LAMBDA-Graphs: A Replacement for PROG

James R. Meehan

ABSTRACT. The PROG feature in LISP can be viewed as a notation for a graph of expressions, corresponding to the standard flowchart. Although PROG and it associated special forms, GO and RETURN, are commonly implemented as LISP functions, we show that PROGs can be transformed into equivalent LAMBDA-expressions that contain no GO or RETURN statements at all, and therefore run faster than PROG by avoiding runtime overhead. The transformation program is included in the report.

Technical Report 135

December 1979

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717

CONTENTS

LAMBDA-Graphs: A Replacement for PROG

James R. Meehan

1.0 Introduction

The "purity" of LISP without the PROG feature derives from the simplicity of the syntax and semantics of Cambridge Polish, LAMBDA-expressions, recursion, and COND. Among such company, the lowly PROG is something of a black sheep, aesthetically. Jumping here and there like an assembly program, PROG seems an all-too-obvious reminder of the underlying architecture. Control flow in the rest of LISP is much more elegant, even in the extended definitions of LAMBDA and COND that permit a sequence of expressions where ur-LISP permitted only one; e.g., (LAMBDA (...) e1 e2 e3) and (COND (p1 e1 e2 e3 ...) ...). Here, the elementary CDR suffices to control the flow of evaluation.

Of course, PROG is a necessary part of LISP, since it permits a more efficient implementation of iteration than is generally possible with recursion. The body of a PROG is also the closest LISP comes to expressing the venerable flowchart, which seems fair to survive as a conceptual tool of programmers.

In Stanford LISP 1.6 and its descendants, PROG is implemented as a compiled FEXPR, one of the so-called "special forms" in LISP. While all FEXPRs control the evaluation of their parameters, PROG also controls the sequencing by means of two more special forms, GO and RETURN, which access global variables in the interpreter that are re-bound every time PROG is called, old values going onto a runtime stack. The expression (GO LOOP) is interpreted by fetching a pointer to the PROG-body and performing a linear search (!) for the atom LOOP. Thus, re-ordering the statements in a PROG can affect its speed, since "upward" GOs are faster than "downward" GOs.

While PROG may be necessary to LISP, that particular implementation of PROG is not, and what we describe here is another implementation, involving a one-time pre-processing cost (transforming the PROG into what we call a LAMBDA-graph). PROG is redefined as a MACRO; RETURN and GO are entirely eliminated from the LISP interpreter.

The LAMBDA-graph runs faster than the original PROG since there is no overhead for PROGs, GOs, and RETURNs, which have all been eliminated. The increase in speed is directly proportional to the frequency with which GOs and RETURNs were executed in the PROG. A "straight-line" PROG with no GOs at all does not benefit measurably from being transformed into a LAMBDA-graph. A small, single-loop function such as DREVERSE runs about 10% faster as a

LAMBDA-graph. The greatest improvements appear in long, "loopy" PROGs with lots of GOs and top-level CONDs, particularly where the GOs are embedded in several layers of CONDs.

In effect, the LAMBDA-graph provides some of the benefits that the compiling LISP code does.

The practical disadvantage of LAMBDA-graphs is that they are, indeed, graph structures, "unprintable" S-expressions, likely to cause havoc with unsuspecting tools of the environment such as the editor or Break Package. Of course, it is quite possible to reverse the transformation by inserting labels and GOs. The result would be equivalent but not identical to the original PROG, since there are several ways one could insert the labels and GOs. (Lacking any jumps whatsoever, LAMBDA-graphs avoid the problems of being sensitive to upward-vs.-downward jumps.)

In UCI LISP, however, macros are expanded in place, retaining both the original and the expanded forms, and most of the environmental tools are smart enough to access the original form, so that the LAMBDA-graphs are not prohibitively troublesome.

Apart from the practical considerations, LAMBDA-graphs are of aesthetic interest, perhaps, because they retain PROG as a notation for flowcharts without actually being interpreted like assembly programs, and also because they

point out that, somewhat unexpectedly, EVAL requires no modification to interpret code that is represented by a data structure -- the graph -- that it was not originally designed to handle.

We begin by presenting the time-honored definition of the LISP interpreter, updated somewhat. From the earliest LISP articles [1,2], this has usually been done in LISP itself, but here we use the more legible notation of MLISP [4,5], since the source language of the interpreter is irrelevant to the discussion. For those unfamiliar with MLISP, a brief introduction appears in Appendix A. The LISP translation of all the MLISP code appears in Appendix B.

2.0 EVAL and APPLY

Our definition of LISP's top-level functions assumes that variables are bound on an association list (ALIST) that must be passed to any function that does its own evaluation (e.g., FEXPRs). We assume that function definitions are stored on property lists.

```
EXPR EVAL (X, ALIST);
   BEGIN NEW Y;
   RETURN
      IF ATOM X THEN
         IF NULL X ¦ X EQ T ¦ NUMBERP X ¦ STRINGP X THEN X
         ELSE IF Y _ ASSOC (X, ALIST) THEN CDR Y
         ELSE ERROR <"UNBOUND VARIABLE", X>
      ELSE IF ATOM X[1] THEN
         ELSE IF Y _ GET (X[1], 'EXPR) THEN
            APPLY (Y, EVLIS (CDR X, ALIST), ALIST)
         ELSE IF Y _ GET (X[1], 'FEXPR) THEN
            APPLY (Y, <CDR X, ALIST>, ALIST)
         ELSE IF Y _ GET (X[1], 'MACRO) THEN
            EVAL (APPLY (Y, <X>, ALIST), ALIST)
         ELSE IF Y _ ASSOC (X[1], ALIST) THEN
            EVAL (CDR Y CONS CDR X, ALIST)
         ELSE ERROR <"UNDEFINED FUNCTION", X[1]>
      ELSE APPLY (X[1], EVLIS (CDR X, ALIST), ALIST);
   END;

EXPR APPLY (FN, ARGS, ALIST);
  (WHILE ATOM FN ¦ FN[1] NEQ 'LAMBDA DO:
     FN _ ATOM FN & GET (FN, 'EXPR) ¦ EVAL (FN, ALIST),
   EVSEQ (CDDR FN, MAPCAR ('CONS, FN[2], ARGS) @ ALIST));

{;; EVLIS evaluates all the expression in the list X and
creates a new list of their values. };
EXPR EVLIS (X, ALIST);
   FOR NEW I IN X COLLECT <EVAL (I, ALIST)>;

{;; EVSEQ evaluates all the expression in the list X and
returns the value of the last expresion. };
EXPR EVLIS (X, ALIST);
   FOR NEW I IN X DO <EVAL (I, ALIST)>;

FEXPR COND (CLAUSES, ALIST);
  BEGIN NEW VAL, EXPRLIST;
1;IF NULL CLAUSES THEN RETURN NIL;
  VAL _ EVAL (CLAUSES[1,1], ALIST);
  IF NULL VAL THEN CLAUSES _ CDR CLAUSES ALSO GO 1;
  EXPRLIST _ CDR CLAUSES[1];
2;IF NULL EXPRLIST THEN RETURN VAL;
  VAL _ EVAL (EXPRLIST[1], ALIST);
  EXPRLIST _ CDR EXPRLIST;
  GO 2;
  END;
```

```
{;; PROG, definition 1. NOTE: this definition is incomplete.};
FEXPR PROG (PBODY, ALIST);
   BEGIN NEW PROGVARS, STATEMENT, BODY;
   PROGVARS _ PBODY[1];
   BODY _ PBODY _ CDR PBODY;
   FOR NEW I IN PROGVARS DO:
      ALIST _ (I CONS NIL) CONS ALIST;
   {;; Bind the PROG-variables to NIL};
1; IF NULL BODY THEN RETURN NIL;
   STATEMENT _ BODY[1];
   IF ATOM STATEMENT THEN {;; It's a label.  Skip it.}
   ELSE IF STATEMENT[1] EQ 'GO
      THEN {;; Find the tail of the PROG-body that starts
                with the label.}
      ALSO BODY _ STATEMENT[2] MEMQ PBODY
      ALSO (IF NULL BODY
                THEN ERROR <"UNDEFINED LABEL",
                   STATEMENT[2]>)
   ELSE IF STATEMENT[1] EQ 'RETURN
      THEN RETURN EVAL (STATEMENT[2], ALIST)
   ELSE EVAL (STATEMENT, ALIST);
   BODY _ CDR BODY;
   GO 1;
   END;
```

3.0 COND inside PROG

In the original version of LISP, SETQ was allowed only inside PROG, and COND was defined differently. We will skip over SETQ here and concentrate on COND. In our current definition of PROG, we made no special check for COND, so it would get passed to EVAL. But that isn't good enough, because there might be a GO or RETURN inside the COND, and EVAL doesn't know anything about those; only PROG does. What we want to do is to treat the expressions inside COND (past the predicates) as if they occurred at the top level of PROG.

In a sense, they already do. After evaluating the expressions that follow the first true predicate, we go on to the statement following the COND (unless we reached a GO or RETURN). So there's no real distinction between the statements that follow a "true" predicate and the statements that follow the COND: they're both "top-level." For example, the following PROGs are equivalent:

```
{;; Example 1.};
   BEGIN NEW A,B,C;
   {;; ...};
   IF A EQ 3 THEN PRINT A ALSO PRINT B
   ELSE PRINT B;
   FOO (BAZ);
   {;; ...};
   RETURN Z;
   END;
```

```
{;; Example 2.};
   BEGIN NEW A,B,C;
   {;; ...};
   IF A EQ 3 THEN GO 1
   ELSE GO 2;
3; FOO(BAZ);
   {;; ...};
   RETURN Z;
1; PRINT A;
   PRINT B;
   GO 3;
2; PRINT B;
   GO 3;
   END;
```

If the PROG doesn't end with a RETURN or a GO, we can insert a RETURN NIL.

So we revise our definition of PROG, illustrating the equivalence of code and data in LISP: Where the control would flow from one sequence to another, we literally APPEND (@) the lists together containing those sequences.

```
{;; PROG, definition 2.};
FEXPR PROG (PBODY, ALIST);
   BEGIN NEW PROGVARS, STATEMENT, BODY;
   PROGVARS _ PBODY[1];
   BODY _ PBODY _ CDR PBODY;
   FOR NEW I IN PROGVARS DO:
      ALIST _ (I CONS NIL) CONS ALIST;
   {;; Bind the PROG-variables to NIL};
1; IF NULL BODY THEN RETURN NIL;
   STATEMENT _ BODY[1];
   IF ATOM STATEMENT THEN NIL
   ELSE IF STATEMENT[1] EQ 'GO
      THEN {;; ... (as before)}
   ELSE IF STATEMENT[1] EQ 'RETURN
      THEN {;; ... (as before)}
   ELSE IF STATEMENT[1] EQ 'COND
      THEN BODY _ FINDTRUECLAUSE (CDR STATEMENT, ALIST) @
          CDR BODY
      ALSO GO 1
   ELSE EVAL (STATEMENT, ALIST);
   BODY _ CDR BODY;
   GO 1;
   END;

EXPR FINDTRUECLAUSE (CLAUSES, ALIST);
  BEGIN
1;IF NULL CLAUSES THEN RETURN NIL;
  IF NULL EVAL (CLAUSES[1,1], ALIST)
     THEN CLAUSES _ CDR CLAUSES
     ALSO GO 1;
  RETURN CDR CLAUSES[1];
  END;
```

The list of expressions that FINDTRUECLAUSE returns is APPENDed to the remainder of the PROG body. If all the predicates evaluated to NIL, then FINDTRUECLAUSE returns NIL, and we APPEND that NIL to the CDR of BODY; i.e., we "drop off" into the next statement.

4.0 Transforming PROG-bodies into LAMBDA-graphs

PROG is now correct, but our hacker's hackles are raised at the thought of having to use APPEND, creating all that new storage. Can we avoid it? Can we use NCONC?

We are suggesting that each branch of the COND, except those that end in GO or RETURN, actually be spliced into the statement following the COND. Since we've already seen that that's the same as using GO, we can do the same for all GO-statements, too, splicing them into their "targets." We can then eliminate the special case for GO. (GO-TO elimination with a vengeance!) There's no need to keep the labels around, either.

Since each branch of the COND will be spliced into subsequent code, we can treat COND as if it were the last statement in the PROG and delete the link to its successor.

Before we do any of this, we must insist that a "true" branch exist in the COND, just as in original LISP. So if there isn't one there, we insert a (T) branch that we later splice into the statement following the COND.

At this point, PROG-bodies no longer resemble sequences; in fact, they're graphs, since our splices can go anywhere inside the PROG.

5.0 Eliminating PROGs altogether

The only thing left that makes PROG unique is the RETURN statement. In one stroke, we eliminate both PROG and RETURN.

In any sequence that includes a RETURN statement, we can delete the link connecting RETURN to the remainder of the sequence (we can't get there, anyway), and reduce (RETURN X) to X. X will now be the last expression to be evaluated, and we can have that value returned because we're going to turn the PROG into a LAMBDA-expression by adding on as many NILs as there are PROG-variables. So (PROG (A B C) ...) becomes ((LAMBDA (A B C) ...) NIL NIL NIL).

(The entire transformation program appears in Appendix C.)

Thus, having eliminated GO and RETURN, we can do without the PROG FEXPR. EVAL and APPLY process the LAMBDA-graph as they would any other LAMBDA-expression.

6.0 A flaw, and a fix

There is a problem, however. We've forgotten what happens to the runtime stack. Whenever a FEXPR such as COND is evaluated, EVAL is eventually going to call itself, which requires putting (at least) a return address on the stack. Consider the function LENGTH:

```
{;; LENGTH, definition 1.};
EXPR LENGTH (L);
  BEGIN NEW COUNT;
  COUNT   0;
1;IF ~ATOM L THEN L _ CDR L ALSO COUNT   COUNT+1 ALSO GO 1;
  RETURN COUNT;
  END;
```

That will be transformed into:

```
{;; LENGTH, definition 2.};
EXPR LENGTH (L);
  LAMBDA (COUNT);
   (COUNT   0,
    IF ~ATOM L
       THEN L   CDR L
       ALSO COUNT   COUNT+1
       ALSO IF ~ATOM L
               THEN L   CDR L
               ALSO COUNT   COUNT+1
               ALSO IF ~ATOM L
                       THEN L   CDR L
                       ALSO COUNT   COUNT+1
                       ALSO {;; ....}
                    ELSE COUNT
            ELSE COUNT
    ELSE COUNT);
  (NIL);
```

Every time we evaluate the COND (IF), the stack grows. We might just as well have defined LENGTH recursively!

Somehow, we've got to get the stack back down to where it was when we first encountered the COND. We don't need any of the information that's on the stack above the COND, so there's no harm in deleting it. We look at two solutions: the Hacker's Delight, and the Better Way.

6.1 The Hacker's Delight

In some LISP systems, there are functions that allow the programmer to access the real, live stack. (Faint-hearted advocates of structured programming are advised to skip to the next section.) In UCI LISP, there is a function called SPREVAL.

> (SPREVAL P V) evaluates its argument V in its local context to get a form, and then it returns to the context specified by [stack pointer] P and evaluates the form in that context, returning from that context with the value. This is very similar to SPREDO except that the EVAL-blip on the stack is changed. [3, page 146]

That's what we need, if only we can figure out how to save the stack pointer. We need one pointer for each level of PROG, so we can't store it in a single global cell. Solution: we include a new LAMBDA-variable called GCL (Graph-COND-Level) whenever we transform a PROG into a LAMBDA-graph, and we change the names of all the top-level CONDs in PROGs to GCOND (so that we don't conflict with CONDs that really do compute a value). We define a FEXPR called GCOND that simply calls COND, but only after checking to see whether the stack needs shrinking. If it's the first evaluation of GCOND in that LAMBDA, GCL will be NIL, in which case we set it to the current stack pointer (via the function SPDLPT). Otherwise, we reset the stack pointer to the value of GCL and go on. The code is very simple:

```
{;; GCOND, definition 1.};
FEXPR GCOND (L, ALIST);
  SPREVAL (GCL ¦ GCL _ SPDLPT(), 'COND CONS L);
```

That resets the stack pointer, changes the current expression on the stack from (GCOND ...) to (COND ...) and picks up from there.

While GCOND is short and simple, it is also slower than the PROG in LISP 1.6, partly because SPREVAL takes some time, and partly because we call CONS each time. (We can't use RPLACA to change GCOND to COND in place, because then we'd never get GCOND on the stack again.)

6.2 The Better Way

We can solve our stack and storage problems if we define GCOND to be the same as COND, with one small addition:

```
{;; GCOND, definition 2.};
FEXPR GCOND (CLAUSES, ALIST);
  BEGIN NEW VAL, EXPRLIST;
1;IF NULL CLAUSES THEN RETURN NIL;
  VAL _ EVAL (CLAUSES[1,1], ALIST);
  IF NULL VAL THEN CLAUSES _ CDR CLAUSES ALSO GO 1;
  EXPRLIST _ CDR CLAUSES[1];
2;IF NULL EXPRLIST THEN RETURN VAL;
  {;; The new statement: };
  IF ~ATOM EXPRLIST[1] & EXPRLIST[1,1] EQ 'GCOND
     THEN CLAUSES _ CDR EXPRLIST[1]
     ALSO GO 1;
  VAL _ EVAL (EXPRLIST[1], ALIST);
  EXPRLIST _ CDR EXPRLIST;
  GO 2;
  END;
```

The extra test (following 2) checks to see if the expression to be evaluated is a GCOND. If so, then we simply go back to the outer loop (1).

APPENDIX A

Introduction to MLISP

MLISP is an ALGOL-like dialect of LISP. That is, MLISP source code is translated into LISP and runs in a LISP core image. It was first developed in 1970 by David C. Smith of Stanford University [5], and has been extended and greatly revised to be compatible and concurrent with UCI LISP [3]. MLISP is now maintained and developed at UCI [4]. With the exception of the READMACRO facility in LISP, all LISP code is expressible in MLISP.

Most MLISP expressions have obvious meanings in LISP, given that the syntax is like ALGOL's. LISP's (CAR A) can be written in MLISP as CAR (A), but it can also be written as CAR A and A[1]. Any one-parameter LISP function can be written as a "prefix" function without parentheses, as in CAR A. Any two-parameter LISP function can be written as an "infix" function without parentheses, as in X CONS Y. Any LISP function at all can be written using parentheses around the parameters, as in MEMQ (I, L). Many common functions

have single-character synonyms: + for PLUS, ~ for NOT, & for AND, ¦ for OR, _ for SETQ, @ for APPEND, and = for EQUAL.

In addition, MLISP uses a variety of syntactic forms for LISP's special forms. Instead of COND, MLISP uses IF-THEN-ELSE. Additional expressions in a COND-clause are denoted by ALSO. E.g.,

(COND ((EQ X Y) (PRINT X) (PRINT Y)))
is written
IF X EQ Y THEN PRINT X ALSO PRINT Y.
Angle brackets indicate lists, so that (LIST 1 2 3) would be written <1,2,3>.

MLISP has a FOR-loop with a wide variety of options. FOR NEW I IN L DO E is translated into a PROG with a local "index" variable I that points to successive CARs of the list L (a la MAPC), evaluating E (any expression) each time, finally returning the last value E had. FOR NEW I IN L DO: E (DO with a colon) always returns NIL, so it translates into (MAPC (FUNCTION (LAMBDA (I) E)) L).

FOR NEW I IN L COLLECT E evaluates E as before, but it also "collects" the values together in a list, APPENDing each value to the end of the resulting list. Its translation is

(MAPCAN (FUNCTION (LAMBDA (I) (APPEND E NIL))) L)

The APPEND ... NIL merely makes a top-level copy of E, for

safety.

If the word IN is replaced with ON, we get functions similar to MAP, MAPLIST, and MAPCON. If the FOR-loop is followed by UNTIL and some expression B, then the loop will stop as soon as the B's value is non-NIL. If the word NEW is missing, then the index variable is not local to the PROG. Finally, if there is no NEW and there is an UNTIL-condition and the UNTIL-condition comes true, the loop will stop, and the index variable will still point to the item in the list for which the condition was true; if the loop goes all the way through the list without the UNTIL-condition becoming true, then the index variable is set to NIL. This is used in EVCONL.

"Subscript brackets" are used in place of CAR. A[1] means (CAR A), A[2] means (CADR A), A[1,2] means (CADR (CAR A)), and so on.

Finally, BEGIN NEW A,B,C; ... END is MLISP's way of saying (PROG (A B C) ...).

LISP translations of all the MLISP code in the text appear in Appendix B. For more details about MLISP, see the reference manual [4].

APPENDIX B

LISP translations of the MLISP code

```
(DEFPROP EVAL
 (LAMBDA (X ALIST)
  (PROG (Y)
        (RETURN
         (COND [(ATOM X)
                (COND [(OR [NULL X]
                           [EQ X T]
                           [NUMBERP X]
                           [STRINGP X])
                       X]
                      [(SETQ Y (ASSOC X ALIST)) (CDR Y)]
                      [T (ERROR (LIST "UNBOUND VARIABLE" X))
                       ])]
               [(ATOM (CAR X))
                (COND [(SETQ Y (GET (CAR X) 'EXPR))
                       (APPLY Y (EVLIS (CDR X) ALIST) ALIST)
                       ]
                      [(SETQ Y (GET (CAR X) 'FEXPR))
                       (APPLY Y (LIST (CDR X) ALIST) ALIST)]
                      [(SETQ Y (GET (CAR X) 'MACRO))
                       (EVAL (APPLY Y (LIST X) ALIST) ALIST)
                       ]
                      [(SETQ Y (ASSOC (CAR X) ALIST))
                       (EVAL (CONS (CDR Y) (CDR X)) ALIST)]
                      [T (ERROR (LIST "UNDEFINED FUNCTION"
                                      (CAR X)))])]
               [T (APPLY (CAR X)
                         (EVLIS (CDR X) ALIST)
                         ALIST)])))))
 EXPR)


(DEFPROP APPLY
 (LAMBDA (FN ARGS ALIST)
```

```
  (PROG NIL
   LOOP (COND [(OR [ATOM FN] [NEQ (CAR FN) 'LAMBDA])
               (SETQ FN
                     (OR [AND [ATOM FN] [GET FN 'EXPR]]
                         [EVAL FN ALIST]))
               (GO LOOP)]))
  (EVSEQ (CDDR FN)
         (APPEND (&VECTOR NIL 'CONS (CADR FN) ARGS) ALIST)))
 EXPR)


{;; EVLIS evaluates all the expression in the list X and
    creates a new list of their values.}


(DEFPROP EVLIS
 (LAMBDA (X ALIST)
  (MAPCAR (FUNCTION (LAMBDA (I) (EVAL I ALIST))) X))
 EXPR)


{;; EVSEQ evaluates all the expression in the list X and
    returns the value of the last expresion.}


(DEFPROP EVLIS
 (LAMBDA (X ALIST)
  (PROG (&V &LST1 I)
        (SETQ &LST1 X)
   LOOP (COND [(NULL &LST1) (RETURN &V)])
        (SETQ I (CAR &LST1))
        (SETQ &LST1 (CDR &LST1))
        (SETQ &V (LIST (EVAL I ALIST)))
        (GO LOOP)))
 EXPR)


(DEFPROP COND
 (LAMBDA (CLAUSES ALIST)
  (PROG (VAL EXPRLIST)
      1 (COND [(NULL CLAUSES) (RETURN NIL)])
        (SETQ VAL (EVAL (CAR (CAR CLAUSES)) ALIST))
        (COND [(NULL VAL)
               (SETQ CLAUSES (CDR CLAUSES))
               (GO 1)])
        (SETQ EXPRLIST (CDR (CAR CLAUSES)))
      2 (COND [(NULL EXPRLIST) (RETURN VAL)])
        (SETQ VAL (EVAL (CAR EXPRLIST) ALIST))
        (SETQ EXPRLIST (CDR EXPRLIST))
        (GO 2)))
 FEXPR)


{;; PROG, definition 1.  NOTE: this definition is
    incomplete.}
```

```
(DEFPROP PROG
 (LAMBDA (PBODY ALIST)
  (PROG (PROGVARS STATEMENT BODY)
        (SETQ PROGVARS (CAR PBODY))
        (SETQ BODY (SETQ PBODY (CDR PBODY)))
        (MAPC (FUNCTION
               (LAMBDA (I)
                (SETQ ALIST (CONS (CONS I NIL) ALIST))))
              PROGVARS)
        {;; Bind the PROG-variables to NIL}
      1 (COND [(NULL BODY) (RETURN NIL)])
        (SETQ STATEMENT (CAR BODY))
        (COND [(ATOM STATEMENT)
               {;; It's a label.  Skip it.}]
              [(EQ (CAR STATEMENT) 'GO)
               {;; Find the tail of the PROG-body that
                   starts with the label.}
               (SETQ BODY (MEMQ (CADR STATEMENT) PBODY))
               (COND [(NULL BODY)
                      (ERROR (LIST "UNDEFINED LABEL"
                                   (CADR STATEMENT)))])]
              [(EQ (CAR STATEMENT) 'RETURN)
               (RETURN (EVAL (CADR STATEMENT) ALIST))]
              [T (EVAL STATEMENT ALIST)])
        (SETQ BODY (CDR BODY))
        (GO 1)))
 FEXPR)


{;; Example 1.}


(PROG (A B C)
      {;; ...}
      (COND [(EQ A 3) (PRINT A) (PRINT B)] [T (PRINT B)])
      (FOO BAZ)
      {;; ...}
      (RETURN Z))


{;; Example 2.}


(PROG (A B C)
      {;; ...}
      (COND [(EQ A 3) (GO 1)] [T (GO 2)])
    3 (FOO BAZ)
      {;; ...}
      (RETURN Z)
    1 (PRINT A)
      (PRINT B)
      (GO 3)
    2 (PRINT B)
```

```
      (GO 3))

{;; PROG, definition 2.}

(DEFPROP PROG
 (LAMBDA (PBODY ALIST)
  (PROG (PROGVARS STATEMENT BODY)
        (SETQ PROGVARS (CAR PBODY))
        (SETQ BODY (SETQ PBODY (CDR PBODY)))
        (MAPC (FUNCTION
               (LAMBDA (I)
                (SETQ ALIST (CONS (CONS I NIL) ALIST))))
              PROGVARS)
        {;; Bind the PROG-variables to NIL}
      1 (COND [(NULL BODY) (RETURN NIL)])
        (SETQ STATEMENT (CAR BODY))
        (COND [(ATOM STATEMENT) NIL]
              [(EQ (CAR STATEMENT) 'GO)
               {;; ...  (as before)}]
              [(EQ (CAR STATEMENT) 'RETURN)
               {;; ...  (as before)}]
              [(EQ (CAR STATEMENT) 'COND)
               (SETQ BODY
                     (APPEND (FINDTRUECLAUSE
                              (CDR STATEMENT)
                              ALIST)
                             (CDR BODY)))
               (GO 1)]
              [T (EVAL STATEMENT ALIST)])
        (SETQ BODY (CDR BODY))
        (GO 1)))
 FEXPR)


(DEFPROP FINDTRUECLAUSE
 (LAMBDA (CLAUSES ALIST)
  (PROG NIL
      1 (COND [(NULL CLAUSES) (RETURN NIL)])
        (COND [(NULL (EVAL (CAR (CAR CLAUSES)) ALIST))
               (SETQ CLAUSES (CDR CLAUSES))
               (GO 1)])
        (RETURN (CDR (CAR CLAUSES)))))
 EXPR)


{;; LENGTH, definition 1.}

(DEFPROP LENGTH
 (LAMBDA (L)
  (PROG (COUNT)
        (SETQ COUNT 0)
      1 (COND [(NOT (ATOM L))
```

```
                (SETQ L (CDR L))
                (SETQ COUNT (ADD1 COUNT))
                (GO 1)])
        (RETURN COUNT)))
 EXPR)


{;; LENGTH, definition 2.}


(DEFPROP LENGTH
 (LAMBDA (L)
  ((LAMBDA (COUNT)
    (SETQ COUNT 0)
    (COND [(NOT (ATOM L))
           (SETQ L (CDR L))
           (SETQ COUNT (ADD1 COUNT))
           (COND [(NOT (ATOM L))
                  (SETQ L (CDR L))
                  (SETQ COUNT (ADD1 COUNT))
                  (COND [(NOT (ATOM L))
                         (SETQ L (CDR L))
                         (SETQ COUNT (ADD1 COUNT))
                         {;; ....}]
                        [T COUNT])]
                 [T COUNT])]
          [T COUNT]))
   NIL))
 EXPR)


{;; GCOND, definition 1.}


(DEFPROP GCOND
 (LAMBDA (L ALIST)
  (SPREVAL (OR GCL [SETQ GCL (SPDLPT)]) (CONS 'COND L)))
 FEXPR)


{;; GCOND, definition 2.}


(DEFPROP GCOND
 (LAMBDA (CLAUSES ALIST)
  (PROG (VAL EXPRLIST)
      1 (COND [(NULL CLAUSES) (RETURN NIL)])
        (SETQ VAL (EVAL (CAR (CAR CLAUSES)) ALIST))
        (COND [(NULL VAL)
               (SETQ CLAUSES (CDR CLAUSES))
               (GO 1)])
        (SETQ EXPRLIST (CDR (CAR CLAUSES)))
      2 (COND [(NULL EXPRLIST) (RETURN VAL)])
        {;; The new statement:}
        (COND [(AND [NOT (ATOM (CAR EXPRLIST))]
```

```
                    [EQ (CAR (CAR EXPRLIST)) 'GCOND])
                (SETQ CLAUSES (CDR (CAR EXPRLIST)))
                (GO 1)])
        (SETQ VAL (EVAL (CAR EXPRLIST) ALIST))
        (SETQ EXPRLIST (CDR EXPRLIST))
        (GO 2)))
FEXPR)
```

APPENDIX C

The PROG-transformer

```
MACRO PROG (X); EXPANDPROG (X);

EXPR EXPANDPROG (X);
   BEGIN NEW LABELS, GOLIST, BODY, SS, P2;
   SS _ BODY _ COPY CDR X;
   {;; Make sure that all the COND's have a final T clause.}
      ;
   FOR NEW I IN BODY DO:
      IF   CONSP I
         & I[1] EQ 'COND
         & (LAST I)[1,1] NEQ T THEN NCONC (I, <<T>>);
   {;; Also make sure that the last statement is a RETURN.}
      ;
   IF (LAST SS)[1,1] NEQ 'RETURN
      THEN NCONC (SS, <<'RETURN, NIL>>);
   {;; Build an association list between labels and
       statements, and delete the labels.};
   WHILE CDR BODY DO:
      IF ATOM BODY[2]
         THEN LABELS   BODY[2] CONS LABELS
         ALSO RPLACD (BODY, CDDR BODY)
      ELSE GOLIST _    (  FOR NEW J IN LABELS COLLECT
                             <J CONS CDR BODY>)
                       @ GOLIST
         ALSO LABELS   NIL
         ALSO BODY   CDR BODY;
   {;; Splice COND-clauses into their successors.};
   FOR NEW I ON SS DO:
      IF I[1,1] EQ 'COND
         THEN FOR NEW J IN CDAR I DO:
                  IF ~((LAST J)[1,1] MEMQ '(GO RETURN))
                     THEN PROGN (NCONC (J,
                                  <<'GO, CDR I>>),
                                 GOLIST   (CDR I CONS CDR I)
                                 CONS GOLIST);
   {;; Build an association list between each statement and
       its successor.};
   P2 _ FOR NEW I ON SS COLLECT <I>;
   FOR NEW P IN P2 DO:
```

```
      BEGIN NEW Q, S, FN;
   1; IF NULL Q _ CDR P THEN RETURN NIL;
      {;; Q is the 'next' statement.};
      S _ Q[1];
      {;; S is 'this' statement.};
      FN _ S[1];
      {;; FN is the function for S.  (No more labels)};
      {;; Splice in the new address.};
      IF FN EQ 'GO
         THEN RPLACD (P, CDR ASSOC (S[2], GOLIST))
         ALSO GO 1;
      {;; COND will be the last expression in a sequence.
          Change its name to GCOND.  Add its clauses to the
          top level.};
      IF FN EQ 'COND
         THEN RPLACD (Q, NIL)
         ALSO RPLACA (S, 'GCOND)
         ALSO FOR NEW I IN CDR S DO:
                 P2 NCONC (
                    FOR NEW J ON I COLLECT <J>);
      {;; RETURN is also last in a sequence.  Keep only the
          parameter.};
      IF FN EQ 'RETURN
         THEN RPLACD (Q, NIL)
         ALSO RPLACA (Q, S[2]);
      END;
   {;; Construct the LAMBDA.};
   RETURN     ('LAMBDA CONS X[2] CONS CDR SS)
         CONS ( FOR NEW I IN X[2] COLLECT <NIL>);
   END;
```

*** LISP translations ***

```
(DEFPROP PROG (LAMBDA (X) (EXPANDPROG X)) MACRO)


(DEFPROP EXPANDPROG
 (LAMBDA (X)
  (PROG (LABELS GOLIST BODY SS P2)
        (SETQ SS (SETQ BODY (COPY (CDR X))))
        {;; Make sure that all the COND's have a final T
            clause.}
        (MAPC
         (FUNCTION
          (LAMBDA (I)
           (COND [(AND [CONSP I]
                       [EQ (CAR I) 'COND]
                       [NEQ (CAR (CAR (LAST I))) T])
                  (NCONC I (LIST (LIST T)))])))
         BODY)
        {;; Also make sure that the last statement is a
            RETURN.}
        (COND [(NEQ (CAR (CAR (LAST SS))) 'RETURN)
               (NCONC SS (LIST (LIST 'RETURN NIL)))])
        {;; Build an association list between labels and
```

```
    statements, and delete the labels.}
(PROG NIL
 LOOP (COND
       [(CDR BODY)
        (COND
         [(ATOM (CADR BODY))
          (SETQ LABELS (CONS (CADR BODY) LABELS))
          (RPLACD BODY (CDDR BODY))]
         [T (SETQ GOLIST
              (APPEND
               (MAPCAR (FUNCTION
                         (LAMBDA (J)
                          (CONS J (CDR BODY))))
                       LABELS)
               GOLIST))
            (SETQ LABELS NIL)
            (SETQ BODY (CDR BODY))])
        (GO LOOP)]))
{;; Splice COND-clauses into their successors.}
(MAP
 (FUNCTION
  (LAMBDA (I)
   (COND
    [(EQ (CAR (CAR I)) 'COND)
     (MAPC
      (FUNCTION
       (LAMBDA (J)
        (COND
         [(NOT (MEMQ (CAR (CAR (LAST J)))
                     '(GO RETURN)))
          (PROGN (NCONC J (LIST (LIST 'GO (CDR I))))
                 (SETQ GOLIST
                       (CONS (CONS (CDR I) (CDR I))
                             GOLIST)))])))
      (CDAR I))]))))
 SS)
{;; Build an association list between each statement
    and its successor.}
(SETQ P2 (MAPLIST (FUNCTION (LAMBDA (I) I)) SS))
(MAPC
 (FUNCTION
  (LAMBDA (P)
   (PROG (Q S FN)
       1 (COND [(NULL (SETQ Q (CDR P)))
                (RETURN NIL)])
         {;; Q is the 'next' statement.}
         (SETQ S (CAR Q))
         {;; S is 'this' statement.}
         (SETQ FN (CAR S))
         {;; FN is the function for S.  (No more
             labels)}
         {;; Splice in the new address.}
         (COND [(EQ FN 'GO)
                (RPLACD P
                        (CDR (ASSOC (CADR S) GOLIST)
```

```
                                    ))
                          (GO 1)])
                    {;; COND will be the last expression in a
                        sequence.  Change its name to GCOND.
                        Add its clauses to the top level.}
                    (COND
                     [(EQ FN 'COND)
                      (RPLACD Q NIL)
                      (RPLACA S 'GCOND)
                      (MAPC
                       (FUNCTION
                        (LAMBDA (I)
                         (NCONC P2
                                (MAPLIST
                                 (FUNCTION (LAMBDA (J) J))
                                 I))))
                       (CDR S))])
                    {;; RETURN is also last in a sequence.
                        Keep only the parameter.}
                    (COND [(EQ FN 'RETURN)
                           (RPLACD Q NIL)
                           (RPLACA Q (CADR S))])))
             P2)
            {;; Construct the LAMBDA.}
            (RETURN (CONS (CONS 'LAMBDA
                                (CONS (CADR X) (CDR SS)))
                          (MAPCAR (FUNCTION (LAMBDA (I) NIL))
                                  (CADR X))))))
EXPR)
```

APPENDIX D

References

1] John McCarthy.
Recursive functions of symbolic expressions and their computation be machine, part I.
Communications of the ACM 3:184-194, 1960.

2] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, Michael I. Levin.
LISP 1.5 Programmer's Manual.
MIT Press, Cambridge, Massachusetts, 1962.

3] James R. Meehan.
The New UCI LISP Manual.
Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1979.

4] James R. Meehan.
The UCI MLISP Reference Manual.
Department of Information and Computer Science, University of California, Irvine, CA 92717. 1979.

5] David Canfield Smith.
MLISP.
Stanford AI Memo 135, October, 1970.