SYNTHESIS OF LISP FUNCTIONS *FROM* EXAMPIES

Steven Hardy
University of Sussex, Falmer, Brighton, Sussex

Abstract

A system, called GAP, which automatically produces LISP functions from example computations is described. GAP uses a knowledge of LISP programming to inductively infer the LISP function 'obviously' intended by a given 'iopair' (i.e. a single input to be presented to the function and the output which must result). The system is written in POPCORN (a CONNIVER-like extension of POP2) and represents its knowledge of LISP procedurally.

Acknowledgements

The research reported herein was carried out with the support of the Science Research Council.

I would like to thank Pat Hayes, Richard Bornat and, especially, Mike Brady for their constant advice and guidance.

Section One - Introduction

Despite the fact that there are infinitely many functional extensions of the input-output ('iopair'): (A B C D) => ((A) (B) (C) (D)) (1) there is only one function that would be regarded by LISP programmers as the 'obviously' intended one, viz: (A B ---- Z) => ((A) (B) --- (Z)).

This paper describes a program, called GAP (Generalizing Automatic Programmer), which attempts to model the LISP programmer to the extent of producing this function. When presented with iopair (1) GAP produces the LISP function:

```
(LAMBDA (X)
(COND ((ATOM X) NIL)
  (T (CONS (LIST (CAR X)) (SELF (CDR X))))))
```

(The LISP system used by GAP acts as if all LAMBDA expressions were implicitly labelled 'SELF').

GAP can be distinguished from those automatic programmers which deduce the wanted program from the given description. These reflect a popular approach to automatic programming and have, theoretically, a number of advantages over GAP. A deductive system can, in principle, produce a correct program - one guaranteed to meet its specification. Furthermore, such a system can, if it contains a complete proof system, produce any program which can be described to it. If the description language uaed is general say, first order predicate calculus - then the automatic programmer will be general. Unfortunately, most theorem provers are not very powerful and this limits the size o f program which can be written. This disadvantage can be overcome, to an extent, by allowing the proof system to employ knowledge in the form of 'control' statements which embody an understanding of how to achieve proofs in the domain of program writing. A more important criticism of deductive automatic programming is that the wanted program must be completely specified. This can be as difficult, and as error prone, as actually writing the wanted program- For example, the function intended by iopair (1) is described in the first order predicate calculus as:

```
LISTIFY (NIL) = NIL
LISTIFY (CONS(X,Y)) =
   CONS (CONS(X,NIL), LISTIFY(Y))
```

and this is barely simpler than the corresponding LISP function.

GAP is written in POPCORN (5) an extension of POP2 (3) that provides some of the features of CONNIVER (9). The program contains a number of heuristic routines embodying knowledge about various program achemas for LISP expressions. When GAP is presented with an iopair these routines examine it for 'cues' which suggest hypotheses about the form of the wanted expression. If an hypothesis seems particularly promising it is examined in great detail by a deductive LISP system which attempts to verify and complete it.

The cue seeking routines are stored in the POPCORN data base, making the addition of new heuristics extremely easy as the remainder of the program need not be altered. It can be *seen* that the detailed flow of control will depend on minor vagaries of the data base controlling routines.

The complete program occupies less than 35K words of core on a PDP-10. It takes two or three seconds of CPU time to code the example given earlier.

In Section Two I explain how GAP works, illustrated by a few simple examples; in Section Three there is a brief discussion of the LISP system and in conclusion I point out some shortcomings of the program and describe ways it could be improved.

Section Two - GAP at Work

The LISP expressions that GAP is capable of producing can be described by a number of program schemas. The type of recursive function to which GAP devotes most attention, called a < LISTFN > , can be represented as:

```
(LAMBDA (X)
     (COND (<TESTS X> NIL)
          (T (APPEND <FORM X>
                     (SELF <PARTOF X> )
                     <FORM X> ))))
```

This schema is completed by replacing the terms <SOMETHING X> by an expression of the appropriate type. <TESTS X> denotes a Boolean expression, <FORM X> denotes any expression that GAP can write, and <PARTOF X> denotes an expression synthesised from the primitive LISP selector functions, CAR and CDR, applied to X.

The various types of expression that GAP can write can be informally described by a 'grammar'. The most general type is 'FORM'; this denotes an expression where the output is formed from the input atoms with no reference to the identity of any particular atoms in the input. This type of expression can be described:

```
<FORM X> :: = <BUILD X>
           :: =  (<LISTFN>X)
           :: =  (<TREEFN>X)
           :: =  ((<FLATFN>X)
           :: =  (LIST<FORM X>---<FORM X>)
```

The notation used is meant purely as an explanatory device - it 'defines' an abstract, rather than concrete, syntax for the expressions GAP writes.

An expression of type BUILD is synthesised from the functions CONS, LIST, APPEND, CAR AND CDR applied to X or NIL. For example, (APPEND X (CDR X)) is of type BUILD.

A <LISTFN> is a function with a list as input and output. The function shown in the introduction is a <LISTFN>, though optimised for efficiency and appearance.

A <TREEFN> takes and returns a tree; the iopair;

((A (B) C) D) => = ((A A (B B) C C) D D)

(which GAP easily handles) would be coded as a <TREEFN>.

The final possibility for an expression of type FORM is an application of a <FLATFN> to a tree to return a list - for example:

((A (B) C) D) && "Q" => = (A Q B Q C Q D Q)

<LISTFN>s embody an essentially iterative process, as they are only singly recursive. <TREEFNS>s and <FLATFN>s call themselves twice.

An expression of type BUILD can be described:

```
<BUILD X>  :: =  (CONS<BUILD X><BUILD X>)
           :: =  (LIST<BUILD X>---<BUILD X>)
           :: =  (APPEND<BUILD X>--<BUILD X>)
           :: =  <SELECT X>

<SELECT X> :: = NIL | (QUOTE <ATOM>) | <PARTOF X>
<PARTOF X> :: = X
           :: =  (CAR<PARTOF X>)
           :: =  (CDR<PARTOF X>)
```

Once GAP has decided that an iopair is of type BUILD the expression is found by exhaustive search - though ordered to favour expressions using APPEND.

The expression to replace <TESTS X> in the expansion of a <LISTFN> can be described:

```
<TESTS X> :: = <PRED X>
          :: =  (OR<PRED X>---<PRED X>)
<PRED X>  :: =  (ATOM <PARTOF X>)
```

The expression actually chosen is completely determined by the recursion line of the function - the test is designed to prevent it causing an error. NIL, the 'boundary condition[1] imposed upon the schema is then a reasonable final value for the schema.

Whenever an iopair is coded as a <FLATFN>, it could have been coded as (<LISTFN> (FLATTEN X)). This would be slightly unnatural and so we include an appropriate combined schema. The only remaining undescribed schema is <TREEFN>:

```
<TREEFN> :: = (LAMBDA (X)
                      (COND ((NULL X) NIL)
                            (T<TREELINE X>))))
<TREELINE X>:: = (APPEND<TREETERM X>(SELF (CDR X)))
             :: = (APPEND (SELF(CDR X))<TREETERM X>)
<TREETERM X>:: = (COND(ATOM(CAR X))( BUILD  X )
                       (T(LIST(SELF(CAR X))))))
```

The schemas described are simplified in that they are presented as if GAP could write only expressions involving one variable, when in fact, any number can be used.

How the schemas are used

It would be possible to represent these schemas explicitly within the computer and GAP could write functions by enumerating possible expressions until one was found which included the given iopair, and, even though the schemas are not explicitly stored, GAP can effectively do this, using the LISP system described in Section Three. Whilst this method of program writing is appropriate on problems with small search spaces - for example, expressions of type PARTOF - it becomes incredibly slow as the complexity of the problem rises. The cue-seeking routines suggest likely replacements for parts of the FORM schema (the most general) and so cut the search space. The heuristics used are best explained by following GAP's progress as it writes several functions.

When GAP is presented with an iopair it tries to find an expression which, when evaluated with an alist built up from the inputs to the iopair, produces the output of the iopair. This expression is then converted to a function and printed to the user. When given the iopair: (A B C D) => = ((A) (B) (C) (D))) GAP tries to find an expression evaluating to ((A) (B) (C) (D)) with the alist ((X.(A B C D))). Initially all that is known of the expression is that it is of type FORM - the most general type. One cue noticed in this case is that the length of the output is an integral number of times the length of an input. (In this case, equal to that of the only input.) This would be the case if the expression were an application of a <LISTFN> where the expressions <FORM X> are replaced by (LIST <FORM X>—<FORM X>) and where the expression

<PARTOF X> is replaced by (CDR I). Such a*LISTFN> is called a CDR-loop. One of the hypotheses suggested by the cue seeking routine is:

(APPEND(LIST< FORM *X*>) (SELF(CDR X))(LIST))

Other heuristic routines, described later, concur with this hypothesis and it is selected for detailed examination. The expression is optimised to;

(CONS < FORM X> (SELF(CDR X)))

and GAP tries to make:

(LAMBDA (X) (CONS < FORM X > (SELF (CDR X))))X)
evaluate to *((*A (B) (C)(D)). For this to be so <FORM X>must evaluate to (A). GAP solves this subsidiary problem by a call on the POPCORN data base - and hence a possible recursive call of GAP itself. If the wanted expression is not already known the iopair: (A B C D) => = (A) is examined and one routine decides it is of type BUILD. It does this by counting the atoms in the output, getting the list (1 0 0 0) (meaning 'A' occurred once and 'B', 'C' and 'D' not at all). The routine expects that if the expression being coded calls a recursive function then there should be some pattern in this list. It can't find any and so the expression is coded by the routine responsible for producing expressions of type BUILD, which produces (LIST(CAR X)).

GAP deduces that the hypothesised recursion line would 'explain' the output if (SELF NIL) evaluated to NIL - this suggests that < TESTS X> evaluates to 'T' when X is NIL. However, GAP 'knows' that the terminating condition of many iterative loops -<LISTFN>s embody an essentially iterative process (8) - is such that one more iteration would have caused an error.

To apply tills knowledge GAP finds the 'minimumrvalue of X if the recursion line is not to cause an error. In this case X must be a pair - since both CAR and CDR are applied to it - and so an appropriate TEST is (ATOM X). GAP inserts this into the <LISTFN> schema and checks that (SELF NIL) does actually evaluate to NIL. This type of redundancy provides a useful consistency check.

Had we given GAP the iopair:

(A B C D) => = ((A) (B) (C) (D) (B)
(C) (D) (C) (D) (D))

then one of the cues noted would have been that the length of the output is proportional to N#(N+1)/2, where N is the length of an input. This can happen when a CDR-loop function calls another as a subroutine. For this iopair, therefore, one of the hypotheses generated is that the first four elements of the output should be split off and an expression evaluating to that segment be found to replace the first <FORM X > in the < IISTFtO schema. Once this has been done the expression is completed in a similar way to the last example.

A common method of problem solving is find some hoinomorphicmapping of the problem, with a smaller search space, which can be easily solved to provide a plan for the solution of the main problem. This technique has been used by a number of researchers, notable (1,6,2,11). GAP applies this method of problem solution when {resented with an iopair with more than one input. Many functions of multiple inputs produce their output by interleaving their inputs in some way. If, therefore, GAP finds for each element of the output from which inputs they have drawn atoms, it may be able to recognise some pattern in the resulting 'origin list'. Consider the iopair:

(A B C D) && "Q" => = (A B Q B C Q C D Q)

Representing the two inputs by X and Y gives the origin list:

((X) (X) (Y) (X) (X) (Y) (X) (X) (Y))

It is trivial to recognise the repeated ((X) (X) (Y)) in this list, and this suggests the recursion line hypothesis:

(APPEND(LIST < FORM X > < FORM X > < FORM Y >)
(SELF < PARTOF X > < PARTOF Y >))

Once this hypothesis has been selected for detailed examination < PARTOF Y > is replaced by Y - the value of Y is atomic and so neither CAR nor CDR can be applied to it. The validation of this hypothesis is more complicated than the earlier example as the recursion step for X is not known. The LISP system, using a simple matcher, deduces that: (LIST < FORM X > < FORM X > < FORM Y >) must evaluate to (A B Q) and can therefore be replaced by: (LIST (CAR X) (CAR (CDR X)) Y)

Whilst anti-evaluating (as this part of the validation process is called) the recursive call of the hypothesised function, all that is known of the value of X is that it is part of (A B C D). However, it is easily found that (CAR X) evaluates to B. X can then only be the CDR of (A B C D) and so < PARTOF X > is replaced by (CDR X).

To complete the hypothesis < TESTS X > is replaced by:

(OR (ATOM X) (ATOM (CDR X)))

and after final optimisation GAP produces the function:

```
(LAMBDA (X Y)
  (COND((OR(ATOM X) (ATOM(CDR X)))NIL)
    (T (CONS(CAR X)
        (CONS(CAR (CDR X))
          (CONS Y(SELF (CDR X) Y)))))))
```

The technique used in the last example - looking for patterns in an origin list - can be extended by regarding multiple occurrences of the same outputs as a single occurrence. For example, the modified origin list for the iopair:

(A B C D) && "Q" => = (A B C D Q B C D Q C D Q D Q)

is ((X) (Y) (X) (Y) (X) (Y) (X) (Y))

The repeated((X) (Y)) in this list suggests splitting of either the first five or the last two elements of the output and finding an appropriate expression and so one of the recursion line hypotheses made is:

(APPEND (APPEND X(LIST Y)) (SELF < PARTOF X > Y))

The cues described so far have to generate a number of hypotheses as they cannot distinguish whether atoms from the 'front' of the inputs occur at the front or back (or both) of the output. If we give GAP an iopair with one input, of length four, and an output of length eight then three

recursion line hypotheses will be generated by the cue first described in this section. These hypotheses can be represented:

(APPEND(LIST<FORM X><FORM X>)(SELF(CDR X)))
(APPEND(LIST<FORM X>)(SELF(CDR X))(LIST<FORM X>))
(APPEND(SELF(CDR X))(LIST<FORM X><FORM X>))

(However, these hypotheses are not generated simultaneously; not until the most likely alternative (the first) is rejected are the other possibilities suggested.)

GAP needs a cheap way oi" rejecting the incorrect hypotheses. One way of doing this is to replace the atoms in the output by numbers representing which element of an input they came from. The 'number list' for the iopair:

(A B C D) =>= (A A B B C C D D)

is (1 1 ? 2 3 3 44). As the 'average ' atom in the inputs to the recursive call of the function is higher than the average atom in the original inputs the average for that segment of the output allegedly due to the recursive call of the function should be higher than that for the whole output - if this is not so the hypothesis is rejected. *This* heuristic will reject the two incorrect hypotheses above.

Additional information can be found by supposing that the wanted function recurs on the CDR of some input and otherwise *refers only* to the CAR *of* that input. If this is so it might be possible to split the output into three segments the outer ones containing no atoms from the CDR of the relevant input, and the inner segment, hopefully due to the recursive call of the function, containing none from the CAR. Of course, there will usually be several possible splittings. For the iopair: (A B C D) && "Q/" -> = (D Q C Q B Q A Q) the routine embodying this heuristic will suggest:

NIL, (D Q C Q B Q), (A Q) and N1L,(D Q C Q B),
(Q A Q))

When used in conjunction with other routines, this heuristic, despite its simple-mindedness, makes a valuable contribution to the selection of the correct hypothesis.

It is clear that we can extend the principle of trying to find which segment of the output of a <LISTFN> is due to its recursive call by guessing which atoms it might contain. Consider the iopair:

(A B C D E F) =>= (A B B C D D E F F)

If we count the atoms in the output we get the atomcount list (1 2 1 2 1 2) - meaning 'A' occurred once, 'B' twice and so on. The repeated (1 2) in this list suggests that the segment due to the recursive call might have the atomcount list (0 0 1 2 1 2). GAP uses a simple matcher to find such a segment and gets (C D D E F F). The two outer segments, (A B B) and NIL, suggest replacements for the terms <FORM X> in the<LISTFN> schema and the length of the repeated atomcount segment (1 2) suggests the recursion step to use and so this routine hypothesises the recursion line:

(APPEND(LIST(CAR X)(CAR(CDR X))(CAR(CDR X)))
(SELF(CDR(CDR X))))

The reader will have noticed that many of the cue-seeking routines appear redundant; several can (and do) suggest the same hypothesis. This redundancy provides the basis of the approach taken to hypothesis generation. Part of GAP, called the 'research director', monitors the hypothesis generation process. If this sees that a hypothesis is particularly popular it interrupts, and tests the promising candidate. Should this be unsuccessful it allows hypothesis generation to continue. Some cue-seeking routines notice failure of their hypotheses and produce fresh alternatives. Should there be no especially popular hypothesis, the research director exhorts the cue-seeking routines to 'try harder'; on receipt of this message the routines release hypotheses previously considered to have too little supporting evidence. This same message causes the recursion lines:

(APPEND <FORM X> (SELF <PARTOF X>))
(APPEND (SELF <PARTOF X>)<FORM X>)

to be hypothesized so that, if desperate, GAP can search blindly *for* a solution J

For a system like GAP, which attempts to write functions by associating an iopair with a particular programming construct, it is natural to ask what must be done to include a new construct. The heuristic routines described so far are all concerned with functions of type<LISTFN> , and I have deliberately omitted references to the routines for functions of type <FLATFN> and <TREEFN> . This reflects the historical development of the program - routines for these two types were added to the working system with only minor modifications being required.

In one sense, the addition of a new schema is trivial - we need only add a routine to the data base which blindly tries the new schema on all iopairs presented for hypothesis generation. Such a routine can be added in a matter of minutes - but unless heuristic routines are added to control the use of the new schema GAP will get involved in huge uncontrolled searches (which will, of course, eventually be successful). As an example of s\ich a large search, the validation of the hypothesis:

(LAMBDA(X Y)
(COND(NULL X) NIL)
((ATOM X)<BUILD X Y>)
(T(APPEND(SELF(CAR X))(SELF(CDR X))))))

for the iopair: ((A (B) C) D (E)) && "Q" =>=
(A Q B Q C Q D Q E Q)

took over a hundred seconds of CPU time; when a heuristic was added to GAP which realised that <BUILD X Y>could be replaced by (LIST<BUILD X Y><BUILD X Y>) the validation time dropped to under six seconds.

Other routines added to control the new schema recognise the concepts of 'treeness', when GAP decides that the input to an iopair is a tree it can ignore the < LISTFN > option for< FORM X> . An extension of the 'average' atom heuristic described earlier determines the choice of recursion for these two schemas and finally an extension of the length heuristic replaces the

<BUILD X> in the <FLATFN> schema by an appropriate (LIST <BUILD X> --- <BUILD X>) expression.

Section Three - The LISP System

As the reader has seen, the hypotheses made by GAP take the form of expression schemas. GAP will usually *know* to what such an expression is to evaluate but not necessarily the value of the variables it contains (since they may be the unknown inputs to a recursive call).

To assess the validity of such hypotheses GAP uses a special purpose LISP theorem prover. At it's simplest this takes a LISP expression and it's alleged value and deduces the values of variables contained in the expression. I call this 'anti-evaluation ' to emphasize the contrast with the deduction performed during LISP evaluation.

A LISP interpreter embodies the rules for the evaluation of LISP in a way that allows a tightly controlled deduction. This suggests the possibility of an 'anti-interpreter' which, when applied to an expression and its value returns possible sets of variable bindings. A LISP function, normally regarded as a program to compute some result from some arguments, could be viewed as an anti-LISP function to compute the arguments from the result:

There are a number of problems with the inversion of computable functions (7)- In general there can be any number - perhaps infinite, perhaps zero - of possible inputs that map onto a given output. (Anti-evaluation is a partial relation, not a total function.) If an expression to be anti-evaluated contains conditional expressions the anti-interpreter must search to find which alternatives could have been taken. (The conditional expression in LISP can be thought of as a non-deterministic statement (1) in anti-LISP.) An additional problem is that information about a variable's value i3 built up gradually during anti-evaluation - :n normal evaluation we know the value completely at all times and can easily represent it by an item on an ALIST.

GAP'S anti-interpreter must also cope with incomplete expressions. If it deduces the value of an expression < SCNETYPE X> it calls upon GAP to write suitable code; for example if told that (APPEND X <PARTOF X>) evaluates to(A B C D B C D) it deduces that one possibility is X = (A B C D), ^PARTOF X>= (CDR X).

Because of its crucial role in validating hypotheses the anti-interpreter should be quick - even at the cost of incompleteness and inconsistency. The search strategy used, and the representation for variable values are both quite weak. They are, however, adequate for GAP's purposes. (In general, such an approximate 'micro-theory' (10) will be of much greater practical use than a complete theory hundreds of times slower).

Section Four - Conclusions

A major criticism of GAP is that the way it forms and represents hypotheses is almost totally ad-hoc. This has two implications. Firstly, the unskilled user cannot understand the system's reasoning and this makes it impossible for him to contribute to hypothesis formation by supplying any information other than a single iopair - which is, of course, insufficient to describe the majority of LISP functions. If GAP used some widely known language (like LISP or predicate calculus) to represent all facts and hypotheses a much richer interaction with the user would be possible. For example, the system could 'think aloud' with the user interrupting or answering questions as necessary.

The second defect is more fundamental. Most of GAP's knowledge of programming consists of function schema's and associated cue-seeking routines. Ihese cue-seeking routines, it will be recalled, assess which schema should be instantiated to realise a particular iopair and suggest likely replacements for the 'slots' of the schema. Uiis knowledge is 'heavily compiled' into POPCORN methods so that to add new heuristics or schemas one must, at least, be able to program in POPCORN. Furthermore since this knowledge is almost totally unstructured, as the number *of* such fragments of knowledge rises so does the difficulty in avoiding undesirable interactions between heuristics. For example, after adding < TREEFN > s and<FLATFN> s to the system I found that heuristics specific to<LISTFN>s were incorrectly preventing the new schemas being tried on some iopairs.

The significance of this can be seen by considering the current version of GAP's behaviour on iopairs such as:

"C" && ((A W)(B X)(C Y)(D Z) => = "Y"
(A C D E) && (B D E F) => = (A B C D E F)

It could not produce the familiar ASSOC and UNION functions because these are not instantiations of known schemas - but there is no way for the user to tell GAP the appropriate schemas, nor, if he could, how to use theml

Bibliography

(1) Bruce Anderson, "Programming Languages for AI - The Role of Non-Determinism" M.Sc. Thesis Edinburgh 1972-

(2) J.R. Buchanan and D.C. Luckham,"On Automating the Construction of Programs" Stanford AI Memo, 1974.

(3) R. M. Bur stall, J.S. Collings and R.J. Popplestone, "Programming in POP2* Edinburgh University Press.

(4) R.O. Duda and P. Hart, "Experiments in the Recognition of Hand Printed Text" Proc. FJCC 1968 pp.1139-1151 .

(5) Steven Hardy, The POPCORN Reference Manual= CSM-1 Essex University 1973.

(0) M.D. Kelly, "Edge Detection in Pictures by Computer using Planning*, Machine Intelligence, 5 Edinburgh University Press.

(7) John McCarthy, "The Inversion of Functions defined by Turing Machines", Automata Studies - Annals of Mathematics Science 1956, Eds, Shannon, J. McCarthy, Vol.jUi Princeton.

(8) John McCarthy, "Towards a Mathematical Science of Computation" Proc. IFIP 1963-

(9) D. McDermott and G.J. Sussman, "The CONNIVER Reference Manual." AI Memo 259, M.I.T.

(10) M. Minsky and S. Papert, "Progress Report on Artificial Intelligence" AI Memo 252, M.I.T. Artificial Intelligence Laboratory.

(11) Y. Shirai, "A Context Sensitive Line Finder for Recognition of Polyhedra" Artificial Intelligence, Vol.4. (1973).

Gerald J. Sussman, "A Computational Model of Skill Acquisition" Ph.D. Thesis, M.I.T. 1974.

(13) R.J. Waldinger, "Constructing Programs Automatically using Theorem Proving* Ph.D. Thesis, Carnegie-Mellon University.

Appendix - Some Examples

```
(A B C D) && "Q" => = (A Q B Q C Q D Q);
**(LAMBDA (X Y)
  (COND((ATOM X) NIL)
  (T(CONS(CAR X)(CONS Y(SELF(CDR X)Y))))))
```

[Time taken was 3.036 seconds]

```
(A B C D E F) => = (A B B C D D E F F);
** (LAMBDA(X)
   (COND((OR(ATOM X)(ATOM(CDR X)))NIL)
   (T(CONS(CAR X)
   (CONS(CAR(CDR X))
   (CONS(CAR(CDR X))(SELF(CDR(CDR X)))))))))
```

[Time taken was 6.42? seconds]

```
(A B C D E F) && "Q" => = (A B B Q C D D Q E F F Q);
**(LAMBDA(X Y)
  (COND((OR(ATOM X)(ATOM(CDR X)))NIL)
  (T
  (CONS(CAR X)
  (CONS(CAR(CDR X))
  (CONS(CAR(CDR X))(CONS Y(SELF(CDR(CDR X))Y)))))))))
```

[Time taken was 4.808 seconds]

```
(A B C D) => = (A B C D B C D C D D);
**(LAMBDA(X)
  (COND((ATOM X)NIL)(T(APPEND X(SELF(CDR X))))))
```

[Time taken was 6.737 seconds]

```
(A B C D) && "Q" => = (A B C D Q B C D Q C D Q D Q);
**(LAMBDA(X Y)
  (COND((ATOM X)NIL)
  (T(APPEND X(CONS Y(SELF(CDR X)Y))))))
```

[Time taken was 10.57? seconds]

```
((A(B C)D)E) => = (E(D(C B)A));
**(LAMBDA(X)
   (COND((NULL X)NIL)
   (T(APPEND(SELF(CDR X))
   (LIST
   (COND(ATOM(CAR X))(CAR X))(T(SELF(CAR X))))))))))
```

[Time taken was 12.472 seconds]

```
(A B C D) => = ((A)(B)(C)(D)(B)(C)(D)(C)(D)(D));
**(LAMBDA(X)
  (COND((ATOM X)NIL
  (T(APPEND((LAMBDA(X)
  (COND(ATOM X)NIL)
  (T(CONS(LIST(CAR X))(SELF(CDR X))))))
   X)
  (SELF(CDR X))))))
```

[Time taken was 11.402 seconds]

```
(A B C D) => = (D C B A);
**(LAMBDA(X)
  (COND((ATOM X)NIL)
  (T(APPEND(SELF(CDR X))(LIST(CAR X))))))
```

[Time taken was 3.685 seconds]

```
(A B C D) && "Q" => = (D Q C Q B Q A Q);
**(LAMBDA(X Y)
  (COND((ATOM X)NIL)
  (T(APPEND(SELF(CDR X)Y)(LIST(CAR X)Y)))))
```

[Time taken was 4.863 seconds]

```
((A(BC)D)E)=>= (A B C D E);
**(LAMBDA(X)
  (COND((NULL X)NIL)
  ((ATOM X)(LIST X))
  (T(APPEND(SELF(CAR X))(SELF(CDR X))))))
```

[Time taken was 4.058 seconds]

```
((A(B C)D)E) && "Q" => = (A Q B Q C Q D Q E Q);
**(LAMBDA(X Y)
  (COND((NULL X)NIL)
  ((ATOM X)(LIST X Y))
  (T(APPEND(SELF(CAR X)Y)(SELF(CDR X)Y)))))
```

[Time taken was 5.003 seconds]

```
((A(B C)D)E) && "Q" => = ((A Q(B Q C Q)D Q)E Q);
**(LAMBDA(X Y)
  (COND((NULL X)NIL)
  (T(APPEND(COND((ATOM(CAR X))(LIST(CAR X)Y))
  (T(LIST(SELF(CAR X)Y))))
  (SELF(CDR X)Y)))))
```

[Time taken was 13.384 seconds]