PROGRAM ABSTRACT COVER SHEET

(1) User Group: FOCUS ☒ VIM ☐ (INCOSL ☐)

Please complete this form according to the instructions on the reverse side

(2) Contributing Organization
Kjeller Institute
Installation Name
Kjeller Norway
City and State

(3) Author Identification
J. Kent
Programmer/Submitter (up to 19 characters)
Revisor

(4) Catalog Identification
L 2 | K C I N | L I S P | Rev.
Cl. Code | Org. Code | Program Name

(5) Operating System and Version: S C O P E

(6) Languages and Dialects (up to 21 characters)
Lisp

(7) Configuration
3 6 0 0
Computer | Other Information

(8) Descriptive Title (up to 56 Characters Including Blanks)
LISP Interpreter

(9) Program Materials Submitted
Write-Up Page Ct: 4 1
Source Record Count:
Source Medium: MT ☒ CD ☐ PT ☐
If MT or PT: No. | Tr/L | Length: 1 | 7 | 2 4 0 0
Other (up to 44 characters)

(10) Date Written
M a r | 1 9 6 6
Original | Revised

(11) Restricted: No ☒ Yes ☐ (Requires ordering information)
Reason: Classified ☐ Geographic ☐ Other ☐

(12) Required Library Routines
Cl. Code | Org. Code | Program Name | Rev.

(13) Entry Point Names

(14) Original/Revised Program Abstract

LISP3600 reads LISP in S-expression format.

(15) Nature of Revision ☐ Proprietary Ordering Information ☐ Additional Information ☐

TEKNISK NOTAT

L2 KCIN LISP 3600

FORSVARETS FORSKNINGSINSTITUTT
Norwegian Defence Research Establishment
Postboks 25 - Kjeller
Norge

FFIE
Teknisk notat E-98
Reference: Job 147/13
Date: March 1966

LISP 3600: USERS MANUAL

by

Jan Kent

Kjeller, 3 March 1966

FORSVARETS FORSKNINGSINSTITUTT
Norwegian Defence Research Establishment
P C Box 25 - Kjeller
Norway

CONTENTS

LISP 3600: USERS MANUAL

1 INTRODUCTION TO THE PROGRAMMING LANGUAGE LISP

LISP is a programming language for manipulating complex data structures. The data structures are built up as S-expressions. LISP programs are also written as S-expressions, because the LISP-interpreter can only read S-expressions.

1.1 S-expressions

1.1.1 Atoms

The most elementary type of S-expression is the atom. An atom is either a string of no more than 82 letters and digits, the first one of which must be a letter, or a number.

Examples: AB, A1, 36

1.1.2 Dot-notation

More complex S-expressions can be built up from atoms and the delimeters ")", "(" and ".". The basic operation for forming S-expressions is to combine two of them to produce a larger one, called a dotted pair. From the two S-expressions AB and A1 one can form the dotted pair (AB. A1). This can f i be dotted with itself to give ((AB. A1). (AB. A1)). S-expressions formed in this way are said to be written in dot-notation.

Examples: ((AB. 36). A1) , ((V. V). (X. (Y. 6)))

1.1.3 List-notation

Large S-expressions are difficult to read when they are written in dot-notation. However, S-expressions can in some cases be written

in an abbreviated form called list-notation. If m_1, m_2, .., m_n are S-expressions the list $(m_1\ m_2 \ldots m_n)$ is identical to $(m_1 \cdot (m_2 \cdot (\ldots \cdot (m_n \cdot NIL) \ldots)))$. The atom NIL serves as terminator for lists. The empty list ( ) is identical to NIL. Lists may have sublists. The dot-notation and the list-notation may be used in the same S-expression. Blank is the usual delimeter in list-notation, but comma may also be used. Blank and comma are equivalent in LISP.

Examples: (A 36 C) = (A, 36, C) = (A · (36 · (C · NIL)))

(A (2 · C)) = (A, (2 · C)) = (A · ((2 · C) · NIL))

1.2 LISP-functions

We shall introduce some elementary LISP-functions. Every example given from now on will, if prefixed with the atom EVAL, constitute complete LISP-programs which may be punched and run. The effect of the prefix EVAL is to call the interpreter.

1.2.1 QUOTE (one argument)

To tell the interpreter what parts of an S-expression are function calls and what parts are arguments for the functions, we "quote" the arguments, with the function QUOTE. In other words QUOTE is used to signify that an expression stands for itself rather than for something to be interpreted further; thus it serves to isolate a program from its data.

1.2.2 CONS (two arguments)

CONS combines the two arguments to make a dotted pair.

Example: (CONS (QUOTE A)(QUOTE(B)))

Upon reading this (with the prefix EVAL) the interpreter calls CONS and calculates and prints out the value:

(A B)

(A B) is equivalent to (A · (B)) but the interpreter will print in list-notation whenever possible.

We see that in LISP the call on a function fn with the arguments x_1, x_2, ..., x_k are written as

$$(\text{fn } x_1 \; x_2 \; \ldots \; x_k)$$

In a list that is not quoted the first element is taken to be a function name. The function is called by the interpreter and applied to the arguments. Functions may as in the example above be nested; the innermost function is computed first.

1.2.3 CAR (one argument)

The argument of CAR should not be an atom. Its value is the first part of its composite argument.

Examples: (CAR(QUOTE (A B))) value A

(CAR(QUOTE ((A · 12) · C) value (A · 12)

1.2.4 CDR (one argument)

The argument of CDR should not be an atom. Its value is the second part of its composite argument. The second part of a list is the rest of the list except the first element.

Examples: (CDR(QUOTE (A B))) value B

(CDR(QUOTE ((A · 12) · C) value C

(CDR(QUOTE (A))) value NIL

That NIL must be the value in this example can be seen from the fact that (A) = (A · NIL).

1.2.5 EQUAL (two arguments) predicate

A function whose value is either true or false is called a predicate. In LISP the values true and false are represented by the atoms T

and F, respectively. A LISP-predicate is therefore a function whose value is either T or F. The atoms T, F and NIL need not be quoted, because they have an inherent meaning for the LISP-interpreter. Numbers need not be quoted, because they are always taken to represent their numeric value.

Examples: (EQUAL 12 13) value NIL

(F is equivalent to NIL)

(EQUAL(QUOTE (A·B))(QUOTE (A·B))) value T

(EQUAL (CDR (QUOTE A)) NIL) value T

1.2.6 ADD1 (one argument)

The argument of ADD1 must be a number. This number is increased with 1.

Example: (ADD1 13) value 14

1.2.7 ATOM (one argument) predicate

The value of the predicate ATOM is T if its argument is an atom, and F otherwise.

Examples: (ATOM (QUOTE A B C D E)) value T

(ATOM (CAR (QUOTE (A·B)))) value T

(ATOM (QUOTE (A·B))) value F

1.2.8 COND (indefinite number of arguments)

More interesting functions may be constructed in LISP with the aid of the conditional expression, definable by the function COND. COND is a special form which takes an indefinite number of argurnents. Each argument is a list of two elements. COND proceeds from argument to argument, evaluating the first element of each; and the value of COND is the second element of the first argument whose first element is true.

Examples: (COND (F (QUOTE A)(T (QUOTE B))) value B

(COND ((ATOM (QUOTE C))(QUOTE FIRST))

((ATOM (CAR (QUOTE (A·B)))) NIL)) value FIRST

(COND ((ATOM (QUOTE (C)))(QUOTE FIRST)

((ATOM (CAR (QUOTE (A·B)))) NIL)) value NIL

1.2.9 DEFINE (one argument)

The LISP-programmer may create names for new functions and reference these names instead of writing the entire function each time it is needed. Up to this point we have defined new functions by nesting known ones. From now on we will allow recursive functions. This is functions which calls itself. It is necessary to create a name for a recursive function, otherwise it would be impossible for the function to reference itself. A function name is also handy when we want the same function computed several times with varying arguments. If we want to define a function without specifying the arguments, we must use variables.

1.2.10 LAMBDA (two arguments)

Variables are introduced by the special form LAMBDA. The first argument of LAMBDA is a list of the variables which will be used in the function which is the second argument of LAMBDA. When the function is computed LAMBDA establishes a correspondence between the supplied arguments and the variables.

The name of a function is created by the function DEFINE. After a function name have been created by DEFINE, the function may be referenced any-where with its name. The argument of DEFINE is a list of function definitions. The definitions are lists of two elements: the first element is the atomic function-name and the second element is the LAMBDA-expression which defines the function.

Suppose we want to define a function called CADR whose value shall be the second element of the list which is supplied as its argument.

The function must contain a variable which we will call X.

The following program will establish CADR as a function in the LISP-system.

```
(DEFINE(QUOTE ((CADR (LAMBDA (X) (CAR (CDR X)))))))
                                                    value (CADR)
```

The value of DEFINE tells us that the definition has been accepted, and CADR is consequently available. Suppose we now want the second element on the list (12 14 18 26). The following program would do this

```
(CADR (QUOTE (12 14 18 26)))  value 14
```

Let us look at a recursive definition: The function FF selects the first atom of any given expression. FF is defined as follows

```
(DEFINE (QUOTE ((FF (LAMBDA (X)
                    (COND ((ATOM X) X)
                          (T (FF (CAR X))))))))) value (FF)
```

The expression (FF (LAMBDA ... can be read: If X is an atom, then X itself is the answer. Otherwise the function FF is to be applied to CAR of X. The use of COND is very important since it assures us that the recursion will end. If X is atomic, then the first branch of the COND which is X will be selected. Otherwise, the second branch (FF (CAR X)) will be selected, since T is always true. Let us now use FF to find the first atom in the S-expression ((A·B)·C).

```
(FF (QUOTE ((A·B)·C)))  value A
```

As a last example we will define the predicate MEMBER of two arguments X and Y. MEMBER is true if the S-expression X occurs among the elements of the list Y. The predicate NULL, which is true if its argument is NIL, is used in MEMBER. Remembering that DEFINE could deal with many definitions at the same time we define both MEMBER and NULL with the following program.

```
(DEFINE (QUOTE(
        (NULL (LAMBDA (J)(EQUAL J NIL)))
        (MEMBER (LAMBDA (X Y)
                (COND ((NULL Y) F)
                      ((EQUAL X (CAR Y)) T)
                      (T(MEMBER X(CDR Y)))
                ))
        )))
(MEMBER (QUOTE (A·B))(QUOTE((E·F) C (A·B))) value T
(MEMBER (QUOTE D)(QUOTE (A B C E))) value NIL
```

There are many other functions in LISP 3600, see Appendix IV for details about some of them. See also the LISP-run in Appendix VI.

2 OPERATING PROCEDURES

The LISP-interpreter is located on a binary tape with tape-label BINARY LISP 3600, which must be requested for a LISP-run.

2.1 Running a program punched on cards

The LISP-program can be punched on cards, free-field, in columns 1 - 72.

The following control cards are necessary to run the LISP-program on the cards.

7/9 JOB, <acc. nr>, <program name>, <time limit>

7/9 EQUIP, 6=(BINARY LISP3600), SV

7/9 EQUIP, 10=60

7/9 EQUIP, 11=61

7/9 LOAD, 6

7/9 RUN, <time limit>, <print limit>

LISP-program

4/8

77
88

Logical unit 10 and 11 are input and output units, respectively; for the LISP-interpreter. These are here equated to the standard input output units 60 and 61. Logical units 10 and 11 may be equated to any other available equipment the programmer may need. The card containing 4/8 column 1 acts as end-of-file mark to the LISP-interpreter, which upon reading it, returns control to SCOPE.

2.2 Prelisting the LISP-program

At the Kjeller Computer Installation it is possible to get a listing of the LISP-program prior to running it. This can be very useful, because with large programs the interpreter print-out is very hard to read. The following control cards should be used.

```
7
9JOB, <acc nr>,<program name>,<time limit>

7
9EQUIP, 6=(BINARY LISP3600), SV

7
9EQUIP, 10=MT

7
9EQUIP, 11=61

7
9LIST, L,O=10
```

LISP-program

```
4
8

77
88

7
9REWIND, 10

7
9LOAD, 6

7
9RUN, <time limit>, <print limit>

77
88
```

2.3 Running a program punched on paper tape

At the Kjeller Computer Installation the LISP-program may also be punched on 8-channel papertape. The papertape's flexowriter codes are translated to Hollerith cardimages on a magnetic tape by the standard program PIMP. The following conventions must be observed when punching LISP-programs on papertape:

a) Use only lowercase letters.

b) Use comma as separator between atoms, never blank.

c) The LISP-program must be preceeded by the character; downstroke, A (↓), and superceded by the character, downstroke, or-sign (∨).

A papertape containing a LISP-program punched according to the above rules, can be run with the following control cards.

```
7
9JOB, <acc nr>,<program name>,<time limit>

7
9EQUIP, 6=(BINARY LISP3600), SV

7
9EQUIP, 1=PT

7
9EQUIP, 11=61

7
9PIMP, I=1, P=10, L

7
9REWIND, 10

7
9LOAD, 6

7
9RUN, <time limit>,<print limit>

77
88
```

These control cards will also give a prelisting of the program.

2.4 Stopping a LISP-program

If a LISP-program should go into an endless loop, it can be stopped in such a way as to give the programmer some information about what went wrong. This is done by pressing jump key 3 on the console. Jump key 3 transfers control to ERROR which prints out all lists bound on the push-downlist.

2.5 Tracing in LISP 3600

Bit 7 in the D-register, which is set by executing SETBIT(7), must be set if tracing is wanted. The bit may be cleared by executing CLEAR-BIT(7). Clearing bit 7 stops all tracing immediately. Tracing is further

controlled by the pseudo-function TRACE, whose argument is a list of functions to be traced. After trace has been executed, tracing will occur whenever these functions are entered. The tracer prints out the name of a function and its arguments when it is entered, and its name and value when it is finished. When tracing of certain functions is no longer desired, it can be terminated by the pseudo-function UNTRACE whose argument is a list of functions that are no longer to be traced.

3 ERROR DIAGNOSTICS

3.1 Syntactical errors

If the reader finds syntactical errors in an S-expression, it inserts special atoms at appropriate places in the S-expression. The special atoms have the following meaning:

atom	meaning
ERRA	,. or) encountered as first non-blank character in an S-expression
ERRB	. (dot) encountered as first non-blank character after a (
DOTERR1	The second S-expression in a dotted pair is not followed by a right parenthesis
DOTERR2	,. or) encountered as first non-blank character after a dot

An illegal character is changed to ? by the reader, and recognized as a syntactical error. A doublet containing one or more syntactical errors is never run.

3.2 Runtime errors

When an error occurs during a LISP-run the following general error-heading will be printed out:

```
**** ERROR        LISTING OF LISTS BOUND ON
                  PUSHDOWNLIST FOLLOWS
```

The appropriate errordiagnostic is inserted into the blank space between ERROR and LISTING before printing. As the heading indicates the lists bound on the pushdownlist are printed out after the heading. Printing of the lists on the pushdownlist will not occur if the error diagnostic was STACK EXCEEDED. The errordiagnostic **** A1 APPLIED FUNCTION CALLED ERROR will be given if a LISP-program calls ERROR. The argument (if any) of ERROR will be printed.

A complete list of error diagnostics is given below, with comments.

A2 APPLY SASSOC	This occurs when an atom given as the first argument of APPLY, does not have a definition either on its property list or on the association list of APPLY.
A3 EVCON	None of the propositions following COND are true.
A4 OR A5 PROG	SET or SETQ given on nonexistent program variable.
A6 GO IN PROG	GO refers to a nonexistent label.
A7 SPREAD	Too many arguments in an EXPR or FEXPR.
A8 EVAL SASSOC	The atom in question is not bound on the association list for EVAL nor does it have an APVAL.
A9 EVAL SASSOC	EVAL expects the first object on a list to be an atom. A8 and A9 frequently occurs when a parenthesis miscount causes the wrong phrase to be evaluated.
F2 PAIR1	The variable list specified by LAMBDA is shorter than the submitted argument-list.

F3 PAIR2	The variable list specified by LAMBDA is longer than the submitted argument list.
DUMP ON JK3	Jump key 3 on the console has been pressed, see 2.4.
STORE IS FULL	The garbage collector is unable to find unused words in free word storage.
STACK EXCEEDED	Recursion is very deep. Nonterminating recursion will cause this error. The list of lists bound on the pushdownlist will not be given on this error.
I2 IN EXPT	First argument is negative in EXPT.
BIG ARG2. EXPT	Abs value of second argument in EXPT is greater than 709.
I3 BAD ARIT ARG	An arithmetic routine has been given an unusable argument.

As indicated above the lists bound on the pushdownlist are printed out if a runtime error occurs. The most recently used list in the stack (the list on top) is printed last. The last printed lists will therefore give a good indication of what caused the error.

Let us assume that none of the functions being interpreted are using the PROG-feature, and that the TRACEIND in the D-register is off.

Under these conditions the lists bound on the pushdownlist will be alternately function calls or definitions and association lists. When reading the pushdownprintout keep in mind that the innermost function are evaluated first, even though the functions are interpreted from the outside in. Thus the call on the function being evaluated when the error occurred will be near the top of the stack.

If the TRACEIND in the D-register was set, the name of an EXPR called will be found on the pushdownlist between the EXPR's definition and the corresponding association list.

The call on a function using the PROG-feature will cause the following lists to appear in the pushdownprintout:

a) The complete function definition (omitting the name of the function).

b) The go-list (see LISP IMPLEMENTATION 3.11).

c) The association list.

d) A list of the uninterpreted statements in the functions starting with the one that was being evaluated when the error occurred.

4 DIFFERENCES BETWEEN LISP 3600 AND LISP 1.5

4.1 Extensions

Alphameric atoms may in LISP 3600 have up to 82 characters.

Fixed point numbers may have absolute values between 2^{47} and 2^{-47}.

Floating point significance on input is 10 digits.

Floating point numbers may have absolute values between 10^{307} and 10^{-307}.

Numbers are considered equal if the absolute value of their difference is less than 10^{-8}.

A completely new function called APPEND1 is included as a SUBR, see Appendix IV for details.

4.2 Omissions

The following functions are not implemented: ARRAY, ERRORSET, RECLAIM, COUNT, UNCOUNT and SPEAK. All other undefined functions in LISP1.5 can be defined from those given in LISP 3600.

4.3 Differences

a) The scale factor in a logical number is an exponent to the base 2.

b) A minussign preceding a logical number will cause the shifted number to be complemented.

c) Blanks are used as fill-in in the fullwords. This makes it impossible to print more than a single blank at a time. But this means that the constant $$$ B$ will print as a single space.

d) The function CLEARBUFF has not been implemented because it should never be needed.

e) The functions INTERN and MKNAM are combined into a single function, namely MKATOM.

MKATOM ≡ INTERN(MKNAM)

f) Because of the reorganisation of all property lists, the printname is CAR of the atom.

g) UNPACK takes an atom as its argument.

h) PRINT should not be used directly after PRINT1 without executing TERPRI between, because PRINT sets the output buffer to blanks before printing, thereby destroying what was put in by PRIN1.

i) GO must only be given atomic labels.

j) + and - should never be used as characters in an atom.

See Appendix VI for the initial object list defined in LISP3600.

5 EXAMPLES OF THE USE OF LISP

A very short mentioning of two interesting and useful LISP-programs will be given here.

5.1 METEOR: A LISP-interpreter for string transformations

Particulars about this program are given in (5) in an article with the above heading as title written by Daniel G Bobrow.

METEOR is a LISP-interpreter for a COMIT-like language. This language is very useful for string manipulation and transformation. METEOR have been debugged and run in LISP3600. The card deck defining METEOR can be obtained from the Kjeller Computer Installation.

5.2 PRETTYPRINT

This program will print the functions, whose names are on its argumentlist. The functions must be EXPRs or FEXPRs which have been defined previously with DEFINE or DEFLIST. The functions are printed in such a way as to make them very readable. In the LISP-run contained in Appendix VI PRETTYPRINT prints itself.

6 INSTALLATION PROCEDURES

The following notes should be read by those who have obtained a LISP3600 systemtape through CO-CP.

6.1 Contents of system tape

The tape distributed through CO-CP has the following contents.

Label: BCD LISP3600

LISP-interpreter

in COMPASS-BCD.

METEOR-BCD

with testcase.

end-of-file

METEOR-BCD

with testcase.

end-of-file

A copy of this tape is included in the program-library at the Kjeller Computer Installation.

6.2 Running the systemtape

The tape was made using SCOPE 6.0 and should therefore never be run on older versions of SCOPE (because of the label). The following run will make a COSY-tape named LISP3600 and a binary tape named BINARY LISP3600.

The METEOR-interpreter lying immediately after the LISP-interpreter will in this run be defined and run on a testcase to test the systemtape.

Control cards necessary:

```
7
9JOB, <acc nr>,<program name> , 25
7
9EQUIP, 3=(LISP3600), SV
7
9EQUIP, 6=(BINARY LISP3600), SV
7
9EQUIP, 2=(BCD LISP3600), SV
7
9EQUIP, 11=61
7
9EQUIP, 10=2
7
9COMPASS, L, R, X=6, C=3, I=2
11
0
7 BANK, (0), ALL
9
```

7
9LOAD, 6

7
9RUN, 5, 5000

77
88

The COSY-tape should be put aside, because any corrections or additions will be in the form of COSY-correction cards.

The binary tape may be used for LISP-runs as described in 2. 1. However, if the computer in question has two banks or more the card

11
0
7 BANK, (0), ALL.
9

must be placed immediately before the LOAD card.

If the METECR-interpreter is wanted, skip the first file and punch it out.

APPENDIX IV

SOME FUNCTIONS IN THE INTERPRETER DEFINED IN M-EXPRESSIONS

Defined functions:

EVALQUOTE
APPLY
EVAL
EVLIS
EVCON
LIST
APPEND1
APPEND
DEFINE
DEFLIST

The defnitions are chosen so as to resemble the actual implementation of these functions as closely as possible.

THE LISP INTERPRETER

```
evalquote [fn; args] = [get[ fn; FEXPR] V get [fn; FSUBR]→
                eval [cons[ fn; args]; NIL]
            T→apply [fn; args; NIL]]

apply[fn; args; a]=[
      null[ fn]→NIL;
      atom [fn]→ [get[ fn; EXPR]→apply [expr;[1] args; a];
                        get[ fn; SUBR]→ { spread [args];
                                          $ ALIST: = a;
                                          TSX subr[1], 4 } ;
                  T → apply [cdr[ sassoc[ fn; a;λ[[ ]; error [A2]]]]; args; a];
      eq [car [fn]; LABEL] → apply [caddr [fn]; args; cons[ cons[ cadr[ fn]; caddr [fn]]; a]];
      eq [car [fn]; FUNARG] → apply [cadr [fn]; args; caddr [fn]];
      eq [car [fn]; LAMBDA] → eval [caddr [fn]; nconc[ pair [cadr [fn]; args]; a]];
      T → apply [eval[ fn; a]; args; a]]

eval [form; a] =[
      null [form] → NIL;
      numberp [form] → form;
      atom [form] → [get [form; APVAL] → car [apval[1]];
                  T → cdr [sassoc [form; a;λ[[ ]; error [A8]]]]];
      eq [car [form]; QUOTE] → cadr [form];
      eq[ car [form]; FUNCTION] → list [FUNARG; cadr [form]; a];[2]
      eq [car[ form]; COND] → evcon [cdr[ form]; a];
      eq [car[ form]; PROG] → prog [cdr [form]; a];[2]
      atom [car[ form]] → [get [car [form]; EXPR] → apply [expr;[1] evlis [cdr [form]; a]; a];
                        get [car [form]; FEXPR] → apply [fexpr;[1] list [cdr[ form]; a]; a];
                        get [car [form]; SUBR] → { spread [evlis [cdr [form]; a]];
                                                   $ ALIST: = a;
                                                   TSX subr,[1] 4 } ;
                        get [car [form]; FSUBR] → { AC: = cdr [form];
                                                    MQ: = $ALIST: = a;
                                                    TSX fsubr,[1] 4 } ;
                        T → eval [cons [cdr [sassoc [car [form]; a; λ[[]; error [A9]]]];
                                                cdr [form]]; a]];
                        T → apply [car [form]; evlis [cdr [form]; a]; a]]
```

[1] The value of get is set aside. This is the meaning of the apparent free or undefined variable.

[2] In the actual system this is handled by an FSUBR rather than as the separate special case shown here.

```
evlis[m; a] = prog [[u; v; w];
              w:= NIL;
    LIST1     u:= car [m];
              v:= eval [m; a];
              w:= append1 [w; v];
              m:= cdr [m];
              [null [m] → return [w]];
              go [LIST1]]

evcon [c; a] = prog[[v];
    EVCON [null [c] → [testbit [4] → return [c]; T → error [A3]]];
              v:= caar [c];
              v:= eval [v; a];
              [null [v] → go [EVCON3]];
              v:= cadar [c];
              v:= eval [v; a];
              return [v];
    EVCON3    c:= cdr [c];
              go [EVCON]]
```

$list[x_1; x_2; \ldots; x_n] = list[l; a] = evlis[l; a]$

(Remember that list is an FSUBR).

```
append1 [x;y] =  prog [[m;n];
                 y: = cons [y;NIL];
                 [null [x] → return [y]];
                 m:= x;
   APPEND2       n: = cdr [m];
                 [null [n] → return [prog 2 [rplacd [m;y];x]]];
                 m:= n;
                 go [APPEND2]]

append [x;y]  =  prog [[u;v];
                 [null [x] → return [y]];
                 u: = NIL;
   APP1          v: = car [x];
                 u: = append1 [u;v];
                 v: = cdr [x];
                 [null [v] → go [APP1]];
                 v: = u;
   APP2          v: = cdr [v];
                 [null [v] → return [prog 2 [rplacd [v;y];u]]];
                 go [APP2]]
```

```
define [x] =  deflist [x; EXPR]

deflist [x; i] =  prog [[u; w; z];
                        u: = NIL;
   DEF2                 z: = car [x]
                        w: = car [z];
                        remprop [w; i];
                        z: = cadr [z];
                        z: = cons [z; NIL];
                        z: = cons [i; z];
                        z: = attrib [w; z];
                        u: = append1 [u; w];
                        x: = cdr [x];
                        [null [x] → return [u]];
                        go [DEF2]]
```

APPENDIX V

RULES FOR TRANSLATING FUNCTIONS WRITTEN IN M-EXPRESSIONS INTO S-EXPRESSIONS

1 If the function is represented by its name, it is translated by changing all of the letters to upper case, making it an atom. Thus car is translated into CAR.

2 If the function uses lambda notation, then the expression λ [[x_1; x_2; ... ; x_n]; ϵ] is translated into (LAMBDA (X1 X2 ... XN) Σ^{**}] where Σ^{**} is the translation of ϵ.

3 If the function begins with a label, then the translation of label [α; ϵ] is (LABEL α^{**} Σ^{**}).

4 A variable, like a function name are translated using uppercase letters. Thus the translation of var1 is VAR1.

5 The obvious translation of letting a constant translate into itself will not work. Since the translation of x is X, the translation of X must be something else to avoid ambiquity. The solution is to quote it. Thus X is translated into (QUOTE X).

6 The form fn[arg_1; arg_2; ... ; arg_n] is translated into (fn** arg_1^{**} arg_2^{**} arg_n^{**}).

7 The conditional expression [$p_1 \rightarrow e_1$; ...; $p_n \rightarrow e_n$] is translated into (COND (p_1^{**} e_1^{**}) ... (p_n^{**} e_n^{**})).

8 Labels (in prog) translate into themselves. Thus they are left unchanged. go [A] translates into (GO A).

9 The assignment symbol := is translated into SETQ. u:=cdr [u]; is translated into (SETQ U (CDR U)).

10 Numbers and the atoms T, F and NIL need not be quoted.

Examples:

M-expression	S-expression
x	X
car	CAR
car [x]	(CAR X)

```
T                                              T
ff [car [x]]                                   (FF (CAR X))
[atom [x] → x; T → ff [car [x]]]               (COND ((ATOM X) X)
                                                     (T (FF(CAR X))))

label [ff; λ [[x];
      [atom [x] → x; T → ff [car [x]]]]]
                                   (LABEL FF (LAMBDA (X)
                                         (COND ((ATOM X) X)
                                               (T (FF (CAR X)))))))
```

APPENDIX VI

A SAMPLE LISP-RUN SHOWING THE COMPLETE INITIAL OBJECT LIST AND PRETTYPRINT PRINTING ITSELF

```
EQUENCE,81
B,3750K-T,JGK,15:
OPE 6,11 KCIN
QUIP,10=60
QUIP,11=61
QUIP,6=(BINARY LISP3600),SV
OAD,6
UN,3,1000
```

PROGRAM NAMES
71507 MAIN 05321:

PROGRAM EXTENS.
ONE

LABELED COMMON
71416 A 00071: 0:71360: BL 00036 0:70064: B 0127

NUMBERED COMMON
12400 1: 07641:

ENTRY POINTS:
00074 SENTRY 0.71510: MAIN

```
ARGUMENTS FOR EVALQUOTE ...
EVAL
(OBLIST NIL)
TIME SPENT IN EVALQUOTE          00000002 MS.  VALUE IS ...
(NIL ERRA ERRB DOTERR1 DOTERR2 APVAL T F EXPR SUBR COND LAMBDA FEXPR FUNARG FSUBR
LABEL *** / **** ;EOR; ;EOF; *T* LPAR ( BLANK RPAR ) PERIOD  CAR CDR CONS ATOM
GET EQ EVAL CDAR PAIR READ NULL NOT FIXP LOGP ADD1 SUB1 CADR CDDR CAAR SET PACK EXPT
APPLY EQUAL EVLIS NCONC NUMBERP PRINT RPLACA RPLACD DEFINE ATTRIB REMPROP ERROR QUOTIENT
FLOATP CADDR CADAR RETURN MINUS LESSP GREATERP DEFLIST APPEND APPEND1 MEMBER ADVANCE
UNPACK MKATOM PRIN1 TERPRI TRACE UNTRACE ZEROP MINUSP TESTBIT SETBIT CLEARBIT GENSYM
OR AND LIST PLUS PROG GO SETQ TIMES LOGAND LOGOR LOGXOR FUNCTION QUOTE OBLIST DIFFERENCE
REMAINDER LEFTSHIFT STARTREAD SASSOC)
ARGUMENTS FOR EVALQUOTE ...
DEFINE
(((LENGTH (LAMBDA (L) (PROG (U V) (SETQ V 0) (SETQ U L) A (COND ((NULL U) (RETURN
V))) (SETQ U (CDR U)) (SETQ V (ADD1 V)) (GO A)))) (MAP (LAMBDA (X F) (PROG (M) (SETQ
M X) LOOP (COND ((NULL M) (RETURN NIL))) (F M) (SETQ M (CDR M)) (GO LOOP)))) (PRETTYPRINT
```

```
(LAMBDA (L) (MAP L (FUNCTION (LAMBDA (J) (PROG (T1) (TERPRI) (PRIN1 LPAR) (PRIN1
(CAR J)) (TERPRI) (PRINTDEF (COND ((SETQ T1 (GET (CAR J) (QUOTE EXPR))) T1) ((SETQ
T1 (GET (CAR J) (QUOTE FEXPR))) T1) (T (QUOTE UNDEFINED)))) (PRIN1 RPAR) (TERPRI)))))))))
TIME SPENT IN EVALQUOTE          00000005 MS,  VALUE IS ...
(LENGTH MAP PRETTYPRINT)
ARGUMENTS FOR EVALQUOTE ...
DEFINE
(((PRINTDEF (LAMBDA (E) (PROG (I IUNITL) (SETQ I 1) (SETQ IUNITL 3) (PRIN1 BLANK)
(PRIN1 BLANK) (PRIN1 BLANK) (SUPERPRINT E) (RETURN NIL)))) (SUPERPRINT (LAMBDA (E)
(COND ((ATOM E) (PRIN1 E)) (T (PROG (EP M) (SETQ EP E) (PRIN1 LPAR) A (COND ((MEMBER
(CAR EP) (QUOTE (AND OR LIST PLUS TIMES COND IF SELECT MAX MIN PROG2))) (GO PL)) ((EQ
(CAAR EP) (QUOTE LAMBDA)) (GO PL)) ((EQ (CAR EP) (QUOTE PROG)) (GO PP))) (SUPERPRINT
(CAR EP)) (SETQ EP (CDR EP)) (COND ((NULL EP) (RETURN (PRIN1 RPAR))) ((ATOM EP) (GO
PD))) (PRIN1 BLANK) (GO A) PK (SETQ I (SUB1 I)) PD (PRIN1 BLANK) (PRIN1 PERIOD) (PRIN1
```

```
BLANK) (PRIN1 EP) (RETURN (PRIN1 RPAR)) PL (SETQ I (ADD1 I)) (SUPERPRINT (CAR EP))
PM (SETQ EP (CDR EP)) (COND ((NULL EP) (GO PJ)) ((ATOM EP) (GO PK))) (ENDLINE) (SUPERPRINT
(CAR EP)) (GO PM) PJ (SETQ I (SUB1 I)) (RETURN (PRIN1 RPAR)) PP (PRIN1 (CAR EP)) (SETQ
EP (CDR EP)) (SETQ I (ADD1 I)) (COND ((NULL EP) (GO PJ)) ((ATOM EP) (GO PK))) (PRIN1
BLANK) (SUPERPRINT (CAR EP)) PY (SETQ EP (CDR EP)) (COND ((NULL EP) (GO PJ)) ((ATOM
EP) (GO PK))) (ENDLINE) (COND ((ATOM (CAR EP)) (GO PZ))) (PRIN1 BLANK) (PRIN1 BLANK)
(PRIN1 BLANK) (PRIN1 BLANK) (PRIN1 BLANK) (PRIN1 BLANK) PX (SETQ I (PLUS I 2)) (SUPERPRINT
(CAR EP)) (SETQ I (PLUS I -2)) (GO PY) PZ (PRIN1 (CAR EP)) (SETQ M (PLUS IUNITL (MINUS
(LENGTH (UNPACK (CAR EP)))))) AA (SETQ M (SUB1 M)) (PRIN1 BLANK) (COND ((NOT (OR
(ZEROP M) (MINUSP M))) (GO AA))) (SETQ EP (CDR EP)) (COND ((NULL EP) (GO PJ)) ((ATOM
EP) (GO PK)) ((ATOM (CAR EP)) (GO PZ))) (GO PX))))))) (ENDLINE (LAMBDA NIL (PROG (J)
(SETQ J I) (TERPRI) A (COND ((ZEROP J) (RETURN NIL)) ((MINUSP J) (ERROR I))) (PRIN1
BLANK) (PRIN1 BLANK) (PRIN1 BLANK) (SETQ J (SUB1 J)) (GO A))))))
TIME SPENT IN EVALQUOTE          00000004 MS.  VALUE IS ...
(PRINTDEF SUPERPRINT ENDLINE)
ARGUMENTS FOR EVALQUOTE ...
```

```
PRETTYPRINT
((PRETTYPRINT SUPERPRINT PRINTDEF ENDLINE))

(PRETTYPRINT
   (LAMBDA (L) (MAP L (FUNCTION (LAMBDA (J) (PROG (T1)
                (TERPRI)
                (PRIN1 LPAR)
                (PRIN1 (CAR J))
                (TERPRI)
                (PRINTDEF (COND
                   ((SETQ T1 (GET (CAR J) (QUOTE EXPR))) T1)
                   ((SETQ T1 (GET (CAR J) (QUOTE FEXPR))) T1)
                   (T (QUOTE UNDEFINED))))
                (PRIN1 RPAR)
                (TERPRI)))))))
```

```
(SUPERPRINT
   (LAMBDA (E) (COND
      ((ATOM E) (PRIN1 E))
      (T (PROG (EP M)
               (SETQ EP E)
               (PRIN1 LPAR)
            A  (COND
                  ((MEMBER (CAR EP) (QUOTE (AND
                     OR
                     LIST
                     PLUS
                     TIMES
                     COND
                     IF
                     SELECT
                     MAX
                     MIN
                     PROG2))) (GO PL))
                  ((EQ (CAAR EP) (QUOTE LAMBDA)) (GO PL))
                  ((EQ (CAR EP) (QUOTE PROG)) (GO PP)))
               (SUPERPRINT (CAR EP))
               (SETQ EP (CDR EP))
               (COND
                  ((NULL EP) (RETURN (PRIN1 RPAR)))
                  ((ATOM EP) (GO PD)))
               (PRIN1 BLANK)
               (GO A)
            PP (SETQ I (SUB1 I))
            PD (PRIN1 BLANK)
               (PRIN1 PERIOD)
```

```
        (PRIN1 BLANK)
        (PRIN1 EP)
        (RETURN (PRIN1 RPAR))
PL  (SETQ I (ADD1 I))
        (SUPERPRINT (CAR EP))
PM  (SETQ EP (CDR EP))
        (COND
           ((NULL EP) (GO PJ))
           ((ATOM EP) (GO PK)))
        (ENDLINE)
        (SUPERPRINT (CAR EP))
        (GO PM)
PJ  (SETQ I (SUB1 I))
        (RETURN (PRIN1 RPAR))
PP  (PRIN1 (CAR EP))
        (SETQ EP (CDR EP))
        (SETQ I (ADD1 I))
        (COND
           ((NULL EP) (GO PJ))
           ((ATOM EP) (GO PK)))
        (PRIN1 BLANK)
        (SUPERPRINT (CAR EP))
PY  (SETQ EP (CDR EP))
        (COND
           ((NULL EP) (GO PJ))
           ((ATOM EP) (GO PK)))
        (ENDLINE)
        (COND
           ((ATOM (CAR EP)) (GO PZ)))
        (PRIN1 BLANK)
        (PRIN1 BLANK)
```

```
            (PRIN1 BLANK)
            (PRIN1 BLANK)
            (PRIN1 BLANK)
            (PRIN1 BLANK)
         PX (SETQ I (PLUS
                    I
                    2))
            (SUPERPRINT (CAR EP))
            (SETQ I (PLUS
                      I
                      -2))
            (GO PY)
         PZ (PRIN1 (CAR EP))
            (SETQ M (PLUS
                      IUNITL
                      (MINUS (LENGTH (UNPACK (CAR EP))))))
         AA (SETQ M (SUB1 M))
            (PRIN1 BLANK)
            (COND
              ((NOT (OR
                      (ZEROP M)
                      (MINUSP M))) (GO AA)))
            (SETQ EP (CDR EP))
            (COND
              ((NULL EP) (GO PJ))
              ((ATOM EP) (GO PK))
              ((ATOM (CAR EP)) (GO PZ)))
            (GO PX))))))


(PRINTDEF
  (LAMBDA (E) (PROG (I IUNITL)
```

```
                (SETQ I 1)
                (SETQ IUNITL 3)
                (PRIN1 BLANK)
                (PRIN1 BLANK)
                (PRIN1 BLANK)
                (SUPERPRINT E)
                (RETURN NIL))))

        (ENDLINE
           (LAMBDA NIL (PROG (J)
                (SETQ J I)
                (TERPRI)
              (COND
                   ((ZEROP J) (RETURN NIL))
                   ((MINUSP J) (ERROR I)))
                (PRIN1 BLANK)
                (PRIN1 BLANK)
                (PRIN1 BLANK)
                (SETQ J (SUB1 J))
                (GO A))))
*** TIME SPENT IN EVALQUOTE          00050468 MS.  VALUE IS ...
    NIL
```