

LISP F1: a FORTRAN  
implementation of LISP 1.5  
by  
Mats Nordström, Erik Sandewall and Diz Bresler

UPPSALA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCES



LISP F1: a FORTRAN  
implementation of LISP 1.5

by

Mats Nordström, Erik Sandewall and Diz Breslaw

The research reported here was supported in part by the Swedish  
Research Institute of national Defence (FOA P) under contract 101-218:1

Summary

Program name: LISP F1

Purpose: Read LISP S-expressions, evaluate them by interpretation, and print the values as S-expressions, all in interactive mode.

Programming language: IBM Basic FORTRAN IV (for a RAX-type TS system)

Programmed by: Diz Breslaw, Mats Nordström, and Erik Sandewall

Completed:

Memory requirements: Program and minor data areas: 36 K bytes. Memory for push-down stack, atom printnames (8 bytes/atom), and free memory (8 bytes/LISP cell) must be added.

Table of contents

1. Introduction
2. Conversation with LISP F1
  - 2.1 The read-eval-print loop
  - 2.2 Input format
  - 2.3 Output format
  - 2.4 Alternative top levels
3. Error messages
  - 3.1 Read errors
  - 3.2 System errors
  - 3.3 LISP errors A2 - A9
4. Tools for debugging LISP programs
  - 4.1 The function break
  - 4.2 The functions a2 - a9
  - 4.3 The function peek
  - 4.4 The LISP editor
  - 4.5 The LISP tracing routine
5. Differences from LISP 1.5
6. Representation of atoms and list structures
7. Recursive programming in FORTRAN
  - 7.1 The stacks
  - 7.2 Recursive calls and returns
8. The main program in the interpreter
  - 8.1 The top level loop
  - 8.2 Transmission of arguments for recursive functions
  - 8.3 Representation of SUBR's and FSUBR's

9. Functions and subroutines in the interpreter
10. Implementation guide
  - 10.1 Memory requirements
  - 10.2 The COMMON area
  - 10.3 Constants set in INIT
  - 10.4 Logical units for I/O
  - 10.5 Non-standard FORTRAN routines
  - 10.6 Use of LISP F1 in interactive mode
  - 10.7 Use of LISP F1 in batch mode
11. How to add new SUBR's and FSUBR's
12. References

Appendices:

- A. M-expression definition of the LISP F1 interpreter
- B. Tables of atoms and functions defined in LISP F1
- C. Flowcharts for IREAD, and GARB
- D. Listing of data deck containing printnames of system atoms
- E. S-expression listing of the edit and trace package

## 1. INTRODUCTION

The LISP interpreter (LISP F1) described here was written in Basic FORTRAN IV for IBM machines by Mats Nordström and Erik Sandewall. An editor and tracing package was then written in LISP by Diz Breslaw and added to the system. The work was performed at the University of Uppsala (Department of Computer Sciences). LISP F1 is designed for interactive use, but can also be run in batch mode. It accepts a dialect of LISP which is essentially LISP 1.5, but contains some of the "small" amendments to LISP that have been proposed and implemented at BBN, i.e. no-spread lambda expressions, break functions, user control of errors in interpretation, etc. (Cf. reference 4)

Sections 2 - 5 of this report are intended as a user manual; sections 6 - 9 as a program description; and sections 10 - 11 as an implementation guide ("how to get the system running, and how to change it"). All parts of the report presume a knowledge of LISP 1.5. Sections 6 - 11 also presume knowledge of FORTRAN.

## 2. CONVERSATION WITH LISP F1

### 2.1 The read-eval-print loop

Welcome to the F1 system. In order to use it, go over to your (teletype) computer terminal. Load LISP F1 by following the instructions that your systems group wrote after they had set up the system on your computer. When F1 has initialized, it asks you for input. (Here in Uppsala, this is indicated by a question mark). Type an S-expression that is suitable for evaluation. (Next section specifies details about the input format). F1 reads this expression, evaluates it, and prints the result. Type another expression. It is also evaluated, and the value is printed back. Here is an example:

```
(DE REVERSE(S U)(COND((NULL S)U)(T(REVERSE(CDR S))(CONS
?(CAR S) U)
REVERSE
?(REVERSE(QUOTE(A(B C)D(E F)G)
(G (E F )D (B C )A )
?(CDR(QUOTE REVERSE))
(EXPR (LAMBDA (S U )
(COND ((NULL S )U )
(T (REVERSE (CDR S )
(CONS (CAR S )U ))))))))
```

Thus the behavioral pattern of the F1 system is a read-eval-print loop. The purpose of sections 2 - 5 in this report is to describe how you as a user can utilize and modify this behavior.

### 2.2 Input format

S-expressions are typed-in in free format in col. 1-72 on the standard input unit (i.e. the value of the variable LUNIM, which is 9 initially).

The following rules apply to S-expressions:

A separator is one of the characters "space" , . ([])  
 (In some of the examples, [] are replaced by <>)

A numerical atom: is a sequence of digits between two separators.  
 Before the first digit + or - may occur. Floating point does not exist.

An alphanumeric atom is any character string with at least one non-digit,  
 all between two separators. Only the first 8 characters in the string  
 will be read and used by the LISP-system. The character % has a special  
 meaning for the function break, and should never be used in any other  
 situation. The "\$\$" facility of some LISP systems does not exist here.

A dotted list has the form (sexpr sexpr ---sexpr . atom). Note that a  
 dotted list must be terminated by ) . The following "S-expr" is meaning-  
 less and will cause a read error: (A B.C D)

S-expressions are then formed in the normal way.

Example:

ABCDEFGHI	the alphanumeric atom ABCDEFGH
123	numerical atom 123
(123.4)	a dotted pair (123 . 4)
123A+	an alphanumeric atom

To facilitate the matching of parentheses, there is a bracket feature  
 using "[" and "]". " " closes the last "[". If "]" occurs, and no  
 corresponding "[" exists, "]" closes the whole expression. "[" also  
 stands for one left parentheses.

Ex. The following S-expressions given as an input are identical:

```
(L1 (((L4 L4 L4))) L1 (L2 (L3 (L4))))
(L1 (((L4 L4 L4))) L1 (L2 (L3 (L4)
[L1 [(((L4 L4 L4) L1 (L2 (L3 (L4)
(L1 [(((L4 L4 L4) L1 (L2 (L3 (L4)
```



### 2.3 Output format

The value of an S-expression is printed in a simple form of pretty-print on the standard output unit (i.e. the value of the variable LUNUT, which is 6 initially). The rules for the pretty-print programmed in the system are given on page 40.

An example of output:

```
?(FAK (LAMBDA (N)(COND((ZEROP N) 1)(T(TIMES N(FAK(SUB1 N)
```

will be printed as

```
(FAK (LAMBDA (N)
      (COND ((ZEROP N) 1)
            (T (TIMES N (FAK (SUB1 N)))))))
```

System printout (value from eval, but not printouts from print and terpri) is switched off by evaluating

```
(SILENCE)
```

and switched on again by doing

```
(TALK)
```

Both functions have NIL as value.

Error-messages are always printed on logical unit 6, and cannot be switched off (except sometimes by redefining the functions A2 - A9, see section 4.2).

### 2.4 Alternative top levels.

In standard usage, the F1 system does read, eval, and print, like e.g. MIT PDP-10 LISP. Some other systems (e.g. 7090 LISP, 3600 LISP) prefer to do read, read, evalquote, print. The F1 system can be changed to this evalquote mode by evaluating

```
(MODE 2)
```

After evaluation of this expression, you are in evalquote mode.

To get back to eval mode, say

```
MODE (1)
```

This is the same function mode, but now you are talking to evalquote.

In some situations, it is desirable to redefine the top-level loop in another fashion. This happens e.g. if you wish to read natural language sentences one at a time, and apply some language processing operations to it. This can be accomplished by first defining the function SYS, and then doing (MODE 3). (The order is important). After these two operations, the system will do sys,print in each cycle. Therefore, sys should be a function of no arguments which reads text from the teletype according to its own conventions, and then does something to it. The value from sys will be automatically printed, so sys does not need to call print. For example, one reasonable definition of sys would be

```
(LAMBDA () (PROG (U)
  L (SETQ U (CONS (RATOM) U))
    (COND ((EQ (CAR U)(QUOTE STOP))
           (RETURN (ANSWERTO (CDR U))) )
          (GO L)))
```

### 3. ERROR MESSAGES

#### 3.1 Read errors

---READ ERROR. S-EXPR. BEGINS WITH )

the first char. in an S-expr is ). Re-type the S-expr!

---READ ERROR . NOT FOLLOWED BY ATOM

---READ ERROR .ATOM NOT FOLLOWED BY )

the rules for forming a dotted pair are not satisfied. Re-type the S-expr.

#### 3.2 System errors

OBJECT STACK IS EXCEEDING n.

No more atoms can be defined. All further attempts to define an atom will give NIL as result.

FUNCTION PDL IS EMPTY

The function stack is empty and the system restarts.

DONT USE % - YOU ARE NOT IN A BREAK

The system restarts.

MODE=3 BUT SYS UNDEF? TALK TO EVAL

See page for proper use of mode. The system acts as an evalsystem.

BIT ARRAY USED OUT OF BOUNDS, INDEX=n

When using clearbit(n), setbit(n) or testbit(n), n is outside (1,24). The system goes down in a break.

DIVIDE n BY ZERO

In quotient(m,n), n is equal to zero. The system goes down in a break.

PUT OUTSIDE ITS DOMAIN

In put(a,l,e), a is not an atom. Put will be ignored.

RETURN OUTSIDE PROG

return(lab) is used outside a prog. The system restarts.

I HAVE UNDERFLOW IN NUMERICAL OPERATION

could be found in any numerical Lisp-function. The system restarts.

I HAVE S-EXPRESSION WHOSE INDEX < 0

what should be a list is in fact (terminated by) a numerical atom. The system restarts.

!?!?!?

the interpreter is entirely confused by the data it has set up for itself. Could be wrong use or rplaca or rplacd. The system restarts.

BAD ARITHM IN ARG: m IN PROGRAM POS N: n

a non-numerical argument is given to a numerical function. n stands for the statement-number in the main program and m is the value of ARG. The system goes down in a break.

INDEX OUTSIDE (0,NFREE) IN POSITION n  
OF INTERPRETER

an argument, supposed to be an atom or list, is not an atom or list. The system restarts.

ARGUMENTS PDL IS FULL; JP=n

the argument stack is full. The system restarts.

FUNCTION PDL IS FULL! IP=n

the function stack is full. The system restarts.

### 3.3 Lisp errors A2 - A9.

I FAIL, DIAGNOSTIC A<sub>9</sub><sup>2</sup>

undefined function.

I FAIL, DIAGNOSTIC A<sub>8</sub>

undefined variable.

FIRST ARG OF SET/SETQ NOT ON A-LIST

error A<sub>4</sub>

GO TO xxxx NOT DEFINED

error A<sub>6</sub>

A<sub>2</sub>, A<sub>4</sub>, A<sub>6</sub>, A<sub>8</sub> and A<sub>9</sub> are ordinary SUBR's, which call break() after printing the error message. For explanation of break() see next page.

#### 4. TOOLS FOR DEBUGGING LISP PROGRAMS

##### 4.1 The function break

When errors A2 - A9 (and some others) occur, the system goes down in a break. This means that the system stores that point in the program where the error occurs, so that evaluation can be continued at a later stage. It then goes into a read-eval-print loop (similar to the top-level loop). Thus the user has here a facility to look at the values of the variables in the S-expression, which caused the error, or of just looking around in that S-expression. When wishing to terminate a break, i.e. restart from the point where the break occurred, type

```
% S-expression
```

and the evaluation of the earlier S-expression will continue with the value of the undefined variable or function.

If an error occurs in a break, the system goes down in a break and

```
% S-expression
```

will take the system up to the first break. No special limit for the depth of a break is given, but as the depth of breaks increases, the pushdown list will ultimately be exceeded.

It is also possible to call break from LISP by the LISP-function break().

Example of using break. (? stands for input-lines)

```
?(DE FOO (A)(PROG (N) (SETQ N (QUOTE ABC)) L1 (SETQ N (CAR H))
? (BREAK) (RETURN N]
?(FOO NIL)
```

```
I FAIL DIAGNOSTIC A8 UNDEFINED VARIABLE H
BREAK
```

```
?N                look at variable N.
```

```
ABC
```

```
?% (QUOTE (X Y Z))    give (X Y Z) as value to H and return
                      from break
```

BREAK CALLED FROM PROGRAM

? N   look at N  
 ?% NIL                                     return from break  
 X   value of (FOO NIL)

```
?(DE FAK (N) (COND((EQ N 1)A)(T(TIMES N(FAK(SUB1 N)>
FAK
?(FAK 3)
```

I FAIL DIAGNOSTIC A8 UNDEF VARIABLE:

BREAK

```
?N                                         look at variable N
1
?(FUNCTION)                               look at the association-list
(FUNARG NIL ((N .1)(N .2)(N .3)(NIL FAK )))
?(CDR (QUOTE FAK>                        look at the definition of FAK
(EXPR (LAMBDA (N )
      (COND ((EQ N 1 )A )
            (T (TIMES N (FAK (SUB1 N ))))))))
?% 1                                       give 1 as value to A and return from
break
6
```

#### 4.2 The functions A2 - A9

As A2 - A9 are defined as normal SUBR's, it is possible to re-define them as EXPR's and let one's own LISP-functions take care of those kinds of errors. For suggestions how to use this feature, see the 3600 LISP U3 user's manual.

#### 4.3 The function peek

There is a LISP-function called peek(), which calls the subroutine TESTUT. With TESTUT it is possible to see how the list-structures are represented internally. For using TESTUT see page 44.

#### 4.4 The LISP editor

The editor is a very simplified version of the one used in the BBN on-line LISP system (see reference 4, chapter IX). In its present rather primitive form, the editor provides a number of commands for modifying function definitions only of the EXPR or FEXPR variety (the BBN editor can be used not only on all functions but also on variables, property lists, and arbitrary expressions). Nor, for the sake of economy of implementation, is there complete error checking - see Error Checking below.

The Structure of the Editor Language. There are three types of command available :-

- a. those for looking around inside the definition of a function, that is for accessing list and sub-list structures.
- b. those for changing the components of lists and sub-lists.
- c. those for modifying the structure of lists.

We will consider the first here.

Entering the Editor. To enter the editor, type

(EDIT FNAME)

where FNAME is the name of the function to be edited.

At any moment the editor is looking at some subexpression of the function definition list, called the Current Level (CL), while the lambda expression of the function, viz. (LAMBDA .....), is the value of what is called the Current Top Level (CTL), that is the reference point from which lower level subexpressions are accessed. (But see Changing the Value of CTL, below). When the editor is first entered both CL and CTL point to the same list, namely (LAMBDA .....).

Sub-list Attention Commands. The current level of attention (CL) is changed by typing an integer n, where n sets CL to the nth subexpression of the current level's present value. For example, if the current level is the list (A B C (D (E)) F), then 3 will set CL pointing to the element C;

while 4 will set it to the expression (D (E)), etc. To access a sub-expression at any depth, one merely types in a succession of integers until CL points to the right level. Thus, to point to the element E in the above example, it is necessary to type

```
? 4 2 1
```

Whereas to point to the list (E), **type** only

```
? 4 2
```

The Print Commands. The list or element to which the current level is pointing, is printed on the console by typing the command P. To avoid wasteful printing, any list with sub-lists nesting to greater than a depth of 3, is printed only to depth 2, after which a couple of ~~xxx~~ signs are printed to indicate continuation of the list. Thus if CL points to (A (B (C (D))))), then

```
? P
(A (B (xxx xxx )))
```

will occur. For purposes of clarity full lists will sometimes be shown in the examples below.

The other print command is PR n where n is a positive integer or zero. The current level is printed to depth n and then ~~xxx~~ signs are substituted for the CDR if n>0. If n=0 the current level is printed in full.

Changing the Value of the Current Top Level. It may sometimes be useful to have CTL pointing to a level lower than the (LAMBDA ....) level, for example when editing is being performed on some deeply embedded sub-structure and it will never be necessary to return right to the top. To set CTL to some lower level, access the required level by suitable integer attention commands, then type the command UP, thus :-

```
? P
(A B C (D (E)) F)
? 4 UP 2 P
(E)
? 0 P
(D (E))
```



But note that the command UP destroys all memory of higher levels, so it is impossible to get back up again without exiting and re-entering the editor.

Exiting the Editor. When editing is completed, type OK which exits the editor and returns the name of the edited function to the console.

Component Changing Commands. There are 3 component changing commands -

Replacing Command. To replace the nth subexpression of the current level by one more expressions, type

(n e<sub>1</sub> e<sub>2</sub> ...) where n is an integer and e<sub>1</sub>, e<sub>2</sub> ... are arbitrary S-expressions. For example

```
? P
(A B C (D (E)) F)
? 4 (2 (G) H (J K))
? O P
(A B C (D (G) H (J K)) F)
```

Inserting Command. To insert one or more expressions immediately before the nth subexpression of the current level, type

(-n e<sub>1</sub> e<sub>2</sub> ...) where -n is a negative integer and e<sub>1</sub>, e<sub>2</sub> ... are arbitrary S-expressions. For example

```
? O P
(A B C (D (E)) F)
? 4 (-2 (G) H) O P
(A B C (D (G) H (E)) F)
```

Deleting Command. To delete the nth subexpression, apply the replacing command with null replacements, i.e. type

(n) where n is a positive integer. e.g.

```
? O P
(A B C (D (E)) F)
? (3) O P
(A B (D (E)) F)
```

Structure Changing Commands. There are six commands which allow alteration of the list structure itself. We will use the list

(A B C (D (E)) F) as an example throughout.

Left Parenthesis Out. The command (LO n) where n is a positive integer, takes out a left parenthesis at the nth subexpression and deletes all elements past the matching right parenthesis. Thus typing

? (LO 4)

produces (A B C D (E))

Left Parenthesis In. The command (LI n) where n is a positive integer, inserts a left parenthesis at the nth subexpression and a corresponding right parenthesis at the end of the list. Thus

? (LI 3)

produces (A B (C (D (E)) F))

Right Parenthesis Out. The command (RO n) where n is a positive integer, moves the right parenthesis of the nth subexpression out to the end of the list. Thus

? (RO 4)

produces (A B C (D (E) F))

Right Parenthesis In. The command (RI m n) where m and n are both positive integers, moves the right parenthesis at the end of the mth subexpression up to the nth subexpression of the mth subexpression, and subordinates the remaining elements one level. Thus

? (RI 4 1)

produces (A B C (D) (E) F)

Both Parenthesis Out. The command (BO n) where n is a positive integer, removes both parenthesis of the nth subexpression and subordinates its elements to the preceding level. Thus

? (BO 4)

produces (A B C D (E) F)

Both Parenthesis In. The command (BI m n) where both m and n are positive integers and  $n > m$ , raises the level of the mth through to the nth subexpression by putting a left parenthesis before the mth and a right parenthesis after the nth. Thus

? (BI Z 3)

produces (A (B C) (D (E)) F)

Error Checking. Should the user attempt to execute

- a. a command that does not exist in the editor's repertoire
- b. an attention command which is impossible, e.g. point to the third subexpression of a list containing only two
- c. a replacing command which is impossible, e.g. insert expressions before the third subexpression of a list containing only one.

then the message

?!?!?

will be output, and the value of CL will remain unchanged.

However, for economy of implementation and speed of execution of the editor, the error checking mechanism does not extend to impossible structure changing commands. Any attempt to execute such will either cause a break, destroy the function, or bring F1 down. Therefore great care must be used when changing the structure of a list.

#### Functions and Atoms Used by the Editor

EDIT, XEDIT, XCHANGE, XADJUST, SPRINT, SPR, APPEND, COPY, MINUS, NTH, LISTN, ILLGC.

#### 4.5 The LISP tracing routine, TRAC

TRAC is a simple tracing package for use on EXPR's and FEXPR's. Any traced function results in the name of the function, the name of each parameter and its value, and the value of the function, being printed whenever the function is called.

Using DEBUG. To set the trace on functions f1, f2, ..., type

```
? (TRACE f1 f2 ....)
```

which returns the value OK when complete. e.g.

```
? (DE COPY (L) (COND ((NULL L) NIL) (T (CONS (CAR (T (CONS (CAR L)(COPY (CDE
(CDR L
```

```
COPY
```

```
? (TRACE COPY)
```

```
OK
```

```
? (COPY (QUOTE (A B)))
```

```
+ COPY L = (A B)
```

```
+ COPY L = (B)
```

```
+ COPY L = NIL
```

```
+ COPY (A B)
```

```
+ COPY (A B)
```

```
+ COPY (A B)
```

```
(A B)
```

```
?
```

To remove the trace from functions f1, f2, ... , type

```
? (UNTRACE f1 f2 ...)
```

which also returns the value OK when complete.

Method Used. Each function to be traced is transformed into a PROG<sub>N</sub> containing the print statements for name and parameters and finally a call to the function itself (called by a different name - this is dealt with by a GENSYM). At the same time, the renamed original function is stored under the indicator TRFL in the atom whose print-name is the name of the function being traced. Because TRACE changes the structure of the function body, it is unwise to attempt any computation on the function body of a traced function.

Untracing puts into the correct function type whatever is stored in the GENSYM atom which itself is in TRFL of the function atom.

Functions and Atoms Used by DEBUG.

TRACE, UNTRACE, TRCL, UTRCL, DMAP, FLIST, OUTP, COPY, PRINL, QUOTE\*, UOT, MAPCARL, PRINZ, LPAR, RPAR, SPC, BLANK, DOT, INT, UNT.

5. DIFFERENCES FROM LISP 1.5

The following general changes have been made to the system specified in the LISP 1.5 programmer's manual:

1. Functions may be called with too many or too few arguments. Missing arguments are taken as NIL. Extra arguments are evaluated but not used.
2. Expressions of the type (LAMBDA U (---)) are permitted. U is then bound to the list of evaluated arguments ("no-spread lambdas"). Expressions like (LAMBDA (U V . W) (---)) are also permitted. U is then bound to the first argument, V to the second argument, and W to the list of remaining arguments.
3. If we "pass through the bottom" of a COND (i.e. no condition is satisfied), the value of the whole expression is NIL. (In LISP 1.5, an error message is obtained).
4. The top-level loop and the responses to errors can be controlled by the user.
5. The system only handles fixed-point (integer) arithmetic. QUOTIENT truncates its value to the nearest lower integer.
6. A number of functions have been deleted and added to the system. In particular, the following functions have been added:

BREAK  
 DE  
 DF  
 DUMP  
 EXIT  
 GENSYM  
 GOTO  
 MEMB  
 MODE  
 OBLIST  
 PACKLIST, UNPACK  
 PRINTPOS  
 READPOS

PROGN  
 RESTART  
 SETBIT, CLEARBIT, TESTBIT  
 SILENCE, TALK  
 TERPRI

The details of these functions are:

BREAK (See sec. 4.1 at page 11)

DE and DF

The simplest way to define s-expressions as expr's and fexpr's is to use DE or DF (where DE stands for Define Expr and DF for Define Fexpr). Use DE as follows:

```
(DE ARG1 ARG2 ARG3)
```

where

ARG1 is the atom to be defined.

ARG2 is a parameter-list.

ARG3 is an S-expression.

The value of DE is ARG1.

Example: For defining add1(x):=plus(x,1) type

```
(DE ADD1 (X) (PLUS X 1))
```

DE will add

```
(EXPR (LAMBDA (X)(PLUS X 1)))
```

to the property-list of ADD1.

DUMP

The function dump(l s) is used for dumping previously defined EXPR's and FEXPR's. l is the output unit number and s is a list of EXPR's and FEXPR's which will be dumped. Note that DUMP itself is a SUBR.

EXIT

Evaluation of exit (with no arguments) terminates the execution of the LISP interpreter.

GENSYM

The function gensym(x) differs from gensym() in Lisp 1.5 in that a new atom is created, whose printname consists of two parts. The first part is the first 4 characters in the atom x. The second part is a running

number, starting from 0000. Example :

```
?(GENSYM (QUOTE ABCD))
ABCD0000
?(GENSYM (QUOTE XXXX))
XXXX0001
```

If x has less than four characters, it is padded zeroes :

```
? ?(GENSYM T)
T0000002
```

### GOTO

goto(x) is defined as go(x) in Lisp 1.5 but will evaluate x first. (The conventional GO also exists).

### MEMB

The function memb(x,y) is analogous to member(x,y), (see Lisp 1.5). The only difference is that memb is defined with eq and member is defined with equal.

### MODE

The function mode(n) sets the variable MODE to n. The variable MODE stands for the kind of top-level function. See page 31.

### OBLIST

oblist(x) will generate an objectlist from the object stack (see fig page 24) starting with the atom x. As the atom T is the last system atom, oblist(T) will generate a list with T at the beginning and then all atoms defined by the user.

### PACKLIST

Given a list x of single-character atoms, packlist(x) will create a new atom, whose printname consists of the characters in the list x. Note that only the first 8 characters are used. Example :

```
(PACKLIST (QUOTE (A B C D)))
xxxx ABCD
```

Packlist can also be used on a list of numerical atoms, or on a list of both numerical and non-numerical atoms. In the first case, the result will be a numerical atom, in the latter case a non-numerical atom.

PRINTPOS

When creating an output line, the variable PRTFNT points to the place in the outputbuffer, where the next character is to be placed.

The value of printpos() is the actual value of the variable PRTFNT - 1.

Printpos(n) sets PRTFNT to n (and returns with n as value).

Note: Printpos(60) will skip to a new line.

READPOS

The input line is first placed in an inputbuffer. Then successive characters are read from this buffer. The characterpointer is the variable RDPNT, and it points to the next character to be read.

The value of readpos() is the actual value of the variable RDPNT-1.

Readpos(n) sets the variable RDPNT to n (and returns with n as value).

Note: Readpos(71) will cause the system to skip to the next input line.

UNPACK

This function "explodes" the name of an atom. If the atom ALPHA is given as argument, unpack(ALPHA) will return the list (A L P H A) of character-atoms. If you want to modify a printname (e.g. remove the last letter), use unpack, modify and then use packlist. The atom to eplode may be a numerical atom.

PROGN

progn(x1,x2,x3,x4,....,xn) evaluates all arguments and returns the value of the last argument. (cf. prog2(x,y) ).

RESTART

restart() will restart the Lisp-F1 system to the same status as at the beginning of the run except that all user-defined atoms are left.

SETBIT, CLEARBIT, TESTBIT

In Lisp-F1 there is a simulated 24-bit register to maintain compatibility with 3600 LISP. The system now uses bit nr 24 for silence() and talk().

The user can manipulate and test the bits by using

setbit(n) puts the n'th bit on. The value of setbit is NIL.

clearbit(n) puts the n'th bit off. The value of clearbit is NIL.

testbit(n) T if the n'th bit is on, else NIL.



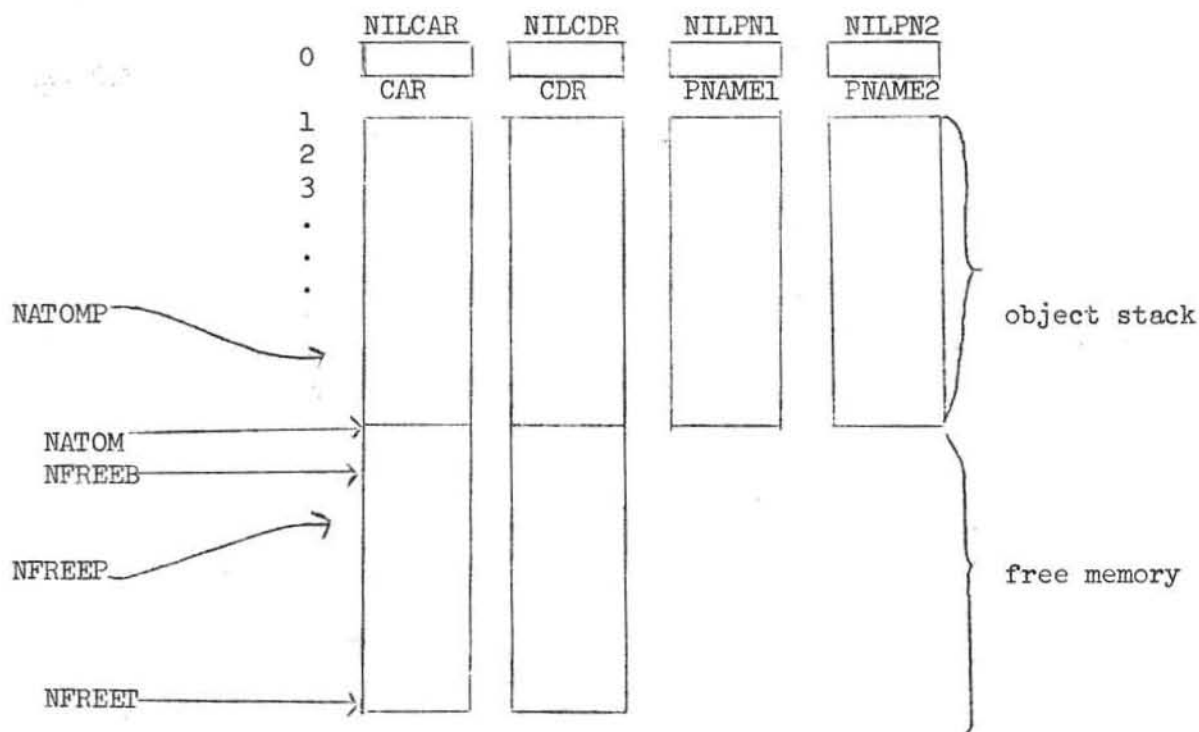
SILENCE, TALK

System printout (value from eval, but not printouts from print and terpri) is switched off by evaluating silence(), and switched on again by doing talk(). Both functions have NIL as value. The system uses bit nr 24 in the simulated register for controlling this.

6. REPRESENTATION OF ATOMS AND LIST STRUCTURES

The following arrays and pointers are used by the system:

Figure 6.1



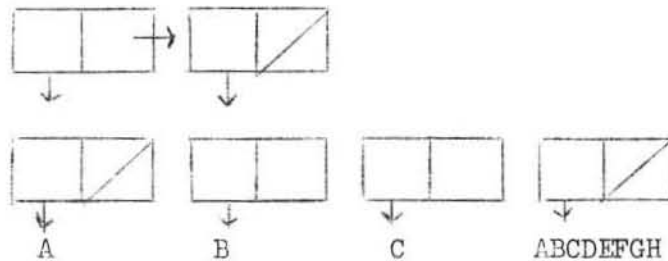
All free memory and the object stack are represented by the integer vectors CAR and CDR. The contents of CAR are (mostly) used as indices to the vector CAR **for** pointing in car-direction. CDR is analogous. NIL is represented by zero. To make it possible to use NIL (= zero) as an index in CAR and CDR, NILCAR and NILCDR are placed immediately before CAR and CDR respectively in the common area.

All atoms are represented by integers between 0 and NATOM (=500 for the present). For atoms, there are two corresponding BCD-vectors PNAME1 and PNAME2, containing the printnames (in this system max. 8 characters). We are now ready to illustrate the representation of a list-structure by an example

Figure 6.2

<u>index</u>	<u>CAR</u>	<u>CDR</u>	<u>PNAME1</u>	<u>PNAME2</u>
0			NIL	
1	-6000	0	A	
2	-6000	0	B	
3	-6000	0	C	
4	-6000	0	ABCD	EFGH
.				
.				
.				
505	506	507		
506	1	0		
507	508	0		
508	2	509		
509	3	510		
510	4	0		

Index 505 represents the list ((A)(B C ABCDEFGH)).



The value of an atom is stored in `CAR(ATOM)`. The contents of `CAR(ATOM)` are

UNDEF (= -6000) until the atom is defined.

The property-list of an atom is stored in `CDR(ATOM)`. When the property-list is empty, the contents of `CDR(ATOM)` are NIL (= zero)

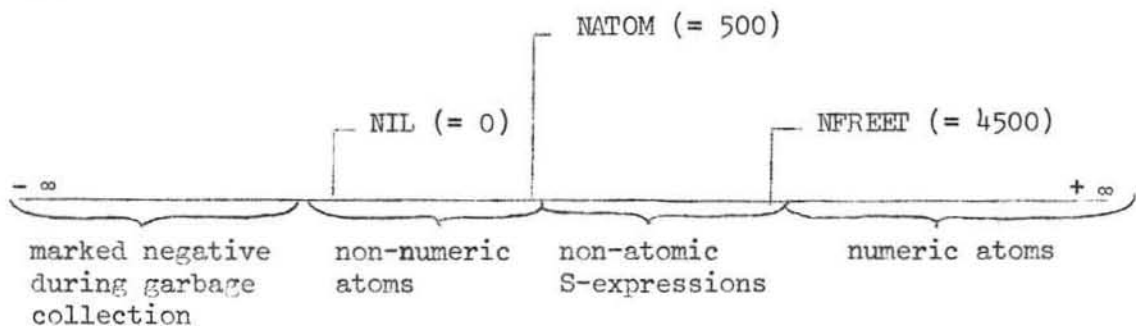
The printname of an atom is stored in `PNAME1 (ATOM)` and `PNAME2(ATOM)`.

If the printname contains less than 8 characters, the end of the printname is marked by the BCD character stored in the FORTRAN variable `ATEND(=%, for the moment)`. Note, that this character can never be printed.

Numerical atoms are not stored like the ordinary atoms, but are represented directly in the list-structures by integers greater than NFREEP (=4500). To get the value of a numerical atom, the system first subtracts NUMADD (=50000). All numerical atoms have integer values. ATEND and NUMADD can be changed by giving them new values in routine INIT.

### Fortran representation of atoms and other S-expressions

Figure 6.3



### Program-variables used as pointers in the free memory (see fig. 6.1)

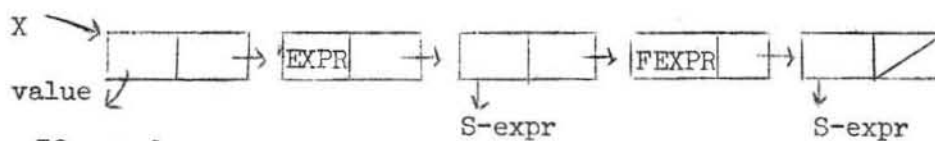
NATOMP the last atom in the object stack.  
 NATOM top of the object stack (=500)  
 NFREEB the first cell in the free memory (=NATOM +1)  
 NFREEP points to the beginning of the free list.  
 NFREEE the last cell of the free memory (=4500)

A property-list in this system is a straight list where indicators and corresponding properties alternate, like in 7090 LISP. For each atom, car(atom) points to the value for that atom (set by cset or csetq) and cdr(atom) points to the propertylist.

(P1 V1 P2 V2 P3 V3.....)

P1 is the first indicator and V1 is the first property and so on.

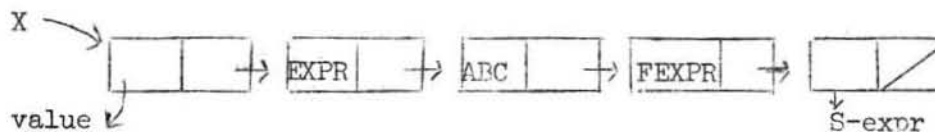
Example of a typical propertylist of the atom X.



If you do

```
(PUT (QUOTE X)(QUOTE ABC)(QUOTE EXPR))
```

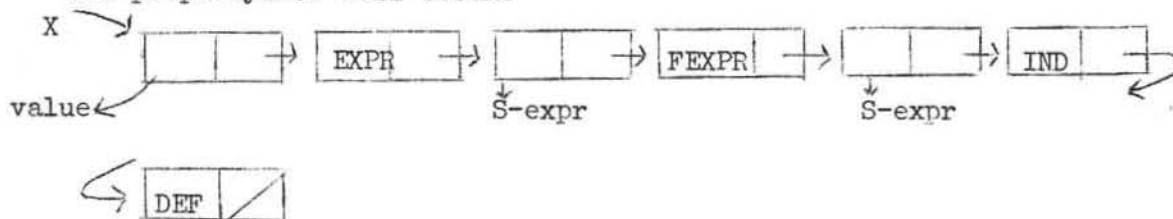
the propertylist will become



and if you do

```
(PUT (QUOTE X)(QUOTE DEF)(QUOTE IND))
```

the propertylist will become



The function `put(x,y,z)` puts `y` on the propertylist of `x` under the indicator `z`, and is defined as follows:

```
put(x,y,z)={if cdr(x)=NIL then rplacd(x, list(z,y))
             elseif cadr(x) = z then rplaca(cddr(x), y)
             else put(cddr(x), y, z) ; y}
```

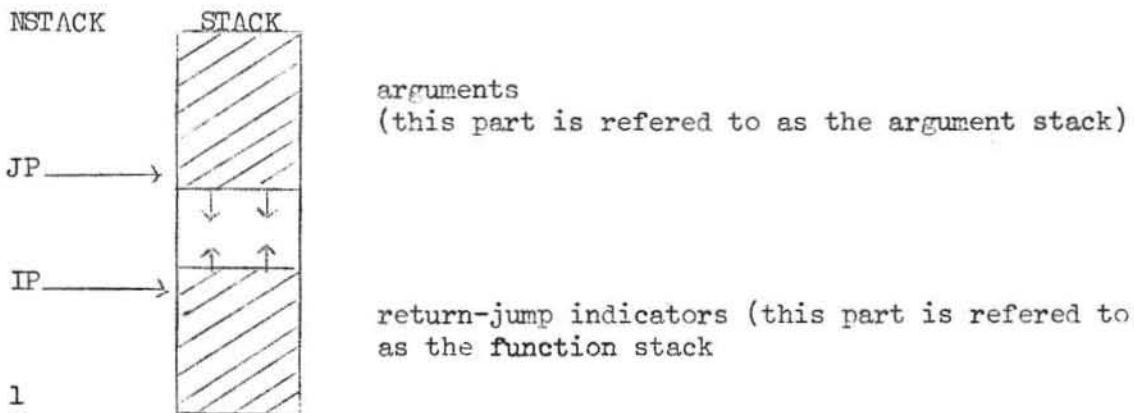
## 7. RECURSIVE PROGRAMMING IN FORTRAN

The LISP F1 interpreter is an almost direct translation from the definition of LISP in LISP 1.5 (see ref. 1, page 53) to FORTRAN. That definition is highly recursive. This chapter explains how recursion has been programmed in FORTRAN.

### 7.1 The stacks.

There are two stacks for recursive calls of "sub-programs". One stack contains saved arguments (the argument stack), and the other contains saved return-jump indicators (the function stack). They are both physically stored in the vector STACK. The size of STACK is NSTACK (=900). IP and JP are the pointers, which hold the current sizes of each stack. To push down in the function stack increase IP and store. To push down on the argument stack decrease JP and store. Vice versa for pop up.

Figure 7.1



The following 4 routines handle the stack:

- APUSH(J)      Push down one argument in argument stack.
- APOP(J)      Pop up one argument from argument stack.

FPUSH(I,J)      Push down one return-jump indicator I in the function stack. J is a dummy variable, only used in a call to have the program more readable for the programmer. Ex. FPUSH(5,1302) means that 5 is an indicator for the statement number 1302.

998 I=STACK(IP) This piece of code (lines 62,63 in the main program)  
       IP=IP-1      pops up one return-jump indicator and places it in I.  
       Statement 998 is the standard return for all recursive routines.

## 7.2 Recursive calls and returns.

All recursive functions (loop, evqt, appyl, eval, evlis, evcon, cassoc, sassoc and mach) are coded in the main program. It means that they are not FORTRAN-subroutines, but just pieces of FORTRAN code.

A call is made by saving necessary arguments with APUSH(ARG) and by saving the return-jump indicator with FPUSH(IND,statement number). IND stands for an indicator in a computed GO TO ( ), IND where IND corresponds to the correct return statement number. After saving, there is an unconditional GO TO to the "subroutine", and then follows the return-statement (with the same statement number as indicated by IND); there, corresponding unsaving of arguments is done by APOP(ARG).

A return from a recursive function is normally done by GO TO 998. At 998 a statement number indicator is popped up from the function stack and used in a computed GO TO which leaves the program control to the calling sequence.

Example from eval-apply:

```

1      998  I=STACK(IP)
2      IP=IP-1
3      GO TO(500,1103,3000,3501,.....),I
      .
      .
      .
4      3500 CALL APUSH(ARG2)
5      CALL APUSH(ARG3)
6      CALL FPUSH(4,3501)
7      GO TO 4000
8      3501 CALL APOP(ARG3)
9      CALL APOP(ARG2)
      .
      .
      .
10     4000 IF(...)
      .
      .
      .
11     GO TO 998

```

Begin at line 4! The lines 4-9 are taken from apply. The lines 10,11 are taken from eval.

line 4,5 save ARG2 and ARG3 (EVAL perhaps destroys them).

line 6 save return-jump indicator 4 (corresponding to statement number 3501).

line 7 jump to eval

line 10 entry in eval. After doing eval,

line 11 GO TO 998, which takes us to the standard return

line 1 there the jump-indicator 4

line 2 is now popped up from the function stack and used in the GO TO( ) statement. We now jump to the 4'th statement in GO TO ( ) and reach 3501 CALL APOP.

line 8,9 Unsave ARG3 and ARG2 (note the reverse order).



## 8. THE MAIN PROGRAM IN THE INTERPRETER

### 8.1 The top level loop

In the top level loop, the system behaves slightly differently according to the value of the FORTRAN variable MODE:

<u>MODE</u>	<u>behavior in each cycle</u>
1.	read an S-expression, send it to eval, and (unless SILENCE has been ordered by the user) print the result
2.	read two S-expressions, send them to evalquote, and (unless SILENCE) print the result
3.	get the EXPR property of the LISP atom SYS and call <u>apply</u> with it as the first argument (and the other arguments NIL). Print the result unless SILENCE has been ordered. The function SYS must do all reading itself.

The value of MODE is set to 1 by INIT (the FORTRAN subroutine that performs initialization in the interpreter). The user can change the value of MODE, using the LISP function mode.

Most of the work in the interpreter is done in the sections of the main program that are labeled eval, apply, etc. The MODE feature is a superficial thing, and can be thought of as specifying three different entry points to the eval/apply complex.

### 8.2 Transmission of arguments for recursive functions

In the main program, all arguments for the recursive routines are held in ARG, ARG2, ARG3, ARG4. Except for sassoc and cassoc, ARG is the first argument, ARG2 is the second, and so on. In sassoc and cassoc the order is ARG, ARG2, ARG4.

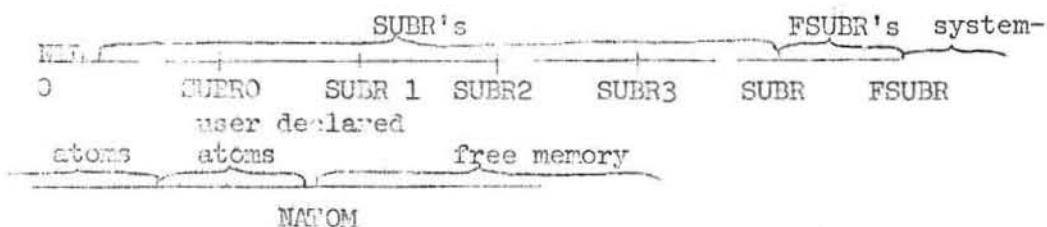
Instead of apply, the interpreter uses appyl(fn,a,args)=apply (fn,args,a) and instead of sassoc the interpreter uses cassoc(atm,a,erf) =cdr(sassoc(atm,a,erf)).

In this way, ARG2 always means the association-list. A call is then done by giving values to ARG, ARG2...., and using the calling-technique mentioned on page 20.

### 8.3 Representation of SUBR and FSUBR

System declared functions are numbered from 1 upwards.

Figure 8.3



SUBR's with no arguments are numbered in the interval (1,SUBRO). SUBR's with 1 argument are numbered in the interval (SUBRO+1,SUBR1) and so on for SUBR's with two and three arguments. In the interval (SUBR3+1,SUBR) we have SUBR's with an indefinite number of arguments (for example; plus). FSUBR's are numbered in the interval (SUBR+1,FSUBR) and after FSUBR come all system and user-declared atoms.

SUBRO,SUBR1,SUBR2... are FORTRAN variables, set by the routine INIT when the atoms are read from logical unit 5.

These conventions make it easy for eval to determine the type of a given S-expr. When eval is called, the S-expr. is held in ARG. If the value of ARG is greater than NFREET it stands for a numerical atom (whose integer value is ARG-NUMADD), there NUMADD is a constant set by routine INIT).

If the value of ARG lies in the interval (NFREEB,NFREET) it stands for a S-expr. If it lies in (0,NATOM) it stands for an atom. The interval

containing the value of ARG, defines the kind of atom. For example, if the value of ARG lies in (SUBR1+1, SUBR2), ARG stands for a SUBR with 2 arguments. By subtracting SUBR1 from ARG we can use the computed value in a computed GO TO-statement and jump directly to the code corresponding to the actual SUBR.

## 9. FUNCTIONS AND SUBROUTINES IN THE INTERPRETER

### Input routines:

IREAD  
JATOM  
RATOM  
SHIFT  
SBYT  
MATOM  
GETCH  
PUTCH

### Output routines:

IPRINT  
PRINAT  
OUTSTR

### Routines used at garbage-collection:

GARB  
MAKFRE  
CONS  
LISTNB

### Routines used from eval:

MEMB  
NBOX  
OK  
ADDAL  
KAR  
KDR  
GET  
APUSH  
FPUSH  
APOP

### Others:

INIT  
TESTUT  
EXCUSE

#### 9.1 FUNCTION IREAD(DUMMY)

IREAD is the subroutine, which constructs new lists from the S-notation input. IREAD uses the routine RATOM.  $ITYP=RATOM(NAT)$  gives the kind of atom in  $ITYP$  and the atom value in  $NAT$ . For  $ITYP$ -codes see page 38.

<u>Typed input</u>	<u>Value of IREAD and the COMMON-variable ARG</u>
a) S-expr.	IREAD=value of S-expr. ARG undefined.
b) % S-expr.	IREAD=-1. ARG=value of S-expr.

The latter return b) is used by the break-function (page 11)

IREAD constructs new lists non-recursively, and the main idea behind IREAD is described by the following example;

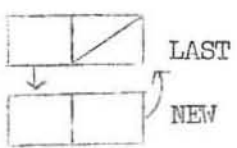
In general, start with a lisp-cell

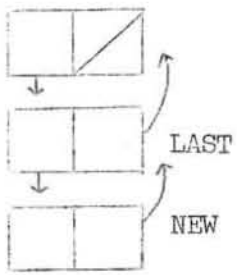
- Then let ( mean "go down one level (add a new cell in car-direction)".
- let atoms mean "add at this level (add a new cell in cdr-direction)".
- let ) mean "go up one level".

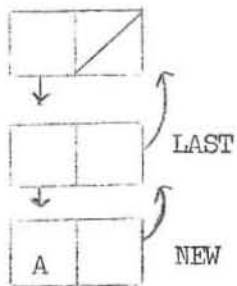
For going up one level, a pointer is always kept to the level above in the cdr of the last cell of the sublist.

Let us now construct the list ((A B) C).

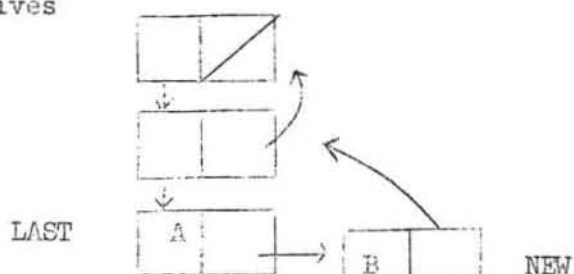
a, start with  NEW, LAST

b. ( gives  corresponding to (

c. ( gives  corresponding to ((

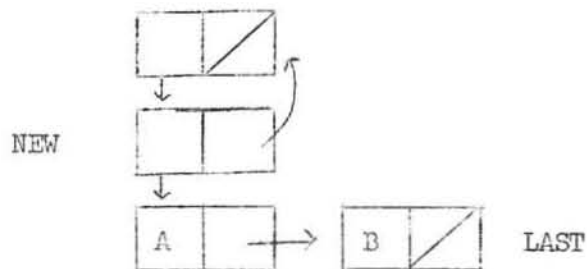
d. A gives  corresponding to ((A

e. B gives

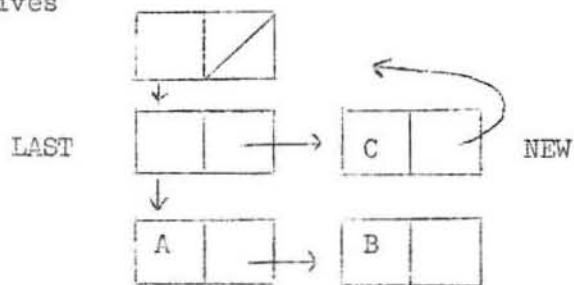


corresponding to ((A B)

f. )

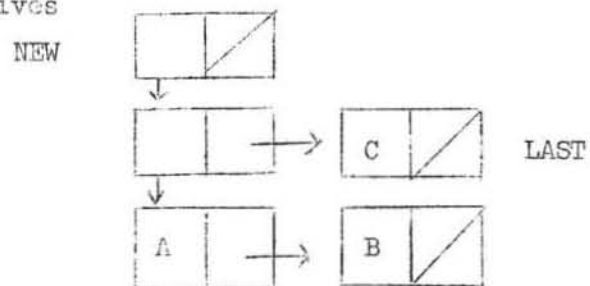


g. C gives



corresponding to ((A B) C

h. ) gives



corresponding to ((A B) C)

`cdr(NEW) = NIL` now indicates, that no more higher levels exist. The list-structure is complete, and the value is stored in `car(NEW)`.

In a dotted pair `atom1.atom2)`, `atom1` is treated as a new atom on its sub-list level and `.atom2)` is treated as `)` [or you can say, that `)` is treated as `.NIL)` ]

The flowchart for IREAD is given in appendix C, page 65.

## 9.2 SUBROUTINE JATOM(ITYP,IATOM)

Reads an atom or a break character. The type of the item is returned by setting ITYP to an integer. If the item type is "atom", then this atom is returned by setting IATOM. This routine is used by RATOM.

The details are: JATOM obtains an input character using SBYT and SHIFT. It classifies the character and sets ITYP as follows:

character	set ITYP to
(	2
)	3
.	4
[	5
]	6
%	7
otherwise	1

If ITYP is set to 1 the character is assumed to be the first character of an atom. (Blanks are disregarded). The atom is then completed (continue reading up to a blank or one of the break characters above), and the character-string for the atom sent to MATOM. The index of the generated atom is returned through IATOM.

## 9.3 FUNCTION RATOM(IATOM)

RATOM is basically like JATOM, with a few differences:

- (a) RATOM keeps "[" and "]" in a separate bracket stack;
- (b) RATOM combines the tip of a dotted list, i.e. . atom ), into one single item, a "generalized right parenthesis";
- (c) Being a function, RATOM returns the type (ITYP) as the value of the function rather than in an argument.

The details are: RATOM calls JATOM once (with some exceptions), and returns a value given by the following table:

characters encountered by JATOM	value put into IFYP by JATOM	value of RATOM
(	2	1
)	3	2
. atom )	4,1,3	2
atom	1	3
%	7	5
[	5	1
]	6	2

In the case of atom and .atom , JATOM is set to the index of this atom. In the case of ) and ], JATOM is set to NIL (zero). When it sees a left bracket, RATOM makes a push operation on its local bracket stack. When it sees a right bracket, it makes a pop and starts a countdown. During the countdown, each call of RATOM will return a rightpar without calling JATOM. Thus RATOM generates the necessary right parentheses until the brackets have been matched.

#### 9.4 SUBROUTINE SHIFT(IC)

Used by JATOM. Uses PUTC.

SHIFT takes one BCD character from the input string and gives it to JATOM, and then, if necessary, reads one new input string from the standard input unit.

#### 9.5 SUBROUTINE SBYT(L,IB,K)

Used by JATOM. Uses PUTC.

Stores one BCD character in an 8-byte buffer, used later by MATOM to form the printname of an atom.

#### 9.6 FUNCTION MATOM(PN1,PN2)

Used by JATOM.

MATOM makes an atom. If an atom with the same printname already exists, that atom will be the value; otherwise, a new atom is stored on the object stack.

The value of MATOM is the index of the atom (the existing or the new one).



### 9.7 GETCH and PUTCH

These are the two assembly-coded routines for character-manipulating.

```
CALL GETCH(LIN,IC,N)
```

puts the N'th character in LIN (counting from the left) into IC, left justified. The remainder of IC is filled with the blank characters. LIN and IC are supposed to contain at least 4 BCD characters.

Example:

```
LIN=
```

```
  ABCD
```

```
after CALL GETCH(LIN,IC,3) IC will be [C ]
```

```
CALL PUTCH(LIN,IC,N)
```

puts the leftmost character of IC into the N'th place in LIN.

Example:

```
LIN=  ABCD
```

```
IC=   A
```

after CALL PUTCH(LIN,IC,3) , LIN will be ABAD ..

### 9.8 SUBROUTINE IPRINT(I)

IPRINT is a non-recursive routine, which performs a print-by-scan on a list I. The scan method is the same as in routine GARB (see page 41). The only difference is that the forward scan is done in the car-direction, while searching for branch-points is done in the cdr-direction. IPRINT, of course, leaves the list unmarked after the printing. (The reader is now recommended to read page 41 for explanation of forward scan and backward scan)

Forward scan (car direction).

If CAR(I) is an atom, print atom and start reverse scan, otherwise print ( and go on with forward scan.

Reverse scan:

If CDR(I) is an atom or NIL, print .atom) or ) and go on with reverse scan, otherwise start forward scan on the sublist.

The print described briefly above is done by call to routine PRINAT(I,IATOM)

there

I=1 means (  
 I=2 means )  
 I=3 means .atom)  
 I=4 means atom

and IATOM is the value of the atom in cases 3 and 4.

### 9.9 SUBROUTINE PRINAT(I, IATOM)

Used by IPRINT. Uses PCHAR, GETCH and OUTSTR.

Depending on I, PRINAT puts the following in an output buffer, which is then written by OUTSTR.

<u>I</u>	<u>ATOM</u>	<u>to put in buffer</u>	
1	-	(	
2	-	)	
3	atom value	.atom)	
4	"	atom	atom here is the printname of the atom

PRINAT also puts spaces in the output buffer in order to get a pretty-print looking output. The rules for pretty-print programmed in PRINAT are as follows:

- the combination )( gives a new line.
- a label in a prog gives a new line.
- every new line begins with spaces, indicating the depth of this level. This is done by calculating the number of unbalanced left parentheses printed previously and printing 2 blanks for each.

### 9.10 SUBROUTINE OUTSTR( IBUFF, IBL, IND)

Used by PRINAT.

First writes one output line on standard output, and then puts a correct number of spaces in the output buffer (IBUFF).

### 9.11 SUBROUTINE PCHAR(IREF, NPL, IND)

Used by PRINAT. Uses FUTCH.

PCHAR takes one BCD character in  $IND$ , and puts it in the BCD vector IREF. Where to put it, is indicated by IND.

### 9.12 Garbage-collection.

When CONS is called, and the free-list is empty, the routine GARB(I) is called with the following lists:

I1	}	The two arguments for CONS
I2		
ARG	}	Lists used and not necessarily saved by EVAL
ARG2		
ARG3		
STACK(I)		All active elements in the argument stack
CAR(I)	}	Each element in the object stack (i.e. <u>car</u> and <u>cdr</u> of every atom)
CDR(I)		
LASTEN		The list under construction by IREAD

GARB(I) then marks these lists. After this there is a call to MAKEFR, which will make a free list of all unmarked cells.

Note. For the present no atom-garbage-collection exists.

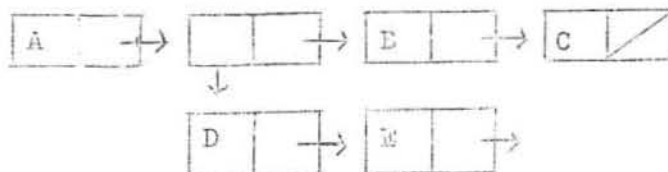
### SUBROUTINE GARB(I)

This is a non-recursive routine, used at garbage-collection time for marking the list  $L$ . The idea is described in CACM Aug 57 (nr 8) and so there follows here only a brief explanation.

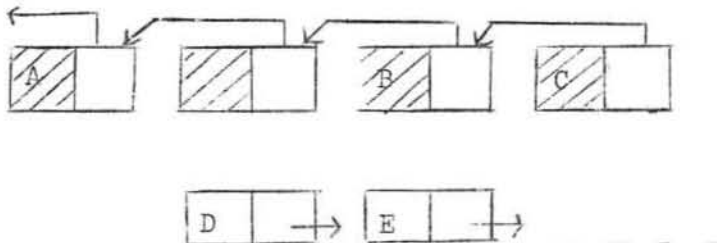
#### FORWARD SCAN:

Scan to the right (in cdr-direction) until the lists ends, or a marked cell is found. While doing that, reverse the cdr's, pointing them to the left, and mark the cells as examined (here indicated as ~~///~~)


Example:



after the forward scan:

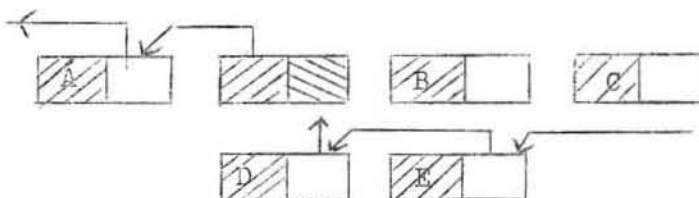


Backward scan:

Go in cdr-direction (which is now to the left) and restore old cdr-values. While doing that, search for any car's pointing to sublists. Call such a cell a branch-point. When a branch-point is found, put old cdr-value in cdr, and put the "reverse cdr" (which just before was the contents of cdr) in car, and mark this cell as a branch-point (here indicated as ) . Then do the forward scan of the sublist found.

Example:

After forward scan of the bottom branch the list above will be



A flowchart for this procedure is given in appendix C page 68.

For marking a cell as examined, do

```
car(i) := -car(i) - 1
```

and for marking a cell as a branch-point, do

```
cdr(i) := -cdr(i) - 1
```

### 9.13 FUNCTION MEMB(I,JJ)

FORTTRAN-routine for the Lisp-function memb(x,y). See also page 21.

### 9.14 FUNCTION NBOX(I,J)

NBOX(I,J) = I-NUMADD. The function checks if the value of I lies in the interval defined for numerical atoms.

9.15 FUNCTION ADDAL(IA,IV,E)

Adds dotted pairs, taken from the elements of IA and IV, to the association list E. M-expression definition:

```
addal [ia,iv,e]= nconc [pair [ia,iv], e]
```

9.16 FUNCTION KAR(I)

Does car(i).

9.17 FUNCTION KDR(I)

Does cdr(i).

In critical points in the program, KAR and KDR are used instead of just the vectors CAR and CDR. The reason for this, is that when debugging the Lisp-system, tests on I could be used in KAR and KDR. (When debugging the system, it turned out that this special test was not necessary and the function may be removed).

9.18 FUNCTION GET(J,I)

FORTTRAN-routine for the Lisp-function get(x,y).

9.19 SUBROUTINE APUSH(J)

Puts one value on the argument stack and increases the stack-pointer.

9.20 SUBROUTINE FPUSH(I,J)

Puts one value on the function stack and increases the stack-pointer.  
J is a dummy-variable.

9.21 SUBROUTINE APOP(J)

Takes one value from the argument stack and decreases the stack-pointer.  
More about APUSH, FPUSH, and APOP can be read on page 28.

9.22 SUBROUTINE INIT(IRESTA)

This routine initiates the Lisp-system by doing the following:

- a) Sets pointers to correct initial values. For actual pointers see page 46.
- b) Reads BCD characters ( ), . and so on from logical unit 5. The characters are used for reference later on.
- c) Reads BCD characters used as printnames for system atoms. This will also be from logical unit 5.
- d) Puts system atoms on the object stack.
- e) Makes a free list by calling MAKFRE.
- f) Initiates the routines SHIFT, GETCH and JATOM.
- g) Define a cell ALIST used as a pointer to the association list in eval.

If IRESTA<0, INIT restarts the system by doing a, e,f,g.

9.23 SUBROUTINE MAKFRE

Generates a free list of all unmarked cells in the free memory space. A marked cell is a cell with negative car. MAKFRE also unmarks all previously marked cells.

9.24 FUNCTION CONS(I1,I2)

Does cons(x,y). In the case of an empty free list, garbage collection is called.

9.25 FUNCTION LISTNB(LISTST)

Counts the elements in a one-way list.

9.26 SUBROUTINE TESTUT

TESTUT is an auxiliary routine for debugging the system and, in some cases, for debugging LISP-functions. From LISP, it is entered by the

LISP-function peek(). With TESTUT you can see how list-structures are represented in core. After calling TESTUT, the following will be written on standard output:

```
IT MIN MAX
?
```

You now have to type in integer values for IT, MIN, and MAX in free format.

If IT is positive, the object stack and/or the free memory, represented by CAR(I) and CDR(I) will be printed with I varying from MIN to MAX.

If IT is zero, the printnames of the atoms, represented by the vectors PNAME1(I) and PNAME2(I) will also be printed.

If IT is negative, TESTUT makes an ordinary return.

Example:

```
?(PEEF)
IT MIN MAX
? 0 1 4
1 -6000 0 A
2 -6000 0 B
3 -6000 0 C
4 -6000 0 ABCD EFGH
IT MIN MAX
? 1 505 507
505 506 507
506 1 0
507 508 0
```

Lines followed by ? are input-lines.  
The example is taken from the example of list structures on page 25.

Note: For the present, TESTUT uses IINPUT(IT,MIN,MAX) for input in free format. This is a library function of Datema timesharing service in Stockholm (see also page 47).

## 9.27 SUBROUTINE EXCUSE

Writes out an excuse-message when the system is confused by the data it has set up for itself.

## 10. IMPLEMENTATION GUIDE

LISP-F1 is at present provided as a FORTRAN program on punched cards (in EBCDIC format). It is written in Basic FORTRAN IV for IBM. For implementation of the LISP F1-system on other computers observe the following rules:

### 10.1 Memory requirements

For the present, the system needs about 85 k bytes in an IBM 360/40. The space is divided into

Program space	35k bytes
Common space	45k bytes

The common space can be changed, and in fact this is recommended in order to get as much free memory as possible.

### 10.2 The COMMON area

The vectors CAR and CDR contain the atom space and the free memory space. The total space for this is now 4500 fullwords. The vectors PNAME1 and PNAME2 contain the printnames for atoms (now 500 fullwords each) and STACK contains the argument-function stack (900 fullwords). Changing the COMMON area must be done in the main program as well as in all subroutines and functions.

Changes in COMMON statements must be supplemented by changes of certain constants that are set in the routine INIT. See the following section.

### 10.3 Constants set in INIT

The following constants are set in the initialization routine INIT:



## Fortran variable Meaning

NFREET	The size of the vectors CAR and CDR. (NFREET stands for the total atom and free memory space).
NATOM	The size of the vectors PNAME1 and PNAME2. (NATOM divides the total space in atom space and free memory space).
NSTACK	The size of the vector STACK.

## 10.4 Logical units for I/O

The variable LUNIN indicates the standard input unit, and LUNUT the standard output unit. The variables LUNIN and LUNUT are set by routine INIT to 9 and 6, respectively. This can be changed by setting the variables LUNIN and LUNUT to any desired values.

Standard I/O units can also be changed from LISP by the LISP-functions inunit(n) and outunit(n) where n is the unit number wanted.

Note: All S-expressions are printed on standard output, but all system-messages (i.e. error messages, dump messages, etc.) are printed on logical unit 6.

The initialization phase of the LISP-system involves reading all system atoms from logical unit 5 via the routine INIT. Also BCD-codes for (), .[] and so on, are read from unit 5 at initialization. The required input for this is given in appendix D page 69.

## 10.5 Non-standard FORTRAN-routines

The routines PUTCH and GETCH are coded in assembly language for IBM-360, and have to be re-coded in the case of other computers. For definitions of PUTCH and GETCH see page 39. The routine IINPUT used in routine TESTUT is a library routine for format-free input at the Datema time-sharing service in Stockholm. In other systems this may have to be re-written. For example in TESTUT, change

```
1      .....  
      CALL IINPUT(IT,MIN,MAX)
```

to

```
1      .....  
      READ(LUNIN,100) IT,MIN,MAX  
100   FORMAT(3I4)
```

## 10.6 Use of LISP-F1 in interactive mode

Before compiling the program (after taking care of points 1 to 5) the data deck containing the system-atoms (appendix D page 69) should be read to logical unit 5. Then compile the program and run it as an ordinary interactive FORTRAN-program.

## 10.7 Use of LISP-F1 in batch mode

The system is written for interactive use, but can be used as a batch-program as well. In that case, notice the following:

10.7.1 If you get a LISP-error A2 - A9, the debugging function break() wants input from the standard input unit. This input is of course not available (in general) on a batch-run. It is therefore recommended to avoid this by replacing the statements

```
GO TO 1000
```

in A2 - A9 by the statement

```
GO TO 998
```

The code for A2 - A9 is placed in the main program between statements nr 9201 and nr 9820+1.

This will have the same effect as if % NIL was given to break(), i.e. NIL is given to the undefined atom (or function) and the system continues the evaluation.

Another way of bypassing the break-function, is to define the LISP-functions A2 - A9 as EXPR's, (see section 4.2).

10.7.2 As the interpreter is written for interactive use, S-expressions given as input are not printed out (since they are normally typed in on the same medium as is used for output). To have the interpreter print out input S-expressions, add the following FORTRAN-statements to the main program:

```

after statement      1100  IARG=IREAD(0)
insert
                    IF(DREG(24)) 1150,1150,1151
                    1150  WRITE(LUNUT,1160)
                    1160  FORMAT ('ARGUMENT FOR EVAL...')
                    CALL IPRINT(IARG)
                    1151  CONTINUE

change statement    1104  CALL IPRINT(IRES)
to
                    1104  WRITE(LUNUT,1161)
                    1161  FORMAT(' VALUE IS...' )
                    CALL IPRINT(IRES)
after statement    1200  IARG=IREAD(0)
insert
                    IF(DREG(24)) 1152,1152,1153
                    1152  WRITE(LUNUT,1160)
                    CALL IPRINT(IARG)
                    1153  CONTINUE

```

Instead of changing the FORTRAN program the top-level function can be changed as described in the example given below

```

(DE SYS NIL (EVAL (PRINT (READ ))NIL ))
(MODE 3)
eval
-----
(DE SYS NIL (PROG (L )
  (SETP L (PRINT (READ )))
  (COND ((OR (GET L (QUOTE FEXPR ))
             (GET L (QUOTE FSUBR )))
        (EVAL (CONS L (PRINT (READ )))NIL ))
        (T (APPLY L (PRINT (READ ))NIL )))))
(MODE 3)
evalquote

```

11 HOW TO ADD NEW SUBR'S AND FSUBR'S

In order to follow the instructions below, the reader is recommended to read chapter 8, especially chapter 8.3 (Repr. of SUBR and FSUBR) on page 32.

To add a new SUBR (or FSUBR) you have to:

- a) Insert the function name in the data deck (see list, appendix D page 69). The location of the name is important and must correspond to a computed GO TO-statement. More about that under point b. The card shall be punched in the format (A1,2X,2A4). The first character describes the type of the function:

0	SUBR	no arguments
1	"	1 argument
2	"	2 arguments
3	"	3 arguments
N	"	no limit in the number of arguments
F	FSUBR	

- b) Change the corresponding computed GO TO-statement, determined by the type of the function:

type:	change statement:
0	9004 + 1
1	-"- (yes, it is the same)
2	9200 + 3
3	9309 + 4
N	9400 + 1
F	9591 + 1

In the corresponding GO TO-statement, insert the statement number referring to your own piece of code, so that the order in the data deck (after insertion) still corresponds to the GO TO-statement. It is also recommended to change the comment card, telling which statement number refers to which function.

c) Put in your piece of code somewhere.

When entering the arguments is hold in ARG,ARG2,ARG3 as shown below:

type:	arguments are in
0	-
1	ARG
2	ARG,ARG3
3	ARG,ARG2,ARG3
N	ARG3(list of the arguments)
F	ARG,ARG2,(arg. list and association-list)

Normal return is then done by setting IRES to the result value and then doing GO TO 998 (in fact, ARG and IRES are equivalent)

Here is an example:

The function lastelem[x] = if cdr[x]=NIL then car[x]else  
lastelem[cdr[x]]

is to be implemented.

a) In the data deck, change:

```
.  
. .  
1 INUNIT  
1 NULL  
. .  
. .  
. .
```

to

```
.  
. .  
1 INUNIT  
1 LASTELEM  
1 NULL  
. .  
. .
```

b) Change:

```
C * 9180, 9109, 9110, 9111, 9112, 9113, 9114, } Part of  
C CDDR CLEARB GENS INUN NULL NUMBP OUTUN } the com-  
GO TO
```

to

```
C * 9108, 9109, 9110, 9111, 9115, 9112, 9113, 9114,  
C CDDR CLEARB GENS INUN LASTE NULL NUMBP OUTUN
```

- c) Insert the following code somewhere in the main program (i.e. after the code for INUNIT)

```
C
C91115 LASTELEM
91115 IF(CDR(ARG)) 91116,91117,91116
91116 ARG=CDR(ARG)

GO TO 91115

91117 IRES=CAR(ARG)
GO TO 998
```

Warning: If you use CONS( , ), variables which earlier have been given a list as a value may afterwards be pointing to the wrong list structure in case of garbage-collection. The following lists are saved at a garbage-collection:

arguments to CONS

ARG,ARG2,ARG3,LASTR

the argument stack

all lists which are referenced by an atom

Example :

- a) TEMP1= CAR(ARG)  
IRES= CONS(I,J)

.  
.  
.

TEMP1 may now have been destroyed

- b) TEMP1 = CAR(ARG)  
TEMP2 = CONS(I,J)

.  
.  
.

TEMP1 may have been destroyed. TEMP2 cannot be destroyed by these statements

- c) TEMP1 = CAR(I)  
TEMP2 = CONS(TEMP1,J)

.  
.  
.

either TEMP1 or TEMP2 can be destroyed by these statements

12. References

1. John McCarthy et al.  
LISP 1.5 Manual  
MIT Press, 1962
2. Carl Weissman  
LISP 1.5 Primer  
Dickenson Publ. Co., 1968
3. Information International, Inc.  
The Programming Language LISP  
MIT Press, 1964
4. Daniel G. Bobrow et al.  
The BBN-LISP system  
Bolt, Beranek, and Newman, Inc., Cambridge, Mass.
5. Jaak Urmi  
3600 LISP U3 User's Manual  
Uppsala University, Computer Sciences Dept.
6. An Efficient Machine-Independent Procedure for Garbage  
Collection in Various List Structures  
(CACM, Aug. 1967, No. 8)

Appendix AThe LISP-F1 interpreter (defined in LISP M-expressions).

```

loop() = if mode=1 then print(eval(read()),NIL))
         elseif mode=2 then print(evalquote(read()),read() )
         elseif TEMP:= get(SYS,EXPR) then print(apply(TEMP,NIL,NIL ))
         else prog2(print(ERROR), print(eval(read()),NIL)) )

```

---

```

evalquote(fn,args) = if get(fn,FEXPR) V get(fn,FSUBR) then
                     eval(cons(fn,args), NIL)
                     else apply(fn,NIL,args)

```

---

```

apply(fn,a,args) = if null(fn) then NIL
                  elseif atom(fn) then
                    if TEMP := get(fn,EXPR) then apply(temp,a,args)
                    elseif TEMP := get(fn,SUBR) then mach(temp,a,args)
                    else apply(cassoc(fn,a,A2) , a, args)
                  elseif car(fn) = LABEL then
                    apply(caddr(fn), cons( cons(cadr(fn), caddr(fn)),a),args)
                  elseif car(fn) = FUNARG then
                    apply(cadr(fn), caddr(fn),args)
                  elseif car(fn) = LAMBDA then
                    eval(caddr(fn), addal(cadr(fn),args,a) )
                  else apply(eval(fn,a), a, args)

```

---

```

mach(le,a,args) = as apply but for SUBR's and FSUBR's

```

---

```

addal(a,b,c) = nconc(pair(cadr(fn),args),a)

```

For nconc and pair see the Lisp 1.5 manual.



```
eval(form,a) = if null(form) then NIL
               elseif numberp(form) then form
               elseif atom (form) then
                 if value(form) then car(form)
                 else cassoc(form,a,A8 )
               elseif atom(car(form)) then
                 if TEMP := get(car(form), EXPR) then
                   appyl(temp, a, evlis(cdr(form), a) )
                 elseif TEMP := get(car(form), FEXPR) then
                   appyl(temp, a, list(cdr(form),a) )

                 elseif TEMP := get(car(form), SUBR) then
                   mach(temp, a, evlis(cdr(form),a) )

                 elseif TEMP := get(car(form), FSUBR) then
                   mach(temp, a, cdr(form) )

                 else eval(cons(cassoc(car(form),a,A9),cdr(form)),a)
               else appyl(car(form),a,evlis(cdr(form),a) )
```

---

```
cassoc(atm,a,erf) =cdr(sassoc(atm,a,erf))
```

---

```
sassoc(atm,a,erf) = if null(a) then appyl(erf,a,cons(atm,cons(a,NIL)))
                   elseif caar(a) = atm then car(a)
                   else sassoc(atm,cdr(a),erf)
```

---

```
value(atm) = if car(atm) ="undefined" then NIL else car(atm)
```

---

Errorfunctions a2,a4,a6,a8 and a9

a2,a9    undefined function  
a4       first argument of set/setq not on the association list.  
a6       undefined label in prog  
a8       undefined variable

```
a2(atm,a) = prog()  
            print(ERROR A2)  
            print(atm)  
            return(break())
```

a4,a6,a8,a9 are analogue.

---

```
break() = prog()  
          if "next expression is in the form % S-expr"  
          then return (loop() ) else loop()
```

## Appendix B

### Table of atoms and functions defined in LISP F1

This appendix contains three tables:

- (1) a list of all functions which are defined in the system;
- (2) a list of all special-purpose atoms;
- (3) a list of I/O and character handling functions with more detailed descriptions of each function.

In table 1, the various columns use the following conventions:

value: the value is defined under the convention that the evaluated first argument to the function is called *x*, the second argument is called *y*, etc. (This convention is used even for FEXPR's, so that e.g. plus is said to have 3 arguments *x*, *y*, and *z* in the expression

(PLUS A B C)

and we let *x* stand for the value of A, rather than A itself. The association-list, or the list of arguments never get involved).

When nothing is said in this column, it means that we have nothing to say (not that the function does not have a value).

side-effects: When nothing is said in this column, it means that there are no side-effects.

7090 reference: This is a reference to pages in the LISP 1.5 manual (reference 1) where the function is further described.

reference here: This is a reference to sections (given as numbers) or appendices (given as letters) in this manual. B3 stands for table 3 in appendix B (i.e. this appendix).

Table B1

function name	type	nr of arg:s	value	side-effects	7090 reference here (page #)	reference here (section)
A2	SUBR	2		prints out error message and does a break		3.3
A4	SUBR	2		se A2		
A6	SUBR	2		- " -		
A8	SUBR	2		- " -		
A9	SUBR	2		- " -		
ADD1	SUBR	1	x+1		26,64	
ADVANCE	SUBR	0	next input character (as a character-atom)	advances input pointer; does <u>not</u> set CURCHAR or CHARCOUNT as in 7090 LISP. Note <u>The first char. following an S-expr will never be reached by advance</u>	88	B3
AND	FSUBR	arb			21,58	
APPLY	SUBR	3			70	
ATOM	SUBR	1			3,57	
BREAK	SUBR	0		The LISP-interpreter goes down in a break		4.1
CAR, CDR,...	SUBR	1	the following exist: CAR,CDR CAAR,CADR,CDAR, CDDR		2,3,56	
CLEARBIT	SUBR	1	NIL	clears the x'th bit in a simulated 24-bit register		5.6
COND	-	arb			18	5.3
CONS	SUBR	2			2,56	

Table B1

function name	type	nr of args	value	side-effects	7090 reference (page #)	reference here (section)
DE	FEXPR	3	x	defines new EXPR		5.6
DF	FEXPR	3	x	defines new FEXPR		5.6
DIFFERENCE	SUBR	2	x - y		26,64	
DUMP	SUBR	2	NIL	Dumps EXPR's and FEXPR's		5.6
EJECT	SUBR	0	NIL	page eject on print		B3
EQ	SUBR	2	T when x and y are same address, NIL otherwise		3,23,57	
EQUAL	SUBR	2			11,26,57	
EVAL	SUBR	2			71	
EVLIS	SUBR	2			71	
FUNCTION	FSUBR	1			21,71	
FORCIGBC	SUBR	0	number of free cells left (specified as a numeric atom)	causes a garbage collection		
GENSYM	SUBR	1	new atom with a fresh name (ABCD0000, XXX000001, etc.)	increases gensym counter to obtain name	66	5.6 <sup>a</sup>
GET	SUBR	2			41,59	
GO	FSUBR	1			30,72	
GOTO	SUBR	1		as GO, but the GO-label is the evaluated argument		5.6
GREATERP	SUBR	2	if x>y then T else NIL		26,64	
INUNIT	SUBR	1	x	switches standard input to be taken from logical unit x		8.0.4

Table B1

function name	type	nr of args	value	side-effects	7090 reference (page #/4)	reference here (section)
LABEL	FSUBR	2	y	binds x to y on the association-list	8,18,70	
LESSP	SUBR	2	if x<y then T else NIL		26,64	
LIST	FSUBR	arb			57	
MEMB	SUBR	2		defined with EQ		5.6
NUMBER	SUBR	2		defined with EQUAL	11,62	
MODE	SUBR	1	x	sets type of top-level function		3.1
NCNFC	SUBR	2	like APPEND	attaches y to the end of x with RPLACD type operation	62	
NULL	SUBR	1	if x=NIL then T else NIL		11,57	
NUMBERP	SUBR	1	if x is an numerical atom then T else NIL		26,64	
OBLIST	SUBR	1	an objectlist, starting with x			5.6
OR	FSUBR	arb			21,58	
OUTUNIT	SUBR	1	x	switches standard (print) output to be given to logical unit x		10.4
PACKLIST	SUBR	1	a new atom			5.6
PAIR	SUBR	2			60	

Table B1

function name	type	nr of args	value	side-effects	7090 reference (page #7)	reference here (section)
PEEK	SUBR	0	NIL	calls the routine TESTUT		4.3
PLUS	FSUBR	arb	$x+y+\dots$		25,63	
PRIN1	SUBR	1	x	puts x in output buffer	65,84	7.7
PRINT	SUBR	1	x	prints x and terminates record; destroys previous contents of output buffer	65,84	7.7
PROG	FSUBR		argument of RETURN. If there is no RETURN clause: cdr of the prog-expression itself		29,71	
PROGN	FSUBR	arb	value of the last argument	evals all arguments		5.6
PUT	SUBR	3	x	adds the property y under the indicator z on p's property-list		
QUOTE	FSUBR	1			10,22,71	
QUOTE ENT	SUBR	2	$x/y$		25,64	7.5
READ	SUBR	0	The S-expr. just read.		65,84	7.7
RESTART	SUBR	0	-	restarts the system		5.6
RETURN	SUBR	1	-	causes exit from a PROG  with x as value of the PROG expression	30,72	

Table B1

function name	type	nr of arg:s	value	side-effects	7090 reference here (page ##)	reference here (section)
RPLACA	SUBR	2	cons(y,cdr(x))	modifies x	41,58	
RPLACD	SUBR	2	cons(car(x),y)	modifies x	41,58	
SASSOC	SUBR	3			60	
SET	SUBR	2	y	changes the value of x	30,71	
SETBIT	SUBR	1	NIL	sets the x'th bit in the D register. Can be used e.g. to control tracing	9	5.6
SETQ	FSUBR	2	y	like SET, but the first argument is not evaluated	30,71	
SILENCE	SUBR	0	NIL	turns off system printout		5.6
SUBL	SUBR	1	x-1		26,64	
TALK	SUBR	0	NIL	turns on system printout after SILENCE		5.6
TERPRI	SUBR	0	NIL	writes present contents of output buffer as one record on printoutput	65,84	5.6, B3
TESTBIT	SUBR	1	state of x'th bit in a simulated 24-bits register (represented as T or NIL)			5.6
TIMES	FSUBR	arb	product of all arguments		26,64	
UNPACK	SUBR	1	list of letters in the printname of x (letters		87	5.6



Table B2 (LISP-CONSTANTS)

constant name	value	purpose	reference here (section)
NIL	NIL		
T	T		
FUNARG	-	used in eval. When (FUNCTION S) is used, (FUNARG S <u>al</u> ) is returned, where <u>al</u> is the current association-list.	
SYS	NIL or set by the user	if MODE=3 the value of SYS is the top-level function.	2.4,8.1,10.7.3
EXPR	-		
FEXPR	-		

TABLE B3: DETAILS OF I/O FUNCTIONS

(supplements table A1)

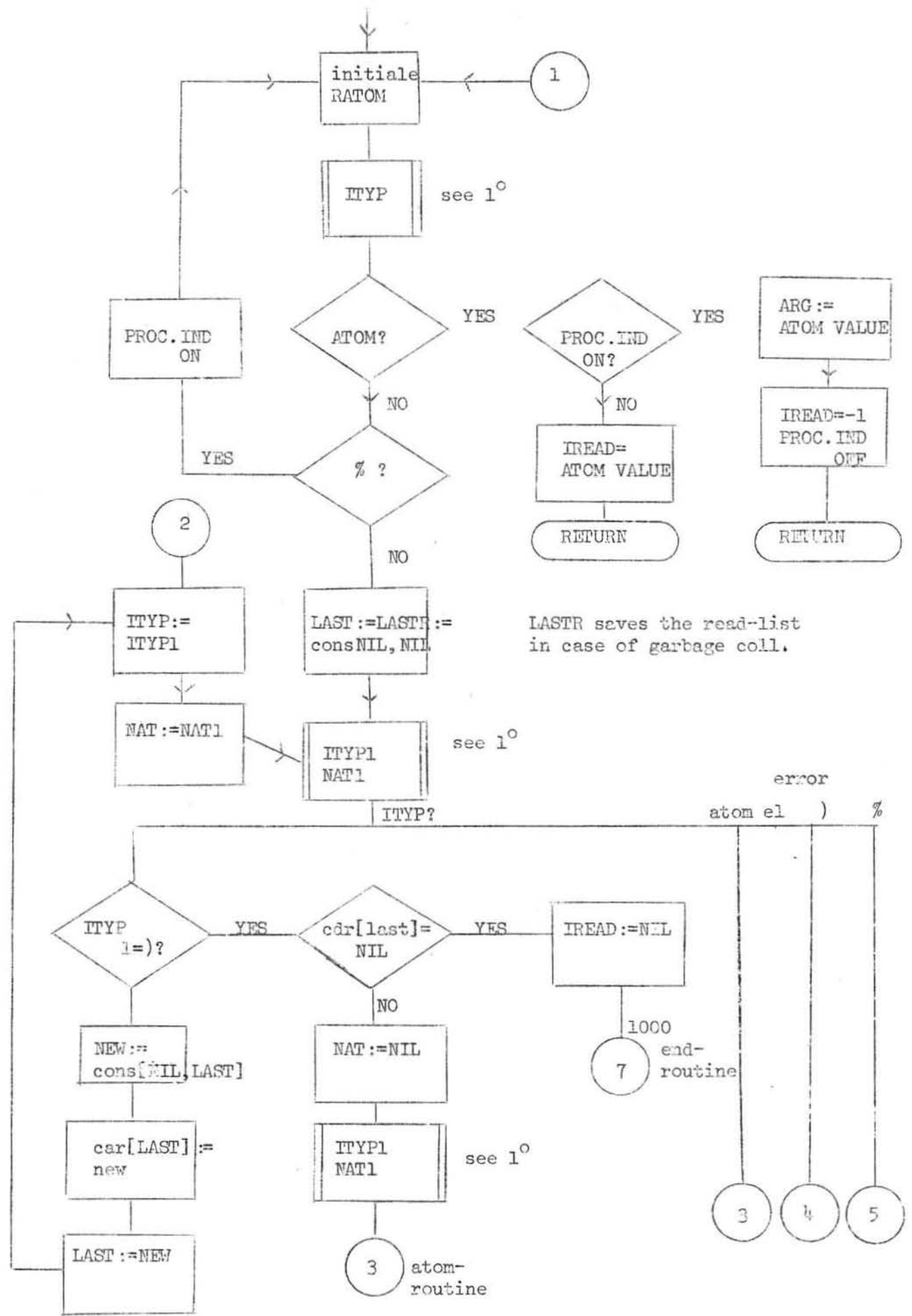
function name	nr of arg:s	argument(s)	side-effects
READ <sup>(x)</sup>	0		reads from standard input, 72 characters per record, until a correct parenthesis match is obtained. Replenishes the input buffer by reading a new record whenever necessary.
PRINT	1	arbitrary S-expression	converts x to a sequence of characters and puts it in the output buffer starting in position n (i.e. overwriting what may previously have been in the buffer). n depends on the depth of the list-structure, and causes the output; look like a pretty-printed output. Writes the buffer on standard output each time it fills up, and after operation is concluded.
ADVANCE <sup>(x)</sup>	0		reads one character from standard input and returns it as a character-atom
PRINL	1	alphameric atom	puts its argument in the output buffer. If this fills the buffer, it is printed on standard output as one record; and the buffer is cleared.,
TERPRI	0		writes the output buffer on standard output and clears the buffer.
EJECT	0		writes a page eject (a record with an l in the first position) on standard output. Does not affect the output buffer.

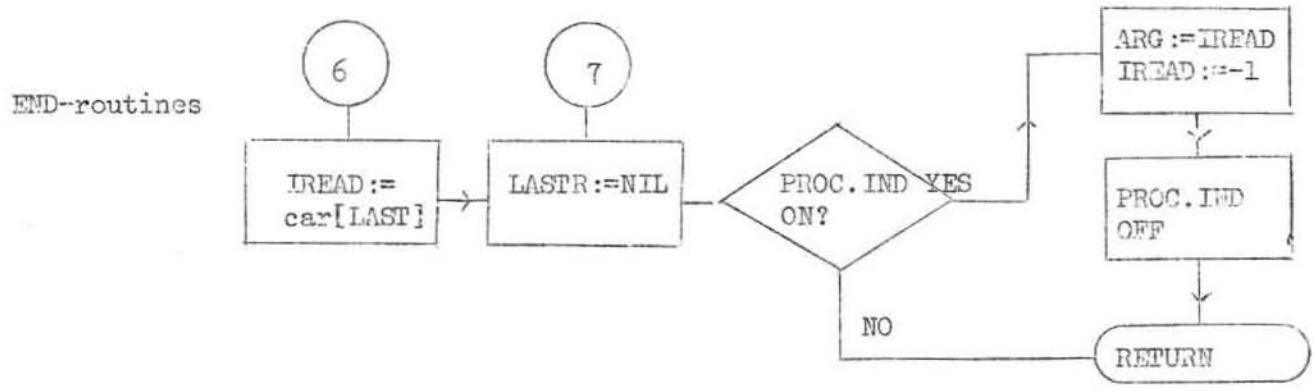
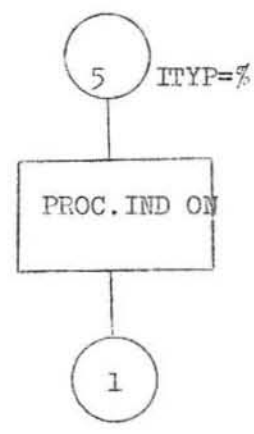
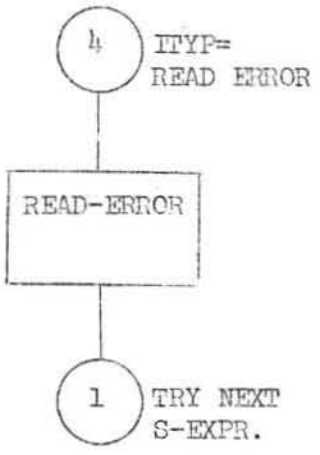
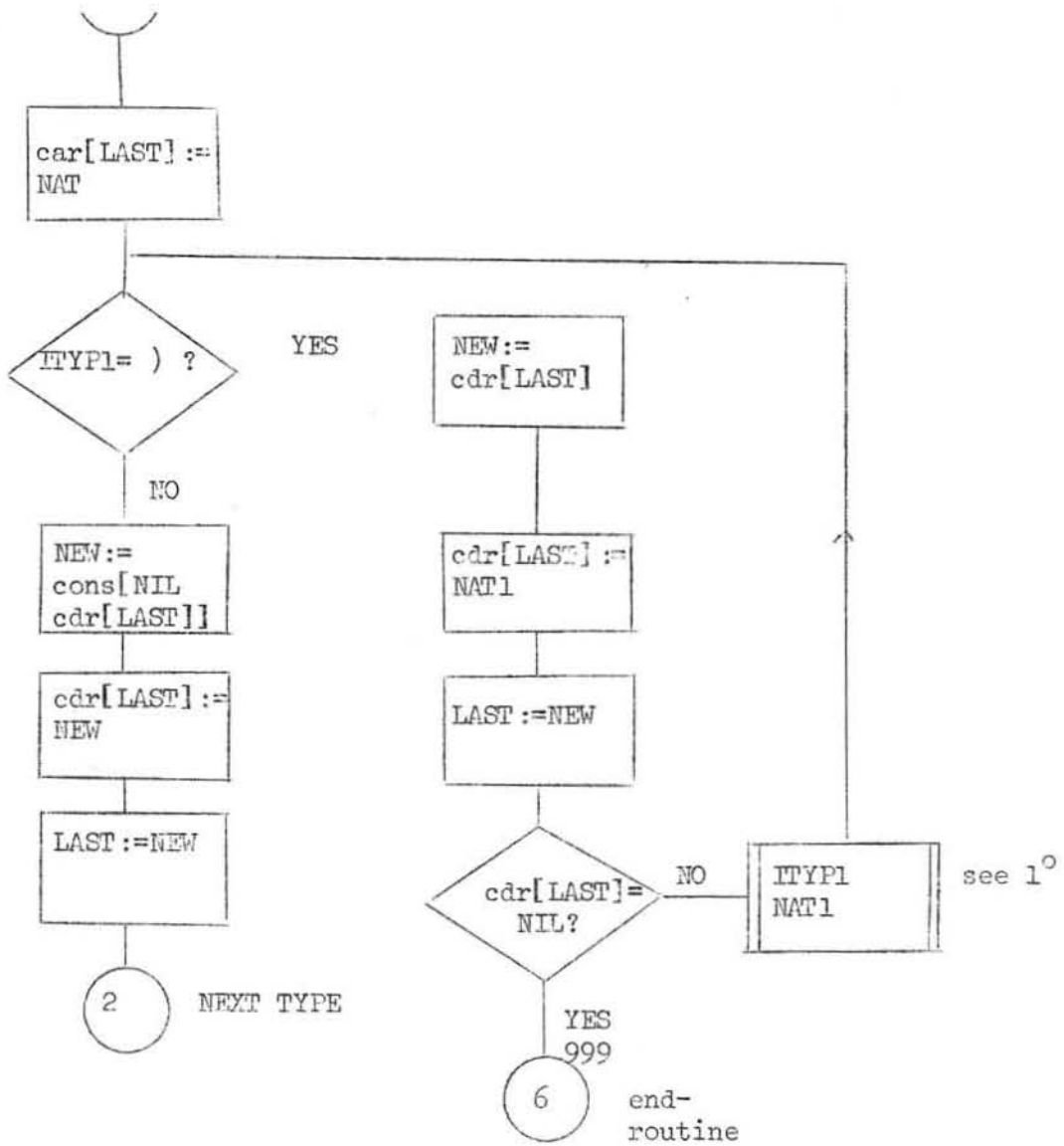
(x)

For the interface between READ, or eval reading of S-expressions on the one hand, and ADVANCE reading on the other, notice the following. READ and eval take in one S-expression plus one character each time they read (this extra character is buffered in a separate place). Therefore, one blank should be allowed between the final parenthesis or character that ADVANCE is to read. Example: the system obtains

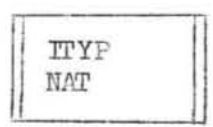
```
EVAL ( (CSETQ COMMA (ADVANCE)) NIL ) ,
```

There is one blank between parenthesis and the ",,".





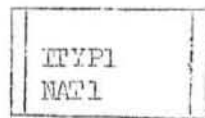
1°



means a call to RAEOM, which gives the type in ITYP and the atom value in NAT (in case of atom-read)

The code for ITYP is

- 1 means (
- 2 " )
- 3 " atom
- 4 " read error
- 5 " %

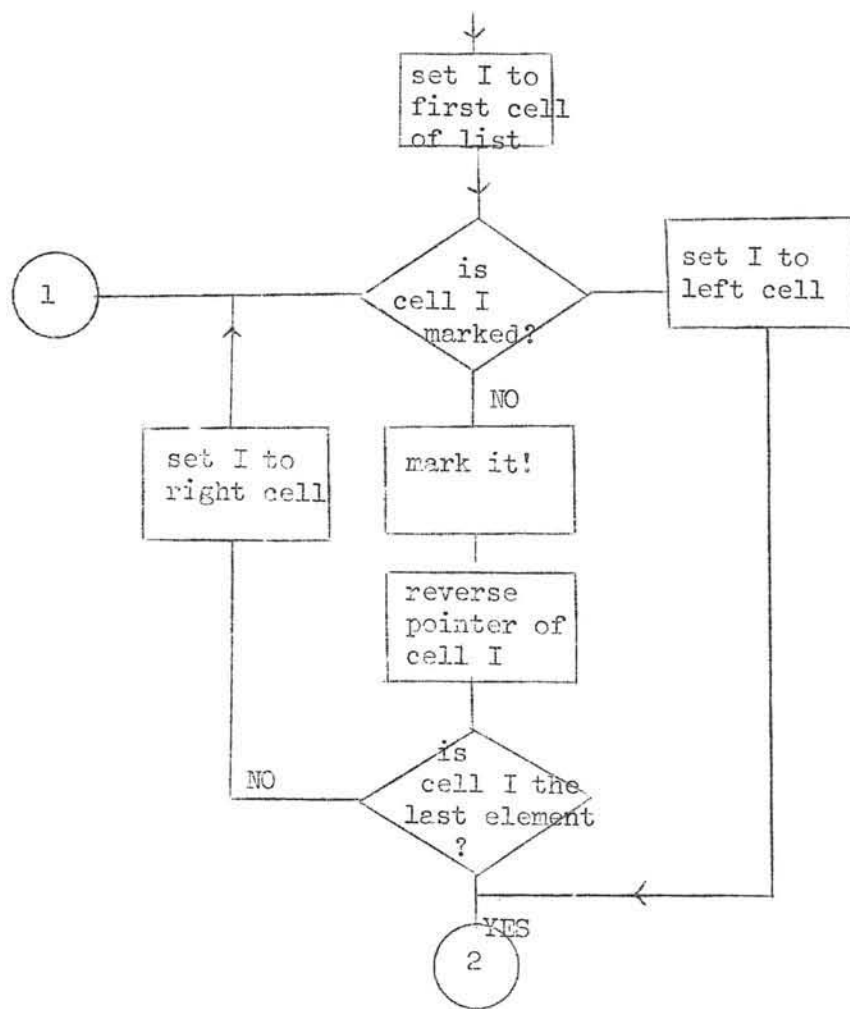


are analogous to ITYP + NAT

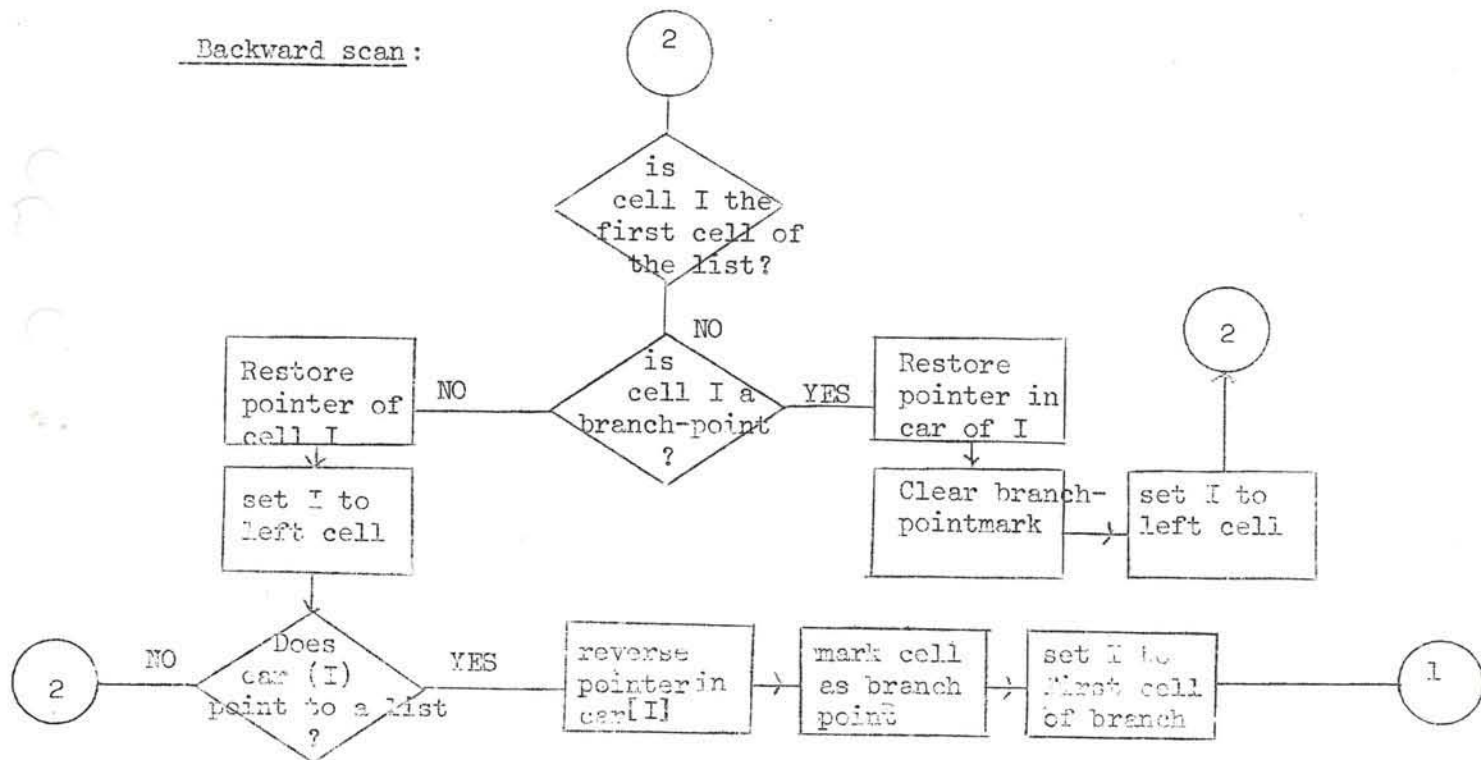
2°

.atom) is always treated as )  
 (which is the same as .NIL )

Forward scan:



Backward scan:



## Appendix D.

This deck should be read to logical unit 5 before running the system.

Note: The order is very important.

Col	12345678901
	012345678901
	012345678901
	0 NIL
	0 ADVANCE
	0 BREAK
	0 EFFECT
	0 MODE
	0 FORCEGBC
	0 PEEK
	0 OBLIST
	0 READ
	0 RESTART
	0 SILENCE
	0 TALK
	0 TERPRI
	0 EXIT
	1 ADD1
	1 ATOM
	1 CAR
	1 CDR
	1 CAAR
	1 CADR
	1 CDAR
	1 CDDR
	1 CLEARBIT
	1 GENSYM
	1 INUNIT
	1 NULL
	1 NUMBERP
	1 OUTUNIT
	1 PACKLIST
	1 PRIN1
	1 PRINT
	1 PRINTPOS
	1 READPOS
	1 RETURN
	1 SETBIT
	1 SUB1
	1 TESTBIT
	1 UNPACK
	1 ZEROP
	1 GOTO
	1 DUMP
	2 A2
	2 A8
	2 A9
	2 A4
	2 A6
	2 CONS
	2 DIFFERENCE
	2 EQ
	2 EQUAL

```

2 EVAL
2 EVLIS
2 GET
2 GREATERP
2 LEESP
2 MEMBER
2 NCONC
2 PAIR
2 QUOTIENT
2 RPLACA
2 RPLACD
2 SET
2 MEMB
3 APPLY
3 PUT
3 SASSOC
N LIST
N PLUS
N PROGN
N TIMES
F COND
F PROG
F AND
F DE
F DF
F FUNCTION
F GO
F OR
F QUOTE
F SETQ
- LAMBDA
- LABEL
- FUNARG
- SYS
- EXPR
- FEXPR
- T
( ) . , [ ] % + - 0 1 2 3 4 5 6 7 8 9 %

```

Note. The last card defines "system characters". The card shall contain in order:

```

one space character
" left par. character
" right par. character
" comma      "
" dot        "
" left bracket "
" right      " "
" character used for return from break (here %)
" plus sign character
" minus sign character
ten figures
one character used as end mark in printnames (here %)
four space characters

```



Appendix B

S-expression listing of the edit and trace package.

E1. The edit package

```
(DE KEDIT (F TT) (PROG (X Y CL CTL)
(SETQ CTL (LIST (GET F TT)))
(SETQ Y CTL)(SETQ CL CTL) A (SETQ X (READ))
(COND ((OR (ATOM X)(NUMBERP X))(GO B))
((ATOM (CAR CL))(PRINT ILLGC))
((NUMBERP (CAR X))(XCHANGE CL X))
(T (XJUST CL X))) (GO A) B (COND
((AND (ATOM (CAR CL))(NUMBERP X)(GREATERP X 0))
(PRINT ILLGC))
((EQ X (QUOTE OK))(GO C))
((EQ X (QUOTE UP))(SETQ CTL CL))
((EQ X (QUOTE P))(SPRINT (CAR CL) 2))
((EQ X (QUOTE PR))(SPRINT (CAR CL)(READ)))
((ZEROP X) (SETQ CL CTL))
((NUMBERP X)(COND ((NULL (SETQ X (NTH X (CAR CL))))
(PRINT ILLGC)) (T (SETQ CL X))))(T (PRINT ILLGC)))
(GO A) C (RETURN (PUT F (CAR Y) TT>
(DE SPRINT (L N) (COND ((OR (ATOM L)(ZEROP N))
(PRINT L))(T (PRINT (SPR (COPY L) 0 N>
(DE SPR (L N1 N2) (COND ((NULL L) NIL)
((ATOM L) L)((EQ N1 N2)(RPLACA (RPLACD L
(QUOTE xxx))(QUOTE xxx))) (T (CONS (SPR
(CAR L)(ADD1 N1) N2)(SPR(CDR L)N1 N2>
(DE XCHANGE (L AL) (PROG (X)
(COND ((GREATERP (CAR AL) 0) (COND ((NULL (SETQ
X (NTH (CAR AL)(CAR L))))(PRINT ILLGC))
(T (RPLACA L (APPEND (LISTN (SUB1 (CAR AL))(CAR L))
(APPEND (CDR AL)(CDR X))))))))))
((LESSP (CAR AL) 0)(COND ((NULL (SETQ X (NTH (MINUS
```

```

(CAR AL))(CAR L)))(PRINT ILLG))
(T (RPLACA L (A YEND (LISTN (SUBL (MINUS (CAR AL)))
(CAR L))(APPEND (CDR AL) X)
(DE ADJUST (L AL) (PROG (X) (COND
((NULL (SETQ X (NTH (CADR AL)(CAR L))))(PRINT ILLG))
((EQ (CAR AL)(QUOTE E0))(RETURN (RPLACA (RPLACD X
(CADR X))(CAAR X)))) ((EQ (CAR AL) (QUOTE LI))
(RETURN (RPLACD (RPLACA X (COPY X)) NIL)))
((EQ (CAR AL) (QUOTE R0))(RETURN (RPLACD(RPLACA X
(COPY (NTH0 (CAR X)(CDR X))) NIL)))
((AND (EQ (CADR AL) L)(EQ (CAR AL)(QUOTE RI)))
(SETQ X (CAR X))) (COND
((EQ (CAR AL)(QUOTE RI))(RETURN(RPLACA (RPLACD X
(COPY (NTH0 (CADR AL) (ADD1 (CAR (CDDR AL)))(CAR X))
(CDR X))))(RETURN (CAR (CDDR AL))(CAR X))))
((EQ (CAR AL)(QUOTE R0))(RETURN (RPLACA (RPLACD
X (COPY (NTH0 (CADR X) (CDR X))) (CAAR X))))
((EQ (CAR AL)(QUOTE E1))(RETURN (RPLACD (RPLACA
X (LISTN (ADD1 (DIFFERENCE (CAR (CDDR AL))
(CADR AL))) X) (NTH (ADD1 (CAR (CDDR AL))
(CAR L)
(DE EDIT (U V) (COND
((GET (CAR U)(QUOTE EXPR))(EDIT (CAR U)(QUOTE EXPR)))
((GET(CAR U)(QUOTE FEXPR))(EDIT(CAR U)(QUOTE FEXPR)))
((GET(CAR U)(QUOTE SUBR))(EDIT(CAR U)(QUOTE SUBR)))
((GET(CAR U)(QUOTE FSUBR))(EDIT(CAR U)(QUOTE FSUBR)))>
(RPLACA (QUOTE ILLG)(QUOTE
>
(DE NTH (N L) (COND ((LESSP N 2)L
((NULL L) NIL)
(T (NTH (SUBL N)(CDR L))))))
(DE LISTN (N L) (COND ((ZEROP N)NIL)
((NULL L)NIL)
(T (CONS (CAR L) (LISTN (SUBL N)(CDR L))))))
(DE COPY (L) (COND ((NULL L) NIL)
((ATOM L) L)
(T (CONS (COPY (CAR L))(COPY (CDR L))))))
(DE PLUS (X) (PLUS X -1))
(DE APPEND (L1 L2)
(COND ((NULL L1) L2)
(T (CONS (CAR (CAR L1))(APPEND (CDR L1) L2))))))
(FUNDEF 9)

```

## E2. The trace package

```

- (DE DMAP(L TRFN) (PROG (X Y)
  LIP (SETQ X (QUOTE (EXPR FEXPR SUBR FSUBR)))
  LOP (COND ((GET (CAR L)(CAR X)) (TRFN (CAR L)(CAR X)))
    (T (PROGN (SETQ X (CDR X)) (GO LOP))))
  (COND ((NULL (SETQ L (CDR L)))(RETURN (QUOTE OK)))
    (T (GO LIP))))))
- (DF TRACE (FL AL) (DMAP FL (FUNCTION TRC1)))
- (DF UNTRACE (FL AL) (DMAP FL (FUNCTION UTRC1)))
- (DE TRC1 (F TT) (PROG (X Y)
  (SETQ X (GET F TT))
  (SETQ Y (GENSYM (QUOTE GGGG)))
  (PUT Y (COPY X)(QUOTE EXPR))
  (PUT F Y (QUOTE TRFL))
  (RETURN (RPLACD (CDR X) (LIST(NCONC(QUOTE x (PROGN
    (PRIN2 INT) (PRIN2 (QUOTE (x.F))))))
    (NCONC (FLIST (CADR X)) (LIST
      (QUOTE x (OUTP (x.F)(x.Y)(x CADR X)>
- (DE FLIST (L) (COND
  ((NULL L) (LIST (QUOTE (TERPRI))))
  (T (NCONC (LIST (QUOTE x (PRIN2 (QUOTE (x CAR L))))
    (QUOTE x (PRIN2 (QUOTE =)))
    (QUOTE x (PRINL (x CAR L)))
    (QUOTE x (PRIN2 SPC))))(FLIST (CDR L)>
- (DF QUOTE x (U V) (UOT (CAR U)
- (DE UOT (X) (COND
  ((ATOM X) X)
  ((EQ (CAR X) (QUOTE x))(EVAL (CDR X) V))
  (T (CONS (UOT (CAR X))(UOT (CDR X)>
- (DE PRINL (L) (COND
  ((ATOM L) (PRIN2 L))
  (T (RPOGN (PRIN1 LPAR)(MAPCAR1 L (FUNCTION PRINL))
    (PRIN1 RPAR>
- (DE MAPCAR1 (L F) (COND ((NULL L) NIL)
  ((ATOM L) (PROGN (PRIN1 DOT)(F L)))
  (T (CONS (F (CAR L))(MAPCAR1 (CDR L) F)>
- (DF OUTP (U V) (PROG (X)
  (SETQ X (EVAL (CONS (CADR U)(CAR (CDDR U)))) V))
  (PRIN2 UNT)(PRIN2 (CAR U))
  (PRINL X) (TERPRI) (RETURN X)>
- (DE UTRC1 (F TT) (PROG (X)
  (COND ((NULL (SETQ X (GET F(QUOTE TRFL))))(RETURN F)))
  (RETURN (PUT F (GET X (QUOTE EXPR)) TT))))))
- (DE COPY (L) (COND ((NULL L) NIL)((ATOM L) L)
  (T (CONS (COPY (CAR L))(COPY (CDR L))))))
- (DE PRIN2 (X) (PROGN (PRIN1 X)(PRIN1 BLANK>
  (RPLACA (QUOTE LPAR)(ADVANCE> (
  (RPLACA (QUOTE RPAR) (ADVANCE> )
  (RPLACA (QUOTE SPC) (ADVANCE> .
  (RPLACA (QUOTE BLANK)(ADVANCE>
  (RPLACA (QUOTE INT)(PACKLIST (LIST
  (ADVANCE)(ADVANCE)(ADVANCE)> -->
  (RPLACA (QUOTE UNT)(PACKLIST (LIST
  (ADVANCE)(ADVANCE)(ADVANCE)> <--
  (RPLACA (QUOTE DOT)(ADVANCE> .
  (INUNIT 9)

```