INFERRING LISP PROCRAMS FROM EXAMPLES

David E Shaw

William R. Swartout

C. Cordell Green

Artificial Intelligence Laboratory
Department of Computer Science
Stanford University
Stanford, California

ABSTRACT

A program is described which infers certain recursive LISP programs from single example input-output pairs Synthesized programs may recur in more than one argument, and may involve the synthesis of auxiliary functions An actual user session with the program, called EXAMPLE, is presented, and the operation of the program and its important heuristics are outlined.

ACKNOWLEDGEMENTS

The authors wish to thank Richard Waldinger and Doug Lenat for the fruits of several valuable discussions held early in the course of this work We are also grateful for the editorial assistance of Avra Cohn, which made possible the preparation of this draft This research was supported m part by the Advanced Research Projects Agency under contract DAHC 15-73-C-CM35 and in part by the State of California through a California State Graduate Fellowship

SECTION I - INTRODUCTION

A common aspect of many definitions of automatic programming is the goal of facilitating program specification In this paper, we consider the specification of programs by examples. To describe a particular program by example, the user supplies only a sample input and output The computer then infers a plausible candidate program

The inductive inference of programs from input-output examples has also been explored by Licklider [1973] and Hardy [1974] More generally, this inference task is related to the problems of program inference from traces [Biermann. 1973] and grammatical inference [Feldman, Crps. Horning and Reder, 1969, Horning, 1969; Biermann and Feldman. 1972, Blum and Blum, 1973]

This paper describes a program, called EXAMPLE, that infers recursive LISP functions from single input-output pairs. Given the input-output specification

```
(A B C D) → (D D C C B B A A),
  input         output
```

for instance, EXAMPLE writes the "reverse-and-double" function

```
f(x) ← if null(x) then nil else
          append(f(cdr(x)),
                 list(car(x),car(x)))
```

EXAMPLE is able to infer a class of functions which perform certain list-to list transformations. In particular, each recursive function in this class steps through the input list from left to right, producing part of the output at each step Consider, for example, the pair

```
(A B C D) → ((A B) (A C) (A D) (B C) (B D) (C D))
             |←---------1--------→| |←----2---→| |←3→|
```

The output is produced in three steps A recursive subfunction produces the sublists I, 2, and 3 in successive steps and the main function appends them together

Let us briefly outline the way this function is synthesized First. EXAMPLE determines which part of the output is produced in the first step of recursion In the above example, sublist I is produced in the the first step It is assumed that this subhst is produced by a subfunction EXAMPLE thus attempts to synthesize the subfunction, generating a new input-output specification which describes this subgoal The arguments A and (B C D) are chosen as input The subfunction specified by

```
A, (B C D) → ((A B) (A C) (A D))
```

may now be synthesized by calling the EXAMPLE program recursively Returning to the synthesis of the main function, we find three remaining steps 1) The recursive call of the mam function is foimrd, 2) the resulting code is embedded in either a CONS or APPEND expression so as to properly conjoin the output from each recursive step, and 3) terminating conditions are selected

We will say that an output has been *realized* when a function has been synthesized which satisfies the given input-output specification Unfoitunately, not all syntheses which simply realize the output will be found acceptable to the user To see why. we consider a trivial synthesis scheme which can realize any output by breaking the input arguments down into their constituent atoms and recombining these atoms mechanically to form the desired output

Using this scheme, the function specified by

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

may be synthesized trivially as

```
(LAMBDA (ARG1)
  (LIST
    (LIST (CAR ARG1)
          (CAR (CDR ARG1)))
    (LIST (CAR ARG1)
          (CAR (CDR (CDR ARG1))))
    (LIST (CAR ARG1)
          (CAR (CDR (CDR (CDR ARG1)))))
    (LIST (CAR (CDR ARG1))
          (CAR (CDR (CDR ARG1))))
    (LIST (CAR (CDR ARG1))
          (CAR (CDR (CDR (CDR ARG1)))))
    (LIST (CAR (CDR (CDR ARG1)))
          (CAR (CDR (CDR (CDR ARG1)].
```

The user probably intended, though, to specify a function which finds all combinations of two elements from an input list of *any* length. The above synthesis is *implausible* since it performs this function only for lists of four elements As we will see, EXAMPLE formulates subgoals in a manner which guards against impIditMble synthesis of this sort.

A discussion of the types of synthesis tasks for which example specification is appropriate, of the problems associated with specification by example, and of the relationship between this and other methods of program specification appears in Green, et al.[197-11

SECTION 2 - AN ACTUAL SESSION

Let us now examine an actual session in which the EXAMPLE program synthesizes several user-specified LISP functions. Material typed by the user appears in lower case and is preceded by an asterisk (*). Responses by EXAMPLE are in upper case, while our comments appear in italics. The session begins when the user types "exampleO" to initiate the specification process.

• • • • •

*example()

WHAT SHALL I CALL THE TOP-LEVEL FUNCTION?

*comb2

OK. NOW TYPE IN AN EXAMPLE ARGUMENT LIST.

*((a b c d))

AND WHAT WOULD BE THE VALUE OF COMB2 WITH THIS ARGUMENT LIST?

*(a b a c a d b c b d c d)

OK, I'LL TRY.

DOES COMB2((B C D)) = (B C B D C D)?

The function of this query will be discussed in section 3.

*yes

I HAVE DEFINED COMB2 AS FOLLOWS.

```
(COMB2
  [LAMBDA (ARG1)
    (COND
      ((NULL ARG1)
       NIL)
      (T (APPEND (COMB2AUX1 (CAR ARG1)
                            (CDR ARG1))
                 (COMB2 (CDR ARG1]))
(COMB2AUX1
  [LAMBDA (ARG2 ARG3)
    (COND
      ((NULL ARG3)
       NIL)
      (T (CONS (LIST ARG2 (CAR ARG3))
               (COMB2AUX1 ARG2
                          (CDR ARG3])
```

The user may now test COMB2.

*comb2((a b c))

(A B A C B C)

*example()

WHAT SHALL I CALL THE TOP-LEVEL FUNCTION?

*pairreverse

OK. NOW TYPE IN AN EXAMPLE ARGUMENT LIST.

*((a b c d e f))

AND WHAT WOULD BE THE VALUE OF PAIRREVERSE WITH THIS ARGUMENT LIST?

*(b a d c f e)

OK, I'LL TRY.

DOES PAIRREVERSE((C D E F)) = (D C F E)?

*yes

I HAVE DEFINED PAIRREVERSE AS FOLLOWS:

```
(PAIRREVERSE
  (LAMBDA (ARG1)
    (COND
      ((NULL ARG1) NIL)
      ((NULL (CDR ARG1)) NIL)
      (T (APPEND (LIST (CAR (CDR ARG1))
                       (CAR ARG1))
                 (PAIRREVERSE(CDR (CDR ARG1])
```

*example()

WHAT SHALL I CALL THE TOP-LEVEL FUNCTION?

*mix

OK. NOW TYPE IN AN EXAMPLE ARGUMENT LIST.

*((a b c)(d e f))

AND WHAT WOULD BE THE VALUE OF MIX WITH THIS ARGUMENT LIST?

*(f c c e b b d a a)

OK, I'LL TRY.

DOES MIX((B C)(D E F)) = (F C C E B B D)?

*no

DOES MIX((B C)(E F)) = (F C C E B)?

*no

DOES MIX((B C)(E F)) = (F C C E B B)?

*yes

```
DOES THIS LOOK RIGHT?
     (MIX
       [LAMBDA (ARG1 ARG2)
         (COND
           ((NULL ARG1)
             NIL)
           ((NULL ARG2)
             NIL)
           (T (APPEND (MIX (CDR ARG1)
                           (CDR ARG2))
                      (LIST (CAR ARG2)
                            (CAR ARG1)
                            (CAR ARG1])

*mix((a b c d)(e f g h))
(H D D G C C F B B E A A)
```

• • • • •

A number of other input-output pairs are included in the appendix, along with the corresponding programs synthesized by EXAMPLE It should be noted that EXAMPLE can not synthesize functions involving counting operations or numerical comparisons (a function that sorts a list of integers by value, for example) Further, all termination checks are null tests which can return only the value NIL. Thus, for example, the function which returns the last element of a list,

(A B C D> → D

can not be synthesized, since an equality test and the ability to return a non-NIL atom would be required. A function which finds the first *ha!j* of a list, which might be specified by

(A B C D E F) → (A B C)

also falls outside the class of functions synthesized by example.

We have tried only to convey a feeling for some of the programs still beyond the reach of EXAMPLE A more precise characterization of the class of functions attacked by the current program is found in sections 4.2 and 4.3.

SECTION 3 - HOW IT WORKS: AN OVERVIEW

The program first determines whether a simple *nonrecursive* realization of the target output is possible The programming constructs available for nonrecursive synthesis will be described in section 4.1.

If the output can not be realized using available nonrecursive constructs, a synthesis involving *recursive* constructs is attempted. The recursive LISP functions synthesized by EXAMPLE produce some part of the output during the original top-level evaluation and the remainder during subsequent recursive calls. Considering the specification

(A B C) → (A A A B B B C C C),

for example, we see that the initial value segment

(A A A ...

Is produced during the first recursive step, while the remainder of the output.

B B B C C C)

is produced by subsequent recursive calls We will refer to the initially produced output segment as the *head* of the output. The remaining segment will be called the *recurrate*

After the dividing point between head and recurrate is found, EXAMPLE attempts to synthesize the code that produces the head in the same way it attempted the original (user-specified) goal. This subgoal is again specified with an input-output pair, with the head appearing as the output:

A → (A A A)

(We ignore for now the question of specifying the input part of the head realization subgoal)

In order to distinguish the head from the recurrate, EXAMPLE divides the output into equal-length *groups* of adjacent elements By way of illustration, we consider a simple variant of COMB2:

(A B C D) → (A B A C A D B C B D C D)

EXAMPLE divides the output into *groups of* two elements, as indicated above

Successive groups are then compared using a template-matching procedure This procedure searches for the first major group which *is* substantially different in some way from *its predecessors,* conjecturing a head-recurrate separation just before this change Comparing successive groups, EXAMPLE discovers a major change aftei the third group, and postulates the following separation

Head -- (A B A C A D ...

Recurrate -- ... B C B D C D)

In the case of some input-output pairs, the serial comparison procedure must in fact proceed *backward* through the output list structure. Simple heuristics are used to select a scanning direction for the output This direction determination is used in several later stages of synthesis The procedures for grouping, matching, and detet mining scanning direction are discussed in section 4.3.

EXAMPLE is now able to reduce the synthesis task to several simpler subgoals The head and recurrate must each be realized, and the resulting blocks of code combined in an appropriate way. along with code for terminating conditions In order to specify the head-trealization subgoal in input-output form, a new set of input arguments must be formulated Arguments used in specifying the subtask must again be carefully chosen to avoid the possibility of implausible synthesis. Still, some of the arguments which form the input of the parent goal may be broken down in specifying input for the subgoal. For example, the initial goal of realizing

(A B A C A D B C B D C D) from (A B C D)

spawns the subgoal of realizing

(A B A C A D) from the *two* arguments

A and (B C D)

The heuristics used to break down parent input arguments are discussed in section 4 4

Once new arguments have been generated, EXAMPLE attacks the head-realization subtask exactly as it did the original problem. If head realization itself requires a recursive synthesis, of course, a separate auxiliary Junction must be synthesized In this case, a call to the auxiliary function (with appropriate arguments) appears as the head realization code

The problems of recurrate realization are different EXAMPLE synthesizes only recursive calls whose arguments are the tails (CDR, CDDR, etc.) of the original lambda-varubles. The number of CDRs within which the original arguments are embedded is postulated using certain clues involving the propagation of argument elements to the recurrate.

If the head and recurrate are successfully realized, they are conjoined using either CONS or APPEND If the original output was interpreted in the forward direction, the head realization appears as the first argument of the joining function, while in the case of backward scanning, the recurrate realization appears first The resulting body of code is embedded in a CONDitional statement, following a set of termination checks. Each terminaiiun form involves a null-check on some tail of a current argument, with the value NIL returned if the result is positive

SECTION 4 HOW IT WORKS: THE WHOLE STORY

4 1 NONRECURS1VE SYNTHESIS

In the current version of EXAMPLE, nonrecursive synthesis is allowed only if the output can be realized simply from the current input arguments without decomposing those arguments No constructs which break down the arguments (such as CAR or CDR, for example) are considered at this stage The effect of this limitation is to prevent the synthesis of an implausible function, which might be generated by breaking down each argument into its primitive components and combining them mechanically to realize the output

At present, EXAMPLE allows nonrecursive synthesis only if the output can be realized with a composition of the functions CONS and LIST over the input arguments. More precisely, the class of functions which may be synthesized without recursion is the union of

1. The identity function over the input arguments themselves
2. All functions of the form (CONS ◦ ◦) and (LIST ◦ ◦), where ◦ represents a function which may be synthesized nonrecursively.

Thus, if the arguments were A and (B C), either (A B C) or (A (B C) A) could be realized nonrecursively, but realization of (B A C) would be not be possible

Because of these restrictions on nonrecursive synthesis, most user-specified functions of interest are not synthesized at this stage. As we wjll see, the importnt function of nonrecursive synthesis is the realization of very simple EXAMPLE-specified subgoals.

4.2 THE RECURSIVE FUNCTIONS

The recursive functions synthesized by EXAMPLE have the form indicated in the following schema:

```
(function-name
  [LAMBDA argument-list
    (COND
      ((NULL argtail) NIL)
      ...
      ((NULL argtail) NIL)
      (T (join-function
           head-realizing-code              ↓ (possibly
           (function-name recursive-args]) ↑  reversed)
```

Here, each "argrail" represents some composition of the function CDR over some input argument "Join-function" may be either CONS or APPEND The "head-realmng-code" is either some nonrecursively synthesized expression or a call to an auxiliary function "function-name AUX 1" with appropriate arguments. The form *of* the rectnsive argument list *will* be discussed later Finally, we note that the order of the "head realizing-code" and the recursive call may be interchanged

4.3 SEGMENTING INTO HEAD AND RECURRATE

Recursive realization requires correct identification of the head and recurrate of the output We recall that a template-matching procedure is used to locate the first major change in successive groups of elements. In the present version of EXAMPLE, each group initially consists of a single element If such a grouping does not allow head-recurrate separation in the template-matching stage, the size *of* the groups *ts* increased

We now examine the template-matching procedure in detail. EXAMPLE loims a template by comparing the first two groups appearing in the output. Consider COMB2

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

In this case, the two-element groups (A B) and (A C) are compared to form a template (A x). where x stands for the differing elements which appear in the two instances. (The first head-recurrate segmentation postulated by EXAMPLE is in fact an inaccurate guess based *on* the *use of* single-element *groups,* the correct segmentation discussed here is found upon subsequent scanning with two-element groups.)

In general, all atoms appearing in corresponding positions are compared for equality If the two atoms are the same, that atom appears in the template Otherwise, a unique variable x, representing the unequal atoms, appears A description of the relationship between the two differing atoms is associated with x Thus, the COMB2 template indicates that C, the second instance *of* x in the output, is the immediate successor in the input list of B. the first instance A template for the function

(A B C D E F) → (A C E),

analyzed with a group size of one. is comprised of a single variable and the associated information that the second instance of this variable is the double-successor of the first

The template is then used to predict the third group of elements, assuming the same relationship between the second and third groups as was observed between the first and second. To predict the third group appearing in COMB2, for example, the successor of C is instantiated for the template variable x. The resulting instantiated template, (A D), in fact agrees with the third group appearing in the output. This template, though, does not correctly predict the fourth group from the third EXAMPLE thus correctly divides the head from the recurrate after the third group For somp function specifications, *no* major change is detected using these heunstics. In this case, the first group of elements is taken to be the head, and the remaining groups the recurrate. The two initial (one element) groups of

F. (A B C D) ->((F AXF BXF CXF D)).

for example, yield the template (F x), which allows prediction of all groups Separation after (F A) is thus assumed. While the head-recurrate separation methods employed by EXAMPLE work reasonably well, it must be emphasized that tney are not universally effective A larger class of functions might be synthesized using better heuristics for this critical decision

It was noted in section 3 that the output *must* sometimes be scanned backward (from right to left) in order to effect the proper head-recurrate separation EXAMPLE chooses a scanning direction by noting whether elements from the front of the input list propagate toward the front or the back of the output If they tend to appear in the end of the output, EXAMPLE assumes that the head will be found at the end of the output list. In this case, a reverse scanning direction is used to distinguish the head and recurrate. In section 4 6, we will see other effects of the decision to scan backward

44 SUBGOAL REALIZING THE HEAD

Let us review the work EXAMPLE has done so far. A scanning direction has been chosen heuristically and noted for later reference. Adjacent groups of elements have been scanned in the chosen direction and compared using a template-matching procedure. By locating the site of the first major change, that part of the output generated during the first step of recursion (the head) has been distinguished from the part produced during all successive recursive calls (the recurrate) If no major change was apparent, a default separation point has been assumed Finally, those atoms appearing only in the head have been distinguished.

EXAMPLE must now attempt to synthesize code which will generate the head when evaluated with the arguments of the top-level function call We implicitly assume that evaluations of this s*ame* code during all subsequent recursive calls will produce the recurrate. As mentioned before, the head realization subgoal is specified with an input-output pan, just as the user specified the original task The target output, of couise, is the head itself Selection of an appropriate input argument list for the head realization subgoal is less trivial

In certain cases, the original input list is a reasonable choice of input for the subgoal For example, consider the following input-output pair

(A B C D) → (A B C D B C D C D D).

Here the head is exactly the same as the original input argument (A B C D) A trivial nonrecursive realization of the head thus results when

(A B C D) → (A B C D)

is specified as the subgoal For a first attempt at head realization, EXAMPLE always tries this first method, in which the input for the subgoal is the same as the original input For some problems, though, the original arguments must be broken down in some manner in order to realize the head In this case, EXAMPLE fails to accomplish the first subgoal, and creates a new subgoal whose input arguments are subparts of the original arguments To illustrate, the first subgoal generated in trying to realize the head of COMB2 it to synthesize a subfunction satisfying the input-output relation

(A B C D) → (A B A C A D)

This output, however, can not be realized from the input (A B C D) EXAMPLE thus generates the new subgoal

A, (B C D) → (A B A C A D)

which will eventually result in the successful synthesis of a two-argument auxiliary function

EXAMPLE generates this new subgoal by decomposing the original input, (A B C D), according to a simple heuristic, First, we note that certain atoms from the input list may appear in the head but not in the recurrate These atoms, called *head diitinguishers,* appear as arguments for the head lealization subgoal. Here A is a head distinguishes since it appears in the head, (A B A C A D), but *not m* the *recurrate,* (B C B D C D), A is thus chosen as an argument Second, the remainder of the original argument after removing the head distinguisher is included unless none of its atoms are found in the head We remark in passing that if EXAMPLE broke down the original argument *completely* into its constituent atoms, head realization would always succeed (nonrecursively) A complete decomposition of this sort, though, is in general dangerous, admitting the possibility of implausible synthesis This danger is the motivation for *selective* input decomposition

Before continuing our discussion of the synthesis of the main function COMB2, let us summarize the synthesis of its subf unction The original goal of synthesizing a function specified by

(A B C D) → (A B A C A D B C B D C D)

spawns the head realization subgoal

A, (B C D) → (A B A C A D).

Template mauhnig with single-element groups separates the head of this subgcul output, (A B), from the recurrate, (A C A D) EXAMPLE thrn decomposes the argument (B C D) into B and (C D), discarding (C D), whose atoms fail to appear in the head.

The resulting subgoal is

A, B → (A B).

The list (A B) is easily synthesized nonrecursively from A and B.

4.5 REALIZING THE RECURRATE

As we have indicated in section 4.1, all recursive calls synthesized by EXAMPLE to realize the recurrate are of the form

(function-name var1 var2 ... var*n*)

where each var*i* is either the *i*th lambda variable or a *tail* of (some composition of CDR over) the *i*th lambda variable. The recursive call of the ALTERNATE function,

(A B C D E F) → (A C E),

for example, is

(ALTERNATE (CDR (CDR ARG1))),

while the function FOO described by

F, (A B C D) → ((F A)(F B)(F C)(F D))

employs the recursive call

(FOO ARG1 (CDR ARG2)).

EXAMPLE must thus determine the number of CDRs within which each reursive argument should be embedded This number is assumed equal to the number of atoms from the beginning of that argument which fail to appear in the recurrate Unfortunately, this method fails for many input-output pairs in which the atoms of the input do not all propagate *to* the output. Certain weak heuristics are used to allow synthesis of some such functions, but the problem is not entirely solved

Once the recursive call has been synthesized, EXAMPLE can check its decision about head-recurrate segmentation by querying the user In the case of the above function FOO, for example, the user is asked if the proposed recursive call in fact realizes the recurrate

"DOES FOO[F, (B C D)) - «F B)<F C)(F D))>"

(The user specifipd identifiers are substituted for the formal variables used in the actual recursive call) A negative response is taken *as* evidence *of* faulty segmentation of head and recurrate, often leading to a revised conjecture regarding scanning group size

4.6 CONJOINING THE HEAD AND RECURRATE

Now that the head and recurrate have been realized, EXAMPLE conjoins the two resulting pieces of code using either CONS or APPEND. If the output was analyzed using a forward scanning direction, the head realizing code appears as the first argument of the joining funrtion, since the head must have been found at the beginning of the output If reverse scanning was used, the head must be at the end of the output, and the head realization appears as the second argument

Several factors are considered in deciding whether CONS or APPEND should be used tor conjunction In the case of backward scanning, APPEND is always chosen The joining function will also be APPEND whenever the head contains more than *one* element In accoidance with usual human programming practice, however, CONS is used in the case of a single-element head *found* by forward scanning EXAMPLE adjusts the outermost list structure of the head to allow the use of the appropriate joining function

4 7 SYNTHESIZING TERMINATING CONDITIONS

We saw in section 4 1 that all terminating conditions synthesized by EXAMPLE rest a tail of some argument, returning NIL if a NULL tail is encountered The number of CDRs involved in each tail depends on the number of CDRs used in the recursive call on that argument Thus COMB2, which is synthesized using CDR recursion, embodies the single null check

if NULL (ARG) then NIL

but the function specified by

(A B C D E F) → (A C E)

requires the deeper termination check

if NULL (CDR (ARG)) *then* NIL

since its recursive argument is

CDR (CDR (ARG)).

It must be acknowledged that this heuristic yields incorrect terminating conditions for some functions which are o:herwise within the target class of EXAMPLE

The resulting block of code is embedded in a function definition call with the user specified name and list of lambda variables. The resulting function is then defined for system use and evaluated with the user-specified input list If this evaluation in fact yields the user-specified *output,* the function is presented to the user for verification and further user testing

SECTION *b* - CONCLUSION

The EXAMPLE program was written in INTERLISP by David Shaw and was revised by William Swattout. A number of EXAMPLE sessions have been observed during the past year, but no formal study has yet been conducted of the programs users actually specify or of the way in which such programs are specified. It seems to us that such further study of actual program specification would be valuable at this point

The exact role input-output examples will play *in* facilitating program specification is not yet clear We believe, however, that the capacity for specification by examples may be a useful component of future automatic programming systems

We conclude with several other LISP functions synthesized by the EXAMPLE program The shorthand notation

<function name> <input list> -» <output>

will represent the user specification of a function <function name> which returns the value <output> when evaluated with the arguments on * input list>.

REVDBL : ((A B C D)) → (D D C C B B A A)

```
(REVDBL
   [LAMBDA (ARG1)
      (COND
         ((NULL ARG1) NIL)
         (T (APPEND (REVDBL (CDR ARG1))
                    (LIST (CAR ARG1) (CAR ARG1])
```

REVERSE : ((A B C D)) → (D C B A)

```
(REVERSE
   [LAMBDA (ARG1)
      (COND
         ((NULL ARG1) NIL)
         (T (APPEND (REVERSE (CDR ARG1))
                    (LIST (CAR ARG1])
```

DOUBLE : ((A B C)) → (A A B B C C)

```
(DOUBLE
   [LAMBDA (ARG1)
      (COND
         ((NULL ARG1) NIL)
         (T (APPEND (LIST (CAR ARG1)
                          (CAR ARG1))
                    (DOUBLE (CDR ARG1])
```

LISTTHRU : ((A B C D)) → ((A) (B) (C) (D))

```
(LISTTHRU
   [LAMBDA (ARG1)
      (COND
         ((NULL ARG1) NIL)
         (T (CONS (LIST (CAR ARG1))
                  (LISTTHRU (CDR ARG1])
```

LISTOFCOMBS : ((A B C D))
→ ((A B) (A C) (A D) (B C) (B D) (C D))

```
(LISTOFCOMBS
   [LAMBDA (ARG1)
      (COND
         ((NULL ARG1) NIL)
         ((NULL (CDR ARG1)) NIL)
         (T (APPEND (LISTOFCOMBS.AUX1
                       (CAR ARG1) (CDR ARG1))
                    (LISTOFCOMBS (CDR ARG1])

(LISTOFCOMBS.AUX1
   (LAMBDA (ARG2 ARG3)
      (COND
         ((NULL ARG3) NIL)
         ((T (CONS (LIST ARG2 (CAR ARG3))
                   (LISTOFCOMBS.AUX1
                      ARG2 (CDR ARG3])
```

TELESCOPE : ((A B C D)) → (A B C D B C D C D D)

```
(TELESCOPE
   (LAMBDA (ARG1)
      (COND
         ((NULL ARG1) NIL)
         (T (APPEND ARG1
                    (TELESCOPE (CDR ARG1])
```

PAIR2 : ((A B C) (D E F)) → ((A D) (B E) (C F))

```
(PAIR2
   (LAMBDA (ARG1 ARG2)
      (COND
         ((NULL ARG1) NIL)
         ((NULL ARG2) NIL)
         (T (CONS (LIST (CAR ARG1)
                        (CAR ARG2))
                  (PAIR2 (CDR ARG1)
                         (CDR ARG2])
```

ALTERNATE: ((A B C D E F)) → (A C E)

```
(ALTERNATE
   (LAMBDA (ARG1)
      (COND
         ((NULL ARG1) NIL)
         ((NULL (CDR ARG1)) NIL)
         (T (CONS (CAR ARG1)
              (ALTERNATE (CDR (CDR ARG1])
```

PAIR1 : (FN (A B C D)) → ((FN A) (FN B) (FN C) (FN D))

```
(PAIR1
   (LAMBDA (ARG1 ARG2)
      (COND
         ((NULL ARG2) NIL)
         (T (CONS (LIST ARG1 (CAR ARG2))
                  (PAIR1 ARG1 (CDR ARG2])
```

SHUFFLE : ((A B C) (D E F)) → (A D B E C F)

```
(SHUFFLE
   (LAMBDA (ARG1 ARG2)
      (COND
         ((NULL ARG1) NIL)
         ((NULL ARG2) NIL)
         (T (APPEND (LIST (CAR ARG1)
                          (CAR ARG2))
                    (SHUFFLE (CDR ARG1)
                             (CDR ARG2])
```

FOO : ((A B C) (D E)) → (A D B D C D A E B E C E)

```
(FOO
   (LAMBDA (ARG1 ARG2)
      (COND
         ((NULL ARG2) NIL)
         (T (APPEND (FOO.AUX1 ARG1
                       (CAR ARG2))
                    (FOO ARG1 (CDR ARG2])

(FOO.AUX1
   (LAMBDA (ARG3 ARG4)
      (COND
         ((NULL ARG3) NIL)
         (T APPEND (LIST (CAR ARG3) ARG4)
                   (FOO.AUX1 (CDR ARG3) ARG4])
```

```
CROSSPROD : ((A B C) (D E))
        → ((A D) (A E) (B D) (B E) (C D) (C E))

    (CROSSPROD
      (LAMBDA (ARG1 ARG2)
        (COND
          ((NULL ARG1) NIL)
          (T (APPEND (CROSSPROD.AUX1
                        (CAR ARG1) ARG2)
                     (CROSSPROD
                        (CDR ARG1) ARG2])

    (CROSSPROD.AUX1
      (LAMBDA (ARG3 ARG4)
        (COND
          ((NULL ARG4) NIL)
          (T (CONS (LIST ARG3 (CAR ARG4))
                   (CROSSPROD.AUX1 ARG3
                          (CDR ARG4])

REVTELESCOPE : ((A B C D))
        → (D C B A D C B D C D)

    (REVTELESCOPE
      (LAMBDA (ARG1)
        (COND
          ((NULL ARG1) NIL))
          (T (APPEND
               (REVTELESCOPE.AUX1 ARG1)
               (REVTELESCOPE (CDR ARG1])

    (REVTELESCOPE.AUX1
      (LAMBDA (ARG2)
        (COND
          ((NULL ARG2) NIL)
          (T (APPEND (REVTELESCOPE.AUX1
                        (CDR ARG1))
                     (LIST (CAR ARG1])
```

REFERENCES

Biermann, A W, and Feldman, J A. "On the Synthesis of Finite-State Machines from Samples of Their Behavior," IEEE Transactions on Computers, Vol C-21, No 6, June 1972, pp b92-597. (also "On the Synthesis of Finite-State Acceptors," Memo AIM-114. Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford. California, April 1970).

Blum, L., and Blum, M, "Inductive Inference A Recursion Theoretic Approach", Information and Control, to appear (Also Memorandum ERL M386, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, California, August 1973).

Feldman, Jerome A., Cips, J, Horning, J. J., Reder, S., "Grammatical Complexity and Inference," Memo AIM-89, Technical Report No. CS 125, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, June 1969

Green, C C, Waldmger, R J., Barstow, D. R, Elschlager, R., Lenat, D B, McCune. B P. Shaw. D E., and Steinberg, L. I, "Progress Report on Program-Understanding Systems," Memo AIM-240, Report STAN-CS-74-444, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, August 1974.

Hardy, Steven. "Automatic Induction of LISP Functions," AISB Summer Conference, University of Sussex, Brighton, England, July 1974, pp 50-62

Horning. James Jay. "A Study of Grammatical Inference." Ph.D. thesis. Memo AIM-98, Report STAN-CS-69-139. Artificial Intelligence Labotatory, Computer Science Department, Stanford University, Stanford, California, August 1969

Licklidei, J C R., in "Automatic Composition of Functions from Modules. Project MAC Progress Report X July 1972 - July 1973, Section III E I. Project MAC, Massachusetts Institute of Technology. Cambridge, Massachusetts, pp 151-156