recursive LISP program is the creation of a conditional expression which correctly computes all of the examples and has the form

$$\begin{aligned} F(x) \leftarrow [\ & p_1(x) \rightarrow f_1(x); \\ & \ldots \\ & p_n(x) \rightarrow f_n(x)\ ] \end{aligned}$$

where f_i is the semitrace generated by the ith example I/O pair. Summers assumes that the user gives a sequence of examples in which the example inputs form a chain, in the sense that for all inputs x_i and x_j either $x_i \leqslant x_j$ or $x_j \leqslant x_i$ where $u \leqslant v$ iff atom(u) or [car(u) $\leqslant$ car(v) & cdr(u) $\leqslant$ cdr(v)]. For example (a.b) $\leqslant$ (d.e) but the s-expressions u = (a.(b.c)) and v = ((d.e).f) could not both appear as inputs since neither u $\leqslant$ v nor v $\leqslant$ u holds. This assumption provides a natural ordering for the branches of the conditional. The predicates $(p_i\ x_i)$ for $1 \leqslant i \leqslant k$ must have the property that $(p_i\ x_i)$ evaluates to T, but $(p_j\ x_i)$ evaluates to nil for all $1 \leqslant j < i$. Since predicates have the form (atom (b x)) where b is a basic function, a mechanism for generating the predicates must find a basic function b_i such that $(b_i\ x_i)$ is an atom yet $(b_i\ x_{i+1})$ is not an atom (thus $(p_i\ x_{i+1})$ = false). The set of basic functions which so distinguish between example inputs x_i and x_{i+1} can be computed by

$$\text{PRED\_GEN}(x_i,x_{i+1}) = \text{PG}(x_i, x_{i+1}, I)$$

(I denotes the identity function)

$$PG(x,y,\theta) = \begin{cases} \phi & \text{if y is an atom} \\ \{\theta\} & \text{if x is an atom and y is not an atom} \\ PG(\text{car}(x),\text{car}(y),\text{car}\hat{}\theta) \cup PG(\text{cdr}(x),\text{cdr}(y),\text{cdr}\hat{}\theta) & \text{otherwise} \end{cases}$$

For example PRED_GEN((A),(A B)) = {cdr}, so a predicate to distinguish between (A) and (A B) is (atom (cdr x)). If Summers' system were given the following set of examples for the function REVERSE

$$\begin{array}{l}\{\ \text{nil} \rightarrow \text{nil}\\ (\text{A}) \rightarrow (\text{A})\\ (\text{A B}) \rightarrow (\text{B A})\\ (\text{A B C}) \rightarrow (\text{C B A})\ \}\end{array}$$

then by combining the semitraces produced by ST and the predicates generated by PRED_GEN, the following loop-free program is produced:

```
F(x) = (cond ((atom  x) nil)
       ((atom (cdr x)) (cons (car x) nil))
       ((atom (cddr x)) (cons (cadr x) (cons (car x) nil)))
((atom (cdddr x)) (cons (caddr x) (cons (cadr x) (cons (car x) nil)))))
```

If recursive function procedures are allowed then mechanisms must be available for detecting recursive patterns in a semitrace, determining the primitive cases for termination, and creating and handling all necessary variables. Summers has shown in his Basic Synthesis Theorem [Summers 1975,1977] the following fundamental result. Suppose that the following recurrence relations hold on a (possibly infinite) set of input-output pairs

$$p_1(x),\ldots,p_k(x),\ p_{k+n}(x) = p_n(b(x)) \text{ for } n \geqslant 1$$

$$f_1[x],\ldots,f_k[x],\ f_{k+n}[x] = C[f_n[b[x]];\ x] \text{ for } n \geqslant 1,$$

where b is a basic function and C is a cons-structure in which $f_n[b[x]]$ occurs exactly once. The function

$$F[x] = \begin{array}{l} f_j[x] \quad \text{if } p_j[x] \ \ \& \ \ \forall_i\ [1 \leqslant i < j = > \neg p_i[x]] \\ \text{undefined} \quad \text{otherwise}\end{array}$$

defined by the recurrence relations can be correctly computed by the following instance of the recursive program schema given above.

$$\begin{array}{l} F[x] \leftarrow [\ p_1[x] \rightarrow f_1[x];\\ \qquad \cdots \\ \qquad p_k[x] \rightarrow f_k[x];\\ \qquad T \rightarrow C[\ F[b[x]];\ x]\ ]\end{array}$$

This theorem and its generalizations establish a link between the characterization of a function by a recurrence relation and a recursive program for computing that function. Thus the synthesis of recursive programs reduces to detecting repetitive patterns in the examples and a decision on when enough instances of a pattern have been found to inductively infer that the pattern holds generally. At least two or three instances of a pattern with no counterinstances have been taken as sufficient to allow induction of the pattern. Any decision on this matter though is subject to easily constructed examples which cause the induction of an incorrect pattern.

Recurrence relations are detected between two semitraces f_i and f_{i+k} by finding a basic function b such that $(f_i(b\ x))$ is a substructure of $(f_{i+k}x)$. For example if

$(f_2\ x)$ = (cons (car x) (cons (caddr x) nil))
and
$(f_3\ x)$ = (cons (car x) (cons (caddr x) (cons (caddddr x) nil)))
then
$(f_3\ x)$ = (cons (car x) $(f_2$ (cdr x))).

One such mechanism makes use of the unification algorithm used in resolution theorem provers. The choice of which semitraces to unify in this way may be done systematically (by trying k = 1,2,...) or by means of heuristics (Kodratoff 1979).

We now can run through a complete example illustrating Summers' synthesis method. Suppose that the user supplies the following examples:

nil → nil
((a b)) → (b)
((a b)(c d)) → (b d)
((a b)(c d)(e f)) → (b d f)

which describes the problem of returning the list of second elements of each sublist of the input list. Applying ST to these examples we obtain the following semitraces

$(f_1\ x)$ = nil
$(f_2\ x)$ = (cons (cadar x) nil)
$(f_3\ x)$ = (cons (cadar x) (cons (cadadr x) nil))
$(f_4\ x)$ = (cons (cadar x) (cons (cadadr x) (cons (cadaddr x) nil)))

Applying the predicate generating algorithm we obtain

$(p_1\ x)$ = (atom x)
$(p_2\ x)$ = (atom (cdr x))
$(p_3\ x)$ = (atom (cddr x))
$(p_4\ x)$ = (atom (cdddr x)).

The following recurrence patterns are obtained

$(f_{i+1}\ x)$ = (cons (cadar x) $(f_i$ (cdr x))) for i=1,2,3 and

$(p_{i+1}\ x)$ = $(p_i$ (cdr x)) for i=1,2,3.

We inductively infer that these patterns hold for all i and using Summers' Basic Synthesis Theorem obtain the following program

(F x) = (cond
((atom x) nil)
(T (cons (cadar x) (F (cdr x))))))

Note that this program computes a partial function since it assumes that the input is a list of lists where each sublist has length at least two. Thus the program is not defined on inputs (a b) or ((a.b)).

For some example sets, such as the examples given above for REVERSE, the recurrence relation detection mechanisms will not work. A fundamental technique used in Summers' system generalizes the semitraces by replacing some sub-semitrace which occurs in each semitrace by a new variable. It is important to ensure that the value of the new variable will be initialized to the value of the sub-semitrace. The system then checks for recurrence relations amongst the generalized semitraces. Consider the semitraces obtained above for the function REVERSE. If the constant sub-expression nil is replaced by variable z then the semitraces become

$$(g_1\ x\ z) = z$$
$$(g_2\ x\ z) = (\text{cons } (\text{car } x)\ z)$$
$$(g_3\ x\ z) = (\text{cons } (\text{cadr } x)\ (\text{cons } (\text{car } x)\ z)))$$
$$(g_4\ x\ z) = (\text{cons } (\text{caddr } x)\ (\text{cons } (\text{cadr } x)\ ((\text{cons } (\text{car } x)\ z)))))$$

and the following recurrence relation is found:

$$(g_{i+1}\ x\ z) = (g_i\ (\text{cdr } x)\ (\text{cons } (\text{car } x)\ z)) \quad \text{for } i=1,2,3.$$

Again we inductively infer that this relation holds for all i, then invoke a form of the Basic Synthesis Theorem to justify the creation of the following reverse-loop program

$$(\text{REVERSE } x) = (F\ x\ \text{nil})$$
$$(F\ x\ z) = (\text{cond } ((\text{atom } x)\ z)$$
$$(T\ (F\ (\text{cdr } x)\ (\text{cons } (\text{car } x)\ z)))).$$

C. Recent Approaches to LISP Program Synthesis from Examples.

Biermann [Biermann 1978] has applied the techniques of synthesis from traces to the domain of LISP. The data type, operators, and control structures used in this approach are the same as in Summers' method.

The program schemas, called *semi-regular LISP schemas,* have the form

$$(F_i\ x) = (\text{cond } (p_1\ f_1)$$
$$(p_2\ f_2)$$
$$\ldots$$
$$(p_k\ f_k)$$
$$(T\ f_{k+1}))$$

where each f_j has one of the following forms: nil, x, $(F_j\ (\text{car } x))$, $(F_j\ (\text{cdr } x))$, or $(\text{cons } (F_g\ x)\ (F_h\ x))$. The predicates p_j are required to have the form $(\text{atom } (b_j\ x))$ where b_j is a basic function and furthermore if $(p_{j+1}\ x) = (\text{atom } (b_{j+1}\ x))$ then $b_{j+1} = b_j \hat{}\, w$ where w is a basic function not equal to the identity function. Certain restrictions are placed on the syntax and interpretation of semi-regular LISP schemas to yield a well-behaved subset of instances called *regular LISP programs.* A semitrace is obtained as in Summers' method and from this a trace is constructed in the form of a nonlooping regular program (i.e., no function F_i is called by more than one function f_j). Of course there are no predicates in the trace so each function in the trace has the form

$$(F_i\ x) = f_{k+1}.$$

From the example

$$((a.b).(c.d)) \rightarrow ((d.c).(b.a))$$

the following trace is constructed

$$\begin{aligned}
&(f_1\ x) = (\text{cons}\ (f_2\ x)\ (f_3\ x))\\
&(f_2\ x) = (f_4\ (\text{cdr}\ x)) \quad (f_3\ x) = (f_5\ (\text{car}\ x))\\
&(f_4\ x) = (\text{cons}\ (f_6\ x)\ (f_7\ x))\\
&(f_5\ x) = (\text{cons}\ (f_8\ x)\ (f_9\ x))\\
&(f_6\ x) = (f_{10}\ (\text{cdr}\ x)) \quad (f_7\ x) = (f_{11}\ (\text{car}\ x))\\
&(f_8\ x) = (f_{12}\ (\text{cdr}\ x)) \quad (f_9\ x) = (f_{13}\ (\text{car}\ x))\\
&(f_{10}\ x) = (f_{11}\ x) = (f_{12}\ x) = (f_{13}\ x) = x.
\end{aligned}$$

A regular LISP program can be viewed as a directed graph whose nodes represent functions. Each arc directed from a node representing function f_i is labelled by a branch of the conditional of f_i, i.e., a predicate and a function call. The construction of the graph structure of a regular LISP program from a given trace is performed by a backtrack search. At each point during the search a certain portion of the target graph has been constructed and it accounts for some initial portion of the trace. At this point if all of the trace is accounted for we are done, otherwise there is some function f_i in the trace which invokes a function f_j such that f_i is accounted for but f_j is not. If the target graph has n nodes currently then there are $n+1$ choices of nodes to identify with f_j (one for each current node plus a new node for f_j if needed). These $n+1$ alternatives are tried in turn. A choice will fail and be pruned from consideration if for various reasons (for example, knowledge about regular programs or general programming knowledge) the identification of a function with a node is incompatible with other functions identified with the node, or if all of the children of the node fail.

If several arcs emanate from a node g_i then predicates must be synthesized in order to distinguish the different cases. First, for each transition we collect all inputs which cause it to be taken. An extension of algorithm PRED_GEN from Summers' method may be used to build predicates distinguishing the resulting set of inputs.

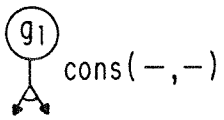
Consider the trace given in above. Initially we create a new node for f_1 as in Figure 1a. For f_2 we have two possibilities: identify f_2 with g_1, or create a new node g_2 as in Figure 1b. The first alternative fails so we explore further the second. Of the three alternative choices for f_3 shown in Figure 1c only the third does not fail. After merging f_4 and f_5 we have the graph in Figure 1d which accounts for all but f_{10}, f_{11}, f_{12}, and f_{13}. To account for f_{10} a new transition is created from g_1 and predicates must be synthesized to distinguish the two arcs from g_1. From the trace we find that the inputs

$$\{\ ((a.b).(c.d)),\ (c.d)\ \}$$

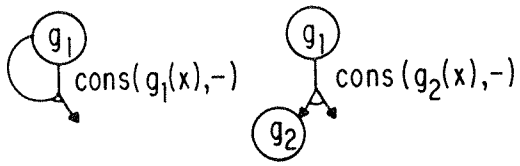
lead to the function call $(\text{cons}\ (g_2\ x)(g_3\ x))$, and the inputs

$$\{\ d\ \}$$

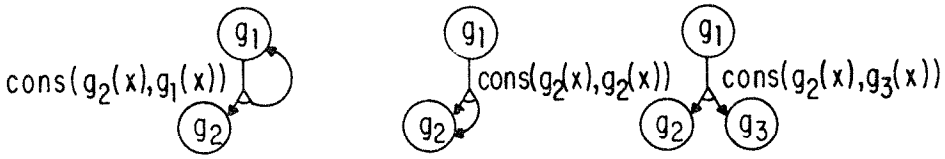
lead to the function call x. The generated predicate which distinguishs these inputs is (atom x). The resulting graph shown in Figure 1e accounts for the entire trace so synthesis halts.



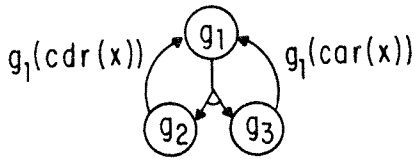
a. A node for f_1



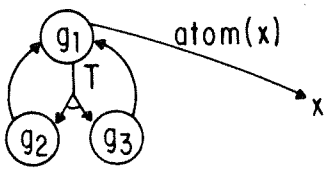
b. Alternatives for f_2.



c. alternatives for f_3.



d. After merging f_4 and f_5.



e. Final graph.

Figure 1. Synthesis of a Regular LISP program.

The resulting program is

$$(g_1\ x) = (\text{cond}((\text{atom } x)\ x)(T\ (\text{cons}\ (g_2\ x)\ (g_3\ x))))$$
$$(g_2\ x) = (g_1\ (\text{cdr } x))$$
$$(g_3\ x) = (g_1\ (\text{car } x)).$$

The system is able to generate any regular program from a finite number of examples. Although the system can be sped up by restricting the schema (for example, to programs consisting of a single loop), in general it faces the combinatorial explosion inherent in backtracking mechanisms.

Biermann and Smith [Biermann and Smith 1977, 1979] have studied the synthesis of a class of programs called scanning programs. A scanning program scans its input variables as it generates its output. Synthesis can be based on a single example and the I/O pairs are assumed to be lists of atoms. Data type, operators, and control structures are the same as those used by Summers.

The target program is viewed as a hierarchy of LISP functions and is constructed in a bottom up fashion. A semitrace is constructed for each level of the hierarchy and has the form of a loopless sequence of function calls. Each function call of a semitrace is either nil or consists of a function and an argument. Each level of a semitrace assumes that the output can be computed from the input by appending the results of the sequence of function calls in the semitrace. Since the example input and output are lists,each atom of the output y can be expressed by a function call to a function which has the form $(P^0\ x) = (\text{cons } (\text{car } x)\ \text{next})$ where next is the function call expressing the next atom in the example output.

For example, if the user gives the input/output pair

$$(a\ b\ c\ d\ e\ f) \rightarrow (a\ c\ e\ f\ d\ b)$$

then the lowest level trace is

$$(P^0\ (x)\ \text{next}),\ (P^0\ (\text{cddr } x)\ \text{next}),\ (P^0\ (\text{cd}^4\text{r } x)\ \text{next}),$$
$$(P^0\ (\text{cd}^5\text{r } x)\ \text{next}),\ (P^0\ (\text{cd}^3\text{r } x)\ \text{next}),\ (P^0\ (\text{cdr } x)\ \text{next}),\ \text{nil}.$$

To obtain the next higher level in the hierarchy the sequence is chopped up into segments such that each segment can be easily generated by either a forward-loop, a reverse-loop, or a straight line function which simply executes a fixed sequence of function calls. The segmenting mechanism works by looking for patterns and extrapolating them as far as they match the sequence of function calls. The above semitrace is segmented as follows

$$\langle(P^0\ (x)\ \text{next}),\ (P^0\ (\text{cddr } x)\ \text{next}),\ (P^0\ (\text{cd}^4\text{r } x)\ \text{next})\rangle,$$
$$\langle(P^0\ (\text{cd}^5\text{r } x)\ \text{next}),\ (P^0\ (\text{cd}^3\text{r } x)\ \text{next}),\ (P^0\ (\text{cdr } x)\ \text{next})\rangle,\ \langle\text{nil}\rangle.$$

The first three function calls can be computed by a forward loop, which we call P^1, which in effect computes (a c e) from (a b c d e f), and the next three functions can be computed by a reverse loop which we call P^2. The second lowest level of semitrace is

$$(P^1\ (x)\ \text{next}),\ (P^2\ (\text{cdr } x)\ \text{next}),\ \text{nil}.$$

There is no good way to segment this semitrace so a straightline routine, called P^3, is constructed and we have the highest level semitrace

$$(P^3\ (x)\ \text{next}),\ \text{nil}.$$

Predicates are generated in a manner similar to Summers' method. The resulting program is

$$
\begin{aligned}
&(P^3\ x) = (P^1\ x\ x)\\
&(P^1\ x_1\ x_2) = (\text{cond}\\
&\qquad((\text{atom}\ x_1)\ (P^2\ (\text{cdr}\ x_2)\ \text{nil}))\\
&\qquad(T\ (P_1^0\ x_1\ x_2)))\\
&(P_1^0\ x_1\ x_2) = (\text{cons}\ (\text{car}\ x_1)\ (P^1\ (\text{cddr}\ x_1)\ x_2))\\
&(P^2\ x\ z) = (\text{cond}\\
&\qquad((\text{atom}\ (\text{cdr}\ x))\ z)\\
&\qquad(T\ (P^2\ (\text{cddr}\ x)\ (P_2^0\ x\ z))))\\
&(P_2^0\ x\ z) = (\text{cons}\ (\text{car}\ x)\ z).
\end{aligned}
$$

Production rule schemas are used to encode both program schemas and the coding knowledge necessary for coordinating the flow of data and control in a hierarchy of functions.

Each function generated by the synthesizer instantiates a rule schema to become a production rule. At the top of the hierarchy is the function F (P^3 in the above example) which when given the example input will produce the example output. The rule for this highest level routine is applied to a start symbol. Each application of a rule adds the code for a routine on the next lowest level of the hierarchy, the correct number and order of variables having been passed down from the coded string, and a sequence of nonterminals, one for each function called by the routine with the correct number and order of variables included to be passed along into the rule for each of these routines. In general it was found necessary to introduce a new variable for each level of nesting of these functions. Rule schemas for forward loops, reverse-loops, straight-line routines, and if-then-else statements were studied.

Kodratoff [Kodratoff 1979] has presented a synthesis technique which is a powerful generalization of Summers' method. The same data type, operators, and control structures are used as in Summers' method. However Kodratoff employs a more powerful schema and technique for detecting looping patterns. Like Summers' method, multiple I/O pairs are required and looping patterns are detected by pattern matching the semitrace of output y_i with a subsemitrace of output y_{i+k} for various values of i and k. In general such a matching divides the semitrace y_{i+k} into three segments $y1_{i+k}$, $y2_{i+k}$, and $y3_{i+k}$, where the middle segment $y2_{i+k}$ matches semitrace y_i. Summers' system stops at this point with the assumption that the initial and the tail segments have no looping structure. Kodratoff's system on the other hand creates two new synthesis problems: one with I/O pairs $<x_i, y1_{i+k}>$ and the other with I/O pairs $<x_i, y3_{i+k}>$. If h is the resulting synthesized program from the former problem, and g is the resulting synthesized program for the latter problem, then these programs can be composed in the following schema:

$$
\begin{aligned}
&(F\ x) = (f\ x, (i\ x))\\
&(f\ x\ z) = (\text{cond}\ ((p_1\ x)\ (f_1\ x\ z))\\
&\qquad\qquad((p_2\ x)\ (f_2\ x\ z))\\
&\qquad\qquad\cdots\\
&\qquad\qquad((p_k\ x)\ (f_k\ x\ z))\\
&\qquad\qquad(T\ (h\ x\ (f\ (b\ x)\ (g\ x\ z)))))
\end{aligned}
$$

where b is a basic function, h and g are programs satisfying the schema, and i is an initialization function for variable z.

Notice that this system works in a top-down fashion in contrast to Biermann and Smith's system which works bottom-up. Another point of contrast is the strikingly different ways that the two systems segment the

output examples. For example, Biermann and Smith's system would segment the output of the example (A B C D) → (A B C D D C B A) into A B C D and D C B A and conjoin the forward and reverse looping routines which produce these segments. Kodratoff's system, given the examples

(A B C) → (A B C C B A)
(A B C D) → (A B C D D C B A)

would match A B C C B A with the sub-expression B C D D C B of the second output, thus segmenting the second output into A, B C D D C B, and A. Here the function h is (h x z) = (cons (car x) z) and g is (g x z) = (cons (car x) z). The complete synthesized program has the form:

```
(F x) = (f x nil)
(f x z) = (cond ((atom x) z)
          (T (h x (f (cdr x) (g x z)))))
(h x z) = (cons (car x) z)
(g x z) = (cons (car x) z)
```

Another feature of Kodratoff's method is the use of an algorithm he calls BMWk in the matching process of one semitrace against another. As noted above, the introduction of new variables is one of the key problems of program synthesis. In attempting to match A B C C B A against B C D D C B as in the example above it will be noticed that if (cdr x) is substituted for x everywhere in the semitrace for the former expression, we obtain the semitrace of the latter subexpression. The fact that a single substitution suffices is the clue that only one variable is involved. Consider on the other hand, the following example pairs.

(A B) → (B A B B B A)
(A B C) → (C A C B C A C B B)

When we attempt to match B A B B B A against C A C B C A we find two different substitutions. For the first, third and fifth atoms we substitute (cdr x) for x, and for the second, fourth, and sixth atoms we substitute x for x (the identity substitution). Here the two different substitutions are the clue that two variables are involved; the first variable is used to obtain the odd numbered atoms and the second is used to obtain the even numbered atoms. The recurrence relation between these examples then has the form:

$$y_{i+1}(x_1,x_2,z) = y_i(\mathrm{cdr}(x_1),x_2,h_i(x_1,x_2,z))$$

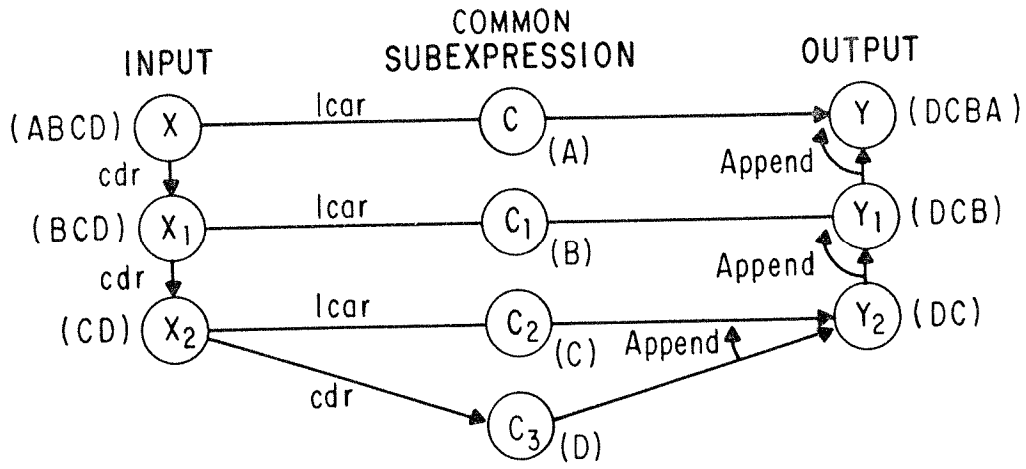
where $h_i(x_1, x_2, z)$ computes the right-most segment of y_{i+1}.

Further development of this method is reported in [Kodratoff and Papon 1980, Jouannaud and Kodratoff 1980, Kodratoff and Jouannaud 1983].

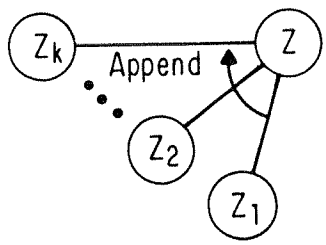
Jouannaud and Guiho [Jouannaud and Guiho 1979] have a system called SISP which works on either single or multiple examples. The class of target programs have a somewhat simpler form than those of the above systems since a more powerful class of basic functions is used. Let us call the following functions JG-basic: lcar((A B C D)) = (A) (the list containing the car of its argument), lrac((A B C D)) = (D) (the list containing the last element of the argument list), cdr((A B C D)) = (B C D), and rdc((A B C D)) = (A B C) (the list formed by deleting the last element of the argument list), and any composition of JG-basic functions. In addition to the JG-basic functions SISP uses the predicate null and a multiple-argument version of append.

Given I/O pair <x,y> in which both x and y are lists of atoms SISP segments x such that x = append(px,c,sx) and y = append(py,c,sy) where c is the longest subexpression common to x and y. Since both x and y are lists it is easy to find a JG-basic function f such that f(x) = c. The synthesis task then reduces to two subtasks. SISP finds the shortest subexpressions x_1 and x_2 of x such that $x_1 \rightarrow$ py and $x_2 \rightarrow$ sy are self-contained examples (chosing px, c, or sx if possible), and recursively tries to construct functions satisfying these new examples.

Consider the synthesis of the function REVERSE from example x = (A B C D), y = (D C B A). The largest common subexpression is (A) (in case of a tie in length, a subexpression on either end of y wins out). Thus c = (A), sx = (B C D) and py = (D C B). x = append(nil,(A),(B C D)), y = append((D C B),(A),nil). Since (A) = lcar(x) we have the partial semitrace y = append(py,lcar(x),nil) = append(py,lcar(x)). The smallest subexpression x_1 of x such that $x_1 \rightarrow$ py is self contained is x_1 = (B C D) = cdr(x). Applying the above mechanism recursively to the example $x_1 \rightarrow$ py we end up reducing it to another simpler example, etc. Jouannaud and Guiho diagram the resulting "approximation" structure as follows



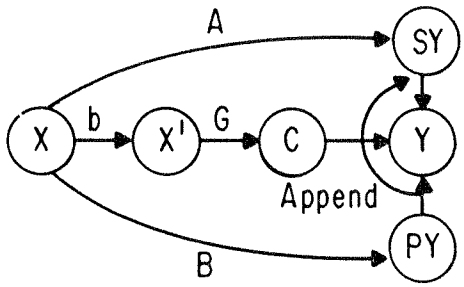
where a transition from Z to Z' that is labeled f means f(Z) = Z', and



means Z = append(Z_1 Z_2 ... Z_k). Adjacent to each node we have supplied that actual value of the node with respect to the example (A B C D) → (D C B A). This structure is similar to the semitrace generated by Summers' method, but its repetitive structure is so clearly brought out that the following recursive program to generate REVERSE is easily created from a single example:

```
F(x) = (cond
        ((null x) nil)
        (T (append (F (cdr x)) (list (car x)))))
```

If SISP fails to find a satisfactory program based on one example then it can ask for another example whose input has the longest length less than x which is defined on the domain of the program. Examples must be carefully chosen so that the inputs are of decreasing length. Given the examples $x \rightarrow y$ and $x' \rightarrow y'$ where $b(x) = x'$ (b is a JG-basic function) SISP segments y into py, c, sx where c now is the largest common segment of y and y'. Synthesis control is based on the following diagram:



The target program is F and it has the form:

$$F(x) = y = \text{append}(py,c,sy) = \text{append}(B(x),G(b(x)),A(x))$$

(If $c = y'$ then G is F). The synthesis problem reduces to synthesizing programs for 1) B, using example $x \rightarrow py$, 2) G, using example $x' \rightarrow c$, and 3) A, using example $x \rightarrow sy$. The system can request more examples of F in order to have more examples for each of these subprograms. In theory any number of levels of this process are possible thus enabling SISP to generate arbitrarily deep nestings of recursive loops. The knowledge necessary to map from the repetitive patterns found in the diagrams to program schemas is given by a small set of theorems which are analogous to Summers' Basic Synthesis Theorem.

D. Comparisons and Characterizations of the LISP Synthesis Methods.

Perhaps the most striking point of contrast between the five methods surveyed in this paper lies in their approaches to finding recursion loops. Summers' and Kodratoff's methods detect recurrence patterns by matching the semitraces obtained from different example outputs. In Biermann's Regular LISP system and Biermann and Smith's system recursion is detected by matching or folding a semitrace into itself. In Jouannaud and Guiho's SISP system the control pattern emerges from the match between an example input and its corresponding output.

Table 1 characterizes each of the synthesis techniques discussed above according to several criteria. The first column compares the time complexity of programs which are synthesizeable by each method. The time complexity is closely related to a bound on the number of atoms in the output list since the time complexity bounds the number of cons operations which can be performed. There is an apparent anomaly in Biermann's regular LISP method in that it can generate programs with any finite number of nested loops yet the complexity of such a program is linear in the number of input atoms. This is explained by the structure of S-expressions and the lack of mechanisms in regular LISP programs for copying and reusing the input variable. By way of illustration consider the problem of producing a list of the last elements of each sublist of a given list as in the example

$$((a\ b\ c)(d\ e)(f\ g\ h)(i)(j\ k\ l\ m)) \rightarrow (c\ e\ h\ i\ m)$$

Table 1. Characterization of Several Classes of Synthesizeable Functions

Synthesis Method	Time Complexity of target programs[a]	Depth of nesting of loops	Number of examples required[b]	Closure under function sequencing[c]	Separate resource and control variables	Multiple uses of a resource variable	Nonlinear recursion
Summers (Summers 1977)	O(n)	1	$\geqslant 3k$	no	yes	no	no
Regular LISP (Biermann 1976)	O(n)	unbounded	enough to exercise all branches in code	no	no	no	yes
Biermann and Smith (Biermann and Smith 1979)	$O(n^k)$ $k \geqslant O$	unbounded	1	yes	no	no	no
Kodratoff (Kodratoff 1979)	$O(n^k)$ $k \geqslant O$	unbounded	$\geqslant k(n+2)$	no	yes	yes	no
SISP	$O(n^k)$	unbounded	1,several	yes	yes	no	no

a. n is the number of atoms in the input S-expression.

b. n is the depth of nesting and k is the number of conditional branches in the top level routine.

c. A class of LISP functions is closed under function sequcing if whenever F and G are functions in the class then H(x) = append(F(x),G(x)) is a function in the class.

A program to compute this function requires a nested loop which scans down each sublist looking for the last element, yet the program's complexity is clearly linear in the number of input atoms. Columns 6 and 7 compare the uses of program variables. A *resource variable* provides atoms for the output list. A *control variable* is used to control the recursion loops. In many programs constructed by the five methods surveyed the resource variables are also used as control variables. Column six shows which synthesis techniques allow separate resource and control variables. Column seven points out Kodratoff's unique technique of allowing copies of a resource variable to be made then used differently in a program.

An important aspect of any program synthesizer is a characterization of the class of target programs. It is important to know that a system is at least sound but preferably complete also over a class. Characterization of several extensions of the class of programs synthesizable by Summers' method are given in [Smith 1977, Jouannaud and Kodratoff 1979]. Biermann's enumerative methods on input traces are sound and complete over their class of target programs. Guiho and Jouannaud [Guiho and Jouannaud 1977] have shown that SISP can correctly synthesize programs over a well defined class of nonlooping functions.

E. Concluding Remarks

The domain of LISP functions has served as a valuable testbed for the exploration of a number of synthesis mechanisms. Summers' insight that any self-contained input-output pair has a unique semitrace provides one of the fundamental starting points for the synthesis techniques described above. The methods surveyed have explored strikingly different mechanisms for detecting looping patterns in semitraces. Knowledge about how to correctly sequence and nest recursive programs has been gained. Another valuable lesson of this research involves the generation and correct use of auxilliary variables. Despite success in generating ever larger classes of programs the question remains of whether these synthesis techniques can be used in any practical way. It may be that there are some special programming domains in which some of these techniques can be applied. However the real importance of this research probably lies in the abstraction and generalization of the techniques which have been found to be sucessful in the LISP domain and the incorporation of these techniques in synthesis systems of more general scope.

References

Barzdin [1977]
J.M. Barzdin, *"Inductive inference of automata, functions and programs," Amer. Math. Soc. Translations,* Vol. 109(2), 1977, pp. 107–112.

Bauer [1979]
M.A. Bauer, "Programming by examples," *Art. Intell.* 12, 1979, 1–21.

Biermann *et al.* [1975]
A.W. Biermann, R.I. Baum, and F.E. Petry, "Speeding up the synthesis of programs from traces," *IEEE Trans. on Computers,* Vol. C–24(2), 1975, pp. 122–136.

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

Biermann and Smith [1977]
A.W. Biermann and D.R. Smith, "The hierarchical synthesis of LISP scanning programs," in *Information Processing 77,* Ed. B. Gilchrist, IFIP, North-Holland Pub. Co., 1977, pp 41–45.

Biermann [1978]
A.W. Biermann, "The inference of regular LISP programs from examples," *IEEE Trans. on Systems, Man, and Cybernetics,* Vol. SMC–8(8), August 1978, pp 585–600.

Biermann and Smith [1979]
A.W. Biermann and D.R. Smith, "A production rule mechanism for generating LISP code," *IEEE Trans. on Systems, Man, and Cybernetics,* Vol. SMC–9(5), May 1979, pp 260–276.

Blum and Blum [1975]
L. Blum and M. Blum, "Toward a mathematical theory of inductive inference," Inform. Control (28), 1975, pp 125–155.

Gold [1967]
M. Gold, "Language identification in the limit," Inform. Control (5), 1967, pp 447–474.

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

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

Jouannaud and Guiho [1977]
J.P. Jouannaud and G. Guiho, "Inference of functions with an interactive system," in *Machine Intelligence 9,* Ed. D. Michie, 1979.

Jouannaud and Guiho [1977]
J.P. Jouannaud and G. Guiho, "Program synthesis from examples for a simple class of non-loop functions," Technical Report, Laboratoire de Recherche en Informatique, Universite de Paris-Sud, 91405 ORSAY.

Jouannaud and Kodratoff [1979]
J.P. Jouannaud and Y. Kodratoff, "Characterization of a class of functions synthesized from examples by a Summers-like method using a 'B.M.W.' Matching Technique," Sixth International Joint Conf. on Art. Intell.−Tokyo 1979, pp. 440−447.

Jouannaud and Kodratoff [1980]
J.P. Jouannaud and Y. Kodratoff, "An automatic construction of LISP programs by transformation of functions synthesized from their input-output behavior," *2nd PAIS special issue on induction,* P.S. Michalsky, Ed., 1980.

Kodratoff and Fargues [1978]
Y. Kodratoff and J. Fargues, "A sane algorithm for the synthesis of LISP functions from example problems: The Boyer-Moore Algorithm," Proc. AISB Meeting, Hamburg, Germany, 1978, pp. 169−175.

Kodratoff [1979]
Y. Kodratoff, "A class of functions synthesized from a finite number of examples and a LISP program scheme," *Int. J. of Comp. and Inf. Sci.,* Vol. 8, No. 6, 1979.

Kodratoff and Papon [1980]
Y. Kodratoff and E. Papon, "A system for program synthesis and program optimization," Proc. AISB, Amsterdam, 1980, 1−10.

Kodratoff and Jouannaud [1984]
Y. Kodratoff and J.P. Jouannaud, "Some specifications for the synthesis of LISP programs," to appear in *Automatic Program Construction Techniques,* A.W. Biermann, G. Guiho, Y. Kodratoff (eds.), MacMillan Pub. Co., 1984.

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

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

Smith [1977]
D.R. Smith, "A class of synthesizeable LISP programs," Technical report CS−1977−4, Duke University, Durham, N.C., 1977.

Summers [1975]
P.D. Summers, "Program construction from examples," Ph.D. dissertation, Yale University,