DESIGN AND DEVELOPMENT DOCUMENT FOR LISP ON SEVERAL S/360 OPERATING SYSTEMS

YORKTOWN HEIGHTS, NEW YORK - 10598
TELEPHONE:914-945-3000

FRED W. BLAIR
JAMES H. GRIESMER
JOSEPH HARRY
MARK PIVOVONSKY

Revised June 24, 1971

CONTENTS

ACKNOWLEDGEMENT

In writing this document the authors have relied heavily on the material that appeared in other documents on the LISP programming language and on implementations of LISP. In particular, the system description rests heavily on the following documents:

1. McCarthy, John, et al.
   LISP 1.5 Programmers Manual
   The M.I.T. Press,
   CAMBRIDGE., 1962

2. Edmund C. Berkely and Daniel G. Bobrow
   The Lisp Programming Language:
   Its Operation and Applications
   The M.I.T. Press,
   CAMBRIDGE., 1964

3. LISP 1.5 Reference Manual for Q-32
   S. L. Kameny
   S.D.C. (TM Series TM-2337/101/00)
   Santa Monica., 1965

4. The BBN 940 LISP System
   D. G. Bobrow, D.L. Murphy, W. Teitelman
   Bolt Beranek and Newman Inc.

   } INTERLISP Reference Manual
   Warren Teitleman
   Xerox Corp. and BBN Inc.

5. Various documents describing the specification
   and implementation of LISP 2.

6. The M44/44X LISP System.
   Private documents by the author:
   Paul Abrahams
   Courant Institute,
   New York Univ., 1966

In addition, many of the functions present in the system are due in part to these previous implementations and LISP persons too numerous to mention.

INTRODUCTION

This manual is intended for those persons familar with the LISP programming language who wish to refer to details of definition and implementation peculiar to this system.

The beginning LISP user is directed to the "LISP 1.5 Primer" by Clark Weisman (Dickinson Publishing Co., Inc., Belmont, Calf., 1967), or "LISP 1.5 Programmer's Manual" by John McCarthy, et al.

The conventions for the specification of syntax are given in APPENDIX F.

As a guide to the user it is suggested that Chapter X. be read first and Chapter II. read only when required for actually getting on the machine.

II. USING LISP ON OS/360, TSS/360, AND CP/CMS.

This chapter is best read after Chapter X if the reader is reading from begining to end.

2.1 General comments on running LISP:

Normally LISP runs are done in the following manner:

A. A LISP module is loaded and given control in one of the following modes:

hot-start: This is the users mode of operation. Major parts of the system (those generated by the core-image generation process) are read into a large area reserved by the LISP module, from a dumped core-image (SOSTAP).

As the dumped core-image was written out in a relativised form, the hot-start (or Bootloader) derelativises. This process yields the state of the system at the time of dumping. Thus if the system was dumped by evaluating the doublet:

```
FILELISP( ... )
```

the hot-start will cause execution to continue with the return from FILELISP in which case,
VALUE = gener (where gener is the current generation number) will be printed, and *SUPV the main LISP supervisor, will continue on to the next doublet.

The hot-start process causes all files which were opened at time of dumping to be reopened. This implies that any system dependent file definition requirements must have been met before control is given to the LISP module.

cold-start: In this mode, which is used for system generation, LISP loads the core-image-generated parts of the LISP system from a file of patch-cards (SOSCOR). The core-image is then dumped to form a binary core-image file (SOSTAP) for future hot-starts. The state of this dumped system is such that upon loading, the system input and the system output files (LISPIT and LISPOT) are opened and *SUPERMAN (the overlord of *SUPV) is executed. *SUPERMAN supplies the input and output filenames for *SUPV and exits only when *SUPV does an exit RETURN upon encountering a FIN doublet.

B. The checkpoint (FILELISP ...) facility:

The FILELISP function which creats a binary core-image

file, provides the user with the ability to capture the state of the LISP system on a temporary file. This file may then be used as SOSTAP in the hot-start mode. The arguments of FILELISP, shown above as "..." vary with the system that LISP is run on. On OS/360 there is one argument, a ddname (usually SOSTMP). On CP/CMS there are three arguments: filename, filetype and filemode.

2.2 Running LISP on OS/360:

In order to run LISP on OS/360, the correct Job Control Language (JCL) cards must be supplied. Those who are unfamiliar with JCL should seek expert help. The following JCL cards may be used to run LISP at the IBM YORKTOWN HTS RESEARCH CTR. on the S/360-91 with MVT/LASP. (note: words written in lower case are subject to change, the local LISP expert will normally supply an up-to-date set of these cards)

```
//jobnam JOB ...
/*FORMAT PR,CONTROL=SINGLE,DDNAME=LISPOT
//stepnm EXEC PGM=LISPLDR,PARM=W,REGION=396K
//STEPLIB DD DISP=SHR,UNIT=2314,VOL=SER=555555,DSN=LISPLDR.G0115
//SOSTAP DD DSNAME=LISPTAP.G0115,
//        UNIT=2314,VOLUME=SER=555555,DISP=(SHR,KEEP),
//        DCB=(RECFM=U,BLKSIZE=1024)
//SOSLLF DD UNIT=2314,SPACE=(CYL,(1,1))
//SOSTMP DD DSNAME=yourfil,
//        DCB=(BLKSIZE=1024,RECFM=U),
//        VOLUME=(PRIVATE,RETAIN,SER=yourdisk),
//        UNIT=2314,DISP=(NEW,KEEP),SPACE=(CYL,(3,1))
//SYSUDUMP DD SYSOUT=A
//LISPOT DD SYSOUT=A,DCB=(BLKSIZE=120,RECFM=F,LRECL=120)
//LISPIT DD *

 Doublets, i.e. function ( args* )
FIN             ))))))))))))))))))))))
FIN FIN FIN
 /*
```

The above JCL illustrate the hot-start mode setup.

2.3 Running LISP on TSS:

Note: Because of low interest, this Section has not been updated, in any revision later than MAY 18, 1971.

In order to run LISP on TSS/360, the user must have some familiarity with the TSS Command Language and terminal operating conventions. For those who are unwilling to study TSS by reading the TSS/360 System Reference Library, the best alternative is the TSS/360 Quick Guide and the tutelage

of an expert user. The user is particularly encouraged to learn to use one of the available data set editors, he must also be aware of the Data Management Commands.

A. The method for acquiring LISP that is currently popular is as follows:

1. The local LISP expert "PERMITs" his latest core-image and LISP module.
2. The user shares these data sets.
   share lis,blair,cor146
   share lib,blair,userlib
3. The user moves the LISP module into his USERLIB.
   ddef fromlib,vp,lib
   ddef tolib,vp,userlib
   movprg
   when the system prompts for the module name:
   atnlis8
4. The user makes his own copy of cor146:
   vv lis,cor146
5. The user deletes his references to the shared data sets:
   delete lis
   delete lib

B. The user can then hot-start the acquired system by:

```
ddef sostap,vs,cor146
lisp
```

which will put him in the LISP supervisor with his keyboard unlocked awaiting the next doublet. The user may exit by typing:

```
FIN
```

Caution: The LISP module is not reentrant,therefore, the user must unload or abend before repeating the above hot-start procedure.

C. The LISP attention conventions:

It is possible to interrupt LISP while it is running. This is done by hitting the ATTN key (in the case of a 2741 terminal), in which case LISP will respond:

```
%
```

the user may then type:

C in which case he will go to the Command Language mode. (typing GO should then return him to LISP.)

U an UNWIND is caused. This will get the user out of most severe problems.

E the ERROR routine is executed.

S the supervisor *SUPV is called. This may be used to turn on function tracing or simply to fix up global parameters. A FIN will cause execution to resume from the point of interruption.

D. The OBEY function: Obey is a LISP function of

one argument, that argument being a LISP string datum, which contaims any legitimate TSS/360 command; the effect of the function is to operate that command. The value of OBEY is GO_ON. Below is illustrated the use of OBEY to make the data set characteristics, of a file named sostmp, known to the TSS/360 system:

```
obey(#ddef sostmp,vs,dsname,dcb=(lrecl=1024,recfm=f)#)
```

2.4 Running LISP on CP/CMS:

The following describes the operation of LISP on CMS. The current method allows any user who has at least 512K of virtual memory to use LISP interactively or it allows any user to use it in the CMS Batch.

1. The interactive LISP command:

```
LISP  cor      vdisk mode
      LISP115  193   B
```

where: cor --- is the filename of the LISP system image that you wish to use, or it is the default, LISP115 the standard LISP system.

Note: All such "core images" are the result of some previous checkpoint of the LISP system. Thus, running LISP can be thought of as a restart. In operations which concern core image files the filetype "SOSTAP" is assumed unless otherwise stated.

vdisk --- is the virtual disk address that you wish LISP to use for a read only extension to your P-disk. If you have no disks currently attached then the default disk 193 should work.

mode --- specifies the filemode (A,B,C,T) you wish the extension to have. The default is mode B.

EXAMPLE: LISP
In this case the user simply wishes to run the standard LISP system, and is unconcerned about what disks and modes are used.

The following lines will be printed:

```
** B (193) READ-ONLY **
EXECUTION BEGINS...
LISP VERSION 115-0
CORE IMAGE:  LISP115  SOSTAP  P1
VALUE = 1
```

at which time your keyboard will unlock and you begin to enter doublets to the *SUPV supervisor which calls EVALQUOTE to evaluate each doublet. Note: The

supervisor reads two s-expressions and APPLY's the first to the second.

It continues to read the doublet until all parentheses are balanced in the second S-expression, then skips the rest of the input record.
You may escape from the LISP supervisor by typing FIN.

At which time the following printout will occur:

```
END_OF_SUPV
END_OF_LISP
ERRORCNT = n
DEV  193 DETACHED
```

The time message is printed and you are returned to CMS.

EXAMPLE: LISP 194
This case is the same as above except that the user wants to use 194 as the disk extension address, probably because he already has a 193 attached.

EXAMPLE: LISP 194 A
The user already has 193 logged in as B, therefore he wishes LISP to use 194 as A.

For the benefit of system programmers at other installations: A listing of the LISP EXEC file can be found in APPENDIX E.

Note: The LISP command links the user to the files of the user BLAIR1 (in the case of our installation) for those files whose filenames are LISP115. The files that user BLAIR1 must must have allowed to be readshared are:
A. LISP115 SOSTAP P1 --- the LISP system standard core image.
B. LISP115 TEXT P1 --- the standard module.
C. LISP115 EXEC P1 --- an exec file consisting of the following:

```
&TYPEOUT OFF          (turns off CMS command typeout)
CHARDEF T             (no special character for tab)
CHARDEF B             (no special character for backspace)
LINEND                (carrier return is the only line end)
LOAD LISP115
START LISPHOT &1 &2 &3
```

2. The Batch LISP command:

```
LISPOFF  in ou  cor     sav
            out LISP115 NO
```

where: in --- is the filename of an input file whose filetype is SYSIN and which contains a sequence of LISP doublets terminated by a FIN card. There is no default for this parameter, it must be supplied.

ou --- is a filename for the output. This file will be created with filetype OUT and will be transmitted by Batch back to the user. A DISK LOAD will be required to put the file on the users p-disk. The default OUT for this parameter causes an OFFLINE PRINT OUT OUT to be executed by Batch and the output is not transmitted back to the user.

cor --- specifies the core image from which the user wishes to restart. The default LISP115 is the basic LISP system.

sav --- is the filename for a checkpoint core image which will be saved at the end of the run. The file "sav SOSTAP P1" will be created and transmitted to the user. The default NO assumes that the user does not wish to save the core image at the end of his run.

Note: As is generally true for the user of Batch, the user must have a file containing: his CP userid followed by at least one blank followed by his account number followed by at least one blank. This file must have the filename userid and the filetype BATCH.

For the benefit of systems programmers at other installations:

A. Contact C. J. Stephenson, IBM Research Ctr., Yorktown Heights, about the CMS Batch facility.

B. The file LISPOFF EXEC can be found in APPENDIX E.

C. The file LISPBCH BATCH is required and is also found in APPENDIX E.

3. CMS attention conventions:

For the users of LISP at IBM Research attention interrupts are created by the named interrupt facility. It is invoked by attentioning twice, typing ZZ, and carrier return. This facility is due to W E Daniels of the Research center and interested parties should contact him. For other installations the external interrupt feature of CMS must be used. This means that one no longer ends up in DEBUG and that one enters the LISP interrupt idler instead.

The LISP interrupt handler only signals that an interrupt has occurred. The LISP interrupt services routine is given control on the next function call. It is possible that a program is in so tight a loop that there is no "next" call! In this rare circumstance the user has no recourse but to re-ipl.

The LISP interrupt service routine when called, responds by typing "%". (The response may be delayed by CMS stacked output.) If the user then hits carrier return, he will be prompted as to what he may request. If the user is familiar with the attention requests he may type the desired request immediately, in which case there is no prompting.

The available requests are as follows:

C Puts the user in CMS subset command language.

U Unwind to the last errorstop.
Has the effect of canceling the last operation.

E Call ERROR for backtrace and diagnostics.

S Recursively reenter the LISP supervisor *SUPERMAN.
A subsequent FIN causes the interrupted program to resume.

D Places the user in CMS DEBUG.
A RET causes him to resume.

P To ignore interrupt and proceed.

III. DATA, TYPES AND TOKENS

3.1 Terminology

Field: An area, container, or box capable of holding information. For example, a computer word.

Setting: The contents of a field.

Locator: A setting from which a field or collection of fields may be found. (e.g., an address constant).

Pointer: A 32-bit field whose 24 low order bits are a locator. Pointers are byte addresses.

Half-word word-locator: A 16-bit field on a half-word boundary from which a pointer may be determined. (Note: In this system, the list nodes are pairs of half-word word-locators, not pointers. However, the half-word word-locators are always translated into corresponding pointer values by CAR and CDR.)

The LISP Language deals with data objects. The external representation of an object will be called a datum. A datum is usually a sequence of characters but it could conceivably be a sequence of sounds or pictures. A data structure is the representation of an object within the LISP system as a collection of fields with settings. A value is a setting that is either the locator of a data structure or is itself a data structure. In the latter case the value is said to be literal.

3.2 DATA

It is the LISP READ and PRINT functions which provide the interface capable of recognizing the data and transforming them into data structures.

A complete description of a datum in the sense of the READ and PRINT functions can be found in APPENDIX F.

The functionS (RDCHR) and (PRINTCH x) provide an interface by which any sequence of characters may be read and printed without regard to that syntax.

The functions (*NEWLIN) and (TERPRI) allow the reception and transmission of arbitrary records from/to currently selected input and output devices.

3.3 DATA STRUCTURES

As should be expected, data structures show a strong correspondence to data. This correspondence is not, however, one to one. For example, integers are represented in three ways; one for $-1024 \leq \text{integer} < 1023$; the other for all other positive and negative integers in the interval

$$-2^{31} \leq \text{integer} \leq 2^{31} - 1 \quad ,$$

and yet another for positive and negative integers outside this range but less than 9000 decimal digits.

It is therefore convenient to classify data structures on a somewhat different basis than in terms of their data class. Principally, this is by their storage maintenance properties.

A. Identifier structures

B. Small integers

C. Fixed size fields which contain no locators

   1. Integers
   2. Floating point reals

D. Contiguous storage vectors

   1. Those which contain no locators
      a. strings
      b. assembled programs
      c. integer vectors
      d. real vectors
      e. large integers
   2. Those which contain locators
      a. general vectors
      b. n-tuples

E. Nodes

A. IDENTIFIER STRUCTURES

Identifier data are represented uniquely by identifier data structures, which contain the information commonly found on description lists in certain other implementations.

The identifier data structure consists of three one-computer-word subfields. These three words are not contiguous but there is a rule for finding any two words given the third. It will be

convenient to refer to these as Wd-1 or the id-head, Wd-2, and Wd-3 or the value-cell.

The id-head contains 2 half-word word-locators named b-list and proplist, respectively.

The left hand locator (b-list) serves to locate the next identifier in the bucket chain. A pointer vector of buckets OBLIST serves to provide a mechanism for mapping an identifier datum into a unique id-head.

proplist is the value of the property list associated with the identifier.

proplist = (CDR id)

Wd 2 consists of 4 fields, namely:

1. type - this 8-bit field contains the following bit-field settings:

Bit 0 = No meaningful setting as yet reserved.

1 = No meaningful setting as yet reserved.

2 = 1 - if this identifier is a literal-real valued function.

3 = 1 - if this identifier is a free variable or function name, referenced by compiled code.

4 = 1 - if this identifier is the name of a function which returns a literal value.

5 = 1 - if this identifier is the name of a function which has an indefinite number of arguments.

6 = 1 - if this identifier is the name of a function which receives its arguments unevaluated.

7 = 1 - if this identifier is a compiled function name.

2. n-l-arg - this 4-bit field contains the number of literal arguments that this function expects, otherwise 0.

3. n-p-arg - this 4-bit field contains the number of pointer arguments that this function expects, otherwise 0.

4. pname - this half-word word-locator designates the string data structure that contains the EBCDIC representation of the identifiers datum.

Wd 3 - This field is a pointer whose high order 8 bit subfield is non-zero only if this identifier is the name of a function which has been called since the last garbage collection. The 24 low order bits contain a pointer value which is either the value associated with this identifier (the so called SPECIAL value) or is the locator of the assembled program associated with this identifier.

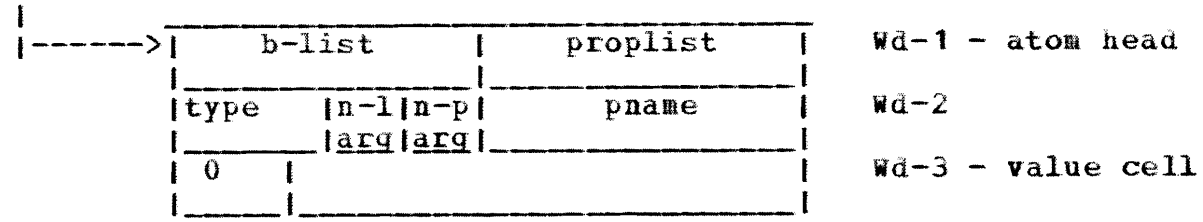


Figure 4. The identifier structure.

See APPENDIX A for functions on identifiers.

B. SMALL INTEGERS

Any pointer value which is a locator of any byte below the space reserved for large integers and above that reserved for stacks, is treated as a small integer by the arithmetic functions, and also by Read and Print. The literal value of the small integer is computed by subtracting QUOTE, and shifting right by 2. (See Fig. 5)

Small integers are thus unique and unalterable. As you might have guessed, small integers are an implementor's trick for consuming time instead of space, avoiding garbage collections, and thus saving time.

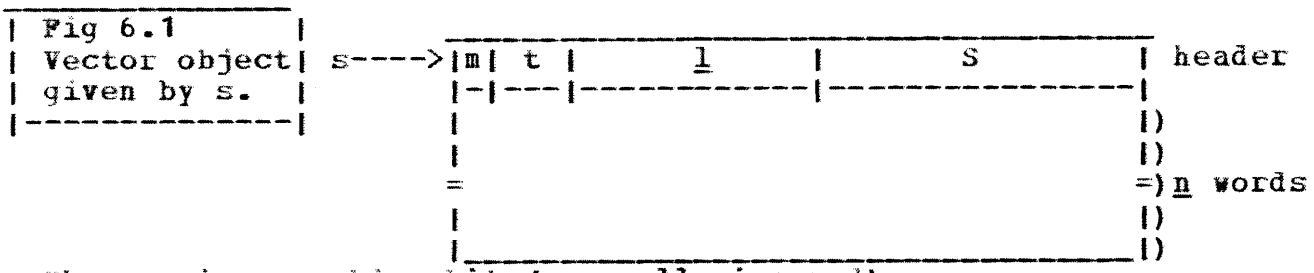
C. FIXED SIZE FIELDS WHICH CONTAIN NO LOCATORS

Two regions of memory are allocated to hold the numeric values of active single-precision integers and floating point reals. Equal numeric values may occur more than once, that is they are not unique. Unlike other programming systems these cells are never intentionally altered in place.

In both these regions, inactive values are reclaimable, i.e., garbage collectable.

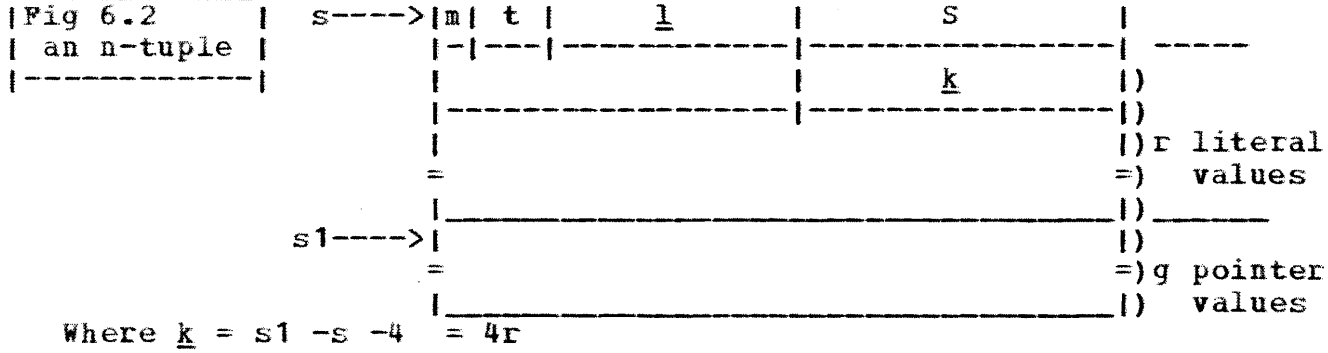
D. CONTIGUOUS STORAGE VECTORS A region of memory bounded in the low address region by the location FSTVEC and on the high end by available space, holds the data structures called vectors. (See Fig. 6) Vectors are not unique and except for assembled

programs, they are alterable.



Where m is a marking bit (generally ignored).
t is a 3 bit type code:
and t = 0 If s is a character string;
t = 1 If s is an assembled program;
t = 2 If s is a vector of literal integers;
t = 3 If s is a vector of literal reals;
t = 4 If s is a large integers;
t = 5 Unassigned;
t = 6 If s is an n-tuple;
t = 7 If s is a vector of pointer values.
l = 4n - 1 (0 < l < 4096) , where n is the vector length in words (not counting the header).
S is the half-word locator equivalent of s.

The header word must never be altered by the user (Note: the system does not prohibit such alterations, therefore this convention is in the hands of the user).

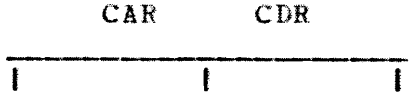


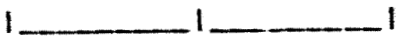
Where k = s1 -s -4 = 4r

G. NODES

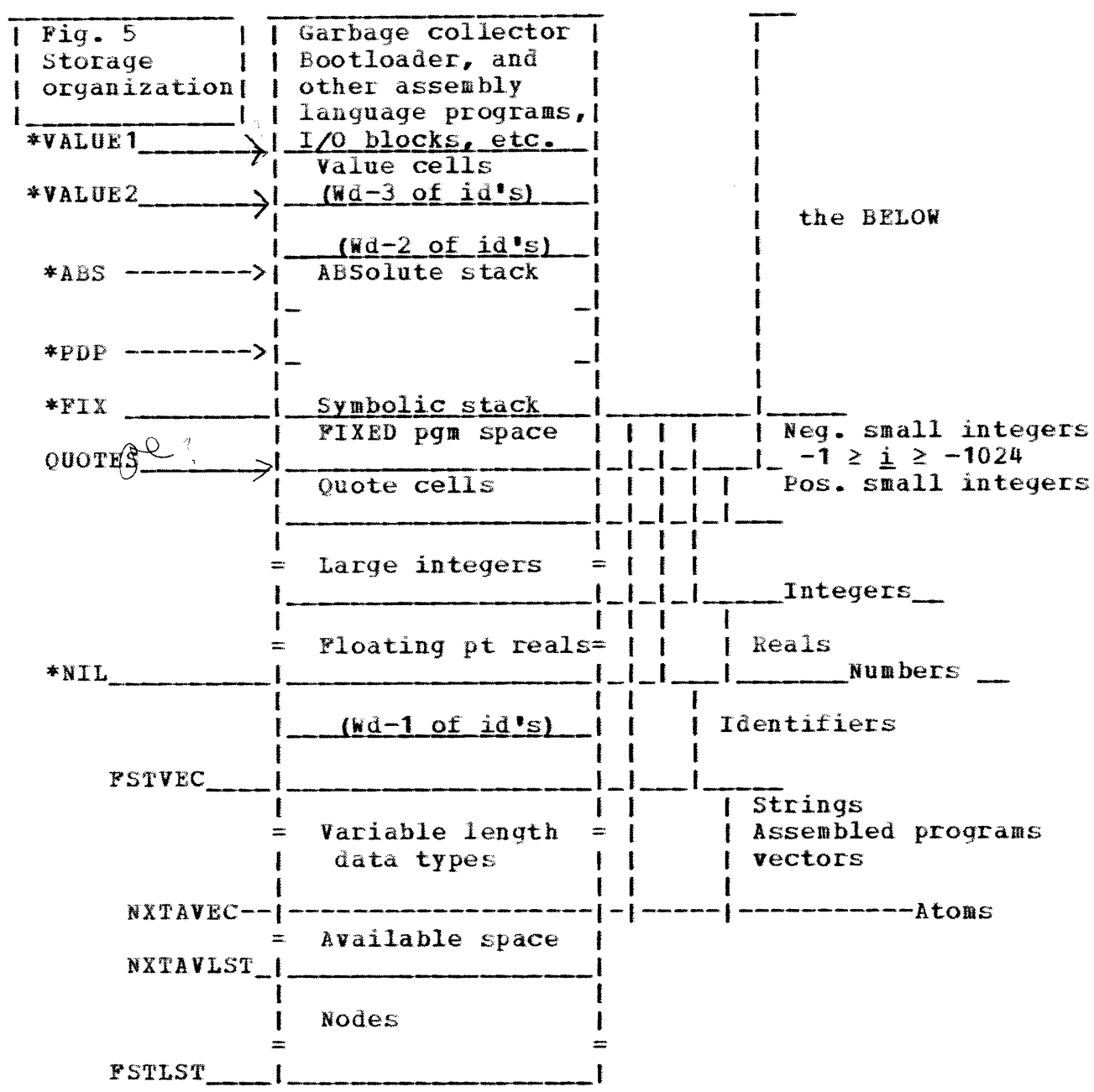
Nodes, commonly called list nodes or list cells, are one word fields consisting of two half-word word-locators. The CAR field is the left-most or lower addressed one. Let it be emphasized again that the value of the function CAR is a pointer, not a half-word word-locator.

Once again the implementor chose to consider space as the more valuable resource.





IV. STORAGE ORGANIZATION



In this implementation a schema for using the CPU Registers and particular layout of core memory have been chosen. This we call the storage organization (See Fig. 5)

Because half-word word-locators are used in nodes, only 2**16 words may be located beyond the address given in *FIX. Any word within this range is said to be Inside. The stacks as shown in Fig. 5 are Below, and certain other features of the system are also Below. Pointers are usually but not necessarily restricted to the Inside. Any object which is outside (above or below) is not

subject to reclamation, nor can its value be CONS'ed.

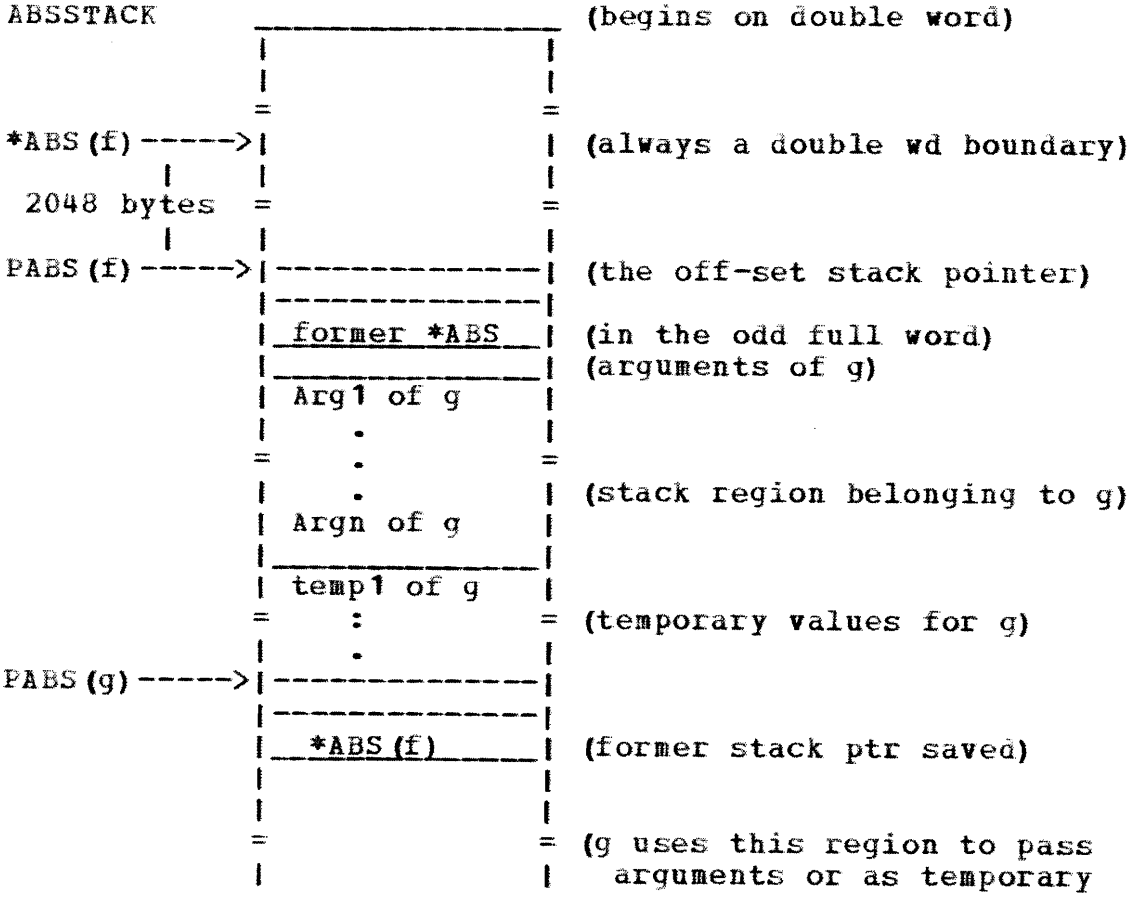
The section on data structures explains the contents of some of the memory partitions and Fig. 5 illustrates their relative order. The blocks labeled ABSolute stack, Symbolic stack, Quote cells, Fixed Pgm. Space, I/O Blocks, Available Space, and the BELOW, need further explanation.

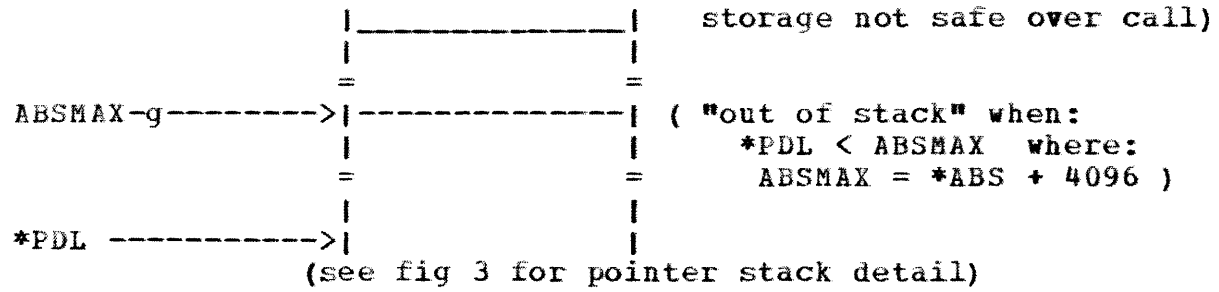
A. STACKS

1. The Absolute Stack:

The absolute stack is a field in memory which is used for the temporary storage of non-locator data structures and for the transmission of non-locator arguments to functions. It is bounded on the low address end by the fixed location ABSSTACK. The current pointer to the stack is given by PABS = *ABS + 2048, where *ABS is the ABSolute stack register. PABS is a value which determines the stack-region belonging to the calling function. PABS is so designed so that references may be made in both positive and negative displacements from it. Each stack region consists of an integral number of double words. The absolute stack should not be used by the casual user. It is used by the system in special and sometimes non-systematic ways. The current LISP COMPILE function does not use it at all.

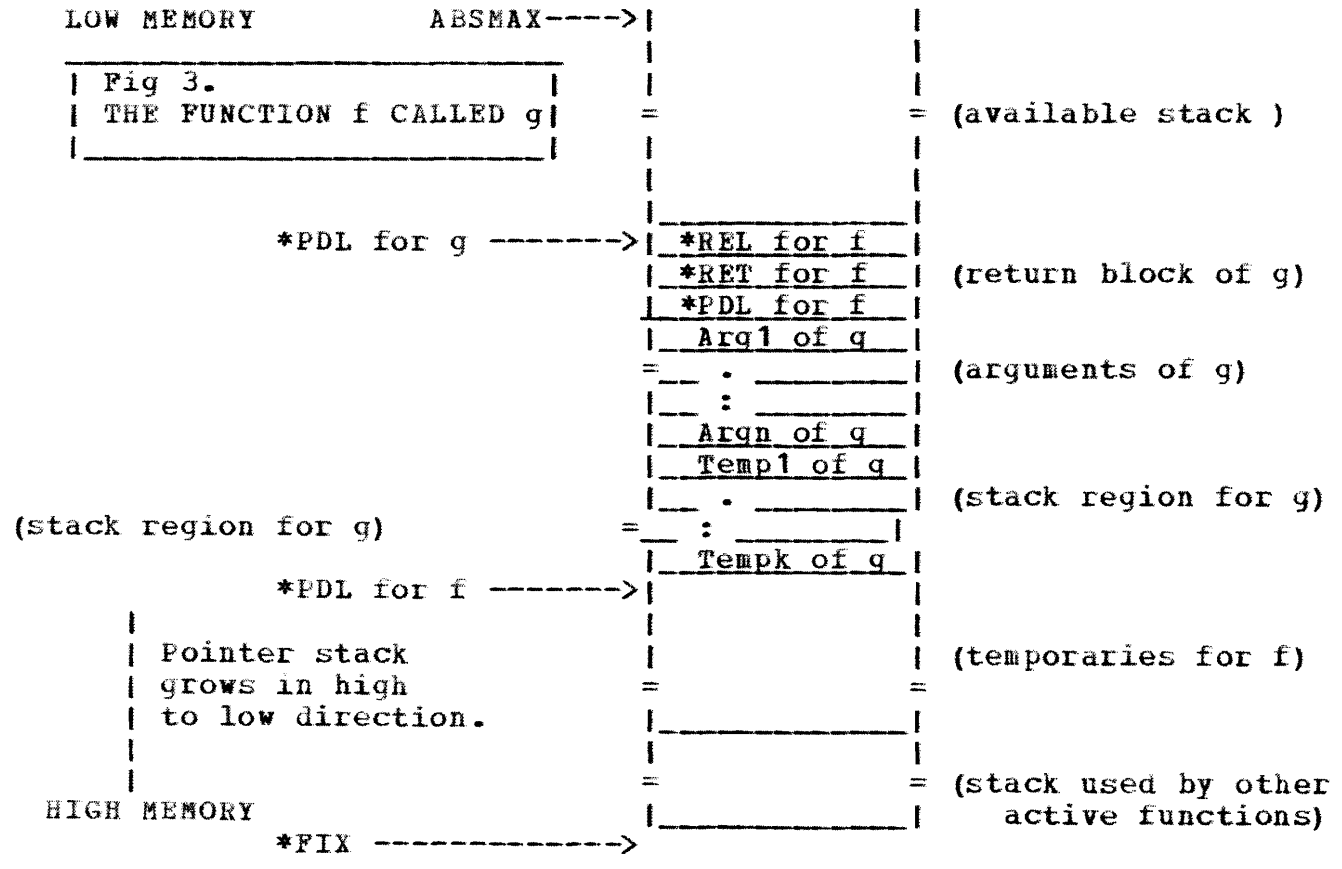
Fig. 2 illustrates the ABS stack.




storage not safe over call)
ABSMAX-g
("out of stack" when:
*PDL < ABSMAX where:
ABSMAX = *ABS + 4096)
*PDL
(see fig 3 for pointer stack detail)

2. The Pointer Stack :

A pointer is a 32-bit field whose 24 low order bits are a subfield containing a setting which is a locator. Pointers are byte addresses. Thus, the pointer stack is a field whose subfields are all pointers. This stack is bounded on its high address end by a fixed address permanently kept in register *FIX, and on its low address end by the current pointer stack pointer contained in register *PDL. This stack is used for the transmission of pointer arguments to functions, for the temporary storage of pointer values, and to provide a systematic method for transferring control from function to function. Fig. 3 illustrates the Pointer Stack.



B. QUOTE CELLS

A field of 1024 words has been permanently reserved for the purpose of holding directly addressable pointer values of compiler generated data structures. These pointers arise from quote expressions which the compiler encounters and are called quote cells.

Quote cells are unique and unalterable. Running out of quote cells would be a catastrophic error Were it

not for the Quotes extension feature. This extension feature uses vector space for additional quotes. The first 1024 quote cells are directly addressable because they all lie within 4096 bytes of the address given by the register called "QUOTE".

All values contained in quote cells are considered active by the garbage collector.

Fixed Program Space

Fixed Program Space is a one page (4096 bytes) field which is permanently allocated and for which addressability is guaranteed by *FIX. Routines and functions which were deemed "most vital" share this page. There is also a subfield of literal and pointer constants which are needed by the system primitives. The contents of fixed program space were hand-coded in OS/360 Assembly Language.

I/O Blocks

A region has been allocated to whatever Input- Output control block that the operating system requires. In the case of OS/360 this allows room for 8 DCB's. This area is now in the BELOW.

Available Space

Requests for new nodes or vectors must be satisfied from available space or else a reclamation occurs. Because of the compacting nature of the reclaim function, available space is always a contiguous block of words.

The BELOW

The BELOW consists of the reclaim function and all its subfunctions, the bootloader, and various other functions all written in OS/Assembly Language. While some LISP data structures may occur in the outside, they are undefined as members of a node (i.e., they cannot be cons'ed). All objects in the outside are not reclaimable and are in fact immobile.

The Register Schema

R0 - *NIL or *ID ; contains NIL; origin of identifiers.
R1 - *LINKR; used to make a function call. (May be used by subroutines (certain LAP360 coded functions which call no other functions and use no pointer stack space) as a base address.)
R2 - QUOTE or *IORG; permanently loaded register used to provide addressability for quote cells; small integer origin (zero).

R3 - *VALUE1)
) permanently loaded registers used to
R4 - *VALUE2) provide addressability for the first 2048
) value cells.
R5 - *FIX; permanently loaded to provide addressability in fixed program space.
R6 - *ABS; the absolute stack register.
R7 - *REL; used to provide addressability within the function currently in execution.
R8 - *RET; used when calling a function to provide a return address.
R9 - *PDL or *PDP; the pointer stack register.
R10- *AC; used to transmit the first argument of a function, and to transmit a function value.
R11- *MQ; used to transmit the second argument of a function.
R12- *SCRE scratch.
R13- *SCRF scratch.
R14- *SCR1 or *RNARG; scratch.
R15- *SCR2 or *MNARG; scratch.

Thus, *NIL, QUOTE, *VALUE1, *VALUE2, and *FIX are all permanently loaded; *ABS, *REL, and *PDL are safe over function calls; and the rest are unsafe over calls and may be used as scratch registers.

The Floating Point Registers

0 - floating point accumulator, used to transmit one literal valued real value, and the value of a real-valued literal-valued function is returned in it.

2 - used for floating point calculation.

4 - currently unused.

6 - scratch register used in moving arguments to their respective stacks.

V. LINKAGE CONVENTIONS

If one is to understand or interfere with this LISP system at the assembly language level, it is necessary to understand the linkage conventions. Linkage conventions are the manner in which control is passed from calling function to called function, and what conditions each is expected to fulfill in order that the system remain well behaved.

As these conventions bear directly on how effectively one can compile code, considerable thought has been put into the conventions. The amalgam of compromises which has resulted is due to the following considerations:

Assembled programs were required to be reentrant, to need no address constant modification when moved, to be as small as possible, to be reclaimable, to be redefinable, and as efficient as possible. Reentrancy of course begot the stacks for alterable storage. It is possible to share such programs provided that when reentered other alterable storage areas pertain. The two stack decision was made to simplify the marking and relocating phases of the RECLAIM function. This cost an additional register (*ABS). Now it appears that there are basically two methods of maintaining stacks. In one method, the calling program is responsible for pushing or bumping (in any case updating) the stack by just the amount required at the time of a call. The other method is that the called function bumps the stack the maximal amount required by itself just once when called. On one hand the first method required only the amount of stack currently holding stored information while the second method would sometimes allocate space that was not required in that particular pass through the function. On the other hand the first method required a longer calling sequence to encompass the variable stack bump. The second method was chosen because stack space was relatively cheaper than binary program space, that is, its permanent requirement for space was smaller. → a pooled function call would change this!

Assembled programs contain no address constants; therefore require no address relocation. They do, however, contain relative addresses (Base-disp) of value cells of identifiers, and of quote cells. But these, once referenced by assembled code, remain in place (i.e., their relative location never changes). Note: While assembly language programs can be exchanged from one copy of the system to another, assembled programs cannot.

The requirement for redefineability and subsequent reclamation of 'dead' assembled programs gives rise to storing them as vector data structures, and accessing

them via the value cell of the function's identifier. Thus redefinition is merely the reassignment of the value cell to a new value. The locator value of such an assembled program vector data-structure is a pointer to the header of the vector. The calling mechanism takes this into account by transferring control into the vector just beyond the header.

A linkage routine exists at the location specified by the register *FIX. The OS/360 version of this routine consists of

```
LH *SCR1,0(*RET)                    1.4
STC *PDP,0(*SCR1,*VALUE1)           1.33
L *LINKR,0(*SCR1,*VALUE1)           1.2
B 4(*LINKR)                         1.2
                                    5.13 u sec
```

and the calling sequence is as short as it can be (32 bits):

```
(BALR *RET,*FIX)                    1.2
                 16

(      x         )
                 16
```

where x is a half-word constant which is a locator for the identifier value cell. The STC instruction is not necessary for linkage but is used in conjunction with the RECLAIM function to measure the activity of a function. The automatic expulsion of assembled programs which takes place dynamically when vector storage space becomes scarce, makes use of this activity measure. The CMS and TSS versions of the linkage routine contain an additional instruction for fielding attention interrupts.

The total time for this sequence is about 6.33 u sec.

This is not significant for functions which require a long time themselves but it is for very short ones. As about 20% of the code produced by the compiler consists of calls to the list accessing functions (CAR, CDR, ..., CADR ..., etc.) and these functions often occur inside tight loops, they were treated specially and require only a 1.2 u sec calling sequence. (The CAR routine requires = 6.58 u sec) The decision to use half-word word pointers has resulted in a factor of 6 loss in time for list accessing. The overall loss in time is probably less than factor of 2 or 3. Hopefully, this can be

overcome completely by the addition of a new instruction in the microprogram of the S/360. YES!

The calling sequence only transfers control to the desired function. The linkage conventions require a systematic method of transmitting arguments, returning values, returning control to the calling program, and for maintaining the stack.

The convention for passing pointer arguments is that the first and second are passed in Registers *AC and *MQ, while the third through 16th must be stored in fixed program space locations ARG3 ... ARG16. This convention dates to the 7094 implementation and is a design error in this S/360 version. Fortunately, it is not felt for functions of 0, 1, or 2 arguments. A better scheme would be one similar to the one used for the absolute stack, in which the ARG's block is unnecessary.

Upon entering a function G called by a function F, G should reserve an area sufficient for its control information, arguments, and the maximum number of temporary locations that it may require. To accomplish this, the assembler recognizes a macro (*MOVE mn rn) where mn is the number of bytes required for the stack region of G and rn is 8 times the number of pointer arguments that G expects. (Thus, if G has 4 pointer arguments, rn = 32.)

The *MOVE macro performs several functions. It

1. reserves a stack region for G.

2. saves *REL, *RET, and *PDP which must be restored when returning control to F.

3. copies the pointer arguments of G into its stack region.

4. checks for out of stack condition.

5. initializes all allocated temporaries to NIL.

6. establishes a new *REL and *PDL for G.

The stack is restored and control returned by the following sequences:

```
LM *REL,*PDP,0(*PDP)    restores *REL, *RET, *PDP

B 2(*RET)               returns to F.
```

The function G is responsible for returning a pointer value in *AC. This completes the descriptions of the

linkage conventions with regard to pointer values, but similar conventions are required for literal arguments, temporaries and values.

The stack for literal values (absolute stack) is also based on the convention that called functions are responsible for allocating a stack region sufficient for their maximal requirement. This stack is somewhat better designed than the pointer stack as it utilizes an offset stack pointer PABS from which it is possible to refer in both directions. Therefore, no ARG's block is necessary. See Fig. 2 which illustrates the absolute stack. The macro (*ABSPUSH m) where m is the number of bytes required for the stack region, is used in the same manner as (*MOVE ...). Literal values are returned in the ARG1 stack position. The absolute stack is restored by:

```
L *ABS,4+2048(*ABS)  .
```

VI. THE COMPILER AND LAP

6.1 Historically LISP has proven to be somewhat slow when evaluated interpretively. This is due in some part to the s-expression model of the evaluating machine and also in some part to the many features that have been added to the language, (i.e., the PROG feature, assignment, go to's, functional arguments, etc.). In any case, it has been possible to implement a compiler which creates code which runs from 40 to 100 times faster than interpreted code. It has also been the case that compiled code usually requires less space than the s-expression representation of the function.

6.2 The Compiler Function

The compiler for LISP is itself a LISP function, or rather, a whole family of functions. The argument of the COMPILE function is a list of defined function names. This list is also the value of the function. The important side effect of the COMPILE function is that for each argument function name an assembled or binary program image (BPI) of executable instructions has been created and the properties associated with the name (identifier) have been altered to reflect that fact. Subsequently that function will be executed rather than interpreted.

It is a characteristic of this compiler that normally only read-only code is generated. The exceptions to this could result from using the *CODE function to create imbedded assembly language instructions.

In this system it is possible to mix compiled and interpreted functions.

6.3 LAP360: The LISP Assembler

(LAP360 listing sym)

The function LAP360 assembles the instructions in listing, plants them in core, and may alter certain properties of identifiers.

listing: = (prologue instruction*) where:

Case 1: (Assembling a patch)

If prologue is a number then the instructions following will be originated at the location computed by adding the number to the system origin given in register *FIX. This is useful for assembling absolute patches.

Case 2: (normal case)

If prologue = (name class number) , then

the instructions will be planted in a vector which the assembler gets dynamically. The identifier data structure for name will be modified to reflect class and number of arguments.

class = (SUBR | SUBR* | FSUBR* | FSUBR)

Indicating whether this function has a definite number of arguments, evaluated (SUBR) or unevaluated (FSUBR); or whether it has an indefinite number of arguments, evaluated (SUBR*) or unevaluated (FSUBR*).

When assembling functions an assembled program vector of the required size is created, and a value pointer for that vector is planted in the value cell for the identifier given as name. In addition, the identifier structure is filled in with the rest of the prologue information.

The format of the assembler's instructions is

instruction = label

macro

call

pseudo-instr

(rrop reg reg)

(rxop reg addr reg)

(ssop reg addr reg addr)

where

label is a literal-atom designation for the location of the following instr. All labels are local to the current listing.

macro = (mac-name args*) and mac-name is the name of a previously defined assembler macro. Such macros can be defined by the function MACLAP, their effect is to insert a sequence of instruction into the listing.

call = (CALL fname number)
Where fname is the called function's identifier.

pseudo-instr = (=F number) for full word constants

(=H number) for half word constant

(*DATAS addr) for base disp constants

(*MOVE number number)

(*ALIGN { F | H }) to provide alignment

addr = (reg disp*)

(QUOTE datum)

(FUNCTION id)

(SPECIAL id)

(LABEL label)

The addr field produces a 16-bit field by shifting the reg field value left 12 and OR-ing it with the sum of the disp values.

(QUOTE datum) creates the base-displacement setting appropriate to locate a pointer value for the s-exp datum. Note - such quoted structures are shared and must not be altered. 1024 such QUOTE value cells have been reserved and when they are all used the assembler resorts to less efficient methods. Namely the use of a QUOTES-extension vector.

(FUNCTION id) creates the base-displacement setting appropriate to locate the value cell of id. Also bit 3 of the identifier type code is set to 1.

5?

(SPECIAL id) same as (FUNCTION id)

In general the symbols for :
rrop, rxop, ssop, reg, disp, and label are all either numbers or id. The id are evaluated by lookup in the symbol table environment given by sym.

sym = ((id . number)*)

If a symbol does not occur on sym, the property list is inspected for a permanent symbol value (called INTSYM).
See APPENDIX G for a list of the assemblers permanent symbol values.

VII. INPUT/OUTPUT

LISP relies heavily on the operating system for its I/O capabilities. I/O functions are essential so that existing data can be read and data structure in memory can be printed. Data exist in records which occur in files, the file being a sequence of records. The individual I/O functions are described in APPENDIX C.

7.1 Describing a file.

The user may process only those files which have been made known to LISP via the (OPEN*SEQ...) function. When the user begins LISP execution two files are automatically made known: LISPIT, the standard input file, and LISPOT, the standard output file.

The function (SHUT x disp) is provided to let the system forget about a file.

Operating systems may impose demands and requirements for files before they may be processed, when running under OS/360 DD cards describing any file which might be used must be included in the Job Control Cards which describe the LISP Job*. The LISP file names correspond to the DDname parameter. All file and record characteristics must be specified on this card. The one great good that comes is that the LISP programs then become quite device independent.

*Needless to say that under OS it may become necessary for the user to supply these cards. See Principles of Operation SRL.

7.2 Selecting Files

The datum and record reading functions are all written to operate upon the currently selected input file. The function (RDS file-name) is available for selecting any known opened file as the current input file. Likewise, (WRS file-name) selects an output file. The following special variables describe the selected input file:

CURINAM = file-name

CURINBUF = string data structure which received the record.

CURINDCB = location of the DCB, data control block

CURCOL = 0 --> n column of CURCHAR

CURCHAR = last character read.

UBUFLIN = index (0-origin) of last char of buffer.

The output file state is given by:

CUROUTNAM = file-name

CUROUTDCB = DCB

CUROUTBUF = string for output record

CUROUTCL = 0 --> n current output column

7.3 Record Processing

The function (*NEWLIN) brings in that portion of the next record of the currently selected input file that will fit in the CURINBUF vector. This allows one to read the n initial characters of a record by specifying a CURINBUF vector of n characters. On the other hand, (TERPRI) fills in blanks from CUROUTCL to the end of the CUROUTBUF buffer and then outputs the record.

7.4 Character Stream Processing

The functions (READCH) and (PRINTCH) allow one to proceed character by character through a file as though it were all one string and the records didn't exist. In this manner arbitrary character strings may be processed without regard to the syntax of LISP data.

7.5 File Positioning - not yet available.

7.6 (READ) the datum reader

(READ) is the function which starts at CURINCOL and continues reading characters until an entire LISP datum has been passed, the value of (READ) is the corresponding data-structure. See Section 3. (READ) employes (*RATOM) to recognize one token . Unless that token was(or %(the value of READ is the data structure for that token. If (or %(was encountered (READ1) is used to collect all the following data into a data structure until an unbalanced) or %) token is encountered.

token = (|) | . | literal-atom | number | string | %(| %)

7.7 (PRINT x)

(PRINT x) outputs the datum x, filling out the final line with trailing blanks.

(PRIN0 x) - same as (PRINT x) without any trailing blanks.

(PRIN1 x) - outputs any literal-atom, number or string without trailing blanks.

VIII. THE CODE FEATURE

In this implementation of LISP it is possible to use LAP360 coded expressions directly. This is possible through the use of a code-exp where

```
code-exp = ( code-op  instr-c* )
```

where

```
code-op = *CODE
          *ICODE
          *RCODE
```

corresponding to pointer valued, integer literal valued, and real literal valued code-exp.

A code-exp can occur in any context that an exp is allowed, but must obey the style conventions imposed by the register allocation schema, the storage organization, and the schema for value transmission.

```
instr-c = instr*
```

The code instr are just like regular LAP360 instructions with the following additional addr field allowed.

```
addr-c = ( *LOCATE id )
```

Which allows the instr to reference variable values that occur in the context of the LISP function in which the code-exp occurs. No provision currently exists for interpreting code-exp. Nor is any envisioned. The code feature is thus purely a compiler artifact.

GETTING ALONG IN LPL ON CMS

This note assumes some knowledge of CMS.

The CMS command that puts one in the environment of the basic LPL supervisor is:

LISP LPL

If this command does not work, then LPL is not available to CMS. (you may make discrete inquires in this case)

If LPL is available and working, you may input one LPL expression. Assuming that the expression is syntactically valid, it will be evaluated. All LPL expressions produce a value which is printed, or they cause the ERROR routine to be entered, or they continue running.

This process is repeated until the expression:

.FIN

followed by a final:

.IN

brings one back to the environment of CMS.

This explanation gives rise to the following questions:

1. What are LPL expressions?
2. How does one input an LPL expression?
3. What if my input is not syntactically correct?
4. What if the evaluation causes ERROR to be called?
5. How can I interrupt the evaluation process?
6. What are the basic primitives and builtin functions available in LPL?

We treate the above questions briefly, in sections 1-6, and then return to a fuller discussion of the meaning of LPL expressions in section 7.

1. What are LPL expressions?

The following symbols are used for syntax description:

{ and } are used for metalinguistic grouping.
[and] are used to indicate optionality.

* is used to indicate one or more.
| is used to separate alternatives.
Vertical alignment is also used for alternatives.

Expressions are the principal linguistic form of the LPL language. An expression may be given an interpretation through evaluation. This new interpretation of the expression, is in a strictly narrow sense, merely a new symbol. The human interpreter of this symbol may think of it as denoting an object. He may consider the expressions themselves as denoting objects and should do so whenever possible.

A conversation in LPL consists of a sequence of expressions received from a terminal or file. The LPL supervisor translates each expression into a data-object (a LISP1.5 S-expression). It then evaluates that object, outputs its value, and repeats this process until a termination is signaled. Should the evaluation of an expression require the reading of user input, then that input must precede the next expression to be evaluated. In addition, printing may occur before the expression's value is printed.

The value of an expression is a data-object. Data-objects are characterized by their location and type. Data-objects may have associated properties, some of which are more directly tied to the location for purposes of efficiency. Data-objects are of two classes:

constants: identifiers, numbers, and strings;
aggregates: pairs, lists and vectors.

It is convenient to give data objects a linear representation (for input and output); this gives rise to a data language. The data objects in this representation are traditionally called symbolic expressions (S-expressions). In LISP1.5 systems constants and vectors are usually refered to as atoms. We shall perpetuate the use of this term, realizing that it has the following operational definition: An atom is an object for which the list selectors CAR and CDR are not applicable. Of all the data objects the sine qua non has proven to be the pair. Thus, the following definition of S-expression:

A data-object is either an atom or a pair of data-objects.

S-expression = atom | (S-expression . S-expression)

We reserve a special atom NIL (also "()") to denote the empty list. We prefer to use list notation which results from replacing all occurrences of dot followed by a left parentheses with a blank, and deleting the balancing right parentheses.

(A . (B . (C . ()))) is the list (A B C)

Names may be associated with values (data-objects) and are

referred to as variables. Functions (more properly function closures) are parameterized definitions of their value object. They are denoted in the LPL language by procedure expressions and are themselves objects. The application of a function closure to arguments is called an operator-operands combination. Combinations in LPL are represented by infix operators or by juxtaposition as described below.

Each expression in the LPL language not only denotes a data object but also has a representation as a S-expression. These data representations include the well known LISP1.5 S-expression language. While not every S-expression is the representation for a meaningful LPL expression, it is possible to represent every LISP1.5 expression in LPL. For example:

```
A + B
```

has the data representation

```
(PLUS A B)
```

The quote operator (') followed by an LPL expression is used to denote the representation of the expression.

```
'(A+B)
```

translates to

```
(QUOTE (PLUS A B))
```

which evaluates to

```
(PLUS A B)
```

One may write expressions directly as an S-expression by prefixing the S-expression with the escape character ("), e.g.
"(PLUS 3 4) evaluates to 7.

Expressions in LPL are written as sequences of tokens. Tokens are represented by the machine dependent character set which is assumed to contain:

letters, digits, space or blank, and a hopefully large set of special characters called marks.

There are three classes of tokens:

Identifiers (id) which are strings of letters and digits that start with a letter and are terminated by a space or a mark. The exclamation mark (!) has been preempted from the set of marks to play a special role. Any character it precedes will be considered a letter, hence we call it the letterizer.

Literals - numeric constants and the string constants. All numeric constants start with a digit and string constants are surrounded by (#)'s.

Mark-symbols - marks and known strings of characters that start with a mark. Compond mark-symbols are made known to the token recognizer as a special property of the initial

mark. The LPL system contains the following compound mark-symbols:
{ ** :=: := .EQ .NE .GT .GE .LE }

Tokens are used as operators and delimiters. Operators are used to form expressions. They may take other expressions as operands and they may require certain delimiters. There are two kinds of operators: Those which require an expression on the left and those which do not.

LPL is said to have an operator precedence grammar because precedence functions are used to determine the scope of operators. Each operator or delimiter has a left binding power (lbp); in addition, operators which take right operands have a right binding power (rbp). Operators which have left operands have an interpretation which is called a left denotation (led). Operators with no left operand have a null denotation (nud). In either case an operator may have optionally many right operands. An operator may have a led or nud or both.

Tokens which are unknown to the grammar are assumed to be variables. The juxtaposition of expressions indicates function application (combination). Combinations associate to the right. For example,
A B C = (A(B(C)))

In the case of tokens which are known to the grammar as operators the interpretation of the operator requiring a left operand is always preferred. For example,
a-b indicates subtraction,
a(-b) indicates application of a to minus b.

If r is an operator that has a right operand, s a token with a lbp, and E is an expression in rEs, then if
rbp(r)≥lbp(s) r gets E.

A set of name-value relations (the environment) gives values to variables. Enriching the environment with new name-value relationships is known as binding. When functions are applied to arguments the parameter variables of the function are bound to the argument values. Binding may also occur by SPECIAL declaration. This establishes a global or shared binding which persists until explicitly removed.

Variables may be assigned any type of data object, provided the name-value relation for that variable has been established in the environment. As a matter of convenience the supervisor creates SPECIAL declarations implicitly for variables that are the subject of top level assignments. For example,
The expression

X := 7.2

is used to assign the value 7.2 to the variable X.

A subsequent

```
PRINT X;
```
would print out
```
7.2
```

Arithmetic operators are indicated in the usual way. For example,
```
X*2-4.4
X*2-4.4/2-2.2
X+-X+5*5-15
3/2*X/7.2*10
(3.0/2-0.5)*X+2.8
```
all evaluate to 10.0 . The arithmetic infix operators used above all associate to the left. Exponentation is indicated by the compound mark-symbol ** which associates to the right. The use of the token "-" above illustrates the use of an operator with two possible denotations.

Expression sequences are two or more expressions separated by semicolons. The e pressions of the sequence are evaluated in left to right orde The value is the value of the last expression evaluated. "PROGN"

The S-expression selectors CAR and CDR are used to select the first element of a list and the list of successors to the first element respectively, for example,
```
A:='(R S T(U V));
```
would evaluate to
```
(R S T (U V))
```
And then,
```
CAR CDR A;
```
would evaluate to
```
S
```
.

The S-expression constructor, CONS, is available either as a function or as the infix operator period, for example,
```
A:= 'R . 'S . 'T . ('U . 'V . NIL) . NIL  ;
```
evaluates to
```
(R S T (U V))
```

The list constructor, LIST, is also available either as a function or as the infix operator comma, for example,
```
A:=('R ,'S ,'T ,('U , 'V)) ;
```
or equivalently
```
A := LIST('R , 'S ,'T ,LIST('U ,'V)) ,
```
evaluates to
```
(R S T (U V))
```
and,
```
A,;  =  LIST(A) ;
```
.

Vector data-objects are denoted in a manner similar to lists, except that angle brackets are used instead of parentheses; for example both
```
B:= '<D E F <F E D>> ;
```
and
```
B:= <'D ,'E ,'F ,'<F E D>> ,
```

evaluate to

```
<D E F <F E D>>
```

Elements of vectors and lists are selected by zero-origin indexing: for example,

```
LIST(A<1>,B<3>) ;
```

evaluates to

```
(S <F E D>)
```

Vector and list elements may also be assigned: for example as a result of

```
B<3,1>:= 'M ;
```

B evaluates to

```
<D E F <F M D>>
```

The result is a change to the existing structure denoted by B (all variables referencing that structure are affected). A similar effect may be achieved on lists by use of the well known LISP functions RPLACA and RPLACD.

A character string of arbritary length may be denoted by surrounding it with the character # for example,

```
A:= #A TYPICAL STRING# ;
```

The concatenation of strings is achieved by the use of the dyadic operator _ for example,

```
B:= A _ #FOLLOWED BY THIS# ;
```

would evaluate to

```
#A TYPICAL STRINGFOLLOWED BY THIS#
```

In addition,

```
C:= SUBSTR(B,12,4)_ # # _ SUBSTR(B,21,6)
```

would evaluate to

```
#RING WED BY#
```

A procedure is a parameterized expression used to define a function. It may be applied to zero or more arguments and may use as yet undefined functions and variables. A procedure expression evaluates to a function closure which has an appropriate internal data form. As an example, the right-hand side of:

```
SUB:='(PROC X,Y,Z;
        COND(Y .EQ Z : X,
             ATOM Z : Z,
             T : SUB(X,Y,CAR Z) . SUB(X,Y,CDR Z))  )
```

is a procedure to be assigned to the variable SUB. The variable can then be used to evoke the procedure in the usual way.

```
SUB('A ,'B ,('C ,'B ,'D ,'B))
```

evaluates to

```
(C A D A)
```

In the procedure SUB above notice the use of the Lisp conditional COND. LPL provides the IF ... THEN ... ELSE... expression but COND may also be used in this direct manner. The use of the list-representation operator colon ":" is also illustrated. This operator is introduced in pursuit of the goal of being able to have an LPL representation for every possible

Lisp1.5 s-expression language expression.

The symbol :=: is provided as a simple way to define new functions. It causes the Lisp representation to be prettyprinted, compiles the PROC expression and associates the code with the function name as a global definition.

The block expression in LPL introduces the sequential mode of programming where statements are executed one after another. Labels and GO statements allow the usual alteration of flow of control. RETURN statements enable the user to indicate the value of the block on exit.

As well as the elementary looping statements, the DO-expression is provided. This may be used to give programs a more understandable appearance through the elimination of labels and GO-statements. The DO-expression is used to express the various classes of loops which one might otherwise construct with labels and GO-statements.

In the simplest case no looping occurs and the DO-expression is equivalent to a block-expression for example,

```
DO; X:=Y ;Y:=2;END;        is equivalent to :
BEGIN;X:=Y;Y:=2;END;              .
```

THE UNTIL and WHILE phrases are used as modifiers which produce loops with exit conditions, e.g.

```
DO WHILE N .GT 0;UNTIL Y=CAR X ; UNLESS NOBLANK;
     PRIN1 BLANK; N:=N-1;X:=CDR X; END ;
```

is equivalent to:

```
     BEGIN;
     A;   IF NULL (N .GT 0)THEN GO B;
          IF NOBLANK THEN GO C;
          PRIN1 BLANK;
          N:=N-1;
          X:=CDR X;
     C;   IF NULL(Y=CAR X)THEN GO A;
     B;   END;
```

The WHILE-phrases generate top-of-the-loop exit-tests. The UNTIL-phrases generate bottom-of-the-loop exit-tests. The UNLESS-phrases are used to conditionally execute the statements of the DO-loop.

A DO-expression may be preceded by a FOR-expression which consists of FOR and a sequence of iteration control clauses. Each iteration control clause specifies a control variable, and possibly its initialization, the exit test, and the manner in which they are reset for each repetition.

Syntacticaly:

```
FOR { name [ := expression [BY-phrase]|
           BY-phrase                  |
           IN expression              |
```

```
            ON expression                ] ; }* DO-expression
```

where, BY-phrase = BY expression [TO | UNTIL] expression .

```
Some examples:
FOR I:=1 BY 1 TO 10;DO ... ;END;
=    BEGIN;    I:=1;
     A;   IF I .GT 10 THEN GO E;
               ...
               I:=I+1;
               GO A;
     B;   END;

FOR x ON y; DO;PRINT x;END;
=    BEGIN;
               x:=y;
     A;   IF NULL x THEN GO B;
               PRINT x;
               x:=CDR x;
               GO A;     B;   END;
```

The syntax and semantics of LPL are completely specified by the LPL to Lisp translator and by the semantics of Lisp. The user of LPL should pursue such detailed information as he may require to those sources.

2. How does one input an LPL expression?

The initial state of the LPL system into which the user enters upon administering the CMS command LISP LPL, is as follows:

A certain amount of typing occurs which may usually be ignored.

If the system is running in the "history keeping" mode a sequence number followed by a colon is printed. In which case a record of computed values and user text is kept. This file has the filetype HISTORY and the default filename of LPLHIST. Text and values may be recalled from this file.

The keyboard at the users terminal unlocks allowing the user to type an LPL expression.

The global free variable SINGLINEMODE has the value true which implies that a carrier return is sufficient to end the expression.

The underscore "_" at the end of a line is the currently defined expression-continued-on-next-line symbol.

The values of expressions will be printed at the terminal.

The Lisp translation of the expression may be printed if the XLATEFLAG is so set.

3. What if my input is not syntactically correct?

Should the LPL expression input be syntactically incorrect, an indication of the error is presented and the LPL editor is entered. The behavior and use of the LPL editor is described below.

The editor is entered with a segment of text that the user has just typed. That segment is revealed to the user by ellipsis printing, that is, as much of the segment as can be printed in 80 columns is displayed with missing parts indicated by ... . If this did not present the user with enough text to correct the error, the user may type FAIL, in which case a larger context will be presented. That being the case the user may use the editor commands to focus in upon a fragment of the original segment and perform deletions, insertions, and substitutions.

The following is a description of the editor commands:

Advance A [n] {* | p}

The left boundary (LB) of the current fragment (FR) is advanced n places. The default value of n is 1 and n may be negative. The meaning of place is established by the last argument:

* If any character is a place.
p If the string p is a place.

CHANGE C p [n] s

Equivalent to "Find p" followed by "Substitute [n] s" . IF p is not found the command has no effect.

Delete D

The current fragment is deleted from the whole segment.

Echo E

The echo flag is flipped. If ON then all editor commands which would not have otherwise caused printing, will cause FR to be printed elliptically.

EXIT

Return from the editor and reparse this segment.

FAIL

Return from the editor for a retry with a larger segment. This gives a larger context in which to repair a bad expression.

Find F p

The string p may be an ellipsis string, e.g.
F #ab...rst#
FR is searched for the indicated string. If not found there is no change, if found LB is readjusted and a new FR is thus defined.

Left-insert L s

The characters of s are inserted to the left of FR. A new FR results which includes s.

Next N

A new FR is defined which consists of everything in the whole segment to the right of FR.

Print P

FR is printed elliptically.

Right-insert R s

FR is extended to include the string s inserted on the right.

Substitute S [n] s

Substitute s for FR n times. The default value of n is 1. n must be positive.

Top T

FR is set to the entire segment.

Whole W

FR is printed in its entirety.

Extend X [n] {* | p}

The right boundary (RB) of FR is extended n places. (see advance)

The text for previously written expressions may be retrieved from the currently selected history file. To retrieve the text for expression "n" one uses the form "(n)" instead of an explicit string for "s" in any of the above commands. By this means old expressions may be revised and reevaluated or they may be inserted into new expressions.

4. What if the evaluation causes ERROR to be called?

At the time ERROR is called you will be so informed and you have several options.

1. You may hit carrier return to be prompted about what your options are.
2. You may type "u" meaning UNWIND. In which case you will be in the LPL editor with the ability to correct or delete the last expression evaluated.
3. You may get a diagnostic backtrace by typing "k" for an abbreviated one or "l" for a long one.

5. How can I interrupt the evaluation process?

The LPL attention handler is invoked through the external interrupt feature of CMS. To use it:

1. Hit Attention once.
2. Type E or EXTERNAL

When this external-interrupt is felt by LPL/Lisp a prompting message is typed at which time the user may indicate which interrupt service he desires or he may hit carrier return for more prompting.

The available requests are as follows:

C Puts the user in CMS subset command language.
U Unwind to the last errorstop.

Has the effect of cancelling the last execution.

B Call ERROR for backtrace and diagnostics.

S Recursively reenter the LISP supervisor.
A subsequent FIN will cause the interrupted program to resume.
A subsequent RET gets him back.

P To ignore the interrupt and proceed.

? Types an expanded prompting message.

This attention handler takes effect at the time of function call. It does not work for tight loops.

In such a situation one should try:

1. Hit attention once.
2. Type--- B 1A790

6. What are the basic primitives and builtin functions available in LPL?

The basic primitives and builtin functions are exactly those available in the Lisp system. These are described in APPENDICES A , B , C, and D.

7. WHAT DOES AN LPL expression mean?

In section 1 an informal overview of LPL was given. A somewhat more complete and more formal definition will be given in this section.

Recall that every LPL expression was said to have a data representation and that form of the expression was called an expression of the Lisp1.5 language. The Lisp1.5 language is an attempt to express programming languages in a quintessential form. It provides us with an evaluation model which explicates the notions of function and variable.

The syntactic form of the Lisp1.5 language is practically devoid of syntactic niceties. Informally these niceties aid in the human recoqnition process. Sometimes in languages they are there to aid in some other recognition process and could hardly be thought of as niceties. The syntax of Lisp1.5 can be thought of as one which simplifies the recognition processes of the evaluator.

The correspondence between LPL expressions and Lisp1.5 expressions is what is described by the LPL to Lisp1.5 translator. This formal system will be dealt with elsewhere. By carefully describing the syntax of Lisp1.5 most of these correspondences will become apparent.

In the past the semantics of Lisp has been given by the process of self description. Perhaps this stems from a desire to illustrate the power of the Lisp language or more pragmatically because this is the method used in a bootstrap implementation of Lisp1.5. Needless to say this approach has some shortcomings from the point of view of definition. A tacit understanding of Lisp is required to read the definition of Lisp and at least a primitive Lisp system is required to begin the bootstrap.

We shall copy the method of P. Landin in creating a metalinguistic description of a machine. The machine itself is described as a complex space consisting of "states" and the state transitions. The meaning of an expression is given by embedding it in the initial state of this machine and when a terminal state is reached after repeated transitions the meaning of the expression may be extracted. The states of the machine are quadruples {S;E;C;D} whose components are called Stack, Environment, Control and Dump respectively. The primary purpose of this state language is to give meaning to the expressions of Lisp1.5. The model is also suggestive of implementation strategies.

In the following syntax and semantics these additional notational conventions will be used:

"(" , ")" , "." , "," , and blanks are all used as special

symbols in forming s-expression representations where an s-expression (also s-exp or datum) is:

{id | c | () | (s-exp* [• s-exp]) }

";" is used as a metalinguistic separator.

Superscripts will be used with brackets to indicate a required one to one correspondence.

Lower case identifiers are used as metalinguistic variables ranging over s-expressions.

Non s-expression constants of the metalanguage are indicated by overprinting or bold italic. They are:
(op, ap, list, eval, pred, stmt, pop, and **go).**

Syntactic classes are represented by underlined lower case identifiers or by italic.

Identifiers (names) given in all upper case letters are Lisp1.5 data objects.

Underlined or italic upper case letters will be used to designate the metalinguistic state components. They are:

S = The value Stack, represented as a list.

E = The Environment, an updatible function from id to values (s-exp). In what follows this function will be represented as a list:

either () or (hE • tE)

where hE = (id • s-exp)* and tE is an E.

The list notation is meant to be suggestive, there is no decree that E must be a list.

C = The Control stack, represented as a list.

D = The Dump of a previous state, which is either () or a previous state {S';E';C';D'} .

The ellipsis "..." is used to denote zero or more objects.

A Lisp expression e is one of:

c a constant.
id is a variable name or identifier.
a lambda-exp which is either a macro represented by:
(MLAMBDA id body) or a procedure (LAMBDA bv body)
where bv the bound variables is ([id]*) and
body is an e.
(LABEL {id e}*) a label-expression.
(SETQ id e) assignment.
(COND [(p g)]*) a conditional-expression

where the predicate p is an e, and
the consequent q is an e.
(PROG pv [s]*) a program-expression
where pv the program-variable part is ([id]*), and
each statement s is a:
label which is an id, or
an e which is not an id.
(GO id) a go-expression.
(RETURN e) a return-expression.
(FUNCTION e) .
(QUOTE s-expression) a quoted s-expression
(rator [rand]*) a combination
where the operator rator is an e, and
each operand rand is an e.

It should be noted that except for constants and variables every Lisp expression is a combination. Some of these combinations are distinguished for semantic reasons.

It should also be noted that the data language of Lisp1.5 s-exp's is somewhat richer than is given above. The full syntax of s-exp is given in implementation documents. The syntax as given is sufficient for the representation of Lisp1.5 and for the explanations of this section.

The following is a listing of the state transitions for the {S; E; C; D} machine.

Halting

{x•S; E; (); ()} ===> x

Value return restoring the former state

{x•S; E; (); {S'; E'; C'; D'}} ===> {x•S'; E'; C'; D'}

Evaluation of a variable

{S; E; id•C; D} ===> {val{E; id}•S; E; C; D}
where val{E; id} is a semantic function which
gives the s-exp value denoted by id when it is defined
by E.
In the case that the variable id is not defined by
E a new state is produced to reflect the error
condition. This state is of the form:
{S; E; (GO ERROR)•C; D}

Evaluation of a lambda-exp

{S; E; lambda-exp•C; D} ===> {(FUNARG lambda-exp E)•S; E; C; D}

Self-denoting expression

{S; E; c•C; D} ===> {c•S; E; C; D}

Quotation

{S; E; (QUOTE s-exp)•C; D} ===> {s-exp•S; E; C; D}

Label expression for self-reference

{S; E; (LABEL id^1 e^1[id^2 e^2 ...])•C; D}
===> {x^1•S; E; C; D}
where x^1 = (FUNARG e^1 E')
and x^2 = (FUNARG e^2 E')
and E' = ((id^1•x^1)[(id^2•x^2)...]•E)

Re-evaluation

{x•S; E; eval•C; D} ===> {S; E; x•C; D}

Transmission list former

{x1•... xn•(); E; list•C; D} ===> {(xn•... x1•())•(); E; C; D}

Operator operand evaluation (decomposition)

{S; E; (rator [rand]*)•C; D}
===> {S; E; rator•op•(rator [rand]*)•C; D}

Repeated evaluation of operator until a closure value results

{x•S; E; op•C; D} ===> {S; E; x•op•C; D}

where x ≠ (FUNARG e E') or c.

Constant operator expressions form lists.

{c • S; E; op • (rator [rand]*) • C; D}
===> {c • (); E; [rand •]* list • (); {S; E; C; D}}

Closure forming function

{S; E; (FUNCTION e) • C; D} ===> {(FUNARG e E) • S; E; C; D}

Operator Classification

Function which expects its arguments all evaluated

{(FUNARG (LAMBDA bv body) E') • S; E; op • (rator [rand]*) • C; D}
===>
{(); E; [rand •]* list • (); {(FUNARG (LAMBDA bv body) E') • S; E; ap • C; D}}

Basic function which expects its arguments all evaluated

{(FUNARG id E') • S; E; op • (rator [rand]*) • C; D}
where id is a basic function of type LAMBDA.
===>
{(); E; [rand •]* list • (); {(FUNARG id E') • S; E; ap • C; D}}

Macro's

{(FUNARG (MLAMBDA id body) E') • S; E; op • (rator [rand]*) • C; D}
===>
{(rator [rand]*) • (FUNARG (MLAMBDA id body) E') • S; E; ap • eval • C; D}

Basic macro's

{(FUNARG id E') • S; E; op • (rator [rand]*) • C; D}
where id is a basic function of type MLAMBDA.
===>
{(rator [rand]*) • (FUNARG id E') • S; E; ap • eval • C; D}

Other closure

{(FUNARG e E') • S; E; op • (rator[rand]*) • C; D}
where e is not a basic function and is not a lambda-exp
===>
{(); E'; e • op • (); {S; E; op • (rator[rand]*) • C; D}}
and
{x • S; E'; op • (); D} ===> {x • S; E'; (); D}
where x = (FUNARG e E'') or c

Application

{x • (FUNARG (lambda-op bv body) E') • S; E; ap • C; D}
where lambda-op is LAMBDA or MLAMBDA

===>
{(); E"; body • (); {S; E; C; D)}
where E" = pairlis{bv; x; E'}
pairlis is a semantic function which creats a new environment E". It pairs the names of bv with corresponding values given by the list x. These pairs(called hE) are appended to the left of the old environment E' to produce E".

Basic function application

{x • (FUNARG id E') • S; E; ap • C; D}
===>
{z • S; E; C; D}
where z is the understood value of id{x}E'.

Distinguished basic functions

The EVAL function

{(e E)•(FUNARG EVAL E')•S; E; ap•C; D}
===>{ (); E; e•(); {S; E; C; D }}

The APPLY function

{(fn args E)•(FUNARG APPLY E')•S; E; ap•C; D}
===> {args • (FUNARG fn E) •S; E; ap•C; D}

Conditional Expression

Default value of conditional is ()

{S; E; (COND) • C; D} ===> {() • S; E; C; D}

Process predicates sequentially

{S; E; (COND(p1 e1) (p2 e2) ...)•C; D}===>
{S; E; p1 • pred • e1 •(COND(p2 e2)...)•C; D}

Continue if false

{()•S; E; pred • e1 • (COND(p2 e2)...)•C; D} ===>
{S; E; (COND(p2 e2)...)•C; D}

Evaluate the consequent when the value of the predicate is true

{x•S; E; pred•e1•(COND(p2 e2)...)•C; D} ===> {S; E; e1•C; D}
for x ≠ ()

Statement Evaluation

Enter statement context

{S; E; (PROG([id]*)[s]*)•C; D}
===> {(); E'; stmt•(s1...)•(); {S; E; C; D}}
where E' =
([(※label1 • label-closure1)...][(id • ())• ...]E)
where ※ is a unique character reserved for this purpose and
label-closure = (LABELCLOSURE y,E',{S; E; C; D})
y = the list of statements following the label.

Leave statement context

{S'; E'; stmt • () • (); {S; E; C; D}} ===> {() • S; E; C; D}

Process statements sequentially

{S; E; stmt•(s1 s2 ...)•(); D} ===> {S; E; s1•pop•stmt•(s2 ...)•(); D}

Statement values are ignored

{x•S; E; pop•C; D} ===> {S; E; C; D}

Go To expression

{S; E; (GO label)•C; D} ===>
{S; E; ※label•go•C; D} ===>
{(LABELCLOSURE(s ...),E',D')•S; E; go•C; D} ===>
{(); E'; stmt•(s ...)•(); D'}

Return expression

{S; E; (RETURN e)•C; D} ===> {(); E; e•(); D}

Assignment

(SETQ id e) =

(CDR(RPLACD(SASSOC(QUOTE id)E(FUNCTION ERROR))e))

```
(ERROR) =     (GO ERROR)

SASSOC = (LABEL SASSOC(LAMBDA(X Y Z)(COND
              ((NULL Y)(Z))
              ((EQ(CAAR Y)X)(CAR Y))
              (T(SASSOC X(CDR Y)Z))    )))
```

It is assumed that the basic functions CAR, CDR, NULL, EQ, etc. are understood. The function RPLACD requires the concept of a store as an addition to the model.

All the transformations given above are in the form:

{S; E; C; D} ===> {S'; E'; C'; D'}

And will be though of as having taken place in a memory M.

{S; E; C; D}M ===> {S'; E'; C'; D'}M'

These memory concepts are described below.

Lisp Extended

The basic Lisp described above has every variable treated in the same manner. They are all equally global and dynamic because of their treatment in variable evaluation, application, and statement context.

The model makes no ordinance about the process of definition. Operator variables may have closures as values or they may have lambda-expressions which evaluate to closures in the run time environment.

In the interactive mode of program development new operators are separately defined and compiled, resulting in an updated environment. The currently popular style of definition is by lambda-expression rather than by closure. When such "definitions" contain free variables the "function" defined cannot be known until run time. Such definitions with free variables which must be bound at run time are said to be more difficult to read and understand than tightly bound definitions. A contrary position to this is that such definitions are simply a short hand for implied parameters and are easier to write, and yield a less verbose program.

As the model given above dictates that a free variable is bound to the latest occurrence on E, the following problem may arise: The binding is not the desired one but rather an inadvertent binding of the same name. This occurs when a bound variable has the same name as an implied parameter and because of the shorthand the conflict went unnoticed. The usual prophylaxis is the use of lexical variables. Lexical variables are textually

bound and are simply invisible to separately compiled functions. Such variables have been the default for Lisp compilers for some time, and the interpreter model (to my knowledge) has never been extended to include them.

An extended Lisp has been defined in terms of the basic Lisp and does include lexical variables. Certain other common features of Lisp systems are also shown to be definable extensions to the basic Lisp model. These other features include functions with unevaluated arguments, functions with indefinite numbers of arguments, lexical and special variables, functions with restrictions on their argument values, and functions with structures as arguments.

All these extensions result from renaming LAMBDA in the above rules to LAMBDA* and creating a macro operator LAMBDA. This macro is of the form :

(LAMBDA bv' body)

where bv' = {var | (var* . var) | ([var]*)}

where var = (QUOTE vari) | vari

and where vari = ident
| structure
| (: ident[{ e | :}[structure]])

and ident = { id | (LOCAL id) | (SPECIAL id) }
and structure = (vari*) | (vari* . vari)

and undergoes the following expression to expression transitions.

Lexical variable renaming (alpha-conversion)

(LAMBDA (id*) body) ===> (LAMBDA* (id'*) body')

where id'is obtained by applying the mapping:
LEXNAM:ID > ID' where
id ∉ ID , and id' ∉ ID'
and ID ∩ ID' is empty.
The inverse function:
UNLEXMAN:ID' > ID is also provided.
and body' results from substituting id' for each free occurrence of id in body, for each id in id* .
note: Substitution never proceedes inside of (SPECIAL id) and (QUOTE s-exp).

Globals are not renamed.

(LAMBDA ((SPECIAL id)) body) ===> (LAMBDA* (id) body)

Mixed lexical and global variables.

```
(LAMBDA([ {id | (SPECIAL id)}¹ ]*)body) ===>
     (LAMBDA*([ {id' | id}¹ ]*)body')
```

Unevaluated arguments.

```
(LAMBDA((QUOTE vari))body) ===>
(MLAMBDA X ((LAMBDA*(Y Z) (CONS
    (COND((IDENTP Y)(REDEF* Y Z))(T Z))
    (LIST(QUOTE QUOTE*)(CADR X)) ))
     (CAR X)    (QUOTE(LAMBDA(vari)body)) ))

     where REDEF* = (LAMBDA(A B)(PROG(U)
          (COND((SETQ U(GET A(QUOTE REDEF*)))(RETURN U)))
          (SETQ U(MKUNIQUEID* A))
          (MAKEPROP A(QUOTE REDEF*)U)
          (COMPILE(DEFINE(LIST(LIST U B)) ))
          (RETURN U)  ))

     and MKUNIQUEID* produces a unique renaming of its
          argument.
     and (QUOTE* s-exp')===> (QUOTE s-exp)
          where each id' in s-exp' becomes
          id = (UNLEXNAM id') in s-exp.
```

The effect of such a macro used as rator is:

```
     ((LAMBDA((QUOTE vari))body)e) ===>
          ((LAMBDA(vari)body)(QUOTE* e))
```

Note: In the following transformations only the rator use is shown.
It should be understood that the LAMBDA-macro first produces an MLAMBDA-expression.

General form for specific arguments unevaluated.

```
((LAMBDA([ {id|(SPECIAL id)|(QUOTE vari)}¹ ]*)body)[ {e|e|e}¹ ]*)
===>
((LAMBDA([ {id|(SPECIAL id)|vari}¹ ]*)body)
     [ {e|e|(QUOTE* e)}¹ ]*)
```

Indefinite number of arguments all evaluated.

```
((LAMBDA vari body)e*)===>((LAMBDA(vari)body)(LIST e*))
```

Indefinite number of arguments all unevaluated.

```
((LAMBDA(QUOTE vari)body)e*) ===>
     ((LAMBDA(vari)body)(QUOTE* (e*)))
```

Indefinite number of evaluated trailing arguments.

```
((LAMBDA({var*}¹ . vari)body){e*}¹ e*) ===>
((LAMBDA({var*}¹ vari)body){e*}¹(LIST e*))
```

Indefinite number of unevaluated trailing arguments.

```
((LAMBDA ({var*}¹ . (QUOTE vari)) body) {e*}¹ e*) ===>
((LAMBDA ({var*}¹vari) body) {e*}¹ (QUOTE※ (e*)))
```

Restricted arguments

```
((LAMBDA ({{ident|(: ident p [structure]¹)}²|structure}³*)
     body) {{e|e}²|e}³*) ===>
((LAMBDA ({{ident|ident}²|x※}³*)
    (COND ((AND p*)
           ((LAMBDA ({[structure]¹structure}³*) body)
            {[ident]¹x※}³*))
          (T (ERROR (QUOTE BAD-ARGUMENT))) ))
  {{e|e}²|e}³*)
```

where the x※ are uniquely generated names
and p is an e. List structured arguments.

```
((LAMBDA ({x|structure}¹*) body) {e|e}¹*)
```

where x is any vari other than structure
and structure = (vari*) | (vari* . vari)

```
===>
((LAMBDA ({x|u※}¹*)
   ((LAMBDA ({{vari*}²|{vari*}³ vari}*) body)
    {{(CAR u※) (CAR (SETQ u※ (CDR u※)))*}²
     |{(CAR u※) (CAR (SETQ u※ (CDR u※)))*}³ u※}* ) {e|e}¹*)
```

These baroque extensions have been included because they are commonly present in Lisp systems. They should in no way be thought of as part of the quintessence.

Storage Model

In order to model the storage organization and management concepts of the LISP system, it will be convenient to expand our concept of a data object. The method used is mainly due to Kurt Walk (ref. SIGPLAN Notices, Feb. 1971). This expanded concept of a value object shall encompass the fact that the object is represented in a store or memory M. It will be convienent to use 'M' and call it memory to avoid confusion with the state component 'S' (stack).

We consider the state of a memory described by a mapping M :

M : L ---> V .

From a set L of locations to a set V of values. We say that l ∈ L has the value v ∈ V, or is associated with v, or has the properties v, if

$\underline{M}(l) = v$, also written $v^{\underline{M}}_{l}$.

The data objects of LISP and also the LISP expressions are called <u>s-expressions</u> which are either <u>atoms</u> or ordered pairs of s-expressions called <u>pairs</u>. These are denoted by two kinds of locations: <u>atomic-locations</u> and <u>pair-locations</u>.

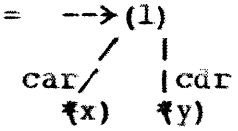
We call the two components of a pair the <u>car</u> and <u>cdr</u>, respectively.

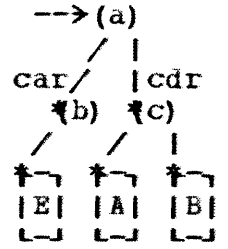
We may characterize <u>V</u>, the set of values for Lisp, as:

"The set of rooted labeled directed graphs with vertices with 0 outgoing edges (atoms), and vertices with 2 outgoing edges (pairs), each vertex labeled with a unique l $\in$ <u>L</u>, and the edges labeled with the component selectors car and cdr, and with exactly one distinguished root vertex".

Pictorially:

Z = --->(l) or  for the atom Z at location l.

l:(x • y) =  for the pair car(l)=x, cdr(l)=y at location l.

a:(b:(E•c:(A•B))•c) = 

Up to this point we have characterized <u>V</u> as a set of unique elements. It is usual that we provide functions on <u>V</u> and these give rise to the notion of equivalence relations. (ref. Birkhoff, MacLane. Algebra).

Given an equivalence relation R on a set V, the equivalence-class under R of any element x ∈ V is:

$$p_R\, x = \{y \mid y \in V \text{ and } yRx\}$$

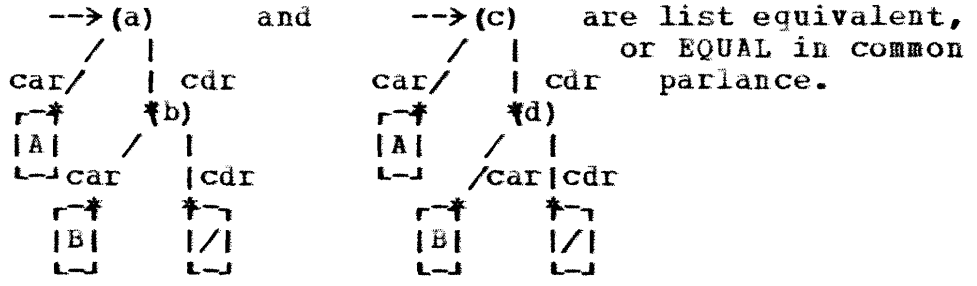
The set of all possible equivalence classes for R is written as:

$$V/R = \{C \mid C \text{ is a subset of } V \text{ and } C = p_R\, x \text{ for some } x \in V\}$$

and is called the quotient set of V by R.

Each equivalence relation R partitions V into equivalence classes and we often choose to consider each of these classes as a data object. We shall call these the R-classes.

For example



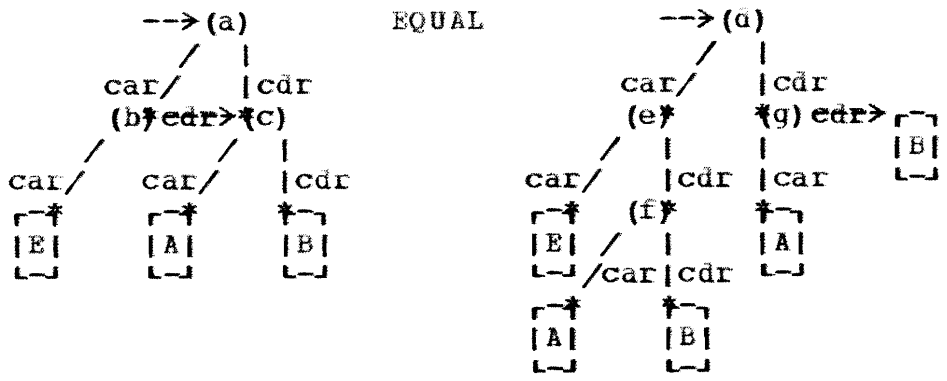
they both represent the list whose linear form is (A B).

This brings us to the question of linear forms. Until now LISP has survived on a linear form data definition as follows:

s-exp = atom | (s-exp* [• s-exp])

but this definition only allows trees. The sharing of subcomponents is undefined. In practice sharing was allowed but not revealed.

Thus,



and they both had the following linear form on printout.

((E A . B) A . B)

We call this representation the tree form of a list structure without cycles. On input this form would generate a structure similar to (d) rather than (a).

There is one equivalence relation namely EQ which is of singular importance because it takes each element of V into a separate equivalence class.

$$V/EQ = V .$$

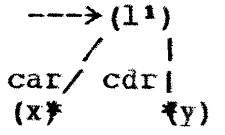
The EQ function is defined as follows:

x .EQ y is true for x =--->(l^1) y =--->(l^2)

if and only if (l^1) = (l^2) .

A linear form for the EQ-class objects would require that each node pair was appropriately labeled,

Thus,



would require the EQ-form:
$l^{1'}$:(x' . y')
where x', y' are the linear forms for x,y; and $l^{1'}$ is some uniqueness preserving canonical representation of l^1.

EQ-form expressions would require labels isomorphic to locations. Thus, EQ-forms explicate not only the intrinsic properties of a value object but also those properties attendant to its addressability. For this reason and for reasons of storage management, EQ-form is not supported as an external

representation form.

The R-classes for other equivalence relations can be simply understood in terms of a new linear form (R-form) which is the result of some transformations on the EQ-form.

For instance in the case of list structures without cycles, labels may be dropped and label references replaced by a copy of the denoted subform. This yields the tree-form.

The following transformations will be applied to EQ-class objects to produce the EQUAL-form:

1. Label renaming - The location (l) of l:(x • y) is replaced by a label ($L^{\prime}$) according to some rule. Such a rule will usually be desirable as it takes one from the domain of locations to the domain of structures. Unless otherwise stated the convention will be that labels will be systematically renamed by READ and PRINT. Labels will be issued from left to right from the list (L^1,L^2,...) by PRINT. READ on the other hand will uniformly substitute new locations for labels.

2. Unmentioned label removal - any label which occurs only once will be removed, i.e. it names a bracket but is not referenced elsewhere in the s-expression.

3. Acyclic label removal - any label which is not mentioned within the scope of parenthesis labeled by that label may be removed, and the resulting expression substituted for any occurrences of that label as a reference outside that scope.

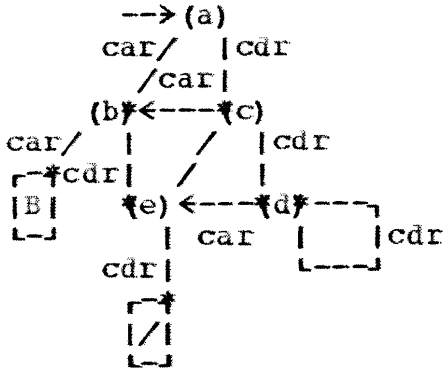
Example:

(... c:(A • B) ... c)

(... (A • B) ... (A • B)).

This transformation will be considered optional. If omitted all sharing is revealed. If applied only the oriented cycles are revealed. For normal Lisp usage the application of this transformation is appropiate. Indeed one must apply it to remain consistant with past usage. We call this list-structure form.

A complex example:



has the following list-structure form with canonical labels.

$(L^1:(B \bullet L^2:((L^1 \bullet L^3:(L^2 \bullet L^3))))L^1 \bullet L^3)$.

We intend that the LISP functions READ, PRINT, and EQUAL should be upgraded as follows:

READ should allow arbitrary use of labels.

PRINT should normally write List-structure-form.

EQUAL should define List-structure-equivalence.

There are three major operations on storage: allocation, freeing, and storing.

Allocation is the inclusion of new locations in the set L of current locations of M. Allocation is described as an operation on an initial M^0. Let l be a location which is independent of all locations in L^0 of M^0,

then allocate$(l)(M^0) = M^{0'}$

where $M^{0'} : L^0 \cup \{l\} \longrightarrow V$

$M^{0'}(l)$ is undefined --- i.e. no value is assigned.

$M^{0'}(l') = M^0(l')$ for $l' \in L^0$

The independence relationship (indep) between two locations

l and l' has the following properties:

(l .indep l) is false,

if (l .indep l') , then also (l' .indep l)

if (l .indep l') , then also (l .indep l'_i)

for all components l'_i of l' .

The major problem of allocation is that memory is finite, so the supply of new l becomes exhausted. The solution to this problem is reclamation, which is described later. Allocation is otherwise well behaved. The set of value objects remain unaffected.

Freeing a location $l \in \underline{L}^o$ correspondingly is defined by:

free(l) ($\underline{M}^o$) = $\underline{M}^{o'}$

$\underline{M}^{o'} : \underline{L}^o - \{l\} \longrightarrow \underline{V}$

$\underline{M}^{o'}(l') = \underline{M}^o(l')$ for $l' \in \underline{L}^o - \{l\}$.

The problem associated with freeing is that:

EQUAL($v^{\underline{M}^o}_{l'}$; $v^{\underline{M}^{o'}}_{l'}$) may become undefined.

This is because $v^{\underline{M}^{o'}}_{l'}$ may contain a subcomponent $v^{\underline{M}^o}_{l}$ which is no longer defined.

Thus freeing must be done with some care.

The process of reclamation occurs when the set of locations from which we are allocating becomes exhausted. The set $\underline{L}^o$ is defined as $\{l \mid \neg(l \text{ .indep } (\underline{S};\underline{E};\underline{C};\underline{D})) \text{ and } l \in \underline{L}\}$. The available cells are then $\underline{L} - \underline{L}^o$.

Storing a value v in a location l of $\underline{M}$, $v \in$ range(l), $l \in \underline{L}$, is defined by:

store(l;v) ($\underline{M}$) = $\underline{M}'$

where

$\underline{M}' : \underline{L} \longrightarrow \underline{V}$

$\underline{M}'(l) = v$

$\underline{M}'(l') = \underline{M}(l')$ for $l' \in \underline{L}$, $(l'$ .indep $l)$.

However,

$\mathrm{EQUAL}(v_{l'}^{\underline{M}}; v_{l'}^{\underline{M}'})$ may be false because of a changed subcomponent.

In Lisp two storing functions RPLACA and RPLACD are provided and defined by:

$$\mathrm{RPLACA}(v_{l^1}^{\underline{M}}; v_{l^2}^{\underline{M}'}) = v_{l^1}^{\underline{M}'}$$

where l^1 is a pair-location, and $\underline{M}$ becomes $\underline{M}'$ and,

$\underline{M}(l^1) = (x \bullet y)$

$\underline{M}'(l^1) = (v_{l^2}^{\underline{M}'} \bullet y)$

$\underline{M}'(l') = \underline{M}(l')$ for $l' \in \underline{L}$, $(l'$ .indep $l)$.

similarly

$$\mathrm{RPLACD}(v_{l^1}^{\underline{M}}; v_{l^2}^{\underline{M}}) = v_{l^1}^{\underline{M}'}$$

$\underline{M}(l^1) = (x \bullet y)$

$\underline{M}'(l^1) = (x \bullet v_{l^2}^{\underline{M}'})$

$\underline{M}'(l') = \underline{M}(l')$ for $l' \in \underline{L}$, $(l'$ .indep $l)$

Issues and Comments

The model given above is intended to model the LPL-Lisp system in its present state (extending slightly into the future). Certain important notions namely, val, pairlis, and error

recovery, were treated somewhat informally.

The introduction of two classes of function abstraction(LAMBDA, and NLAMBDA) has led to a more complex formal definition than is usual for Lisp. It seems important however to bring these long term denizens of Lisp systems to first class status.

The model described above treats GO, and RETURN as first class objects. This is at variance with current pragmatics wherin they are treated as strictly local to a PROG.

It is noted that the formation of label-closures in this formalism is somewhat restricted and that a label-closure forming expression should be added to the language to make labels first class citizens.

It has been suggested by H. F. Ledgard (ref. MAC-TR-60 (Thesis)) that the lambda-calculus formalism should be combined with a rewrite or production system. That these "Markov algorithms" explicate the notion of an algorithm operating on a string and are ideal for describing structure transformations. The productions are used to define primitive structure transformations, and the lambda-calculus is used to define new functions from these primitives. The current model does not describe such a combined formalism because pragmatically it does not yet exist. It should.

APPENDIX A.

Common LISP Function Descriptions: In the following description all functions are shown as rator (operator) in a (rator rand*) combination.

(AND x1 x2 ... xn) Predicate

AND is a special form with an indefinite number of arguments. Its arguments are evaluated in succession until one of them is found to be NIL. In this case the value of AND is NIL (false). If no argument evaluates to NIL, the value of AND is not NIL. The value of (AND) with no arguments, is true.

(APPEND x y) SUBR

If x is not an atom, APPEND returns a copy of x in which y replaces the CDR of the last cell at the top level. If x is an atom, APPEND returns y.

Examples:
1) APPEND ((A B) (C D)) = (A B C D)
2) APPEND ((A B) C) = (A B . C)
3) APPEND ((A . B) C) = (A . C)
4) APPEND (A (B C)) = (B C).

For both arguments in the form of lists (Example 1), the result is the same as in 7090 LISP. The other cases are undefined and cause errors in 7090 LISP.

(APPLX fn args) SUBR

APPLX applys the function fn to the arguments given in list args. The arguments are not evaluated but are given to fn directly. If fn does not directly denote a function (i. e. it is not the name of a assembled program, or if it has no such property as EXPR, EXPR*, FEXPR, or FEXPR*), then fn is treated as an expression (form) to be evaluated and that value is APPLX'ed to args.

During this application the SPECIAL free variable $ALIST defines the environment.

(APPLY fn args a) SUBR

APPLY applys fn to args in the environment a.

(ATOM x) SUBR Predicate

ATOM returns *T* (true) if x is any atom, and NIL otherwise. (ATOM x) evaluates to NIL if and only if x is a node.

(ATTRIB x e) SUBR

ATTRIB concatenates x to e using NCONC. The value of ATTRIB is e.

(CAR x) SUBR

CAR gives the first element of a list x, or more precisely, the left element of a node (node means dotted pair). CAR assumes that x is a node, i.e. a pair of contiguous half-word word-locator fields; the value of CAR is the pointer equivalent of the contents of the first of these fields.

CAAR _ CDDDDR are all defined in LISP as composition functions of CAR and CDR.

(CASEGO e (q1 lb1) (q2 lb2) ...(qn lbn)) MACRO

This CASEGO operator is equivalent to the expression
(COND((EQ E (QUOTE q1))(GO lb1))
((EQ E (QUOTE q2))(GO lb2))
... ((EQ E(QUOTE qn))(GO lbn))) but is some what more efficient. As CASEGO expands into a *CODE expression it cannot be executed interpretively.

(CDR x) SUBR

CDR gives the tail of a list x (the rest of the list after the first element). More precisely, CDR gives the right element of a node. CDR assumes that x is a pointer to a pair of contiguous half-word word-locator fields. The value of CDR is pointer equivalent of the contents of the second of these fields.

CAAR-CDDDDR are all defined in LISP as composition functions of CAR and CDR.

(COMMON x) SUBR

The list x contains the names of variables that are to be declared COMMON. The flag COMMON is put on the

property list of each literal-atom in x. At compile time variables which have been declared common are bound on the A-list. Assignment and variable evaluation by the compiled function takes this into consideration. (See chapter X, section I. Environment conventions, representations and problems.)

(COMPILE x) SUBR

The list x contains the names of previously defined functions. They are compiled. A name denotes a defined function if it has the property EXPR or FEXPR*.

COMPILE also recognizes the property FEXPR* for functions which recieve a list of unevaluated arguments. FEXPR* is equivalent to the FEXPR of M.I.T. LISP 1.5, McCarthy et. al, that is, they are treated as two-argument functions. The first argument is a list of unevaluated rands; the second is the A-list which pertained at the time the combination was evaluated.

(COMP360 x) SUBR

The list x is the same as for DEFINE. A list of function defining forms. They are compiled directly.

(CONC x1 x2 x3 ... xn) SUBR*

CONC acts like an APPEND of many arguments and concatentates its arguments onto one new list. The first n-1 arguments are copied; the last argument is not.

(COND (p1 e1) (p2 e2) ...(pn en)) Special form FSUBR*

The special form COND takes an indefinite number of argument clauses in the form of pairs (pi ei), where pi is a predicate and ei is an expression.

The parentheses in COND have a different meaning than they do in any other LISP form in that (pi ei) does not mean to apply function pi to arguments. Instead, pi's are evaluated from left to right until the first one, say pt, is found that evaluates to true (more precisely, whose value is not EQ to NIL). The value of the entire COND is that of the associated form et; all other ei (i $\neq$ t) and pi (i > t) are not evaluated. pi may be any form in LISP except the specific statement forms (GO label) or

(RETURN value) since these yield no value.

If COND is used anywhere except as a top level statement within PROG, then at least one of the pi must be true (typically, the last pn is the atom T). If none are true, an error will be detected at operate time. (If none of the pi are the atom T, the compiler inserts the pair (T (*CONDERR)) at the end of the COND.)

If COND is used at the top level of a PROG, then it is a statement executed for effect not value (except when an ei is of the form (RETURN value)), and the following differences occur:

1. The requirement that one of the pi be true is waived. If all pi are NIL, the COND falls through to the next statement.

2. Any of the forms ei may be of the form (RETURN v), where v is an expression to be evaluated and is the value of the PROG.

3. Any of the forms ei may be a statement of the form (GO l) where l must be a label which is used in this PROG (see PROG).

(CONS x y) SUBR

CONS is the basic function for allocating list nodes. It creates a node whose CAR is x and whose CDR is y and returns that node as its value. More precisely, CONS takes a full word field from available space, maps the pointer values x and y into half-word word-locators, and places these in the left and right half-word portions of the word, respectively, and returns the pointer to the word as value.

(COPY x) SUBR

This function makes a copy of the list x.

```
(COPY(LAMBDA(X)(COND
        ((ATOM X) X)
        (T(CONS(COPY(CAR X))(COPY(CDR X))))   )))
```

(CSET a v) SUBR

This function is used to establish a zero-level special binding of an identifier. It also creates an APVAL property (see Chapter X. section on the APVAL kludge).

CSET (PI 3.14159) sets the value of the id PI to the value 3.14159. (Note that both arguments of CSET are quoted by Evalquote.

The form (CSET a v) produces the following results: If a is not an id (i.e., does not have a literal id as its value) an error is detected; otherwise a is treated like a Special and the value cell of the id which is the value of a is given the value of v.

(CSETQ a v) Special Form

This special form is like CSET except that it quotes its first argument which must be an atom.

(DEFINE x) SUBR

The argument of DEFINE, x is a list of k pairs

((n1 d1) (n2 d2) ... (nk dk)),

where each ni is a name of a function and di is the corresponding LAMBDA-expression for the function.

The execution of DEFINE is as follows:

(DEFINE (LAMBDA (L) (DEFLIST L (QUOTE EXPR))))

(DEFLIST x ind) SUBR

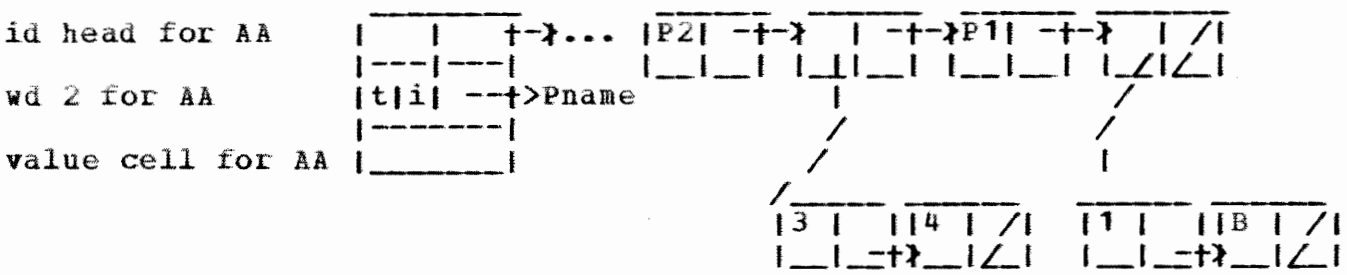
The first argument of DEFLIST x is a list of pairs ((n1 d1) (n2 d2) ...), as for DEFINE, and the second argument ind is an atom. DEFLIST places each expression di on the property list of the corresponding id ni under the indicator ind.

If DEFLIST is used twice on the same id with the same indicator, the old expression on the property list is replaced by the new one. DEFLIST places new properties on the property list to the left of all old properties.

For example:

DEFLIST (((AA (1 B))) P1) DEFLIST (((AA (3 4))) P2)

results in the following structure for id AA



DEFLIST treats the ind values EXPR, EXPR*, FEXPR, or FEXPR* as very special cases in which case it also resets the type code t of id n to no longer indicate that n is the name of a compiled function. It also sets the value cell of n to point to the assembled function INTERFACE.

(DELETEL b m) SUBR

DELETEL deletes from list m all elements which are members of list b, and reCONSes the remaining elements into a new list. It does not change m. (DELETEL (LAMBDA (B M) (MAPCON M (FUNCTION (LAMBDA (J) (COND ((MEMBER (CAR J) B) NIL) (T (LIST (CAR J)))))))))). The new list is returned as the value of DELETEL.

(ED fn) SUBR CMS

Invokes the file editor CEDIT (a CMS command) for filename fn whose filetype is assumed as SYSIN.

```
(ED(LAMBDA(X)(PROG2(SETQ INFILE X)(EDIT))))
```

(EDIT) SUBR CMS

Invokes the file editor CEDIT for the SYSIN file named by the special value INFILE.

(EFFACE x l) SUBR

Deletes the first occurrance of x from the list l. Note: this function alters the existing structure l. Example: EFFACE(A (B A C A)) = (B C A)

(ELT vec n) SUBR

ELT is the basic indexing function for contiguous storage vectors. The first argument vec must be a vector, otherwise an error will occur. The index n is

zero-origined. i.e. the first element of a vector is given by the index 0. Index n should be less than 4095 for string data, otherwise less than 1023.

(EMBED fname redef) SUBR

The EMBED function provides an unusual facility whereby one may redefine a function whose name is fname and get to use the old definition of fname in the redefinition. The redefining LAMBDA-expression is given as the s-expression value of redef. Furthermore, the UNTRACE function will cause the previous definition to be restored. For example, to cause the single argument of a function FOO to print out before execution, one might issue the following doublet:

```
EMBED(FOO(LAMBDA(X)(PROG2(PRINT X)(FOO X))))    ,
```

To later remove this embedded definition one issues UNTRACE((FOO)).

(EQ x y) SUBR

EQ tests for equality of two pointers. If the values x and y are the same pointer, the result is T. Otherwise, the result of EQ is NIL.

(EQUAL x y) SUBR

EQUAL tests x and y for equality. Two data structures are EQUAL if and only if:

x and y are the same pointer value,
or if x and y are numbers, then (*EQP x y) is true,
or if x and y are list nodes, then (EQUAL (CAR x) (CAR y)) and (EQUAL (CDR x) (CDR y)),
or if x and y are vectors whose type, length and contents are identical,
or x and y are pointer vectors of the same length whose corresponding elements are EQUAL,
or x and y a n-tuples and (N*TUPLEQ x y) is true,
or x and y are vectors of reals whose corresponding elements are *EQP.

(ERROR msg) SUBR

ERROR causes the value of its argument msg to be printed and then induces an error unwind of the LISP system.

(ERROR2 msg1 msg2) SUBR

ERROR causes the value of both of its arguments to be printed and then induces an error unwind of the LISP system.

(EVA1 exp) SUBR

EVA1 performs the evaluation of the s-expression exp. The evaluation is performed with respect to the environment (as represented by the current state of special values, APVAL's and $ALIST).

(See chapter X EVALUATION...)

(EVAL exp a) SUBR

EVAL simply sets $ALIST to a and returns (EVA1 exp)

(EXCISE id) SUBR CAUTION

The literal atom id is destroyed and its space is returned to the list of available identifiers. EXCISE is used intelligently by the UNTRACE function. Users beware.

(FLAG k ind) SUBR

FLAG puts the flag ind on the property list of every id in the list k. The value of flag is NIL. No property list ever recieves a duplicate flag. (see REMFLAG)

(FUNCTION fn) Special Form

FUNCTION is used to transmit functional values. fn can be the name of a true function (not a Macro or Special Form) or a LAMBDA or LABEL expression for a function. The value of such a FUNCTION expression is the closure:

(FUNARG fn $alist)

where $alist is the value of the special variable $ALIST at the time the closure was formed. Note, this is not a complete adequate representation of the environment as it captures only a-list or COMMON variables. Refer to the section on LAMBDA expressions

for naming conventions which should overcome the systems inability to capture the name-value relationship for other than a-list variables.

FUNCTION expressions may occur anywhere an expression is allowed.

(GENSYM) SUBR

Each call to (GENSYM) generates a fresh and distinct atomic symbol of the form %G00001. GENSYMS are not placed on the OBLIST and are collected by the garbage collector if they are not in use. The initial letter g for gensyms is given by the special value GENSYMCH which is normally G.

(GET x y) SUBR

GET searches list x for an element EQ to y. If the test succeeds, GET returns the CADR of the list (i.e., the next element on the list). If the test fails, GET returns the value NIL. GET treats the properties, APVAL, PNAME, SUBR, SUBR*, FSUBR, FSUBR* in a special manner as they are indicated in the identifier structure itself instead of occurring on the property list. The use of GET does not change because of this, but the user must refrain from altering the property list directly to achieve these properties. The simplest advice is to use MAKEPROP for attributing values to identifiers.

(GETBPI n) SUBR MACRO

GETBPI gets from available vector space an empty assembled program vector sufficient to accommodate n bytes of code. (see section F on Contigious Storage Vectors).

(GETCH strng n) SUBR

GETCH is a character string indexing function. n = 0 to retrieve the initial character of the string. The value of GETCH is the corresponding character object. No check is made to see if n is too large.

(GETCVEC n) SUBR

Returns a symbolic vector of n/4 elements. Each element is initialized to NIL.

(GETDEF l) SUBR CMS

Retrieves and defines the definitions of the functions whose names are given in the list l. The definitions are retrieved from the file named by the special variable INFILE. The value of INFILE is of the form : (filename {filetype | * })

Note: ED leaves INFILE selected to the last file edited.

(GETIVEC n) SUBR MACRO

GETIVEC gets from available vector space an empty vector sufficient to accomidate n/4 integers. The vector is of type 2 (a vector of integer literals).

(GETRVEC n) SUBR MACRO

Same as GETIVEC except the returned vector is of type 3 (a vector of literal reals).

(GETSTR n) SUBR

GETSTR returns as value a string vector sufficient to hold n characters. The last word of the vector is initialized to zeros, the fill code for strings.

(GETSVEC n) SUBR MACRO

Returns an uninitialized pointer vector. See GETCVEC.

(GO l) Special Form PROG only

GO is a special form valid only within PROG. (GO l) causes the flow of the program to move to the label l within the PROG. See PROG.

(IDENTP id) SUBR

IDENTP is *T* if id is an identifier else NIL.

(INTERFACE) SUBR LAP

INTERFACE is a routine which allows the compiled code to call interpreted functions.

DEFINE puts the address of INTERFACE into the special value cell for each identifier being defined.

The compiler treats all calls as though they were to compiled functions. INTERFACE assumes control and performs the interfacing tasks.

(IVECP v) SUBR MACRO

Returns true if v is a vector of literal integers.

(LABEL name (LAMBDA (bv*) body)) Special Form

LABEL is a special form used to give a LAMBDA-expression a name so that it can be called recursively from within the LAMBDA-expression. LABEL causes compilation to occur in a manner similar to DEFINE in LISP, but with two differences:

1. the name used in LABEL is local, and can be seen only within this LABEL form. It thus can cause no conflict with other functions or atoms used in the system.

2. LABEL defines a single function, while DEFINE can take many function expressions.

(LAMBDA (bv*) body) LAMBDA-expression Special form

A LAMBDA-expression is used to specify a LISP function. The bv are bound variable names, which are identifiers in the normal case where the corresponding argument expression is assumed to be pointer valued.

A LAMBDA-expression is not by itself a complete (or closed) specification of a function. Variables which occur free in body must have their name-value relationship specified for the function specification to be closed.

The LAMBDA-expression can be considered as an expression which evaluates to a closure. The closure is given as the s-expression form (FUNARG LAMBDA-expression environment) which combines the LAMBDA-expression with the environment. The environment gives the name-value relationships for variables which occur free in body.

In interpreted LISP, variables are bound on the a-list. The a-list has several desirable characteristic as a model of the environment: Firstly, all name-value relationships which pertain are captured by a single pointer value. Thus, saving the environment is a simple matter of saving this pointer. Secondly, binding does not destroy existing environment but just appends new name-value relationships to it. The drawback of using the a-list is that evaluation of a variable requires a search.

In compiled LISP variables are treated in several ways:

1. Bound variables which are not declared SPECIAL or COMMON are treated as purely local (or dummy) variables. The compiler assigns a stack location relative to the stack frontier to contain the value of the local variable. Such variables are said to occur vacuously and cannot be referenced by name as free variables in functions called by the defined function. The use of these variables by compiled functions is very direct and efficient.

2. Variables declared SPECIAL have their values associated with their corresponding identifier structure in the value cell. These identifier value cells can be referenced directly by any function and thus provide an efficient device for free variables for compiled code. It is necessary that any compiled function which binds such a variable must first save the old value and restore it upon exit. While this gives this global enviornment the same binding characteristic as the a-list, a given environment cannot be captured so easily.

Remark: This presents a rather serious problem in forming closures. The traditional solution in LISP has been to require that the user declared COMMON any variable which required the correct closure mechanism.

3. Variables which occur free as operator in combinations are treated by compiled functions as if they were SPECIAL, and furthermore they were assumed to have a value which is an assembled program.

4. Variables which have been declared COMMON are bound on the a-list and the interpreter is used to evaluate them. The form (SETQ I J) where I has been declared COMMON is transformed to (SETC (QUOTE I) J) which uses the a-list.

Remark: If only unusual names are used for SPECIAL

variables which are bound by functions having functional arguments (arguments that are closures), then the user avoids conflicts with SPECIAL variables that may be bound at the time the closure is applied. If such is the case the necessity of catching the environment at argument evaluation time is removed.

(LAP360 ...) see chapter VI, The Compiler and LAP.

(LAST x) SUBR

LAST scans a list x and returns the last element at the top level of the list. It will cause an error if applied to an atom or to a list terminated by a non-NIL atom.

Example:

LAST ((A B C)) = C

LAST ((A B (C))) = (C)

LAST (A) = LAST ((C . B)) = LAST ((A B . C)) = error.

(LENGTH x) SUBR

LENGTH applied to a list x returns an integer equal to the number of elements in the top level of the list. Applied to an atom it yields zero.

```
(LENGTH (LAMBDA (M) (PROG (N) (SETQ N 0)
A (COND ((ATOM M) (RETURN N)))
(SETQ M (CDR M)) (SETQ N (ADD1 N)) (GO A)))).
```

(LENGTHCODE vec) SUBR

If vec is a vector, and k is the number of bytes in vec then LENGTHCODE returns the value k-1. If vec is not a vector datum, the value will be some unexpected integer x, where $0 \le x \le 4095$

(LIST x1 x2 ... xn) Special Form

LIST takes an arbitrary number of arguments, and constructs a list out of them. The compiler handles the special form LIST by constructing open code using

the function *LIST.

*LIST calls CONS and the effect is the same as

(CONS x1 (CONS x2 (... (CONS xn NIL) ...)))

Remark: the actual method employed is considerably more efficient in terms of length of compiled code and speed of operation if n > 2.

(LISTP x) SUBR Predicate

LISTP's value is *T* if x is a node, otherwise NIL. Thus LISTP is false for numbers, strings, vectors, n-tuples, literal atoms, and NIL.

(LIST2IVEC l) SUBR

Creates an integer vector whose elements are the literal values of the integers of the list l. No integer vector may be longer than 1023 elements.

(LIST2RVEC l) SUBR

Creates a real vector whose elements are the literal values of the reals of the list l. No real vector may be longer than 1023 elements.

(LIST2SVEC l) SUBR

Creates a pointer vector whose elements are the elements of the list l. No pointer vector may be longer than 1023 elements.

(LIST2VEC l) SUBR

Transforms a list l of the form:

1. (INTEGER x1 x2 ... xn)
   where the xi are integers into a vector of n literal integers.
2. (REAL r1 r2 ... rn)
   where the ri are reals, into a vector of n reals.
3. (SYMBOL s1 s2 ... sn)
   where the si are any S-expression values,
   into a symbolic(pointer) vector of n elements.
4. (s1 s2 ... sn)
   where s1 is not INTEGER,REAL, or SYMBOL,
   into a symbolic vector of n elements.

(MACLAP listing) SUBR

Expands the LAP MACROS which occur as instructions in listing. LAP MACRO definitions are one argument lambda expressions which produce a list of instructions as value. These instructions are in turn expanded.

(MAKEPROP id attr val) SUBR Pseudo-function

MAKEPROP makes val the value of the attribute attr on the property list of id. The attributes PNAME, SUBR, SUBR*, FSUBR , and FSUBR* do not occur on the property list, but their values are embedded in the three word identifier data structure as illustrated for DEFLIST. (see also sec. III-C)

(MAP x fn) SUBR Functional

MAP applies the function fn to x and to successive CDRs of x until x is reduced to a single atom (usually NIL) which is returned as the value of MAP.

```
(MAP (LAMBDA (X FN) (PROG (M) (SETQ M X) LP (COND
((ATOM M) ( RETURN M))) (FN M) (SETQ M (CDR M)) (GO
LP))))
```

MAP cannot be input as the top level function to Evalquote since the functional argument must be evaluated, unless written as in the following examples:

```
MAP ((A B C) (FUNARG PRINT ( )))
produces the following lines of printout:
   (A B C)
   (B C)
   (C)
   VALUE = ( )
```

(MAPCAR x fn) SUBR Functional

MAPCAR constructs a new list whose value is a list of elements each of which is obtained by applying the function fn to the corresponding element of the list x. MAPCAR is non-recursive, and uses ATOM to find the end of the list. MAPCAR cannot be input as the top level function to Evalquote, except as illustrated for MAP. Examples of the use of MAPCAR are:

```
(LAMBDA (L) (MAPCAR L(FUNCTION SUB1))) ((0 1 2 0.3))=(-1
0 1 -0.7)
```

(LAMBDA (L) (MAPCAR L (FUNCTION (LAMBDA (J) (COND ((ATOM J) (QUOTE ATOM)) (T NIL)))))) ((A B (C) D)) = (ATOM ATOM NIL ATOM)

MAPCAR ((0 1 2.3) (FUNARG SUB1 ())) = (-1 0 1.3)

(MAPCON x fn) SUBR Functional

MAPCON maps list x onto a new list ((fn xi)*) using NCONC, so that the resulting list is formed by concatenation, and uses ATOM to find the end of list x.

(MAPCON (LAMBDA (X FN) (COND ((ATOM X) X) (T (NCONC (FN X) (MAPCON (CDR X) FN)))))).

The top level restrictions on the use of functional values as illustrated in MAP hold for MAPCAR also. Because of the use of NCONC, MAPCON will damage the system or will cause an endless loop or both, unless the function fn is chosen carefully. (See DELETEL for an example of the use of MAPCON.)

(MAPLIST x fn) SUBR Functional

MAPLIST maps the list x onto the list ((fn xi)*). It performs the same function as MAP except that it produces an output list by CONSing together all of the results of the form (fn xi) computed during the mapping.

MAPLIST is non-recursive, and uses ATOM to find the end of list x. MAPLIST has functional arguments so the restriction on its use at the top level is the same as MAP.

Example:

(LAMBDA (X) (MAPLIST X (FUNCTION (LAMBDA (J) (CONS (QUOTE B) J))))) ((A B C D)) = ((B A B C D) (B B C D) (B C D) (B D)).

(MDEF exp) SUBR

The expression exp is macro-expanded. The result is that expansion. (See chapter X. III. MACROS)

(MEMBER a b) SUBR Predicate

MEMBER is a predicate which is true if a is a member of list b, and NIL otherwise. EQUAL is used to perform the equality test. Hence MEMBER (1.0 (A B 1

2)) = T.

(N*TUPLEQ ntuple1 ntuple2) SUBR

The corresponding literal elements of ntuple1 and ntuple2 are compared for exact identity. The corresponding pointer elements are compared to see if they are EQUAL. If all elements correspond in this manner N*TUPLEQ has the value true, otherwise NIL.

(NCONC x y) SUBR

NCONC appends list y onto the end of list x without copying x. The value of NCONC is the new value of x. The ATOM test is used to find the end of the list x. If x is atomic, NCONC appends a y onto the end of the property list of atom x.

```
(NCONC (LAMBDA (X Y) (PROG (M) (COND ((NULL X) (RETURN
Y))) (SETQ M X) A (COND ((ATOM (CDR M)) (GO B))) (SETQ
M (CDR M)) (GO A) B (RPLACD M Y) (RETURN X) ))).
```

(NOT x) Special Form

This is regarded as a Special Form by the Compiler and is always changed to the equivalent form (NULL x).

(NQSUBST x y z) SUBR

Substitutes x for all occurrances of y in z, except when z is inside a QUOTE expression. Example: NQSUBST(A B (A B (QUOTE (A B)) A B)) = (A A(QUOTE(A B))A A)

(NULL x) SUBR

NULL is compiled as open code when used as a predicate. For other uses, the definition used is:

```
(NULL(LAMBDA(X) (COND((NULL X)T) (T NIL))))
```

(OBEY strng) SUBR CMS,TSS

The string strng contains a command of the command language for the time shared system. On cp67-CMS one must not use those commands which overwrite memory above 16000x. The command is obeyed (operated).

(OR p1 p2 ... pn) Special Form

The arguments of OR are evaluated from left to right until the first true (non-NIL) predicate is found. If a true predicate is found, the value of OR is T; if the end of the list of arguments is reached, the value of OR is NIL. The value of (OR) of no arguments is NIL.

(PAIR x y) SUBR

PAIR requires its inputs x and y to be lists of equal length.

x = (x1 x2 ... xn) y = (y1 y2 ... yn).

PAIR returns a list of dotted pairs:

((x1 . y1)(x2 . y2) ... (xn . yn))

as its value if this condition is met.

If the two lists are of unequal length, PAIR induces an error.

(PROG vars s1 s2 ... sn) Special Form

PROG is a Special Form that permits LISP programs to be written in the form of a series of statements to be executed. In form, PROG looks like a function of an indefinite number of arguments.

Its first argument vars must be either an empty list or a list of bound variables (v1 v2 ... vn), called program variables. Any program variable which is not in SPECIAL status at compile time is merely a cell on the pushdown stack. If a program variable is in SPECIAL status during compilation, its previous SPECIAL binding is saved on the pushdown stack at entrance to the PROG and is restored at exit, and the current binding is stored in the corresponding identifier value cell. Thus in either case, the binding of a program variable is visible only within the PROG. However, if the variable is SPECIAL, it is also visible when used free by any function called from within the PROG. If not SPECIAL, it is invisible except in the body of the PROG.

The other arguments s1 s2 ... sn of a PROG can be either identifiers (id) or statements.
A statement may be any standard LISP combination (rator rand*) but it may also include GO or RETURN statements. If there are no GO or RETURN statements, the statements

s1 s2 ... sn are executed by evaluating each statement and ignoring the value. (Id are disregarded since evaluating an id and discarding the value is of no consequence.) The control "falls through" the end of PROG with NIL as the value.

The form (GO l), where l is a label within the PROG can occur at the top level of the PROG as one of the si or can be used at the top level of a COND which itself is at the top level of the PROG. If evaluated, (GO l) causes transfer of control to the label l in the PROG.

The form (RETURN v) can occur under the same conditions as (GO l) but causes v to be evaluated, and exit from the PROG with v as the value of the PROG.

Within a PROG, COND does not require a T alternate, since control simply "falls through" to the next statement.

(PROG2 a b) SUBR

PROG2 causes its first argument to be evaluated and then returns the value of its second argument. It is defined by: (PROG2(LAMBDA(X Y)Y))

(PROP x y fn) SUBR Functional

PROP searches the list x for a property EQ to y. If one is found, the value of PROP is a pointer to the CDR of the list. If the property is not found, the value of PROP is (fn), a function of no arguments.

NOTE: Because of the special treatment of the properties PNAME, APVAL, SUBR, SUBR*, FSUBR*, and FSUBR which formerly occurred on the property list, PROP will not work the same on these values of y.

(PUNWORD h1 h2 h3 loc n) SUBR

PUNWORD is a function used by LAP360 to plant code in the core. n bytes are planted starting at location BPORG + loc. BPORG is a special variable. h1, h2, h3 are integers whose low-order 16 bits are concatenated to form a 48 bit setting (6 bytes), of which the high-order n bytes are planted. n < 6.

(QUOTE v) SUBR Special Form

The value of the Special Form QUOTE is the CADR of the

list whose CAR is the id QUOTE. Thus, when evaluated,

(QUOTE A) = A

(QUOTE (A B)) = (A B) , etc., but

(QUOTE A B) = (QUOTE A . B) = A

A quoted expression stands for itself, and is not evaluated.

In compiled LISP the form (QUOTE A) is represented by a quote cell which contains the pointer value A; similarly the form (QUOTE (A B)) has a quote cell which contains the value (A B). NOTE: Quote cells are shared and must not be altered.

Constants, T, F and NIL need not be quoted in LISP because such a constant n is treated as (QUOTE n). (F becomes (QUOTE NIL))

(RECLAIM) Pseudo-function

causes an explicit garbage Collection.

(REMFLAG l flg) SUBR

REMFLAG removes all occurrences of the flag flg from the property list of each identifier in the list l.

(REMOB id) Pseudo-function

This function is used by the system to remove literal atoms which are known to be inactive. It should not be used by the casual user.

(REMPROP x ind) Pseudo-function

Removes the attribute and value of x for the attribute ind. This function alters property lists to undo the effects of ordinary MAKEPROP's.

(RETURN exp) Special Form

If (RETURN exp) is encountered in evaluation of a PROG, the expression exp is evaluated and its value is the value of the PROG.

(REVERSE a) SUBR

The function REVERSE has for its value a list whose elements are the top level elements of list a taken in reverse order, e.g.,

REVERSE ((A (B C) D (E F))) = ((E F) D (B C) A).

When applied to an atom or to a list terminated by an atom other than NIL, REVERSE is undefined.

(RPLACA x y) SUBR

RPLACA replaces the CAR of the cell pointed to by x with the half-word word-locator equivalent to pointer y. Its value is x but x has been replaced by (CONS y (CDR x)).

The use of RPLACA on an id or vector can be disastrous. Likewise, altering quoted data structures can be disastrous.

(RPLACD x y) SUBR

RPLACD replaces the CDR of the cell pointed to by x with the half-word word locator equivalent to pointer y. Its value is x but x has been replaced in value by (CONS (CAR x) y).

The use of RPLACD on an id will alter its property list and this may be disastrous. The use of RPLACD on the header of a vector probably will be disastrous.

(RVECP vec) SUBR MACRO

RVECP returns the value true if vec is a vector of literal reals. For any other argument the value is false.

(SASSOC x y fn) SUBR functional

SASSOC searches y which is a list of pairs (usually but not necessarily dotted pairs), for the first pair whose first element is EQ to x. If the search succeeds, the value of SASSOC is the pair. If the search fails, the value of SASSOC is the value of (fn), a function of no arguments.

Because of its functional argument, SASSOC cannot be input as a function at the toplevel of Evalquote.

```
(SASSOC (LAMBDA (X Y FN) (PROG ( ) A (COND ((NULL Y)
(RETURN (FN))) ((EQ (CAAR Y) X) (RETURN (CAR Y))))
(SETQ Y (CDR Y)) (GO A)))).
```

(SELECT ao (a1 e1) (a2 e2) ... (an en) eo) Special Form

The expression ao is evaluated, then each of the ai are evaluated in turn and tested until the first one is found that satisfies (EQ ao ai). The value of SELECT is then the corresponding ei. If no such ai is found, the value of SELECT is eo.

SELECT can be used at the top level of PROG in much the same way as COND. In this application GO and RETURN forms are not legal for ei and eo. Also, eo cannot be omitted, but maybe NIL. The compiler converts SELECT to the equivalent form

```
((LAMBDA (G) (COND ((EQ G a1) e1) ((EQ G a2) e2) ...
((EQ G an) en) (T eo))) ao)
```

where G is an arbitrary gensym. (If eo were omitted, the syntax of the COND would be incorrect.)

(SETC a b) or (SET a b) SUBR

SETC may be used in compiled functions, but it only alters the values which occur on $ALIST. SETC searches the ALIST until the pair (a . c) is found. It then uses RPLACD to change the pair to (a . b).

(SETIV vec n val) SUBR

This is the primitive assignment statement on integer vectors. n is the zero-origin index, vec is an integer vector, and val is a pointer-valued integer. SERIV checks for out of range, and for the correct type of val. The value of SETIV is val.

(SETRV vec n val) SUBR

SETRV is the primitive assignment statement for real vectors. n is the zero-origin index, vec is a real vector, and val is a pointer-valued real. SETRV checks for out of range, and for the correct type of val.

(SETV vec n val) SUBR

SETV is the primitive assignment statement for pointer vectors. n is the zero-origin index, vec is a pointer vector, and val is any datum. SETV checks for out of range.

(SETQ a v) Special Form

SETQ is a special form which evaluates its second argument v and assigns this value, which is also the value of SETQ, to the variable given as its first argument a. In general, a is treated as if it were quoted. If a is not an atom, an error results. If a is not in SPECIAL status and is bound in a function by LAMBDA or PROG, SETQ affects only the cell on the pushdown list of the function. In which case it is said to be a local variable.

If a is in SPECIAL status and has had a previous CSET binding SETQ changes the value of that binding.

SETQ can be used in series to set many variables to the same value as (SETQ X (SETQ Y Z)) which sets both X and Y to the value of Z.

(SPECIAL x) SUBR

The argument of SPECIAL x is a list of identifiers. SPECIAL sets a flag in the identifier structure of each id of x, and puts the attribute SPECIAL on the property list of each id of x, and returns the list x as value.

The SPECIAL flag on an identifier serves mainly to tell the compiler that if this atom is bound by LAMBDA or PROG, the old binding of the atom must be saved and the current binding stored in the value cell (rather than in the pushdown list).

(STRCONC s1 s2 ... sn) SUBR*

STRCONC concatenates the strings given as arguments and creates the resultant string as value.

(STRINGP s) SUBR

STRINGP is *T* if s is string datum, otherwise NIL.

(STRLENGTH s) SUBR

Returns the number of non-fill characters in the string

s. (Fill characters are trailing 00X bytes.)

(SUBST x y z) SUBR

SUBST substitutes x for each occurrence of the list structure y in the list structure z. The function EQUAL is used to perform the test, so that x and y can have the most general form. But z must be a list.

Examples:

SUBST (A B (B C E)) = (A C E)

SUBST (A B (B (B . C) (B))) = (A (A . C) (A))

SUBST (A (B) (C B)) = (C . A)
since (C B) = (C . (B.NIL))

SUBST (A (B) (B C)) = (B C)

SUBST (2 3 (3 4 . 5)) = (2 4 . 5).

(SVECP vec) MACRO - *CODE

SVECP is the predicate used to test if vec is a pointer vector.

(TEMPUS-FUGIT) SUBR CMS,OS

TEMPUS-FUGIT returns the elapsed task-time in thousandths of a second.

(UNCOMMON id) SUBR

Defined by: (UNCOMMON (LAMBDA (L) (REMFLAG L (QUOTE COMMON))))

(UNSPECIAL x) SUBR

UNSPECIAL is similar to SPECIAL. x should be a list of identifiers. For each atom in x, UNSPECIAL removes the attribute SPECIAL and its value from the PROP list. The value of UNSPECIAL is x.

(VECP vec) MACRO - SUBR

The predicate VECP is true if and only if vec is a pointer into vector space.

APPENDIX B.

ARITHMETIC FUNCTIONS AND PREDICATES

The syntax of numerical objects is given below:

```
number = integer | hex | octal | real

integer = ddigit ddigit*

octal = odigit odigit* Q [ integer ]

hex = ddigit hdigit* X

odigit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

ddigit = odigit | 8 | 9

hdigit =ddigit | A | B | C | D | E | F

real = integer { E [ integer ] | .[ integer ][ E
[integer]]}
```

Semantically, hex and octal numbers are treated like integers. On input they become integers MOD 2**32. Large integers may only be input as integers. The reals are stored as floating point numbers and are currently stored as 32-bit words: 8 bits of character plus 24 bits of mantissa.

Arguments of all functions described below are assumed to be numbers unless otherwise stated.

SUBR*s:

The following arithmetic functions allow an indefinite number of arguments:

MAX, MIN, LOGOR, LOGAND, LOGXOR, PLUS and TIMES.

Unless otherwise stated the arithmetic functions are defined for integers, reals or combinations of reals and integers. The result is real if any argument is real, integer only if all arguments are integers. LOGOR, LOGAND and LOGXOR use $FIX on their arguments and produce integers as answers. These logical functions also d not work on large integers.

Other Arithmetic Functions

(ABSVAL x)

Computes the absolute value of the number x.

(ADD1 x)

Adds 1 to X, e.g. (ADD1 (LAMBDA (X) (PLUS X 1)))

(DIFFERENCE x y)

Subtracts y from x, e.g. (DIFFERENCE(LAMBDA(X Y) (PLUS X (MINUS Y))))

(DIVIDE x y)

Equivalent to (LIST (QUOTIENT x y)(REMAINDER x y))

(EXPT x y)

Raises x to the y power. The result is an integer if x is an integer and y is a positive integer; otherwise the value is a floating point number.

($FIX x)

Computes the integer part of x for positive x and - integer-part of -x for negative x.

(LEFTSHIFT x y)

(LEFTSHIFT x y) produces an integer number equal to the literal value of x shifted left by y bits, with zero brought in at the right to replace the shifted bits. If y is negative, a right shift results and zeros are brought in at the left end of the word. If both x and y are negative, the sign of x is not extended, and the resulting value of LEFTSHIFT will be positive. The acceptable range for y is -31 < y < 31.

(MINUS x)

Returns -x for number x.

(QUOTIENT x y)

For fixed-point arguments, the value is the number theoretic quotient, i.e. the value is an integer q

such that x=qy+r where r is the number theoretic remainder, 0≤r≤y-1 . If either x or y is a floating point number, the answer is the floating point quotient.

(REMAINDER x y)

Computes the number theoretic remainder for fixed-point arguments, and floating point residue for floating- point arguments. (see QUOTIENT)

(SUB1 x)

Subtracts 1 from x, e.g.

```
(SUB1 (LAMBDA (X) (PLUS X -1)))
```

Arithmetic Predicates:

(*EQN x y)

Tests two numbers for equality of representation. Thus

```
(*EQN 0 -0)      = T      (*EQN 0  0)      = T
(*EQN 1  1Q)     = T      (*EQN -0  -0)    = T
(*EQN 1  1.0)    = NIL    (*EQN 1.0 1.0)   = T
(*EQN 1Q 1.0)    = NIL    (*EQN 1Q  1Q)    = T
(*EQN 3Q4 30Q3) = T       (*EQN 1.0 1.0E0)= T
(*EQN 1.0  1.00000000001) = T.
```

The last case holds because the last decimal place is lost in the internal representation.

(*EQP x y)

Tests two number for approximate equality. In general:

```
(  T if |x - y| < 3.E-6                )
(                                      )
(  T if |x - y|                        )
(       |-----|     < 3.0E-6 ,        )
(*EQP x y)  =      (       |x + y|  )
(                   )
(     NIL otherwise.                   )
```

(FIXP x)

Is true if x is a fixed point number, and false othewise.

(FLOATP x)

Is true if x is a floating point number, and false otherwise.

(GREATERP x y)

Is true if x > y and false if x ≤ y.

Note: that it is possible for both (*EQP x y) and (GREATERP x y) to be true simultaneously, but (AND (LESSP x y) (GREATERP x y) is always NIL.

(LESSP x y)

Is true if x < y and false if x ≥ y.

Note that it is possible for both (*EQP x y) and (LESSP x y) to be true if either x or y is a floating point number.

(MINUSP x)

Is true if x is negative, and false otherwise.

(NUMBERP x)

Is true if x is a pointer denoting a LISP number, and false otherwise. In particular, NUMBERP(NIL) evaluates to NIL.

(ZEROP x)

Is true if x = 0 and false otherwise.

(ZEROP (LAMBDA (X) (*EQP X 0)))

APPENDIX C.

Input/Output Functions and Predicates

(BACKCHR) SUBR Pseudo-function

Three global special variables CURINBUF, CURCOL and CURCHAR describe the currently selected input file. The value of CURINBUF is the input buffer string; the value of CURCOL is a number n=0,1,2,3,.... which serves as an index pointing to a character within CURINBUF; the value of CURCHAR is the character object corresponding to the character in CURINBUF indexed by CURCOL.

BACKCHR reduces the value of CURCOL by one, and sets CURCHAR to the appropiate character object. This allows backup within the current input record. BACKCHR returns the new value of CURCOL.

BACKCHR does not work properly if CURCOL by any chance is not a small integer. Since BACKCHR is used extensively by read routines, this limits the input buffer size to a maximum of 1024 characters.

If the value of CURCOL is zero or negative, BACKCHR may be issued, but the resulting values are meaningles and should not be used for anything until an application (possibly repeated) of RDCHR makes CURCOL nonnegative again. A backup from one input record into another is not possible.

(BUCKET id) SUBR Caution!!!

BUCKET is normally not to be invoked by the user. BUCKET returns a pointer to the pointer within the OBLIST vector for the bucket corresponding to the identifier id. This is formed by hashing the first 8 bytes of the PNAME of id, dividing by 599 and using the remainder as an offset.

The garbage collector is quite sensitive to pointers into vectors. BUCKET should be used in such a way that the garbage collector does not see the pointer returned by BUCKET. Otherwise, a garbage collection may lead to a program interrupt.

BUCKET is used by REMOB which is used by the garbage collector.

(CHARP c) SUBR, MACRO Predicate

CHARP is a predicate that tests for character atoms. The value of CHARP is *T* if c is a character object and NIL if c is not.

(CHR2FIX c) SUBR

Converts the digit character object c to its fixed point integer equivalent. Undefined if c is not a digit.

(CLEARBUFF) SUBR Pseudo-function

Clears the string which is the value of special variable BUFFO. Clearing consists of setting contents of the string BUFFO to all hex zeros and setting special CURCOLX to 0. CLEARBUFF is used for side effects only, the value returned by it is unpredictable.

The string BUFFO is used by PACK, INTERNX, MKNAM and NUMOB for the creation of numbers and literal identifiers. It is initialized by *SUPERMAN as a string vector capable of holding 132 characters. The value of CURCOLX is a number n=0,1,2,3,.... which serves as an index pointing to the first unused character position in the string BUFFO.

(CLOSE dcb opt) SUBR Pseudo-function OS, TSS

CLOSE issues an OS or TSS CLOSE macro and returns NIL. CLOSE is used by SHUT to close an input or output file. The parameter dcb is the address of a DCB (data control block). The parameter opt is DISP, LEAVE or REREAD for OS, irrelevant for TSS (REREAD being assumed regardless of the value of opt).

CLOSE is normally not to be invoked by the user; SHUT should be called to close files.

(DD fn ft fm) SUBR Pseudo-function CMS

Opens the file described by the identifiers fn, ft, fm, and returns a list of the names of the open files. Opening a file consists of assigning a string as an I/O buffer for the file, creating a CMS file control block, and placing information about the file into the OPENEDFILES list. If fn is the name of an already open file, DD just returns a list of the names of the open files.

An asterisk or a NIL is acceptable in any argument position (a NIL is interpreted as an asterisk). If filename is unspecified for a new file, the default filename is LISPOUT. If filetype is unspecified for a new file, the default filetype is OUT. If filemode is unspecified for a new file, the default filemode is P1.

If the file exists, the I/O buffer length is as

needed for the file, otherwise it is set to the default value of 120. The useful input record length for the file is set to 71 for filetype SYSIN, and to full record length for all other filetypes. The useful input record length determines the number of characters read from each record. It is ignored while doing output.

Input files may be of either F or V record format. Output files are always of F record format. Input records may not be longer than 1024 characters.

The filetypes LOAD, TIN, and TOUT are reserved for special purposes.

The filetypes TIN and TOUT are used to open terminal input and terminal output files, respectively. A filename for a TIN or TOUT file is needed so that other LISP functions could refer to the file after it is opened. Such filename has no significance for the CMS operating system. A filemode for a TIN or TOUT file determines the mode of input or output line processing by CMS, as follows:

```
Input:
   U    Editing, upper-case translation,
                                  blank filling
   *        same as U
   NIL      same as U
   S    Editing and blank filling
   V    Editing and upper-case translation
   T    Editing only
   X    Input line read exactly as is

Output (CMS/360):
   K    Type line directly from CUROUTBUF
   *        same as K
   NIL      same as K
   B    Move line to free storage before typing

Output (CMS/370):
   B    Type line in black
   *        same as B
   NIL      same as B
   R    Type line in red
```

Filetype LOAD causes a core image file to be opened. One issues a function to open a core image file in an anticipation of **DUMP, **LOAD or DUMPOVER. Core image files are not described in the OPENEDFILES list and for that reason their filenames may duplicate those of other open files. However, regardless of name, only one core image input file and only one core image output file may be open simultaneously.

Issuing

```
(DD fn (QUOTE LOAD) (QUOTE INPUT))
```

is equivalent to

(DISKDEF fn (QUOTE SOSTAP) (QUOTE *) 0 (QUOTE LOAD))

Issuing

(DD fn (QUOTE LOAD) (QUOTE OUTPUT))

is equivalent to

(DISKDEF fn (QUOTE SOSTMP) (QUOTE P1) 0 (QUOTE FILE))

Filemodes other than INPUT or OUTPUT are not allowed with filetype LOAD.

The list of the names of the open files returned as value by DD function is obtained from the OPENEDFILES list. If a core image file is being opened the value returned does not contain the name of that file.

(DD strng) SUBR Pseudo-function TSS

Performs the TSS command

DDEF strng

where strng is a string containing a TSS parameter list for the DDEF command. Returns *T* if successful. In case of an error, prints an explanatory message and returns a number which is a TSS error code (See DDEF macro in TSS manuals). DD function defines a data set but does not open it; to open a data set, issue DD first and then either DISKDEF or OPEN*SEQ.

(DDEF strng) SUBR Pseudo-function TSS

Identical to (DD strng)

(DDIN fn) SUBR Pseudo-function CMS

DDIN is a function used to open a LISP input file given by filename only. If a file named fn is already open, DDIN just returns NIL. Otherwise, it searches for an existing disk file with filename fn. If none is found it types a message to that effect and returns NIL. If one is found, the function DD is executed to open that file and DDIN returns a list of the names of the open files.

If several files with filename fn exist, the first file in the order of precedence of filetypes is opened. The order of precedence of filetypes for DDIN is determined by special variable INTYPELIST which is initially set to (LISP SYSIN), giving the highest

precedence to filetype LISP, next highest to filetype SYSIN, and the lowest to any other filetype.

(DDOUT fn) SUBR Pseudo-function CMS

```
(DDOUT (LAMBDA (X)
   (COND ( (SASSOC X OPENEDFILES NIL)  NIL )
         ( T  (DD X (QUOTE OUT) (QUOTE P1)) ) )))
```

(DIGIT c) SUBR, MACRO Predicate

The value of DIGIT is NIL if c is not a digit character object, otherwise the value is c.

(DISKDEF ddnam org opt1 opt2 n) SUBR Pseudo-function OS,TSS

The arguments of DISKDEF are:

ddnam	the ddname of the file to be opened
org	QUEUED | LOAD (for OS),
	LINE | VI | VS | QUEUED | LOAD (for TSS)
opt1	INPUT | OUTPUT | RDBACK | UPDAT (for OS)
	INPUT | OUTPUT | OUTIN | UPDAT (for TSS)
opt2	LEAVE | REREAD | DISP (for OS)
	ignored (for TSS)
n	the length in bytes of I/O buffer

DISKDEF opens a file for reading or writing and returns a list of the names of the open files. The file must have been declared to the system by means of a JCL DD statement (OS), DD command (TSS), or LISP DD function (TSS). Opening a file consists of assigning a string as an I/O buffer for the file, creating a data control block, and placing information about the file into the OPENEDFILES list. If ddnam is a name of an already open file, DISKDEF is unpredictable.

If the value of org is LOAD, the file opened is a LISP core image file, and its name is not included in the list of names of open files returned by DISKDEF. LOAD is used only before issuing **LOAD, **DUMPER or DUMPOVER. Open core image files are not described in the OPENEDFILES list. Only one core image file may be open at a time. If org is LOAD, opt1 must be either INPUT or OUTPUT.

The value QUEUED denotes sequential file organization for OS, is a synonym of VS for TSS (more about TSS file organization and I/O buffers in a separate memo).

For the significance of opt1 see OPEN macro in OS and TSS manuals.

The value of opt2 is used as the second argument of

SHUT when SHUT is called by FINCLOS, which is invoked automatically at the normal exit from LISP. For the significance of opt2 see CLOSE macro in OS manuals. The option opt2 is ignored if the user issues SHUT or CLOSE before exiting from LISP.

The argument n should be a multiple of 4. If it is not, the next higher multiple of 4 is assumed. If it is desired to read only a part of each input record, a value of n smaller than the record length may be used. E.g. the value 72 is frequently used while reading 80 column card images. For LOAD files, n should be 4 (although no LISP buffer is needed for LOAD files, GETSTR in the definition of DISKDEF should be given a valid argument).

(DISKDEF fn ft fm urecl n) SUBR Pseudo-function CMS

This function opens a file for input or output. It is similar in its operation to

(DD fn ft fm)

The first three arguments are the same as for the DD function. The additional arguments are:

urecl := integer | NIL n := integer | NIL | string | FILE | LOAD

For output operations and all core image saving and loading, urecl is irrelevant. For input operations, urecl determines the number of bytes actually read in each record. If urecl is NIL or 0, the number of bytes read is 71 for filetype SYSIN and equal to the buffer length for all other filetypes.

If n is a string, that string is used as an I/O buffer. If n is a positive integer (not greater than 1024), it determines the length of the I/O buffer assigned to the file. If n is not a multiple of 4, the actual buffer length is the next higher multiple of 4. For input files, the buffer length should not be less than the record length of the file. For output files, the buffer length determines the output record length. If n is NIL, the buffer length is determined by the same rules as in the DD function.

If the value of n is LOAD, a LISP core image file is opened for input. If the value of n is FILE, a LISP core image file is opened for output. LOAD should be issued only before a **LOAD. FILE should be issued only before **DUMP or DUMPOVER.

(**DUMPER) SUBR Pseudo-function

**DUMPER saves the current LISP core image as a new LISP system on the core image file open for

output. The file must have been opened via the DD (CMS), DISKDEF or OPEN*SEQ function. The following evalquote doublets will open a file, save a core image on it and close it:

On OS or TSS:

```
DISKDEF (ddname LOAD OUTPUT NIL 4)
**DUMPER ()
SHUT (ddname LOAD)
```

On CMS:

```
DISKDEF (fn ft fm 0 FILE)
**DUMPER ()
SHUT (fn LOAD)
```

There is a global special variable GENER (generation number), the value of which is increased by one in every saving-loading cycle. The value of **DUMPER is the generation number of the system currently in use. When the saved system is later loaded, the first occurring VALUE=n printout shows the generation number increased by one.

(DUMPOVER) SUBR Pseudo-function

DUMPOVER is similar to **DUMPER. It is used to create overlaying systems which have the same DCB or File Control Block environment. Such overlay core images when loaded by LOADLISP or **LOAD do not get their files reopened. As the DCBs and FCBs of a system are "below" and not saved with a core image, while the OPENEDFILES list is saved, strict agreement in the control block environment must be maintained between the system being overlayed and the overlay. One way of guaranteeing this agreement is to write a function to open all the shared files and to have that function executed very early in the creation of each overlay.

If an overlay core image saved by DUMPOVER is used for initial loading, then its files are all opened, as they always are for regular core images saved by **DUMPER.

DUMPOVER should be used after a DISKDEF or OPEN*SEQ function has opened a file for saving. e.g.
CMS --- (DISKDEF fn ft fm 0 FILE)
OS --- (DISKDEF ddname LOAD OUTPUT NIL 4)
As core image files are not included in the OPENEDFILES list, the control block environment restriction does not apply to them.

EODAD Special variable OS, TSS

Contains the name of a function of no arguments which will be executed when the end-of-file condition is encountered.

The value of EODAD is reset from the special value EODADIN by *NEWLIN before a new record is read. EODADIN is initialized to STANDEODADIN.

The value of EODAD is reset from the special value EODADOUT by TERPRI prior to writing out a record. EODADOUT has STANDEODADOUT as its initial value. Which is of no consequence since OS and TSS never reflect an output end-of-file condition to LISP.

(ERASE fn ft fm) SUBR Pseudo-function CMS

If a file 'fn ft fm' exists, a CMS command
'ERASE fn ft fm'
is typed and obeyed, and *T* is returned as the value. If no such file exists, ERASE returns NIL without typing a message. Asterisks and NILs are acceptable as arguments (NILs being converted by ERASE to asterisks).

(**EXPEL n) SUBR Pseudo-function

Causes the binary program for the function named by the identifier whose delta is n, to be copied from vector space to auxiliary storage (OS, CMS) or upper core (TSS). The original copy of the function becomes subject to removal by the garbage collector. There are numerous conditions which may block expelling of a function. **EXPEL returns the length in bytes (including the header) of the function just expelled, or 0 if expelling did not take place (CMS, OS). Delta is the displacement (in bytes) of the intended identifier from NIL. (MKFN id) will return the delta for a given id. Should the expelled function be called, it will be automatically brought back into vector space for execution (OS, CMS) or executed in place (TSS). Binary programs residing outside of the LISP-addressable space are never expelled.

(*EXPEL fn) SUBR Pseudo-function

Expels the binary program for the function named fn. Returns its argument whether the expelling occurred or was blocked. Calls **EXPEL to do the work.

(EXPELFN list) EXPR Pseudo-function

(EXPELFN (LAMBDA (L)

```
(MAPCAR L (FUNCTION *EXPEL))))
```

(EXPLLOC n) SUBR Pseudo-function

Identical to HEXEXP.

(EXPLODE a) SUBR Pseudo-function

If EXPLODE's argument is a number (of any type) or an identifier which is not a gensym, its value is a list of the character objects which comprise the printed representation of the atom a. If the argument is a generated symbol, EXPLODE's value is NIL. For all other data types, the argument's hexadecimal location is returned as a list of 8 character objects. EXPLODE uses UNPACK for identifier arguments and EXPLODE1 for all others.

Note: If an argument is an object outside of the LISP-addressable space, the hex location is correct only if the argument has never been CONSed and then restored by CAR or CDR, before being passed to EXPLODE.

(EXPLODE1 n) SUBR Pseudo-function

EXPLODE1 is used by print routines to convert numbers from the internal LISP representation to character string representation. EXPLODE1 returns as value a string containing the representation of the number n in terms of EBCDIC characters. If n is not a number, it returns a string containing the representation of the hex location of n in terms of EBCDIC characters. EXPLODE1 uses string named PR1BUF as working storage, and unless n is a long integer, returns its value in that string.

Note: If an argument is an object outside of the LISP-addressable space, the hex location returned by EXPLODE1 is correct only if the argument has never been CONSed and then restored by CAR or CDR, before being passed to EXPLODE1.

The exploding of numbers can be controlled by changing the value of the free variable PRTCONTL.

PRTCONTL is a special variable whose value is an integer of the form 65536*h+w
If both h and w are 0, the print format is free (the default case). The quantity w indicates the print field width. If a number does not fit in w columns its high-order digits are truncated. If a number has less than w digits, it is justified to the right. The quantity h controls exploding of floating point numbers. It indicates the location of the decimal point

within the print field of w columns. If w=0, h is ignored. However, if h=255 all floating point numbers are exploded into the E format, regardless of w. If h=255, w should be either 0 or at least as large as 10. PRTCONTL does not affect the exploding of long integers.

(FILELISP fn ft fm) SUBR Pseudo function CMS
(FILELISP ddname) SUBR Pseudo-function OS,TSS

A LISP restart core image is saved. FILELISP opens the file, performs a garbage collection, issues **DUMPER, and shuts the file. The argument(s) identify the saved LISP file.

(FINCLOS) SUBR Pseudo-function OS, TSS

Causes all open files to be shut. This function is automatically executed on exit from LISP.

(FINCLOS) SUBR Pseudo-function CMS

Performs CMS command
FINIS * * *
thus closing all files as far as CMS is concerned and writing out all directories from core to disk. The files remain open as far as LISP is concerned. No reopening is necessary to continue input or output, since CMS will reopen automatically.

(*FINIS fcb) SUBR Pseudo-function CMS

Performs CMS FINIS function on the file described in the file control block fcb and returns NIL. *FINIS is used by RDS and WRS functions when the operation is switched from reading to writing or vice versa.

(FORTLOAD name n) SUBR Pseudo-function

Generates a calling sequence for the FORTRAN compiled function
name(a1 a2 ... an)
so that it can be called by LAP instruction
(CALL name n)
The only possible argument types are single-precision fixed and floating point numbers. Small integers may be not used as arguments. Any small integers must be converted by LAP code to the regular LISP integer representation before a call to name is

made. Long integers are not permitted as arguments either.

The value of the FORTRAN function is returned as a true integer in *AC, or as a floating-point number in fl.point registers 0,1 . The caller should issue

```
(LA *SCR1 (0 6|8))
(BAL *RET (EXTERNAL $PLANTNM))
```

where 6 idicates an integer and 8 indicates a floating point number, in order to convert the returned value to LISP representation.

The FORTRAN function should reside in a program library suitably defined to the operating system. FORTLOAD uses the operating system LOAD macro to bring the FORTRAN function into the core.

In OS and TSS systems, functions established by FORTLOAD are not preserved by FILELISP, **DUMPER or DUMPOVER; FORTLOAD has to be issued again every time a LISP system is loaded.

In CMS systems, functions established by FORTLOAD are preserved in the core images.

(HEXEXP n) SUBR Pseudo-function

Resets the string named PR1BUF to hex zeros, then stores in it the print representation of the hex location of n, and returns the string as the value.

Note: If n is an object outside of the LISP-addressable space, the hex location is correct only if n has never been CONSed and then restored by CAR or CDR, before being passed to HEXEXP.

(INITIALOPEN) SUBR Pseudo-function

INITIALOPEN is issued automatically at LISP loading time (unless LISP is being overlayed by an overlay core image, see DUMPOVER) to open files for input and output. The files LISPIT and LISPOT are always opened by INITIALOPEN. The file REPLY is always opened for a conversational task. Furthermore, all files are opened which were open at the time the core image being loaded was saved.

A user may embed around INITIALOPEN any operations which should occur at LISP loading time before any LISP doublets are read.

(INTERN strng) SUBR Pseudo-function

Finds, and if not found, creates the unique identifier, whose print name is given by the string strng. The value of INTERN is that identifier.

(INTERNX) SUBR Pseudo-function

Same as INTERN except that it uses the current contents of the string BUFFO between its beginning and the point indicated by the index CURCOLX, for the print name of the identifier it finds or creates.

(LINEUPINPUT)

Not longer in the system.

(LINTEXP n) SUBR Pseudo-function

If n is a large integer, LINTEXP returns a string containing its print representation. If n has more than 4096 digits, the low-order digits are truncated and a message to that effect is printed. If n is not a large integer, LINTEXP is unpredictable and hazardous. The predicate LINTP may be used prior to issuing LINTEXP to check whether an item is a long integer. LINTEXP uses PR1BUF as working storage.

(**LOAD) SUBR Pseudo-function

**LOAD loads a LISP core image from a file where it was saved. The file must have been opened via the DD (CMS), DISKDEF or OPEN*SEQ function. The following evalquote doublets will open a file and load a LISP core image from it:

On OS or TSS:
```
DISKDEF (ddname LOAD INPUT NIL 4)
**LOAD ()
```

On CMS:
```
DISKDEF (fn ft fm 0 LOAD)
**LOAD ()
```

**LOAD is used both for restart core images (saved by FILELISP or **DUMPER) and overlay core images (saved by DUMPOVER). See DUMPOVER for the distinction. **LOAD is automatically invoked for the initial loading of LISP. **LOAD returns the value of special variable GENER, which is at this time greater by one than its value at the time the core image was saved. Control returns to the point in the new core image where **DUMPER, DUMPOVER or FILELISP was called when the core image was saved.

(LOADLISP fn ft fm) SUBR Pseudo-function CMS
(LOADLISP ddname) SUBR Pseudo-function OS, TSS

LOADLISP loads a LISP core image from a file where it was saved. It operates by first opening the file and then invoking **LOAD. LOADLISP is used both for restart core images (saved by FILELISP or **DUMPER) and overlay core images (saved by DUMPOVER). See DUMPOVER for the distinction. LOADLISP returns the value of special variable GENER, which is at this time one greater than it was at the time the core image was saved. Control returns to the point in the new core image where **DUMPER, DUMPOVER or FILELISP was called when core image was saved.

(MKNAM) SUBR Pseudo-function

This function copies the contents of the string BUFFO, between its beginning and the point indicated by the index CURCOLX, into a new string, which it returns. BUFFO is cleared to hex zeros and CURCOLX is reset to 0. MKNAM is a subroutine of INTERNX.

(*NEWLIN) SUBR Pseudo-function

*NEWLIN causes a record to be read from the file whose name is the value of CURINNAM, i.e. the file curently selected for input. The record is read into the string CURINBUF. CURCOL is set to 0 and CURCHAR remains unchanged. The value returned by *NEWLIN is undefined and may easily change from one release of LISP to another. See BACKCHR for the description of the special variables.

If *NEWLIN encounters an end-of-file condition, it fills the buffer CURINBUF with the repeated occurrences of the string X'401340135C', where X'40' is of course a blank, X'5C' is a right paren, and X'13' is the PNAME of the character object which is the value of special EOF.

(NOTE fn) SUBR CMS

If fn is a name of an open disk file, NOTE returns the record sequence number of the last record read or written on the file. If fn is a name of an open terminal file, NOTE is undefined. If fn is not a name of an open file, there is an error condition.

(NUMEXPLD ... Implementation subroutine with non-standard calling sequence.

NUMEXPLD is the underlying subroutine used by the exploding functions EXPLODE1 and LINTEXP to create the print representation of numbers. The value of NUMEXPLD is the string which is the value of PR1BUF, which string contains the exploded number. NUMEXPLD is normally not to be invoked by the user.

(NUMOB) SUBR

Creates the LISP number representation from the EBCDIC representation of the number datum contained in the string BUFFO, between its beginning and the point indicated by the index CURCOLX, and returns the created LISP number. NUMOB is used in conjunction with PACK by *RATOM to produce numbers.

(OPEN*SEQ fn ft fm urecl s) SUBR Pseudo-function CMS
(OPEN*SEQ ddnam org opt1 opt2 s) SUBR Pseudo-function OS,TSS

OPEN*SEQ opens a file for input or output and returns the value of OPENEDFILES list describing currently open functions. If the file being opened is a core image file, OPEN*SEQ opens it and returns NIL (OS, TSS), or a the value of OPENEDFILES list (CMS). OPEN*SEQ is used by DD (CMS) and DISKDEF functions to do the actual work. The first four arguments of OPEN*SEQ are the same as for DISKDEF.

For OS and TSS, the fifth argument s is a string which will serve as the I/O buffer. The length of the string may not exceed 1024 bytes. For output files, the length of s determines the output record size. For input files, the length of s determines the number of characters actually read in each input record. For a LOAD file (whether input or output), s is ignored.

For CMS,

s := NIL | string | FILE | LOAD

with the same significance as in DISKDEF.

OPENEDFILES Special variable

OPENEDFILES is a special variable whose value is a list of descriptions of currently open files (except core image files). Each description is a list of 13 items. To check whether a file is open, issue

(SASSOC fn OPENEDFILES NIL)

If the file fn is not open, SASSOC will return NIL, if it is open, SASSOC will return a list of 13 items describing the file. The OPENEDFILES list is maintained by OPEN*SEQ, RDS, WRS and SHUT functions. The format of the description of an open file is different for each

operating system:

For CMS:

item 1: filename (an identifier)
item 2: an identifier whose special value is fcb
(a CMS file control block)
item 3: I/O buffer (a string)
item 4: unused
item 5: saved CURCOL (input), unused (output)
item 6: unused
item 7: saved CURCHAR (input), unused (output)
item 8: unused
item 9: unused (input), saved CUROUTCL (output)
item 10: saved UBUFLIN (input), unused (output)
item 11: filemode (disk files),
CMS terminal control character (terminal files)
item 12: filetype | TIN | TOUT
item 13: unused

For OS:

item 1: ddname (an identifier)
item 2: an identifier whose special value is dcb
(an OS data control block)
item 3: I/O buffer (a string)
item 4: unused
item 5: saved CURCOL (input), unused (output)
item 6: unused
item 7: saved CURCHAR (input), unused (output)
item 8: unused
item 9: unused (input), saved CUROUTCL (output)
item 10: opt2 (see DISKDEF function)
item 11: opt1 (see DISKDEF function)
item 12: QUEUED (the only value supported)
item 13: unused

For TSS:

item 1: ddname (an identifier)
item 2: an identifier whose special value is dcb
(a TSS data control block)
item 3: I/O buffer (a string)
item 4: unused
item 5: saved CURCOL (input), saved LINENO (output)
item 6: unused
item 7: saved CURCHAR (input), unused (output)
item 8: unused
item 9: unused (input), saved CUROUTCL (output)
item 10: unused
item 11: opt1 (see DISKDEF function)
item 12: LINE | VI | VS | QUEUED
item 13: unused

(OSOPN dcb opt1) SUBR Pseudo-function OS, TSS

OSOPN issues an OS or TSS macro OPEN for the data control block dcb. See DISKDEF for the supported values of opt1. OSOPN returns *T* if it succeeds in opening the file and NIL otherwise. OSOPN is used by OPEN*SEQ. It is not normally invoked by the user.

(PACK c) SUBR

PACK concatenates the character c to the right hand end of the composition buffer BUFFO. Used in conjunction with UNPACK, MKNAM, NUMOB, and INTERNX.

(PLANTDDNAM name dcb) Psuedo-function OS
(PLANTNAME name block displ length) Psuedo-function CMS

On OS the function is used to modify/prepare the ddname field of the DCB. On CMS it is used to prepare file control blocks.
These functions are used by OPEN*SEQ.

(PRETTYPRINT l) SUBR

The function definitions given in the list l are printed in a more readable fashion. The list l has the same form as the argument of DEFINE.
This function uses the subfunctions PRINDEF, SUPERPRINT, and ENDLINE.

(PRINHEX n) SUBR

The 16 loworder bits of the number n is printed as 4 hex digits.

(PRINSCHR x) SUBR

Prints the atom x unless x is EQ to LBRAC or RBRAC, in which case %(or %) is printed.

(PRINT x) SUBR

(PRINT (LAMBDA (S) (PROG () (PRIN0 S) (TERPRI) (RETURN S)))).

PRINT prints one S-expression in standard format.

(PRINTCH c) SUBR

If c is a character atom (such as 'A), PRINTCH enters the corresponding character (A) into the output buffer at the next available byte position. PRINTCH returns c as its value. Any attempt to PRINTCH anything other than a character atom results in the error message P2.

(PRINTEXP stng) Pseudo-function

Assumes stng to be a string of characters, which it transfers into the output buffer, issuing a TERPRI when it fills the buffer. This is a subfunction of PRIN1 and is used with HEXEXP, NUMEXPLD, and EXPLODE1.

(PRIN0 x) SUBR

PRIN0 is used by PRINT to decompose an S-expression x into a string of atoms, parentheses, dots and spaces, it calls PRIN1 to fill the print buffer. PRIN0 does all of the work of PRINT except for the final (TERPRI). The value of (PRIN0 x) is x. To print two S-expressions A and B on the same line, one can follow (PRIN0 A) by (PRINT B), e.g.

(LAMBDA (A B) (PROG ( ) (PRIN0 A) (PRINT B)))) (FIRST SECOND).

This will result in the following printout:

FIRST SECOND

= NIL.

(PRIN1 a) SUBR

PRIN1 accepts any atom a and packs its print name into the print buffer. It is the main function of the system which packs the output buffer. All other printing functions (such as PRIN0) use PRIN1 as the basic building block. The value of (PRIN1 a) is a.

Example:

PRIN1 (A) results in AA

(LAMBDA (A B) (PROG () (PRIN1) (PRIN1 BLANK)

(PRIN1 B) (BLANKS 3) (PRIN1 B) (PRIN1 A) (PRIN1

PERIOD)

(TERPRI))) (FIRST SECOND) results in the following printout:

FIRST SECOND SECONDFIRST.

= NIL (the value of the PROG).

(PRIN1B a) SUBR

Defined by: (PRIN1B (LAMBDA (A) (PROG () (PRIN1 A) (PRIN1 BLANK))))

(PRINV vec) SUBR

Prints all vectors except ntuples. It prints strings, vectors of literal integers, vectors of literal reals, large integers, pointer vectors and NIL. If vec is an assembled program vector PRINV attempts to print it as a string. If vec is not a vector the location vec is printed as a hex location.

(PRTSTRG stng) SUBR

Same an PRINTEXP, except PRTSTRG tests stng to see if it is a string datum.

(*RATOM) SUBR

*RATOM is the LISP token reading function, which always returns a single atom whenever called. *RATOM uses RDCHR to provide it with a stream of input characters. *RATOM scans the input and returns a single atom which is either LPAR, RPAR, LBRAC, RBRAC, PERIOD, a number, string or symbol. An identifier not already there is added to the OBLIST. (See Section VII on Input Output.)

(RDCHR) Pseudo-function

RDCHR reads the next character from the current input file. Returns CURCHAR as value, updates CURCOL and CURCHAR, using (*NEWLIN) to bring in more characters as needed.

(RDS in) Pseudo-function

If in is already the currently selected input file RDS simply returns in, otherwise, RDS puts the CURCHAR and CURCOL information for the current file CURINNAM back in its entry on the OPENFILES list. (see OPEN*SEQ for the format of OPENFILES.) RDS then establishes in as the currently selected file by filling in the following global variables from the OPENFILES entry corresponding to in:

CURINDCB CURINBUF EODADIN CURINNAM CURCOL SYNADIN CURCHAR in any case it returns the value in. UBUFLIN also?

(READ) SUBR

READ calls for one S-expression to be read from CURINNAM, using functions (READ1) and (*RATOM). READ calls ERROR if a right parenthesis or period (not a decimal point) occurs, and calls READ1 every time it sees a left parenthesis.

(READ0) SUBR

Subfunction of READ which uses the special value GENL which is bound by READ. READ0 is the recursive part of READ. GENL provides a list which allows gensyms with the same spelling to be EQ. Note: such gensyms are guaranteed EQ only over a single read, and not between S-expressions read separately.

(READ1) SUBR

READ1 is a function used by READ to read nodes and non-symbol elements. READ1 is entered after one left parenthesis has been encountered. It calls *RATOM or READ1 successively until the matching right parenthesis is read and calls CONS to tie atoms together appropriately to build the corresponding list structure in core. If an illegal structure is encountered, READ1 produces a diagnostic and calls ERROR. READ1 has been extended to read vectors.

(SHUT ddnam opt2) Pseudo-function

Closes the file whose name is ddnam with the disposition opt2. Removes the entry for ddnam from the OPENEDFILES list, and restores the name of the dcb area, (BOX?) to the list EMPTYBOXES.

(STANDEODADIN) Psuedo-function OS,CMS

Used as the dtandard end of data set encountered on input routine. CURINBUF is filled with the sequence blank, tape-mark,blank, tape-mark, ")". i.e. in hex 401340135d .
This function causes the value EOF to be returned as the value of *NEWLIN and RDCHR.

(STANDEODADOUT) unnecessary function CMS,OS

(STANDSYNADIN) Psuedo-function CMS,OS
(STANDSYNADOUT) Psuedo-function CMS,OS

These standard error routines are executed on input or output error respectively. On OS an error message and backtrace is printed and a dump taken. On CMS the error code is analyzed and an action appropiate to that code taken.

SYNAD Special variable OS

Contains the name of the routine that is to be applied on input or output error.
Reset by TERPRI from the special value SYNADIN. br
Reset by *NEWLIN from SYNADOUT.
The nominal values of SYNADIN and SYNADOUT are STANDSYNADIN and STANDSYNADOUT.

(TAB n) SUBR

Resets CUROUTCL to n.

(SNAPS x y z) Pseudo-function

SNAPS is a desperation debugging device which has proved useful for the system debugging. It gives a hex dump of the registers, locations in memory between x+FIXED and y+FIXED, where x and y are LISP numbers. If (NULL z) then SNAPS will return NIL, otherwise SNAPS will terminate the task.

(TEREAD) SUBR

TEREAD causes the read buffer to be reset so that the next call to READ (or to *RATOM) will call (*NEWLIN). If READ is called without TEREAD and if at the last READ there were any right parentheses left in the buffer, READ would call ERROR. TEREAD prevents this.

The value of TEREAD is NIL.

(TERPRI) SUBR

TERPRI causes the contents of the print buffer to be printed, and resets the print buffer.

Consecutive TERPRIs result in skipping print lines. The value of TERPRI is NIL.

(UNPACK strg) Pseudo-function

Returns as a list of character objects the contents of the string strg.

(WRS out) Pseudo-function

Selects out as the currently selected output file after first saveing the CUROUTCL parameter for the file CUROUTNAM, which out replaces as the selected file. WRS sets the following global Special variables with parameters obtained from the OPENEDFILES list entry for out: CUROUTDCB CUROUTBUF CUROUTNAM EODADOUT CUROUTCL SYNADOUT ,the value of WRS is the file name out.

APPENDIX D.

SUPERVISORY AND DEBUGGING FUNCTIONS

(EMBED fname redef) SUBR

The EMBED function provides an unusual facility whereby one may redefine a function whose name is fname and get to use the old definition of fname in the redefinition. The redefining LAMBDA-expression is given as the s-expression value of redef. Furthermore, the UNTRACE function will cause the previous definition to be restored.

(ERROR msg) SUBR

ERROR causes its argument msg to be evaluated and its value printed and then induces an error unwind of the LISP system.

(ERROR2 msg1 msg2) SUBR

Defined by: (ERROR2(LAMBDA(A B)(PROG2(PRINT A)(ERROR B))))

(ERRORSET e n m a) SUBR

The arguments n and m are ignored and are present only for compatability with 7094 LISP. If no error occurs then ERRORSET can be defined as (LIST (EVAL e a)). If an error occurs inside of ERRORSET then the value is NIL. If variable bindings bound outside of ERRORSET have not been damaged by CSET, SET, DEFINE and other such pseudo-functions, it is usually possible to continue with some other computation. ERRORSET is useful for building supervisors.

(EVALQUOTE fn args) SUBR

EVALQUOTE is defined by:

```
(EVALQUOTE
   (LAMBDA (FN ARGS) (COND
       ((OR
          (GET FN (QUOTE FEXPR))
          (GET FN (QUOTE FSUBR)))
         (EVAL (CONS FN ARGS) NIL))
         (T (APPLY FN ARGS NIL)))))
```

This definition shows that EVALQUOTE is capable of handling special forms (such as COND, PROG, etc.) as a special case. An alternate definition of EVALQUOTE is simply:

```
(EVALQUOTE
   (LAMBDA (FN ARGS) (EVA1 (CONS FN ARGS)))).
```

(SNAPS x y z) Pseudo-function

SNAPS is a desperation debugging device which has proved useful for the system debugging. It gives a hex dump of the registers, locations in memory between x+FIXED and y+FIXED, where x and y are LISP numbers. If (NULL z) the SNAPS will return NIL, otherwise SNAPS will terminate the task.

(*SUPV a b)

The supervisor *SUPV selects file a for reading Evalquote doublets and for input to user READs, file b is selected for printing the value of the doublets, and b is selected for user PRINTs.

The user may change the file selected for input to his READs by executing the doublet RDS(c), or evaluating (RDS c) from within a doublet. The users output file may likewise be switched, in which case the file selected for doublet input remains unchanged as does the file on which the doublets value is printed.

If instead of a doublet the identifier FIN is encountered this function returns the value T. A side effect of this function is that the user input and output file selection remains in effect, even though the return to a higher level *SUPV means that doublet input and output files revert to a former selection. *SUPV is defined by:

```
(*SUPV
   (LAMBDA (IN OUT) (PROG (X I O)
                          (SETQ I IN)
                          (SETQ O OUT)
                          (RDS IN)
                          (WRS OUT)
                      A   (SETQ XERMAN (READ))
                          (COND((EQ  XERMAN  (QUOTE  FIN))  (RETURN
T)))
                          (COND((EQ    CURINNAM    (QUOTE    LISPIT))
(TEREAD)))
                          (PRIN1 BLANK)
                          (PRINT YERMAN)
                          (RDS I)
```

```
                    (WRS O)
                    (SETQ X(CAR(ERRORSET (LIST
                       (QUOTE EVALQUOTE)
                       (LIST (QUOTE QUOTE) XERMAN)
          (LIST (QUOTE QUOTE) YERMAN)) 1000000 T NIL)))
                    (SETQ I CURINNAM)
                    (SETQ O CUROUTNAM)
                    (RDS IN)
                    (WRS OUT)
                    (PRIN1 (QUOTE VALUE =    )) } or {(PRIN1B (QUOTE VALUE))
                    (PRINT X)                         (PRIN1B (QUOTE =))
                    (GO A))))
```

(TRACE x) SUBR

The argument of TRACE x, is a list of function names (f1, f2, f3 ... fn). The effect of tracing is to EMBED a redefinition of f. The redefinition of f is such that if f is called while being traced, a trace printout occurs.

Example:

ARGUMENTS of f arg1

arg2
.
.
.

VALUE of f value

TRACE uses TRACE1 to effect a trace of this format. Tracing must be done with some caution to avoid excessive printout, it is usually not possible to trace special forms, also tracing of the print functions or MAPCAR will surely result in infinite printout.

(TRACE1 x redef)

TRACE1 is a more general tracing mechanism than TRACE as the redefinition prototype of the functions, whose names are given by the list x, is given as the argument redef. redef is a s-expression representation of a LAMBDA-expression (it is not a functional argument). The s-expression redef may contain the identifiers ARG*, FUN*, RAGS and FNARGS. TRACE1 substitutes for ARG*, a list of generated symbols (g1, g2, ...) corresponding to the number of arguments that the function expects. The name of the function being redefined fn for FUN*, (LIST g1, g2, ...) for RAGS, and (fn, g1, g2, ...) for FNARGS. The resulting

redefinition redef of fn is embedded for fn by (EMBED fn redef).

This is done for each function named in x.

(UNTRACE x) SUBR

UNTRACE undoes the effect of a previous EMBED.

APPENDIX E.

```
      %         THE LPL PARSER
      %    THIS PARSER WAS SUGGESTED TO ME BY VAUGHAN R. PRATT
      %
PARSER:=:PROC RBP; BEGIN !&LEFT;
     !&LEFT := NUD() ;
B;   IF RBP .LT LBP TOK THEN !&LEFT := LED()ELSE RETURN !&LEFT ;
     GO B ;  END ;
LBP:=: PROC A;  IF NUMBERP A THEN 99  ELSE
                IF !|TEM := GET(A,'LBP) THEN !|TEM ELSE 99 ;
NUD:=: LAMBDA  ; BEGIN ;
     IF NUMBERP TOK THEN GO A ;
     CASEGO(TOK,("!# :STR),("!" :DAT),("!. : DOT))     ;
H;   IF TOK .EQ  "$EOF$ THEN  (PRINT 'ENDOFFILE ; UNWIND()) ELSE
     IF !|TEM := GET(TOK,'NUD) THEN GO B ELSE
     IF ¬ GET(TOK,'LBP) THEN GO A ELSE
     IF GET(TOK,'LED) THEN RETURN((
      PRINTEXP # MISSING LEFT OPERAND, ASSUMED NIL, INSERTED BEFORE # ;
      PRINT TOK ; !|DUMMY ))  ;
A;   !|TEM:=TOK ; NXTTOK();
E;   IF NUMBERP TOK THEN GO G ELSE
     IF GET(TOK,'LED) THEN RETURN !|TEM ELSE
     IF GET(TOK,'NUD) THEN GO G  ELSE
     IF GET(TOK,'LBP) THEN RETURN !|TEM ;
G;   RETURN( !|TEM . NUD().() );
B;   NXTTOK() ;
     !|TEM:= SUBRFUN((),!|TEM) ;     GO E;
STR;!|TEM:=LPLSTRING();GO E;
DAT;QUOTSWT := 'NXTCHR;!|TEM:=READ();QUOTSWT:=();NXTTOK();GO E;
DOT;IF COMPSEQ('"(!. P I N)) THEN UNWIND();   GO H;
 END;
LED:=: PROC ; IF AND(¬NUMBERP TOK,!|TEM:=GET(TOK,'LED)) THEN
         (NXTTOK();SUBRFUN((),!|TEM)) ELSE (!&LEFT,PARSE 50) ;
DEF("THEN ,0)          ;
DEF("ELSE ,0)          ;
DEF("END ,0)           ;
DEF("EOL ,0)           ;
DEF("IN ,0)            ;
DEF("ON ,0)            ;
DEF("BY ,0)            ;
DEF("TO ,0)            ;
DEF("UNTIL ,0)         ;
DEF("WHILE ,0)         ;
DEF("UNLESS ,0)        ;
DEF("!) ,0)            ;
DEF("!> ,0)            ;
DEF("!; , 1 , LED, 'PROGN . !&LEFT . GETLIST('"!; ,1))     ;
DEF("!,  ,2,LED, 'LISS . !&LEFT . GETLIST('"!, ,2),
          NUD, 'LISS . NIL . GETLIST('"!, ,2)   )                    ;
DEF("!: , 3,LED, !&LEFT . GETLIST('"!: ,3) )     ;
DEF("!. ,14 ,LED, ('CONS,!&LEFT,PARSE 13) )                          ;
```

```
DEF("!- ,20,LED, ('DIFFERENCE,!&LEFT,PARSE 20),
          NUD, ('MINUS, PARSE 20)  )                                    ;
DEF("!*!* ,22,LED,('EXPT,!&LEFT,PARSE 21) )                              ;
DEF("!:!= ,36,LED,('GSETQ,!&LEFT, PARSE 2) )                             ;
DEF("!:!=!: , 40,LED, ('GDEF ,!&LEFT, PARSE 2) )    ;
DEF("!( ,  99,NUD, IF TOK .EQ '"!) THEN (NXTTOK(),NIL) ELSE
       ANDDO( PARSE 0,RPAREN '"!)) ,
          LED ,  (!&LEFT . ANDDO(GETLIST('"!, ,2),RPAREN '"!))) )  ;
DEF("!< ,  99,NUD, ANDDO('VECTOR . GETLIST('"!, ,2),RPAREN '"!>),
           LED,'GELT . !&LEFT . ANDDO(GETLIST('"!, ,2),RPAREN '"!>) );
DEF("!# , 99, NUD, NUD())                                                 ;
DEF("!" , 99, NUD, NUD())   ;
DEF("PROC ,99,NUD, ('"LAMBDA . DCLR2()) )  ;
DEF("LAMBDA ,99,NUD,'"LAMBDA . DCLR2()  )  ;
DEF("FPROC ,99, NUD,'"FLAMBDA . DCLR2() )  ;
DEF("FLAMBDA ,99,NUD,'"FLAMBDA . DCLR2() )  ;
DEF("IF ,99,NUD,('COND .((PARSE 2,PROG2(RPAREN '"THEN ,PARSE 2))
                        . ELSEEXPR()))   )   ;
DEF("BEGIN , 99, NUD, BEGIN !|PV ;
         !|PV := IF TOK .EQ '"; THEN PROG2(NXTTOK(),,) ELSE
            ANDDO(GETVAR('"!, ),RPAREN '"; )  ;
          RETURN('PROG .(!|PV . FLAT!|(ANDDO(GETLIST('"; ,2) ,
                RPAREN '"END))))   ;  END )                               ;
INFIX("!_ ,19,STRCONC)                                                    ;
INFIX("!.EQ ,19,EQ)                                                       ;
INFIX("!.NE ,19,NE)                                                       ;
INFIX("!.GT ,19,GT)                                                       ;
INFIX("!.GE ,19,GE)                                                       ;
INFIX("!.LE ,19,"*LE)                                                     ;
INFIX("!.LT ,19,LT)                                                       ;
INFIX("!+ ,20,PLUS)                                                       ;
INFIX("!* ,21,TIMES)                                                      ;
INFIX("!/ ,21,QUOTIENT)                                                   ;
PREFIX("!¬ ,10, NOT)                                                      ;
PREFIX("!' ,99, QUOTE)                                                    ;
 .FIN
```

APPENDIX E.

CMS EXEC Files.

LISP EXEC

```
&TYPEOUT OFF  &MOD1 = 0 &IF  &INDEX GT 4 &GOTO  -BDPARM &IF
&INDEX EQ 3 &ARGS &1 &2 &3 LISP115 &IF &INDEX EQ 2 &ARGS &1
&2 B  LISP115 &IF &INDEX  EQ 1 &ARGS  &1 193 B  LISP115 &IF
&INDEX EQ 0 &ARGS LISP115 193  B LISP115 CP LINK BLAIR1 191
&2 (NOPASS)  &IF &INDEX0  NE 0  &GOTO -BDLINK  &IF &3  EQ P
&GOTO -BDMODE &IF &3  EQ A &MOD1 = A,P &IF &3  EQ B &MOD1 =
B,P &IF &3 EQ  S &GOTO -BDMODE &IF &3 EQ C  &MOD1 = C,P &IF
&3 EQ T &MOD1  = T,P &IF &MOD1 EQ 0  &GOTO -BDMODE LOGIN &2
```

```
&MOD1 LISP115 &IF &INDEX0 NE 0 &GOTO -BDLOG EXEC &4 &1 &X =
0  -FIN &CONTINUE  RELEASE  &2 &3  CP  DETACH  &2 &EXIT  &X
-BDPARM &PRINT  BAD PARAMETERS  &X =  1 &GOTO  -FIN -BDLINK
&PRINT BAD  LINK &INDEX0 &PRINT TRY  IT AGAIN &X =  2 &GOTO
-FIN -BDMODE &PRINT &3 IS A BAD MODE &PRINT TRY IT AGAIN &X
= 3 &GOTO -FIN -BDLOG &PRINT  BAD LOGIN, ERROR CODE &INDEX0
&X = 4 &GOTO -FIN
```

```
LISPOFF EXEC

&TYPEOUT OFF &IF &INDEX GT 5  &GOTO -BDPARM &IF &INDEX EQ 4
&ARGS &1 &2  &3 &4 LISP &IF &INDEX  EQ 3 &ARGS &1  &2 &3 NO
LISP &IF &INDEX EQ 2 &ARGS &1 &2 LISP115 NO LISP &IF &INDEX
EQ 1  &ARGS &1 OUT  LISP115 NO LISP  &IF &INDEX EQ  0 &GOTO
-ERR &IF &1  EQ &2 &GOTO -TR2 BATCH (NOFILE)  LISPBCH &3 &1
&2 &4 &5 &EXIT &INDEX0 -ERR  &PRINT NO INPUT FILE PARAMETER
GIVEN -E2 &PRINT THE CORRECT FORMAT  IS : &PRINT LISPOFF IN
OU COR SAV &PRINT WHERE: IN IS INPUT FILE NAME &PRINT OU IS
OUTPUT FILE NAME IF IT IS TO  BE KEPT. &PRINT OU DEFAULT IS
PRINT OFFLINE AS OUT OUT &PRINT COR IS LISP CORE IMAGE FILE
THAT YOU WISH  TO RUN FROM. &PRINT COR  DEFAULT ... LISP115
&PRINT SAV  IF YOU WANT A  FILELISP (SAV SOSTAP P1)  AT THE
END. &PRINT SAV = NO IS THE DEFAULT &EXIT 10 -BDPARM &PRINT
TOO MANY PARAMETERS &GOTO -E2 -TR2 &PRINT ERROR ARG1 = ARG2
&EXIT 11
```

```
LISPBCH BATCH

CP LINK &USERID 191 194 (NOPASS) LOGIN 194 A,P &BEGSTACK
 CSET(QUIET())
 EMBED(CONVERSATIONAL(LAMBDA() ()  )) RDF( &END STACK &STACK
&2 &STACK &3 &BEGSTACK )))
 UNTRACE((CONVERSATIONAL)) &END  STACK &IF  &4 EQ  NO &GOTO
-SKP &BEGSTACK FILELISP( &END STACK &STACK &4 &BEGSTACK
 SOSTAP P1  ) &END STACK  -SKP &CONTINUE &STACK  FIN &ERROR
&GOTO -ERR EXEC &5 &1 &GOTO  -END -ERR CP MSG &USERID ERROR
IN LISP BATCH COMPILATION-- &INDEX0  &GOTO -FIN -P1 OFFLINE
PRINT OUT OUT &GOTO  -P2 -END &IF &3 EQ OUT  &GOTO -P1 DISK
DUMP &3 OUT -P2 &CONTINUE &IF &4 EQ NO &GOTO -FIN DISK DUMP
&4 SOSTAP P1 -FIN &ERROR &CONTINUE RELEASE 194 A CP DET 194
CP MSG &USERID BATCH JOB FINISHED
```

APPENDIX F.

datum = nil | node | literal-atom | non-symbol-element 1/

nil = ( ) | NIL

node = (datum*+1 [. datum])

literal-atom = identifier | genid | special-spelling | $$-artifact | quoted

identifier = letter {letter | digit | . }*

genid = % G digit*+1

special-spelling = % string

$$-artifact = $$ a (any_character_except a)*+1 a

quoted = ' character | quoted {letter | digit | . }*

string = # {non-string-delimiter | ' character }*+1 #

non-string-delimiter = any_character_except { ' | # }

non-symbol-element = number | string | vector | n-tuple

vector = %([vector-typer] vector-element* %)

vector-element = datum

vector-typer = SYMBOL | INTEGER | REAL

n-tuple = %(n-tuple-name datum*+1 %)

n-tuple name = identifier

number = integer | octal | real | hex

Throughout this document "|", "{", "}", "[", "]", "*", and "*+1" are used for the specification of syntax.

Underlines and capitals are used to denote literal occurrance.
| is used to separate alternatives.
{ and } are used for meta-linguistic groupers.
[and] are used to denote optionality.
* as a suffix indicates an indefinite number of.
*+1 as a suffix indicates one or more.

1/ Data are required to be separated by at least one blank. Blanks can be eliminated by the rules shown in Table 1.

Table 1. Optional transformations for the elimination of spaces between a and b.

a \ b	1	2	3	4
1 string node () $$-artifact n-tuple vector				
2 number	X	X	X	X
3 genid quoted special-spelling identifier	X	X	X	X
4 .		X	X	Y

An "X" indicates that a space is required between a and b, no "X" indicates that the space may be eliminated. "Y" indicates an unallowed syntax condition.

APPENDIX G.

Assembler symbol table

The assembler LAP uses a permanent symbol table which is found on the property list of each identifier as the attribute INTSYM. It is possible to remove these if the assembler is not required. The current settings are given in the following table:

$ALIST	0208	$FLOAT	06F8	$NUMVAL	06A4	$PLANTNM	0766
$ROUND	0688	*ABS	0006	*AC	000A	*FIX	0005
*ID	0000	*IORG	0002	*LINKR	0001	*LIST	05F0
*MNARG	000F	*MQ	000B	*NIL	0000	*PDL	0009
*PDP	0009	*REL	0007	*RET	0008	*RNARG	000E
*SCRE	000C	*SCRF	000D	*SCR1	000E	*SCR2	000F
*VALUE1	0003	*VALUE2	0004	A	5A00	AASYSREF	0158
ABOTTEND	01FC	ABS	0006	AD	6A00	ADR	2A00
AER	3A00	AL	5E00	AR	1A00	ARG1	020C
ARG10	0230	ARG11	0234	ARG12	0238	ARG13	023C
ARG14	0240	ARG15	0244	ARG16	0248	ARG2	0210
ARG3	0214	ARG4	0218	ARG5	021C	ARG6	0220
ARG7	0224	ARG8	0228	ARG9	022C	ASAVEARE	017C
ATTON	0188	B	47F0	BAL	4500	BALR	0500
BC	4700	BCR	0700	BCT	4600	BCTR	0600
BE	4780	BH	4720	BIG1	0104	BL	4740
BM	4740	BNE	4770	BNH	47D0	BNL	47B0
BNZ	4770	BO	4710	BXH	8600	BXLE	8700
BZ	4780	C	5900	CAAAAR	0444	CAAADR	04D4
CAAAR	03B4	CAADAR	048C	CAADDR	051C	CAADR	03FC
CAAR	036C	CADAAR	0468	CADADR	04F8	CADAR	03D8
CADDAR	04B0	CADDDR	0540	CADDR	0420	CADR	0390
CALL	0585	CAR	034C	CARMASK	007C	CDAAAR	0456
CDAADR	04E6	CDAAR	03C6	CDADAR	049E	CDADDR	052E
CDADR	040E	CDAR	037E	CDDAAR	047A	CDDADR	050A
CDDAR	03EA	CDDDAR	04C2	CDDDDR	0552	CDDDR	0432
CDDR	03A2	CDR	035C	CDRMASK	0080	CE	7900
CER	3900	CH	4900	CHAROB$	01BC	CHAR46	0100
CH255	0258	CL	5500	CLC	D500	CLI	9500
CLR	1500	CONS	058C	CR	1900	CVD	4E00
D	5D00	DBFLTONE	00D0	DBLWD	00C0	DBLZERO	0040
DDR	2D00	DER	3D00	DISKPBL	0CA4	DISPATCH	032C
DR	1D00	DSPMSK	0074	EIGHT	0064	ERRORSTP	01E4
EX	4400	EXTERNAL	0005	FIVE	0058	FOUR	0054
FSTCHAR	01BC	FSTFIX	01A8	FSTFLT	01B4	FSTID	01B8
FSTLST	01D8	FSTQT	0190	FSTVAL1	019C	FSTVAL2	01A0
FSTVAL3	025C	FSTVAL4	0260	FSTVAL5	0264	FSTVAL6	0268
FSTVEC	01CC	FSTWD2	01A4	IC	4300	ID	0000
INFCB	0108	INLIS	015C	IORG	0002	L	5800
LA	4100	LABEL	0007	LCDR	2300	LCR	1300
LD	6800	LDR	2800	LE	7800	LER	3800
LH	4800	LINKR	0001	LM	9800	LNDR	2100
LNR	1100	LOGDISK	0D40	LPER	3000	LPR	1000

LR	1800	LSTID	01C8	LSTQT	0198	LTR	1200
M	5C00	MAXAVID	01C4	MD	6C00	MDR	2C00
MINUSONE	006C	MOVE	0284	MR	1C00	MVC	D200
MVI	9200	N	5400	NI	9400	NILF	01DC
NILH	00E0	NINE	0068	NINEDIGIT	0170	NUMTEMP1	0028
NUMTEMP2	002C	NUMTEMP5	0038	NXTAVEC	01D0	NXTAVFIX	01AC
NXTAVFLT	01B0	NXTAVID	01C0	NXTAVLST	01D4	NXTAVQT	0194
O	5600	OBEYCOM	0D48	OI	9600	ONE	0048
OR	1600	OUFCB	0CF0	PDLOC	018C	PWROF10	0180
QUOTES$	0190	REL	0007	S	5B00	SCANBF	0168
SCRE	000C	SCR1	000E	SDR	2B00	SER	3B00
SEVEN	0060	SIX	005C	SLDA	8F00	SLL	8900
SLR	1F00	SPCBASE$	019C	SPECBIND	0564	SPECRSTR	0588
SPM	0400	SR	1B00	SRA	8A00	SRDA	8E00
SRDL	8C00	SRL	8800	ST	5000	STC	4200
STD	6000	STE	7000	STH	4000	STM	9000
SVC	0A00	TERMRPBL	0C84	TERMWPBL	0C94	THREE	0050
TM	9100	TNZ	4770	TOLRNCE	00E4	TRAPOFF	0C58
TWO	004C	TZE	4780	UNPK	F300	VALUE1	0003
VALUE2	0004	XC	D700	ZERO	0044	ZERODIGIT	016C

APPENDIX H.
The Permanent Environment of LISP.

The permanent Environment of LISP is given by the following table. Each identifier is given allong with its attributes. A series of flags, labeled bits 1 - 5, have the following meanings:

Bit 1 : 1 if that identifier is a compiled function.
Bit 2 : 1 if that function recieves unevaluated arguments.
Bit 3 : 1 if indefinite number of arguments.
Bit 4 : 1 if function with literal value.
Bit 5 : 1 if is used free in a compiled function.

After the five flags is given the number of arguments if a function.

The last thing on the line is the current special value setting. If the value VECTOR is printed, it indicates a pointer into vector space. In the case of compiled functions it indicates the adress of the assembled program. If BELOW is given it indicates a pointer to the area below fixed program space.

If the identifer has a property list that is printed on the following line.

	1	1	1	1	1	0	NOVAL		
$ALINE$	0	0	0	0	1	0			
(SPECIAL (NIL))									
$ALIST	0	0	0	0	1	0	NIL	87	
(INTSYM 520 SPECIAL (NIL))									
$ARGSCNTR	0	0	0	0	1	0	BELOW		
EOF	0	0	0	0	1	0			
(SPECIAL (NIL))									
$FIX	1	0	0	0	0	1	VECTOR	91	
**DUMPER	1	0	0	0	1	0	BELOW	99	
**EXPEL	1	0	0	0	1	0	BELOW	101	
**LOAD	1	0	0	0	1	0	BELOW	105	
*CONDERR	1	0	0	0	0	0	VECTOR		
*EQC	1	0	0	0	0	2	VECTOR		
*EQN	1	0	0	0	0	2	VECTOR	92	
*EQP	1	0	0	0	0	2	VECTOR	92	
*EQUAL	1	0	0	0	0	3	VECTOR		
*EXPAND	1	0	0	0	1	2	VECTOR		
*EXPEL	1	0	0	0	1	1	VECTOR	101	
*FINIS	1	0	0	0	0	1	VECTOR	103	
*MAX	1	0	0	0	1	2	VECTOR		
*MIN	1	0	1	0	1	2	VECTOR		
*NEWLIN	1	0	0	0	0	0	VECTOR	106	
*RATOM	1	0	0	0	0	0	VECTOR	111	
*SUPERMAN	1	0	0	0	0	0	VECTOR		
*SUPV	1	0	0	0	1	2	VECTOR	116	
ABSVAL	1	0	0	0	0	1	VECTOR	98	

```
AC                0 0 0 0 1        0     NOVAL
ADD1              1 0 0 0 1        1     VECTOR
APOSTROPHE        0 0 0 0 1        0     '
     (SPECIAL (NIL))
APPEND            1 0 0 0 1        2     VECTOR
APPLX             1 0 0 0 0        2     VECTOR
APPLY             1 0 0 0 0        3     VECTOR
ATOM              1 0 0 0 0        1     VECTOR
     (MACRO %G-859)
ATTACH            1 0 0 0 0        1     VECTOR
ATTRIB            1 0 0 0 0        2     VECTOR
B                 0 0 0 0 1        0     NOVAL
     (INTSYM 18416)
BACKCHR           1 0 0 0 0        0     VECTOR
BIGUNWIND         1 0 0 0 0        0     VECTOR
BKTRCE            1 0 0 0 1        0     VECTOR
BLANK             0 0 0 0 1        0
     (SPECIAL (NIL))
BOMB              1 0 0 0 0        0     -98
BPI$DIR           0 0 0 0 1        0     BELOW
BPORG             0 0 0 0 1        0     VECTOR
BUCKET            1 0 0 0 0        1     VECTOR
BUFFO             0 0 0 0 1        0     VECTOR
     (SPECIAL (NIL))
BUFFO2            0 0 0 0 1        0     VECTOR
     (SPECIAL (NIL))
BX@1              0 0 0 0 1        0     BELOW
BX@2              0 0 0 0 1        0     BELOW
BX@3              0 0 0 0 1        0     BELOW
BX@4              0 0 0 0 1        0     BELOW
BX@5              0 0 0 0 1        0     BELOW
BX@6              0 0 0 0 1        0     BELOW
BX@7              0 0 0 0 1        0     BELOW
BX@8              0 0 0 0 1        0     BELOW
CAAAAR            1 0 0 0 0        1     VECTOR
     (INTSYM 1092 MACRO %G-916)
CAAADR            1 0 0 0 0        1     VECTOR
     (INTSYM 1236 MACRO %G-892)
CAAAR             1 0 0 0 0        1     VECTOR
     (INTSYM 948 MACRO %G-940)
CAADAR            1 0 0 0 0        1     VECTOR
     (INTSYM 1164 MACRO %G-904)
CAADDR            1 0 0 0 0        1     VECTOR
     (INTSYM 1308 MACRO %G-880)
CAADR             1 0 0 0 0        1     VECTOR
     (INTSYM 1020 MACRO %G-928)
CAAR              1 0 0 0 0        1     VECTOR
     (INTSYM 876 MACRO %G-952)
CADAAR            1 0 0 0 0        1     VECTOR
     (INTSYM 1128 MACRO %G-910)
CADADR            1 0 0 0 0        1     VECTOR
     (INTSYM 1272 MACRO %G-886)
CADAR             1 0 0 0 0        1     VECTOR
     (INTSYM 984 MACRO %G-934)
```

91 AND 66

66 66 66 67 67 94 94 95, 105, 106, 107, 109 67 67

```
CADDAR          1 0 0 0 0       1    VECTOR      67
     (INTSYM 1200 MACRO %G-898)
CADDDR          1 0 0 0 0       1    VECTOR
     (INTSYM 1344 MACRO %G-874)
CADDR           1 0 0 0 0       1    VECTOR
     (INTSYM 1056 MACRO %G-922)
CADR            1 0 0 0 1       1    VECTOR      67
     (INTSYM 912 MACRO %G-946)
CALL            1 0 0 0 0       2    VECTOR
     (INTSYM 1413)
CALLEDFNNAM     0 0 0 0 1       0    ZEROP
CAR             1 0 0 0 1       1    VECTOR      67
     (INTSYM 844 MACRO %G-868)                        <-- CASEGO  67
CDAAAR          1 0 0 0 0       1    VECTOR
     (INTSYM 1110 MACRO %G-913)
CDAADR          1 0 0 0 0       1    VECTOR
     (INTSYM 1254 MACRO %G-889)
CDAAR           1 0 0 0 0       1    VECTOR
     (INTSYM 966 MACRO %G-937)
CDADAR          1 0 0 0 0       1    VECTOR
     (INTSYM 1182 MACRO %G-901)
CDADDR          1 0 0 0 0       1    VECTOR
     (INTSYM 1326 MACRO %G-877)
CDADR           1 0 0 0 0       1    VECTOR
     (INTSYM 1038 MACRO %G-925)
CDAR            1 0 0 0 0       1    VECTOR
     (INTSYM 894 MACRO %G-949)
CDDAAR          1 0 0 0 0       1    VECTOR
     (INTSYM 1146 MACRO %G-907)
CDDADR          1 0 0 0 0       1    VECTOR
     (INTSYM 1290 MACRO %G-883)
CDDAR           1 0 0 0 0       1    VECTOR
     (INTSYM 1002 MACRO %G-931)
CDDDAR          1 0 0 0 0       1    VECTOR
     (INTSYM 1218 MACRO %G-895)
CDDDDR          1 0 0 0 0       1    VECTOR
     (INTSYM 1362 MACRO %G-871)
CDDDR           1 0 0 0 0       1    VECTOR
     (INTSYM 1074 MACRO %G-919)
CDDR            1 0 0 0 1       1    VECTOR
     (INTSYM 930 MACRO %G-943)
CDR             1 0 0 0 1       1    VECTOR      67
     (INTSYM 860 MACRO %G-865)
CEQ             1 0 0 0 0       2    VECTOR
CHARP           1 0 0 0 0       1    VECTOR      94
     (MACRO %G-797)
CHR2FIX         1 0 0 0 0       0    VECTOR      95
CLEARBUFF       1 0 0 0 1       0    VECTOR      95      <-- CLOSE  95
COMBIND         1 0 0 0 1       2    VECTOR
COMBOOL         1 0 0 0 0       4    VECTOR
COMCODE         1 0 0 0 0       1    VECTOR
COMCOND         1 0 0 0 0       2    VECTOR
COMLIS          1 0 0 0 0       1    VECTOR
COMMA           0 0 0 0 1       0    ,
```

```
     (SPECIAL (NIL))
COMMON          1 0 0 0 0       1     VECTOR      67
COMP            1 0 0 0 0       2     VECTOR
COMPACT         1 0 0 0 0       2     VECTOR
COMPILE         1 0 0 0 0       1     VECTOR      68
COMPLY          1 0 0 0 0       2     VECTOR
COMPROG         1 0 0 0 0       3     VECTOR
COMPSEQ         0 0 0 0 1       0     NOVAL
COMP360         1 0 0 0 0       1     VECTOR      68
COMRSTR         1 0 0 0 1       1     VECTOR
COMVAL          1 0 0 0 0       3     VECTOR
COM1X           1 0 0 0 0       4     VECTOR
COM2            1 0 0 0 0       4     VECTOR
CONC            1 1 1 0 1       2     VECTOR      68
CONC1           0 0 0 0 1       0     VECTOR
     (EXPR (LAMBDA (L) (COND ((NULL L) NIL) (T (APPEND (EVAL (CAR
 L) A) (CONC1 (CDR L)))))))
COND            1 1 1 0 0       2     VECTOR      68
     (FEXPR (LAMBDA (X Y) (EVCON X)))
CONS            1 0 0 0 1       2     VECTOR      69
     (INTSYM 1420 MACRO %G-862)
CONVERSATIONAL  1 0 0 0 0       0     VECTOR
COPY            1 0 0 0 0       1     VECTOR      69
COPYID          1 0 0 0 0       1     VECTOR
CSET            1 0 0 0 1       2     VECTOR      69
CSETQ           1 1 0 0 1       2     VECTOR      70
CURCHAR         0 0 0 0 1       0     )
     (SPECIAL (NIL))
CURCOL          0 0 0 0 1       0     81
     (SPECIAL (NIL))
CURCOLX         0 0 0 0 1       0     2
     (SPECIAL (NIL))
CURINBUF        0 0 0 0 1       0     VECTOR
     (SPECIAL (NIL))
CURINDCB        0 0 0 0 1       0     BELOW
     (SPECIAL (NIL))
CURINNAM        0 0 0 0 1       0     IDPROP
     (SPECIAL (NIL))
CUROUTBUF       0 0 0 0 1       0     VECTOR
     (SPECIAL (NIL))
CUROUTCL        0 0 0 0 1       0     0
     (SPECIAL (NIL))
CUROUTDCB       0 0 0 0 1       0     BELOW
     (SPECIAL (NIL))
CUROUTNAM       0 0 0 0 1       0     OUU
     (SPECIAL (NIL))
DASH            0 0 0 0 1       0     -
     (SPECIAL (NIL))
DDIN            1 0 0 0 1       1     VECTOR      97
DDOUT           1 0 0 0 1       1     VECTOR      98
DEFINE          1 0 0 0 1       1     VECTOR      70
DEFLIST         1 0 0 0 0       2     VECTOR      70
DELETE          0 0 0 0 1       0     NOVAL
DELETEL         1 0 0 0 0       2     VECTOR      71
```

DD 95
DDEF 97

```
DEXP              1 0 0 0 1       1     VECTOR
DIFFERENCE        1 0 0 0 1       2     VECTOR        91
DIGIT             1 0 0 0 0       1     VECTOR        98
     (MACRO %G-852)
DISKDEF           1 0 0 0 1       5     VECTOR        98
DIVIDE            1 0 0 0 1       2     VECTOR        91
DLOG              1 0 0 0 1       1     VECTOR
DOLLAR            0 0 0 0 1       0     $
     (SPECIAL (NIL))
DUMPEX            1 0 0 0 0       0     BELOW
DUMPOVER          1 0 0 0 0       0     VECTOR        100
ED                1 0 0 0 1       1     VECTOR        71
EDIT              1 0 0 0 1       0     VECTOR        71
EFFACE            1 0 0 0 0       2     VECTOR        71
ELT               1 0 0 0 1       2     VECTOR        71
EMBED             1 0 0 0 0       2     VECTOR        72,115
EMPTYBOXES        0 0 0 0 1       0     (BX@5 BX@6 BX@7 BX@8 EODAD SYNAD
 PLANTDDNAM)                                           112
     (SPECIAL (NIL))
ENCODE            1 0 0 0 1       2     VECTOR
ENDLINE           1 0 0 0 0       0     VECTOR
EODAD             0 0 0 0 1       0     BELOW         100
EQ                1 0 0 0 0       2     VECTOR        72
EQUAL             1 0 0 0 1       2     VECTOR        72
EQUALN            1 0 0 0 0       2     VECTOR  <---- ERASE   101
ERINLAP           0 0 0 0 1       0     NOVAL
ERROR             1 0 0 0 1       1     VECTOR        72,115
     (RENAME %G-667)
ERRORCNT          0 0 0 0 1       0     0
     (SPECIAL (NIL))
ERRORSET          1 0 0 0 0       4     VECTOR        115
ERROR2            1 0 0 0 1       2     VECTOR        73,115
ERROR3            1 0 0 0 0       3     VECTOR
ESCAPE            0 0 0 0 1       0     %
     (SPECIAL (NIL))
EVAL              1 0 0 0 0       2     VECTOR        73
EVALQUOTE         1 0 0 0 0       2     VECTOR        115
EVA1              1 0 0 0 1       1     VECTOR        73
EVCON             1 0 0 0 0       1     VECTOR
EVLIS             1 0 0 0 0       2     VECTOR
EVLIS1            1 0 0 0 0       1     VECTOR
EVPROG            1 0 0 0 0       1     VECTOR
EXCISE            1 0 0 0 0       1     VECTOR  <---- 73      EXPELIN  101
EXPLLOC           1 0 0 0 0       1     VECTOR        102
EXPLODE           1 0 0 0 1       1     VECTOR        102
     (RENAME %G-373)
EXPLODE1          1 0 0 0 0       1     VECTOR        102
EXPT              1 0 0 0 1       2     VECTOR        91
EXPT*2            1 0 0 0 1       3     VECTOR
EXTENDQT          1 0 0 0 0       1     VECTOR
FENCE             0 0 0 0 1       0     #
     (SPECIAL (NIL))
FILELISP          1 0 0 0 1       3     VECTOR        103
FINCLOS           1 0 0 0 0       0     VECTOR        103
```

```
FIXP            1 0 0 0 1       1     VECTOR      92
     (RENAME %G-398)
FLAG            1 0 0 0 1       2     VECTOR      73
FLOATLARGE      1 0 0 0 1       1     VECTOR
FLOATP          1 0 0 0 0       1     VECTOR      92
     (MACRO %G-788)
FN              0 0 0 0 1       0     NOVAL
FORM            0 0 0 0 1       0     NOVAL
FORTLOAD        1 0 0 0 0       2     BELOW       103
FSUBRFUN        1 0 0 0 0       2     VECTOR
FUNC            1 0 0 0 1       2     VECTOR
FUNCTION        1 1 0 0 0       2     VECTOR      73
     (FEXPR (LAMBDA (X Y) (LIST (QUOTE FUNARG) (CAR X) Y)))
G               0 0 0 0 1       0     NOVAL
GENER           0 0 0 0 1       0     1
GENL            0 0 0 0 1       0     NOVAL       112
GENNUM          0 0 0 0 1       0     298
GENSYM          1 0 0 0 1       0     VECTOR      74
GENSYMCH        0 0 0 0 1       0     G           74
     (SPECIAL (NIL))
GET             1 0 0 0 1       2     VECTOR      74
GETBPI          1 0 0 0 0       1     VECTOR      74
     (MACRO %G-845)
GETCH           1 0 0 0 1       2     VECTOR      74
GETCVEC         1 0 0 0 0       1     VECTOR      74
GETDEF          1 0 0 0 1       1     VECTOR      75
GETFLT1         1 0 0 0 0       0     VECTOR
GETFLT2         0 0 0 0 1       0     NOVAL
GETFXD e ?      1 0 0 0 0       0     VECTOR
GETIVEC         1 0 0 0 1       1     VECTOR      75   GETRVEC   75
GETSTR          1 0 0 0 1       1     BELOW       75   GETSVEC   75
GETTYP          1 0 0 0 0       1     VECTOR
GO              1 1 0 0 0       1     VECTOR      75
     (FEXPR (LAMBDA (X Y) (MKSPCVAL (MKSP (QUOTE PROGGO)) (CAR
X))))
GOLIST          0 0 0 0 1       0     NOVAL
GREATERP        1 0 0 0 1       2     VECTOR      92
GS              0 0 0 0 1       0     NOVAL
G12288          1 0 0 0 0       1     VECTOR
G138752         1 0 0 0 0       1     VECTOR
G140800         1 0 0 0 0       1     VECTOR
G17792          1 0 0 0 0       1     VECTOR
G18416          1 0 0 0 0       1     VECTOR
G2              0 0 0 0 1       0     NOVAL
G2719           1 0 0 0 0       1     VECTOR
G2722           1 0 0 0 0       1     VECTOR
G36864          1 0 0 0 0       1     VECTOR
G36864          1 0 0 0 0       1     VECTOR
G4462           1 0 0 0 0       0     VECTOR
G45236          1 0 0 0 0       1     VECTOR
G45324          1 0 0 0 0       0     VECTOR
G4555           1 0 0 0 0       1     VECTOR
G4849           1 0 0 0 0       1     VECTOR
G4851           1 0 0 0 0       0     VECTOR
```

```
G4928          1 0 0 0 0      1    VECTOR
G4985          1 0 0 0 0      1    VECTOR
G4995          1 0 0 0 0      1    VECTOR
G5011          1 0 0 0 0      0    VECTOR
G5062          1 0 0 0 0      1    VECTOR
G5064          1 0 0 0 0      0    VECTOR
G52692         1 0 0 0 0      1    VECTOR
G6320          1 0 0 0 0      1    VECTOR
G66304         1 0 0 0 0      1    VECTOR
G7611          1 0 0 0 0      1    VECTOR
HEXEXP         1 0 0 0 0      1    VECTOR
I              0 0 0 0 1      0    NOVAL
INFILE         0 0 0 0 1      0    NOVAL
     (SPECIAL (NIL))
INITIALOPEN    1 0 0 0 0      0    VECTOR
INTERFACE      1 0 0 0 0      0    VECTOR
INTERN         1 0 0 0 0      1    VECTOR
INTERNX        1 0 0 0 0      0    VECTOR
INT2LINT       1 0 0 0 1      1    VECTOR
IUNIT          0 0 0 0 1      0    NOVAL
IUNIT1         0 0 0 0 1      0    NOVAL
J              0 0 0 0 1      0    NOVAL
L              0 0 0 0 1      0    NOVAL
     (INTSYM 22528)
LAC            1 0 0 0 0      1    VECTOR
LAPEVAL        1 0 0 0 0      1    VECTOR
LAPLISTFLAG    0 0 0 0 1      0    NIL
     (APVAL (NIL) SPECIAL (NIL))
LAP360         1 0 0 0 0      2    VECTOR
LARGEEXPT      1 0 0 0 1      2    VECTOR
LAST           1 0 0 0 0      1    VECTOR
LBKTRCE        1 0 0 0 0      0    NOVAL
LBRAC          0 0 0 0 1      0    <
     (APVAL (<) SPECIAL (NIL))
LCOPY          1 0 0 0 1      1    VECTOR
LDIFF          1 0 0 0 1      2    VECTOR
LEFTSHIFT      1 0 0 0 0      2    VECTOR
LENGTH         1 0 0 0 0      1    VECTOR
LENGTHCODE     1 0 0 0 0      1    VECTOR
LESSP          1 0 0 0 1      2    VECTOR
LINENO         0 0 0 0 1      0    NOVAL
LINEUPINPUT    1 0 0 0 0      0    VECTOR
LINTEXP        1 0 0 0 1      1    VECTOR
LINTP          1 0 0 0 1      1    VECTOR
     (MACRO %G-729)
LINT2INT       1 0 0 0 1      1    VECTOR
LIST           1 1 1 0 1      2    VECTOR
     (MACRO %G-1003)
LISTING        0 0 0 0 1      0    NOVAL
LISTOPOBJECTS  1 0 0 0 1      0    VECTOR
LISTP          1 0 0 0 0      1    VECTOR
     (MACRO %G-770)
LIST2IVEC      1 0 0 0 0      1    VECTOR
LIST2RVEC      1 0 0 0 0      1    VECTOR
```

IDEMTP 75
IVECP 76
LABEL 76
LAMBDA 76

```
LIST2SVEC       1 0 0 0 0       1     VECTOR      79
LIST2VEC        1 0 0 0 0       1     VECTOR      79
LNCDOP          1 0 0 0 0       1     VECTOR
LOADEX          1 0 0 0 0       0     BELOW
LOADLISP        1 0 0 0 1       3     VECTOR      1Ø6
LOC             0 0 0 0 1       0     NOVAL
LOCATE          1 0 0 0 0       1     VECTOR
LOGAND          1 0 1 0 0       2     VECTOR      9Ø
LOGOR           1 0 1 0 0       2     VECTOR      9Ø
LOGXOR          1 0 1 0 0       2     VECTOR      9Ø
LPAR            0 0 0 0 1       0     (
     (SPECIAL (NIL))
LPLUS           1 0 0 0 1       2     VECTOR
LQUOT1          1 0 0 0 1       2     VECTOR
LTIMES          1 0 0 0 1       2     VECTOR
LTIMES1         1 0 0 0 1       2     VECTOR
MACLAP          1 0 0 0 0       1     VECTOR      8Ø
MACRO           1 0 0 0 0       1     VECTOR
MAKEPROP        1 0 0 0 0       3     VECTOR      8Ø
MAP             1 0 0 0 0       2     VECTOR      8Ø
MAPCAR          1 0 0 0 1       2     VECTOR      8Ø
MAPCON          1 0 0 0 1       2     VECTOR      81
MAPLIST         1 0 0 0 0       2     VECTOR      81
MAX             1 0 1 0 1       2     VECTOR      9Ø
     (MACRO %G-426)
MDEF            1 0 0 0 0       1     VECTOR      81
MEMBER          1 0 0 0 1       2     VECTOR      81
MENGTH          0 0 0 0 1       0     NOVAL
MIN             1 0 1 0 0       2     VECTOR      9Ø
     (MACRO %G-424)
MINUS           1 0 0 0 1       1     VECTOR      91
     (RENAME %G-359)
MINUSLINT       1 0 0 0 1       1     VECTOR
MINUSLINTP      1 0 0 0 1       1     VECTOR
MINUSP          1 0 0 0 1       1     VECTOR      92
     (RENAME %G-338)
MKNAM           1 0 0 0 1       0     VECTOR      1Ø6
MKQT            1 0 0 0 0       1     VECTOR
MODE            0 0 0 0 1       0     NOVAL
MYGREATP        1 0 0 0 1       2     VECTOR
N*TUPLEQ        1 0 0 0 0       2     VECTOR      82
NAME            0 0 0 0 1       0     NOVAL
NCONC           1 0 0 0 1       2     VECTOR      82
NILFN           1 0 0 0 0       0     VECTOR
NODESLESSP      0 0 0 0 1       0     NOVAL
NOT             1 0 0 0 0       1     VECTOR      82 ——— NOTE 1Ø6
NOVAL           0 0 0 0 1       0     NOVAL
     (SPECIAL (NIL))
NOVALUE         1 0 0 0 1       0     NOVAL
NQSUBST         1 0 0 0 0       3     VECTOR      82
NULL            1 0 0 0 0       1     VECTOR      82
NUMBERP         1 0 0 0 1       1     VECTOR      92
     (RENAME %G-411)
NUMEXPLD        1 0 0 0 0       0     BELOW       1Ø6
```

```
NUMOB            1 0 0 0 0       0     BELOW        107
OBEY             1 0 0 0 1       1     VECTOR       82
OBLIST           0 0 0 0 1       0     VECTOR
     (SPECIAL (NIL))
OLDERRORSTP      0 0 0 0 1       0     BELOW
     (SPECIAL (NIL))
OPEN*SEQ         1 0 0 0 1       5     VECTOR       107
OPENEDFILES      0 0 0 0 1       0     ((IDPROP BX@4 #TEST()     107
                                                    00001280# NIL
1023 NIL   NIL 0 71 P1 SYSIN NIL) (OUU BX@3 #1023 NIL   NIL 0 71 P1 SYSI
                                          # NIL 1023 NIL   NIL
 0 72 P1 OUT NIL) (LISPOT BX@2 #

        # NIL 1023 NIL   NIL 0 120 * TOUT NIL) (LISPIT BX@1 #RDF(IDPROP
OUU)
                                                  # NIL 133 NIL

) NIL 0 130 * TIN NIL))
     (SPECIAL (NIL))
P                0 0 0 0 1       0     NOVAL  <------ OR OSOPN   83 109
PACK             1 0 0 0 0       1     VECTOR       109
PAFORM           1 0 0 0 0       2     VECTOR
PAFORM1          1 0 0 0 0       2     VECTOR
PAIR             1 0 0 0 0       2     VECTOR       83
PAIRMAP          1 0 0 0 0       4     VECTOR
PALAM            1 0 0 0 0       2     VECTOR
PART             1 0 0 0 1       1     VECTOR
PASSONE          1 0 0 0 0       2     VECTOR
PA1              1 0 0 0 0       1     VECTOR
PA11             1 0 0 0 0       2     VECTOR
PA12             1 0 0 0 0       1     VECTOR
PA14             1 0 0 0 0       1     VECTOR
PA15             1 0 0 0 0       1     VECTOR
PA2              1 0 0 0 0       1     VECTOR
PA3              1 0 0 0 0       1     VECTOR
PA4              1 0 0 0 0       3     VECTOR
PA5              1 0 0 0 0       2     VECTOR
PA6              1 0 0 0 0       2     VECTOR
PA7              1 0 0 0 0       2     VECTOR
PA8              1 0 0 0 0       3     VECTOR
PA9              1 0 0 0 0       2     VECTOR
PERIOD           0 0 0 0 1       0     .
     (SPECIAL (NIL))
PHASE2           1 0 0 0 0       2     VECTOR
PI1              1 0 0 0 0       1     VECTOR
PI2              1 0 0 0 0       2     VECTOR
PI3              1 0 0 0 0       3     VECTOR
PLANTDDNAM       0 0 0 0 1       0     BELOW        109
PLANTNAME        1 0 0 0 0       4     VECTOR
PLUS             1 0 1 0 1       2     VECTOR       90
     (MACRO %G-430)
PLUSS            0 0 0 0 1       0     +
     (SPECIAL (NIL))
PRETTYPRINT      1 0 0 0 1       1     VECTOR       109
PRINHEX          1 0 0 0 1       1     VECTOR       109
```

```
PRINSCHR        1 0 0 0 0       1     VECTOR      109
PRINT           1 0 0 0 1       1     VECTOR      109
PRINTARG        1 0 0 0 0       0     VECTOR
PRINTCH         1 0 0 0 0       1     VECTOR      110
PRINTDEF        1 0 0 0 0       1     VECTOR      109
PRINTEXP        1 0 0 0 1       1     VECTOR      110
PRINTVAL        1 0 0 0 0       0     VECTOR
PRINV           1 0 0 0 0       1     VECTOR
PRIN0           1 0 0 0 1       1     VECTOR      110
PRIN1           1 0 0 0 0       1     VECTOR      110
PRIN1B          1 0 0 0 1       1     VECTOR      111 ---- PRINV 111
PRO             0 0 0 0 1       0     NOVAL
PROG            1 1 1 0 0       2     VECTOR      83
     (FEXPR (LAMBDA (X Y) (EVPROG X)))
PROGGO          0 0 0 0 1       0     NOVAL       131
PROGITER        1 0 0 0 0       2     VECTOR
PROGRET         0 0 0 0 1       0     NOVAL
PROGVAL         0 0 0 0 1       0     NOVAL
PROG2           1 0 0 0 1       2     VECTOR      84
PROP            1 0 0 0 0       3     VECTOR      84
PROPERTIES      1 0 0 0 1       1     VECTOR
PROPHDPRNT      1 0 0 0 0       1     VECTOR
PROPS           1 0 0 0 1       1     VECTOR
PRTCONTL        0 0 0 0 1       0     0           102 [SPECIAL ??]
PRTSTRG         1 0 0 0 0       1     VECTOR      111
PR1BUF          0 0 0 0 1       0     -958        102,105
PT              1 0 0 0 1       3     VECTOR
PUNWORD         1 0 0 0 0       5     VECTOR      84
Q               0 0 0 0 1       0     NOVAL
     (SPECIAL (NIL))
QSORT           1 0 0 0 1       1     VECTOR      QUICKSORT ?
QTEXTEN-NUM     0 0 0 0 1       0     NOVAL
QTEXTEN-VEC     0 0 0 0 1       0     NOVAL
QUIET           0 0 0 0 1       0     *T*
     (SPECIAL (NIL))
QUOTE           1 1 0 0 0       1     VECTOR      84,15
     (FEXPR (LAMBDA (X Y) (CAR X)))
QUOTED          0 0 0 0 1       0     NOVAL
QUOTIENT        1 0 0 0 1       2     VECTOR      91
RBRAC           0 0 0 0 1       0     >
     (APVAL (>) SPECIAL (NIL))
RDCHR           1 0 0 0 1       0     VECTOR      111
RDF             1 0 0 0 1       2     VECTOR
RDS             1 0 0 0 1       1     VECTOR      111
READ            1 0 0 0 1       0     VECTOR      112
READNCH         1 0 0 0 0       2     VECTOR
READ0           1 0 0 0 0       0     VECTOR      112
READ1           1 0 0 0 0       0     VECTOR      112
RECIP           1 0 0 0 0       1     VECTOR
RECLAIM         1 0 0 0 1       0     BELOW       85
REMAINDER       1 0 0 0 1       2     VECTOR      91
REMFLAG         1 0 0 0 0       2     VECTOR
REMOB           1 0 0 0 0       1     VECTOR      85
REMPROP         1 0 0 0 0       2     VECTOR      85
```

```
RETN            0 0 0 0 1       0     NOVAL
REVERSE         1 0 0 0 0       1     VECTOR
RPAR            0 0 0 0 1       0     )
     (SPECIAL (NIL))
RPLACA          1 0 0 0 0       2     VECTOR
     (MACRO %G-979)
RPLACD          1 0 0 0 0       2     VECTOR
     (MACRO %G-976)
SASSOC          1 0 0 0 1       3     VECTOR
SAVLIS          1 0 0 0 0       2     VECTOR
SCRUB           0 0 0 0 1       0     NOVAL
SELECT          1 1 1 0 1       2     VECTOR
SELECT2         0 0 0 0 1       0     VECTOR
     (EXPR (LAMBDA (B M) (COND ((NULL (CDR M)) (EVA1 (CAR M)))
((EQUAL B (EVA1 (CAAR M))) (EVA1 (CADAR M))) (T (SELECT2 B (CDR
M))))))
SETC            1 0 0 0 0       2     VECTOR
SETIV           1 0 0 0 0       3     VECTOR
SETQ            1 1 0 0 0       2     VECTOR
     (FEXPR (LAMBDA (X Y) (SET (CAR X) (EVA1 (CADR X)))))
SETRV           1 0 0 0 0       3     VECTOR
SETSV           1 0 0 0 0       3     VECTOR
SHUT            1 0 0 0 1       2     VECTOR
SLASH           0 0 0 0 1       0     /
SNAPS           1 0 0 0 0       3     BELOW
SPECIAL         1 0 0 0 0       1     VECTOR
ST              0 0 0 0 1       0     NOVAL
     (INTSYM 20480)
STANDEODADIN    1 0 0 0 0       0     VECTOR
STANDEODADOUT   1 0 0 0 0       0     VECTOR
STANDSYNADIN    1 0 0 0 0       1     VECTOR
STANDSYNADOUT   1 0 0 0 0       1     VECTOR
STAR            0 0 0 0 1       0     *
     (SPECIAL (NIL))
STOMAP          0 0 0 0 1       0     NOVAL
STORE           1 0 0 0 0       2     VECTOR
STRCONC         1 0 1 0 1       2     VECTOR
STRINGP         1 0 0 0 0       1     VECTOR
     (MACRO %G-734)
STRLENGTH       1 0 0 0 1       1     VECTOR
SUBRFUN         1 0 0 0 0       2     VECTOR
SUBST           1 0 0 0 1       3     VECTOR
SUB1            1 0 0 0 1       1     VECTOR
SUPERPRINT      1 0 0 0 0       1     VECTOR
SUPVINFILE      0 0 0 0 1       0     IDPROP
     (SPECIAL (NIL))
SUPVOUTFILE     0 0 0 0 1       0     OUU
     (SPECIAL (NIL))
SWITCH          0 0 0 0 1       0     NOVAL
SYNAD           0 0 0 0 1       0     BELOW
TEMPUS-FUGIT    1 0 0 0 0       0     VECTOR
TEREAD          1 0 0 0 1       0     VECTOR
TERPRI          1 0 0 0 1       0     VECTOR
TEST            1 0 0 0 1       0     VECTOR
```

86 RETURN 84,85

RVECP 86

87

88

116

112

113

113

113

88

88

89

SVECP 89

113 TAB 113

89

92

109

113

114

```
TIMES           1 0 1 0 1       2     VECTOR      90
     (MACRO %G-428)
TR*PRINT        1 0 0 0 0       1     VECTOR
TRACE           1 0 0 0 0       1     VECTOR      117
TRACECOL        0 0 0 0 1       0     NOVAL
TRACEMARGIN     0 0 0 0 1       0     NOVAL
TRACE1          1 0 0 0 0       2     VECTOR      116
UBUFLIN         0 0 0 0 1       0     71          108
     (SPECIAL (NIL))
UNCOMMON        1 0 0 0 0       1     VECTOR      89
UNPACK          1 0 0 0 1       1     VECTOR      114
UNSPECIAL       1 0 0 0 0       1     VECTOR      89
UNTRACE         1 0 0 0 0       1     VECTOR      72, 118
UNWIND          1 0 0 0 1       0     -494
UPI-FLOAT       0 0 0 0 1       0     NOVAL
VECGTP          1 0 0 0 1       2     VECTOR
VECP            1 0 0 0 0       1     VECTOR      89
     (MACRO %G-761)
VECTLESSP       0 0 0 0 1       0     NOVAL
WRS             1 0 0 0 1       1     VECTOR      114
X               0 0 0 0 1       0     NOVAL
XERMAN          0 0 0 0 1       0     TEST
     (SPECIAL (NIL))
YERMAN          0 0 0 0 1       0     NIL
     (SPECIAL (NIL))
ZERMAN          0 0 0 0 1       0     DONE
     (SPECIAL (NIL))
ZEROP           1 0 0 0 1       1     VECTOR      92
     (RENAME %G-385)
```