TR-546 (supersedes TR-400)
MCS-76-23763 June, 1977

VLISP for PDP-11s with Memory Management

Robert L. Kirby

Computer Science Center
University of Maryland
College Park, Maryland 20742

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

TR-546 (supersedes TR-400)
MCS-76-23763 June, 1977

VLISP for PDP-11s with Memory Management

Robert L. Kirby

Computer Science Center
University of Maryland
College Park, Maryland 20742

ABSTRACT

A new large scale implementation of LISP, VLISP, for PDP-11s with memory management is described as implemented at the University of Maryland. The implementation is modelled after the University of Wisconsin's UNIVAC 1100 LISP. Four versions are available: an interpreter for use with the Virtual Operating System (VOS) being developed at the University of Maryland; a version compatible with DEC's Disk Operating System (DOS) using a VOS emulator; a stand-alone version which also emulates VOS; and a version for use with Bell Laboratories' UNIX operating system. This documentation 1) explains how to use the implementation; 2) discusses the problems, limitations, and internal configuration; 3) briefly describes the available system software including a Pretty Printer, an S-expression editor, a LISP function compiler, and micro-PLANNER; and 4) provides a synopsis of the pre-defined LISP functions.

The support of the Mathematical and Computer Sciences Division, National Science Foundation under Grant MCS-76-23763 is gratefully acknowledged, as is the help of Prof. Azriel Rosenfeld, Prof. Chuck Rieger, Ms. Joan Weszka, Mr. Mache Creeger, and Mr. Ken Hayes in the preparation of this document.

COPYRIGHT, 1977, Robert L. Kirby

This document may be copied for non-profit purposes or for any purpose of the United States government, provided that any copy includes the copyright notice and this statement.

Table of Contents

1. The LISP interpreter.

VLISP for PDP-11s with memory management is modelled on the University of Wisconsin's LISP for the UNIVAC 1100 series computers. Familiarity with LISP is assumed. The manuals describing Wisconsin's UNIVAC 1100 LISP give a more detailed description of the language. The differences and peculiarities of this PDP-11 LISP dialect are cescribed here. The first chapter gives a brief overview of VLISP, comparing it to Wisconsin's UNIVAC 1100 LISP. The second chapter describes the internal configuration of the interpreter. A knowledge of the internal configuration may then be used to write extensions to the interpreter as described in the third chapter. The appendices give implementation instructions and a synopsis of pre-defined LISP functions.

1.1. General Capabilities.

VLISP is a moderate-scale, in-core implementation using two stacks, deep or global bindings, multiple data types, and type determination through address location. The initial code and data for the LISP interpreter occupy approximately 9K words of core. The Virtual Operating System / Distributed Computer Network (VOS/DCN) developed at the University of Maryland, a VOS emulator for use with DEC's Disk Operating System (DOS), and a stand-alone system which emulates some VOS functions are operating systems which support the LISP interpreter. Conditional assembly instructions select the host operating system. With a small operating system, the LISP interpreter needs nearly 16K words of core just to sign on. Expanding just the potential data area, 42K words of core could be supported. 28K words of core would probably be needed for much useful computing. Compiled and assembled code could use yet anctner 24K words. With the maximum supportable configuration, about 14K CONSed nodes could be used for data, assuming that almost all programs are compiled. The amount of core which can be supported is reduced on PDP-11s which do not support seperate instruction (I) and data (D) spaces (e.g. the PDP-11/40). The host computer must also support the Extended Instruction Set (EIS) consisting of the MUL, DIV, ASH, and ASHC instructions. PDP-11/45s and PDP-11/70s support EIS with standard hardware.

Bell Laboratories' UNIX operating system also supports VLISP. However, before the LISP compiler can be used under UNIX on PDP-11/45s or PDP-11/70s, the UNIX operating system must be extended. The extensions consist of two pages of straightforward additions to the "c" code of the UNIX operating system. The extended UNIX can support for each process an additional, writable I-space following the write-protected code in seperated-I-and-D-space mode. The VLISP compiler can then write instructions into the I-space area to increase efficiency. A

second, smaller improvement to the UNIX operating system provides one-line-at-a-time input from files other than teletypes. After the improvement, which creates a new "sys" call by adding 20 lines of "c" code, callers can read input from file systems and pipes up to and including the first new-line, line-feed character encountered. Without the second UNIX modification, LISP requests ASCII character input one character at a time so that input may alternate between files. VLISP only reads characters on the current line that is to be immediately processed. Unfortunately this slows down VLISP input processing. LISP software may access the powerful features of UNIX such as FORKs, EXECs, PIPEs and the other "sys" and "shell" calls described in the UNIX Programmer's Handbook.

1.2. Available Functions.

Except as noted in subsequent sections, the following functions have been implemented consistently with the definitions in the UNIVAC 1108 LISP Reference Manual [Norman 1969] and the additions produced at the University of Maryland. An appendix provides a synopsis of these pre-defined functions.

ADD1 ALIST AMB AND APPEND ARRAY ARRAYL ARRAYP ASSOC
ATOM ATSYMP ATTEMPT

BACKSP BREAK

CLEARBUFF CLOSE COMPLEMENT COMPRESS COND CONS CSET
CSETQ CURRCOL

DEFINE DEFMAC DEFSPEC DELIM DIFFERENCE DO DOUBLE DUMP

ENTIER EQ EQUAL ERASE ERROR EVAL EXEC EXPLODE EXPLODE2

FIXP FLOAT FLOATP FLAG FORK FUNCTION

GENSYM GET GO GREATERP

IFFLAG IFTYPE INDEX INTO

LAMBDA LAMDA LEFTSHIFT LENGTH LESSP LISP LIST LOAD
LOGAND LOGOR LOGXOR

MANIFEST MAP MAPC MAPCAR MAPLIST MEMBER MINUS MINUSP

NCONC NOT NTH NULL NUMBERP

OBLIST ONDEX ONTO OPEN OR

PIPE PLENGTH PLENGTH2 PLIMIT PLUS PRINT PRIN1 PRIN2
PROG PROP PUT

QUOTE QUOTIENT

READ READCH READMAC REMAINDER REMOB REMOBP REMPROP
REQUEST RETURN REVERSE RPLACA RPLACD

SET SETCOL SETQ SH SINGLE SPACE STACK STRING SUBST SUB1
SYS

TERPRI TIME TIMES TOKEN TRAP

UNBREAK UNFLAG

WAIT

ZEROP

*BEGIN *CAR *CDR *CHAIN *DEF *DEPOSIT *EMIT *EPT *EXAM
*MACRO *ORG *REVERSE *SPEC

1.3. Additional Features.

VLISP on PDP-11s has some new features that are not provided on Wisconsin's UNIVAC 1100 LISP. VLISP provides array and array utility functions, upper-lower case characters, an empty atom and string, file opening and closing functions, and specific functions for interacting with the host operating system.

1.3.1. Arrays.

Array functions manipulate a numerically-indexed contiguous area of S-expressions, logical data, or numerical data.

1.3.1.1. Creating Arrays

Evaluating the LISP expression

(CSETQ ARR (ARRAY SIZE TYPE))

creates a one dimensional array of logical length SIZE and globally binds it to the SYMBOLic atom ARR. The actual physical size in bytes of the array depends on the array type given by the second parameter, TYPE, a fixed-point number. Permissible values of the TYPE parameter are described below. If TYPE is omitted, ARRAY produces an array of pointers. The value of the ARRAY function is a function whose values may be obtained by evaluating:

(ARR X)

where ARR evaluates to the created function and X to a positive, fixed-point number. An element of this array may be set to the value of VAL by evaluating:

```
(ARR X VAL)
```

The created array function, ARR, returns the value of VAL regardless of what ARR stores in the array.

Both the logical length (specified by the parameter SIZE) and the physical size in bytes must be positive, non-zero, fixed-point numbers (octal or integer) that a 15-bit number (i.e. less than 32,768 and greater than zero) can express. If ARRAY attempts to create an array with an improper SIZE parameter, the LISP interpreter will produce ar internal error -9 as if it were evaluating

```
(ERRCR -9) .
```

If a current ATTEMPT invocation catches error -9, processing continues at the restart point. Otherwise, the interpreter prints the message

```
WARNING, X BAD INDEX
```

where x is the offending SIZE parameter. The interpreter then restarts at the latest level of LISP supervision by requesting a new expression to evaluate. The ARRAY function rounds the creation size of the bit array types (logical and binary) up to the next multiple of 8 to simplify array index checking.

The function created by ARRAY checks that the array index, the first parameter of the created array function, is a fixed-point number which lies between 1 and the logical length inclusive. The created array function makes no conversion of the array index from a floating-pcint number into a fixed-point number. Array indices out of range also produce error -9 and, if undetected by ATTEMPT, the same message as above where X is the offending array index. However, ATTEMPT may be used while sequentially referencing array elements tc catch the error -9 of an out-of-range reference. Thus, the programs need not explicitly check for the last element of arrays during sequential references. For example, evaluating the following S-expression defines a function CARRAY that creates an array of any desired size and fixed-point type whose elements are their integer indices.

Figure 1 - Define CARRAY to create arrays.

```
(CSETQ CARRAY  ? Establish a global binding
 (LAMBDA A     ? Make a list of the function's parameters
         (SETQ A (ARRAY (STACK A))) ? Use list of SIZE and TYPE
         (ATTEMPT [PROG <(X 1)>     ? Start indexing at one
                   LOOP <A X X>     ? Initialize array value
                        <SETQ X (ADD1 X)> ? Increment index
                        <GO LOOP>]
                  [-9 A]            ? Return array when done
          )))
```

The function CARRAY handles logical and floating-point arrays differently than fixed-point arrays. The elements of logical type arrays created by CARRAY will all be T (true). Floating-point arrays use the fixed-point indices without conversion.

1.3.1.2. Types of Arrays.

The second parameter of ARRAY (or its absence) specifies the type of array:

TYPE ARGUMENT	MEMBER DESCRIPTION	INITIAL VALUE	RANGE
Omitted	Pointer	Undefined	Any S-expression
0	Pointer	Undefined	Any S-expression
1	Logical	NIL	T (true) or NIL (false)
2	Binary	0	0 or 1
3	Signed byte	0	-128 to 127
4	Unsigned byte	0	0 to 255
5	16-bit integer	0	-32768 to 32767
6	2-word floating	0.0E0	Single precision floating point
7	4-word floating	0.0D0	Double precision floating point

All elements of type 0, pointer arrays, are initially undefined. Values should be assigned to pointer array elements before they are referenced. If not, an error occurs after referencing an undefined, pointer-array value. The VLISP interpreter uses an error -8 as if the S-expression

(ERROR -8)

were evaluated. If a current ATTEMPT call catches error -8, processing continues at the restart point. Otherwise, the interpreter prints the message

WARNING, X IS UNBOUND ,

where X is the index of the unbound array element, followed by the solicitation

Help:

that requests an expression whose value may be used instead of the undefined array element.

1.3.1.3. Array Utility Functions.

Two utility functions for retrieving an array's specifications are available.

If the parameter of ARRAYP, the array predicate, is an array, then ARRAYP returns a number indicating the array type. If the parameter of ARRAYP is not an array, then ARRAYP returns NIL (false) as its value. For example, suppose the SYMBOLic atom, ARR, has been given an array value by evaluating

(CSETQ ARR (ARRAY 17 1)) .

Then the expression

(ARRAYP ARR)

will return a value of 1, which specifies a logical array, an array that stores either NIL (the initial value) or T for non-NIL values.

ARRAYL, the array length function, returns the logical length of an array given as a parameter. If the parameter of ARRAYL is not an array, ARRAYL returns NIL. The logical length specified by SIZE during the creation of logical and binary arrays is rounded up to the next multiple of 8 to align logical and binary arrays (represented by bits) on a byte boundary. In the example above, when 17 is given for the length of a logical or binary array, ARRAY produces ar array of logical length 24. Hence,

(ARRAYL ARR)

evaluates to 24. Other types of arrays just use the length specified by the first parameter of ARRAY as the logical length.

1.3.2. Full ASCII Character Set.

The full ASCII character set is available for use in atom names and strings. However, to avoid a proliferation of atom names that differ only in character case, upper case letters are automatically converted to lower case before being used in symbol names. This feature can be overridden by using an escape character (') before each upper case letter which is not to be converted to lower case.

While using the DOS or stand-alone versions of the LISP interpreter, lower case ASCII may not be desired or supported. In this case, the assembly line

```
.ENABLE LC ; Use lower case
```

should be commented out of the code module TRAPS.MAC of the DOS versions of the interpreter source code. In an interpreter assembled without the above line commented, READ and TOKEN convert lower case characters encountered to upper case unless the escape character, which is initially exclamation point (!), precedes them.

1.3.3. Empty Atom and String.

An empty string and an atom whose print name is empty have been provided. Both have a print length of zero. When READ or TOKEN encounter a single pair of double quotation marks (""), they reference the empty string. However, READ and TOKEN cannot directly scan the empty atomic symbol. The expression

```
(ATSYMB "")
```

will evaluate to the empty atom, if it is needed.

1.3.4. VOS and DOS operating system calls.

Only VOS and the VOS emulator under DOS versions of VLISP provide the VOS and DOS operating system call functions.

1.3.4.1. The VOS and DOS TRAP Function.

The TRAP function provides an interface to the VOS operating system via the TRAP instruction with offset zero (0). The VOS emulator may then use appropriate DOS system calls. The first parameter of TRAP is a fixed-point number that is placed in CPU register R5 for use as the TRAP instruction offset by the operating system. The other, optional parameters are used to place values in registers from R0 to R4 to be passed to the operating system. A NIL parameter or omitted parameter passes zero. Both strings and arrays pass pointers to the first words of their data. Atoms pass their print names which are strings. Fixed-point numbers pass their values. Floating-point numbers pass a pointer to their values. CONSed nodes pass pointers to their CARs. The value of the TRAP function is a CONSed pair of octal numbers giving the values returned in registers R0 and R1 by the operating system.

1.3.4.2. The VOS and DOS OPEN function.

The VOS and DOS OPEN function provides the subset of the services provided by the TRAP function with offset 370 (octal). The OPEN function may be used with from one to three parameters. For example, when evaluating

(OPEN FILE-STRING MODE FILE-NUMBER)

the first parameter, FILE-STRING, evaluates to a string or atomic symbol, the external name of the file to assigned an internal logical file number. Under DOS each permissible internal file number is associated with a default external file name on the system device, "SY:". The default external name is the internal number followed by the suffix ".LSP". The OPEN command replaces the external association with a new one given by the first parameter, FILE-STRING. The first time the file is accessed for either input or output, the VOS emulator searches for a file with the given external name using any User Indentification Code (UIC) given as part of the file name in the standard DOS syntax. If no UIC was specified, the VOS emulator first searches the current UIC directory. If the VOS emulator does not find the file under the current UIC directory, the VOS emulator then searches the supervisor UIC ("[1,1]") directory. Most LISP system software should be available under the supervisor UIC directory. Inability to locate a file produces a system error and a restart. If MODE, the second parameter of OPEN, is NIL or zero, or is omitted, then the first write to the logical file-number attempts to first .INIT and .OPEN in contiguous mode (013) in case the file is contiguous. If the file is not contiguous, the first write attempts to .OPEN the file in extension mode (3) which is synonymous with output mode if the file did not previously exist. On the other hand, the first read directed to the file attempts to .INIT and .OPEN the file in input mode (4). If MODE is non-NIL and non-zero, the first I/O attempt uses the given mode if possible. This provides a way to .INIT and .OPEN contiguous files in update mode (1), so that the emulator may use random access. When FILE-number, the optional third parameter of OPEN, specifies a fixed-point number, the VOS emulator uses that number as the internal logical file-rumber instead of searching for an available, unused, logical file-number. OPEN returns the logical file-number which may be used by CLEARBUFF, TERPRI, and LOAD to access the new external file association. The external file-name string may also contain both switches and a second file name following the standard input-and-output-file-seperator character ("<"). The five possible modes available through the second parameter, MODE, can also be specified by switches:

Switch	Value	Description
/I	4	Input from existing file,
/E	3	Extension of an existing file,
/O	2	Output to a new file,
/U	1	Update an existing contiguous file, and
/C	013Q	Contiguous file started empty.

The logical number association available through a third parameter, FILE-NUMBER, could also be given by a numeric switch (e.g. "/5"). An allocation size for creating contiguous files given as a 64-byte-block count is specified by the switch "/AL:#". For example, in evaluating

```
(OPEN "NEWFIL.LSP/AL:32/C")
```

32 is a switch parameter allocating 32 contiguous blocks of file space. OPEN may also rename and append files through the switches "/RE" and "/AP", respectively. In order to append the linked file "FILE2.LSP" tc the end of the linked file "FILE1.LSP", evaluate

```
(OPEN "SY:FILE1.LSP<SY:FILE2.LSP/AP")
```

DOS pads FILE1.LSP with nulls before appending the other linked file. In DOS BATCH mode, the opening facility with switches is available to command strings in the run stream with the standard syntax, i.e. preceded by a number sign (#). For example, when the BATCH command

```
#NEWFIL.LSP/4<OLDFIL.LSP/RE
```

is encountered, the file originally named "OLDFIL.LSP" is renamed "NEWFIL.LSP" and associated with logical file-number 4.

1.3.4.3. The VOS and DOS CLOSE function.

The VOS and DOS CLOSE furction calls the operating system to close and release any external file and device associated with the parameter, a fixed-point number. The OPEN function would usually have associated external file names to the parameter, a logical file number. The logical file-number given may be reassigned by a subsequent OPEN call to a different external association. CLOSE makes the buffer space and device control blocks in the DOS monitor available provided more recently opened files are also closed, since the DOS monitor allocates buffer and control block space from a stack. The CLOSE function returns NIL.

1.3.5. UNIX VLISP Operating System Calls.

UNIX VLISP provides complete access to operating system calls. Either specific system calls using the

```
sys 0 ; buff / irdirect system call
```

machine instruction like "c" language and "as" assembler programs, or general calls to the shell, "sh", may be made. UNIX VLISP facilitates passing strirgs endec by a zero byte that many system-call syntaxes require. The LISP interpreter converts internal types STRING, SYMBOL, and CONSED into STRING and insures that the oata is followed by a zero byte, even if this forces creating a slightly longer copy of the original. The interpreter passes the print name of SYMBOLic atoms, the LISP variables, as a string except for NIL for which the interpreter passes zero. If a system call receives a CONSed node parameter, VLISP assumes the node heads a list of single-character atoms, single-character strings, or fixed-point, ASCII-character values. The interpreter

concatenates the implied characters into a string. The system call eventually passes a pointer to the first word of the string followed by a guaranteed zero byte. Programs themselves need not supply the zero byte after strings. In system calls, VLISP passes the value of fixed-point-number parameters and a pointer to the first word of floating-point-number parameters. The interpreter handles function LINKERs in two ways. System calls pass the I-space address (*CDR) of non-array LINKER parameters so that signals may be caught by user-written, machine-code routines. When VLISP receives an array function LINKER, a pointer to the first word of the array data is passed. If the array data, that has internal type STRING, were used directly as a parameter, the interpreter might create a copy of the array in order to satisfy the zero byte requirement. Hence, in order to pass an array of data via a system call, e.g. "gtty" or "fstat", the invocation should use the array LINKER, not the array data. The invocation must provide arrays long enough to receive all data returned by system calls. The operating system overwriting the area following an array does grave damage to VLISP storage allocation.

When the UNIX operating system detects a venial error during a system call, the operating system returns from the call with the carry (C) bit on in the processor status (PS) register to signal an error condition. When VLISP detects this error condition after any system call except CLOSE or EXEC, VLISP generates an internal error 0 as if the S-expression

(ERROR 0)

had been evaluated. If a current ATTEMPT invocation catches error 0, processing continues at the restart point. Otherwise, the interpreter prints the message

WARNING, X SYSTEM ERROR

where X is the integer error number returned in CPU register R0 by the UNIX system call, and restarts at the latest level of LISP supervision.

1.3.5.1. The SYS Function.

The SYS function allows access to most of the UNIX operating system calls. UNIX VLISP provides other functions for the cases in which SYS cannot efficiently handle the syntax. Programs should invoke the SYS function with at least one parameter, the SYS offset number. In

(SYS ARG0 . . . ARGn)

the interpreter converts the first parameter, ARG0, the offset number, to integer type, and uses it to construct a machine instruction in D-space

```
ibuff: sys arg0 / start of indirect buffer
```

for use in an indirect system call in I-space

```
sys 0 ; ibuff / indirect.
```

The interpreter converts any remaining parameters according to the above rules and places them after the system call in the indirect call buffer. SYS also places the last two parameters in CPU registers R1 and R0, respectively, just before the indirect system call. If the system call returns with an error condition, i.e. the C (Carry or error) bit is on, the interpreter uses the LISP-system-error procedure. Otherwise, SYS returns the value that the UNIX operating system returned in CPU register R0 in integer representation. The system call section (Part II) of the UNIX Programmer's Manual contains the particulars of each UNIX system call. Following these instructions for assembly language format calls, the program supplies, in order, the SYS offset, any in-line parameters, and any values to pass in registers. The LISP interpreter converts most parameters to the natural, UNIX-system-call format to minimize programming effort. The available support software includes a file, "/lisp/sy", that gives examples of system calls.

1.3.5.2. The UNIX OPEN Function.

The OPEN function calls the UNIX operating system to obtain an internal, logical file-number used to access a pre-existing file. The OPEN function gets one or two parameters, e.g.

```
(OPEN EXTERNAL-NAME I/O-MODE) .
```

The first parameter, EXTERNAL-NAME, specifies an external file name that OPEN will convert to internal type STRING, ended with a zero-byte, if needed. The second optional parameter, I/O-MODE, a fixed-point number, sets one of the permissible I/O modes: zero (0) for reading only, one (1) for writing only, or two (2) for both reading and writing allowed. If the second parameter, I/O-MODE, is omitted, OPEN uses zero (0) to set read-only mode. If UNIX opens a file for writing, UNIX starts placing output at the beginning of the file, overwriting any existing data without first truncating the file. To extend an existing file, before sending any output, evaluate the S-expression

```
(SYS 19 0 2 FILE-NUMBER)
```

to perform a seek (sys 19.) to the end of the file (offset=0 and ptrname=2) where FILE-NUMBER is bound to the value returned by the OPEN call. In order to create or truncate a UNIX file, a function CREAT could be defined by evaluating the following S-expression.

Figure 2 - Define CREAT function.

```
(CSETQ CREAT                    ? Define constant binding
 (LAMBDA (NAME . MODE)          ? Optional mode parameter
         (SYS 8 NAME            ? Call system to create external name
          (COND [MODE <CAR MODE>] ? Use any given mode
                [666Q]))))) ? Else default to read/write for all
```

CREAT could then return a logical, internal file-number of a new or previously existing, truncated file. If UNIX detects an error while processing an OPEN, CREAT, or SEEK call, the interpreter generates an internal-type-0 system error that an ATTEMPT call may intercept.

1.3.5.3. The UNIX CLOSE Function.

The CLOSE function removes the external file connection to the internal, logical file-number given by the CLOSE parameter, a fixed-point number, for example:

(CLOSE FILE-NUMBER) .

If CLOSE removes such a connection, CLOSE returns NIL. However, if the internal to external file connection does not exist or the parameter is out of range, CLOSE returns the integer error number returned by the UNIX operating system. If CLOSE gets the parameter NIL, CLOSE disconnects the standard input, logical file-number zero (0). As CLOSE removes the last internal connection to a file, the operating system may perform other actions such as rewinding magnetic tape, returning end of file to the receiving end of a pipe, or reclaiming file space that is no longer referenced by any directory.

1.3.5.4. The PIPE Function.

The PIPE function, a function of no arguments,

(PIPE)

calls the UNIX operating system to obtain a pair of PIPE file descriptors. PIPE returns the pair as a CONSed node of two integers: the read and write internal, logical, PIPE-file-numbers. The current invocation of VLISP and any subsequent offspring created by the FORK request may share the PIPE-file-numbers for inter-process communication. A process receives output in the order sent by any other process on any one PIPE. Processes not intending to use one side of the PIPE or pass further copies of that PIPE descriptor to offspring should CLOSE the unused side of the PIPE descriptor so that 1) receiving processes may detect an end of file when all other processes have finished sending data, and so that 2) sending processes may be stopped when no other process intends to read the data sent via the PIPE. If a PIPE call is unsuccessful, the LISP interpreter

generates an internal, type-0 error.

1.3.5.5. The FORK Predicate.

The FORK predicate, a function of no arguments,

```
(FORK)
```

creates a second process that is a copy of the original process. Each process maintains a distinct copy of the data area and any user code in the writable I-space. The two processes determine their identity by examining the result of the FORK predicate. FORK returns NIL (false) to the child process but returns the Process IDentification (PID) (true) of the child process to the parent process. If UNIX cannot create a second process, FORK generates a LISP internal system error 0, that a current ATTEMPT invocation could catch. Any files that were open before the FORK call, including any PIPE files that inter-process communication could use, are available to both processes. The child process suppresses the prompting message, the value return prefix used by the LISP supervisor, and the restart sign-on message. Thus the parent process may continue sending prompting messages to the user while the child process suppresses prompting messages in order to converse cleanly with the parent through redirected standard input/output files. For example, in order to redirect the standard output to a previously created PIPE on which the parent may receive data, the child process would 1) close the standard output file, 1, by evaluating

```
(CLOSE 1) ;
```

2) duplicate the write descriptor of the dotted-pair descriptor, PIPE-PAIR, by evaluating

```
(SYS 41 (CDR PIPE-PAIR)) ? System DUP call
```

which allocates the lowest available number to the file descriptor; and 3) close the child's unused copies of the file descriptor by evaluating

```
(DO [CLOSE (CDR PIPE-PAIR)] [CLOSE (CAR PIPE-PAIR)])
```

so that logically unused pipes may return end-of-file status. Similarly, the parent would close the PIPE write descriptor. Thereafter, the parent would read the standard output of the child, without any "Eval: " prompt or "Value: " prefix, using the PIPE read-file descriptor. Either process, but usually the child process, may overlay itself using the EXEC function, to perform a different activity as a satellite of the other process. Finally, using the WAIT function, the parent process may suspend its own activity until the completition of the child process.

1.3.5.6. The UNIX VLISP EXEC Function.

UNIX allows a process to overlay itself with a replacement activity, whose initial data anc code any executable, UNIX file may define, e.g.

(EXEC ARG0 ARG1 . . . ARGn) .

The initial parameter, ARG0, of the UNIX VLISP EXEC function gives a complete external file name that UNIX passes as the parameter of the "exec" call tc replace the LISP interpreter activity. EXEC calls the function STRING to convert all of the parameters into strings terminated by a zero (null) byte, constructs an array of pointers to the heaa of each null-terminated string, and passes the array as the second parameter in a constructed, indirect "exec" call to the UNIX operating system. By convention, UNIX expects the initial element of a string-pointer array to specify the overlay file. The other parameters often specify option strings, usually starting with minus (-), and external file names manipulated by the replacement activity. The standard I/O files used by the replacement activity may be redirected before calling the EXEC function. If the EXEC function returns to LISP instead of overlaying LISP, EXEC returns the integer error number that the UNIX operating system returned in CPU register R0, rather than gererating a LISP internal error.

1.3.5.7. The WAIT Function.

The WAIT function

(WAIT)

suspends activity in the current process until any one of its previously createa children terminates. WAIT removes the remnants of a terminated child and returns a dotted pair of two integers. The CAR is the Process IDentification (PID) of the terminated child. The CDR is the status value returned by UNIX in CPU register R1, and is composed of the child's exit-value byte and the child's terminatior status in the high and low order bytes respectively. If a termirated child has not been waited for previously, the call to WAIT will continue immediately without suspending activity. If the calling process has no remaining children, WAIT generates an internal LISP error 0 condition. Since SH, the shell command interpreter, waits for a specific terminatea child, SH silently removes the remnants of any other terminated children, who disappear without further announcement of their demise.

1.3.5.8. The Shell Command Interpreter Function, SH.

The SH function provides convenient access to the UNIX commana language interpreter, the shell. SH may get one optional parameter, e.g.

(SH ARG) .

SH converts the optional parameter, ARG, to a string followed by a null (zero) byte that the shell command interpreter uses with an implicit "-c" option as a single command line. If the optional parameter is omitted, SH calls the command interpreter to receive commands from the current standard input up to an end-of-file. SH expects to find the shell command interpeter named "/bin/sh". While the shell command interpreter processes commands, SH suspends activity, ignoring the standard, delete-key (DEL) interrupt and the QUIT, file-seperator (FS) interrupt (Control-SHIFT-L or Control-Backslash), waiting until the shell command interpreter terminates. While waiting, SH removes any other children who terminate without returning any status about the terminated children. When the shell-command interpreter terminates, SH restores the previous LISP interrupt handling and returns the octal number returned by UNIX in CPU register R1 as the termination status word.

1.4. Differences in Implementation.

Due to machine architecture differences, some features are implemented differently in VLISP than in Wisconsin's UNIVAC 1100 LISP. VLISP calculates slightly different arithmetic values, uses different TERPRI and CLEARBUFF parameters, redefines the compiled code handling functions, and omits some features.

1.4.1. Arithmetic.

Unlike the UNIVAC 1100 series machines that use 36-bit-word, one's-complement arithmetic, DEC PDP-11s use 16-bit-word, two's-complement, fixed-point arithmetic and signed-magnitude, 32-bit-single-precision and 64-bit-double-precision floating-point arithmetic. VLISP provides one-word, 16-bit, octal-and-integer-representation, fixed-point numbers and stores negative, fixed-point integers in two's complement. Integers from -32767 to 32767 may be created by the READ and TOKEN routines. The function MINUS produces the integer two's complement negation of a fixed-point parameter. VLISP defines a new function COMPLEMENT to provide an octal representation of the one's-complement negation of its parameter, i.e. COMPLEMENT reverses each of the 16 bits.

Signed-magnitude, floating-point-arithmetic hardware is optional with PDP-11s. If the host PDP-11 provides floating-point arithmetic, VLISP can support floating-point data types depending on the setting of flags for concitional assembly statements in the interpreter source code. VLISP may support floating-point, signed-magnitude data types that are either 2-word, single-precision; 4-word, double-precision; or both. If VLISP supports any floating-point type, VLISP also supports mixed-mode arithmetic between any floating-point-type or fixed-point-type number. The stancard multi-parameter, arithmetic functions

PLUS, DIFFERENCE, TIMES, QUOTIENT, and REMAINDER

and the standard comparison functions

EQUAL, LESSP, and GREATERP

convert an operand with lesser precision to the type of the operand with greater precision before computing each intermediate result. If VLISP supports floating-point numbers, then the final result of arithmetic functions, including the single-argument functions

ADD1, SUB1, and MINUS

have the same type as the parameter whose precision is greatest. If VLISP does not support floating-point numbers, then the arithmetic functions use all parameters as fixed-point integers and return an integer result. The TIMES function converts any fixed-point-multiplication, intermediate result which overflows into a numeric type with the highest floating-point precision available in order to avoid losing information. If VLISP supports any floating-point type, VLISP defines additional floating-point conversion functions and predicates:

ENTIER, FIXP, FLOAT, and FLOATP.

The function FLOAT, which Wisconsin's UNIVAC 1100 LISP does not pre-define, converts any fixed-point parameter into a lowest-available-precision, floating-point result and returns floating parameters unchanged. If VLISP supports both the single and double floating-point type, VLISP defines two additional conversion functions

SINGLE and DOUBLE

that convert parameters to the appropriate floating-point precision.

The bitwise logical functions

COMPLEMENT, LEFTSHIFT, LOGAND, LOGOR, and LOGXOR

treat any parameter as a fixed-point number and return octal-representation, 16-bit results. The bitwise logical functions of VLISP, like their Wisconsin UNIVAC 1100 LISP counterparts, treat floating-point parameters as 16-bit quantities without conversion using the high-order, most significant word.

1.4.2. CLEARBUFF and TERPRI Parameters.

The LISP I/O functions CLEARBUFF and TERPRI can take an optional parameter which can be a fixed-point number or NIL. The parameter specifies a new temporary input or output device, respectively. NIL may be used to return to the standard file. If CLEARBUFF or TERPRI get no parameter, the appropriate buffer is handled without changing the current I/O file, unlike what is done in Wisconsin's UNIVAC 1100 LISP. System messages are always sent to a standard file. Also, after a system message, input is expected from the standard file. TERPRI and CLEARBUFF save their most recent parameter as the constant binding on the variables, *TERPRI and *CLEARBUFF, respectively.

1.4.3. System Commands.

VLISP does not implement the Wisconsin's UNIVAC 1100 LISP system commands that begin with a colon (:) in column 1. These include

:BACK :EXEC :LISP :OOPS :PEEK :STOP and :TIME .

1.4.4. Utility Functions Not Implemented.

Other utility functions included in Wisconsin's UNIVAC 1100 LISP are as yet unimplemented. These are:

BACKTR CONCAT DATE DTIME GCTIME GROW MEMORY and *PACK.

1.4.5. Compiler Functions.

Functions used with the LISP compiler to manipulate generated code, namely

*BEGIN, *DEPOSIT, *EMIT, *EPT, *EXAM, *ORG, DUMP, and LOAD

are not defined in the same way as in Wisconsin's UNIVAC 1100 LISP. Since most of the compiler functions are machine dependent, and would have little utility for programs other than the compiler, the differences have little effect on the transportability of code, except that DUMP and LOAD have different purposes. Instead of using DUMP to output compiled code as is done in Wisconsin's UNIVAC 1100 LISP, the Pretty Printer should be used as described below. The LOAD function could then restore the code into VLISP by reading S-expressions intermixed with binary code modules instead of restoring an absolute loader format file as is done in Wisconsin's UNIVAC 1100 LISP. If VLISP does not support compiled code, as is the case with VLISP on a PDP-11/40, VLISP does not pre-define the functions

*BEGIN, *DEPOSIT, *EMIT, and *ORG

but instead defines the functiors

*EXAM and DUMP

so that they return NIL when called, and defines the functions

*EPT and LOAD

with a reduced capability. The setting of an assembly-time flag, CPLCPL, in the module "TRAPS.MAC" determines if VLISP will support compiled code.

1.5. LISP Systems Software.

Systems programs, written in LISP, are available to help the programmer. They are kept on file in a form that can be brought into core by evaluating the LISP S-expression

(LOAD FILE)

where the SYMBOLic atom, FILE, evaluates to the logical file-number of the program. Under UNIX, the FILE parameter may also specify the name of a file that UNIX VLISP opens, repetitively reads and evaluates, and closes. The system software includes a Pretty Printer, an S-expression editor, a debug package, micro-PLANNER, and a compiler.

1.5.1. Pretty Printer.

The Pretty Printer, PRETTYP, displays non-circular LISP objects in an orderly, indented format that can be read as input to recreate the objects. The function, PRETTYP, takes from one to three parameters, e.g.

(PRETTYP DUMP-LIST ASCII-FILE BINARY-FILE) .

The first parameter, DUMP-LIST, evaluates to a list of 1) atomic symbols with constant bindings to be displayed; or 2) sublists, the CAR of which is a property or flag of the subsequent atoms in the sublist to be displayed. If the second parameter, ASCII-FILE, is given, it specifies that output will be sent to a logical file-number instead of to the keyboard. If PRETTYP gets the second parameter, ASCII-FILE, ar internal, fixed-point, logical file-number, PRETTYP sends the S-expression output to the specified file instead of the current file. PRETTYP sends binary output of compiled code to the internal, fixed-point, logical file-number given by the last parameter, BINARY-FILE, provided that the parameter is non-NIL. If the last parameter is NIL, or if PRETTYP gets only one parameter, PRETTYP produces no binary output. The second parameter may also be the last so that ASCII-character output of S-expression representations and binary output of compiled code will be appropriately interleaved in the same file. The ASCII and binary logical file-numbers should have

previously been given an external association by a call to OPEN or a similar function such as PIPE under UNIX. PRETTYP returns a list of the atoms in the first parameter, DUMP-LIST, which had no constant binding, and sublists with two elements giving a name and atom whose property list did not contain either the property or a flag with the name mentioned in a sublist of the first parameter, DUMP-LIST. When Pretty Printing compiled code, the expression bound to the master LINKER, the function entry to the start of the compiled code area, should be output first so that the expression may later be restored. Usually the safest way to output expressions which have been compiled is to output them all with a single call to PRETTYP, passing as the first parameter a list of atoms bound to the compiled functions in the same order as the functions were compiled. After Pretty-Printing, the files could be re-read to re-establish the indicated bindings by evaluating the S-expression

(LOAD ASCII-FILE BINARY-FILE)

where ASCII-FILE and BINARY-FILE are internal, fixed-point, logical file-numbers, previously associated with an external file-name of files containing the S-expression representations and compiled code images, respectively. If LOAD gets only one parameter, it may input a file of interleaved ASCII and binary information. LOAD repetitively reads S-expressions until reaching an end of file. Under UNIX VLISP, the first parameter (but not the second) of LOAD may specify an external file which the interpreter will open, read, and close.

1.5.2. The LISP Expression Editor.

The LISP editor special form, EDIT, and function, EDIT1, allow the programmer to easily alter in-core expressions and function definitions. Once the editor is invoked, for example, by evaluating

(EDIT FUNC)

the following simple commands can:

```
M #       - move the focus horizontally without descending;
+#        - move the focus horizontally in list and descend;
-#        - ascend # times in a list structure;
P         - print the currert focus;
PP        - Pretty Print the focus (if PRETTYP is loaded);
E EXP     - evaluate the expression EXP;
I EXP     - insert the value of EXP before the focus;
D         - delete the current focus and ascend one level;
R EXP     - replace the current focus with the value of EXP;
S ATM     - save current focus as fluid binding of ATM;
C OLD NEW - replace all occurrences of OLD with NEW;
RESTORE   - start over from the top; or
STORE     - install the edited object and return.
```

Note that # represents any integer, its sign giving the direction of travel. EXP represents any LISP S-expression. ATM represents an atomic symbol, a variable.

1.5.3. The Debug Package.

The debug package provides four routines utilizing the system functions, BREAK and UNBREAK, whose first parameter is a list of variables or atoms with constantly bound functions, macros, or special forms.

$TRACE traces the call and exit of constantly bound functions, macros, and special forms and gives parameter and exit values.

$BREAK is similar to $TRACE but stops, querying the user for expressions to evaluate until reading the expression T.

$TRACEV prints the new values of variables as CSET, CSETQ, SET, and SETQ alter them. Tracing is ineffective for variables altered by compiled code.

$UNBUG removes tracing from the atoms in its parameter list or, if no parameter list is provided, $UNBUG removes all tracing.

If possible, use a compiled version of the debug package to avoid internal conflicts between traced variables and functions. If compiled code is not available, evaluate

(SMANIFEST DB-LIST) ;

after loading the debug package and before initiating tracing to remove some of the conflicts involved in tracing functions using functions which might be traced.

1.5.4. Micro-PLANNER.

A version of Micro-PLANNER can be used on PDP-11/45s on a small data base. A 32K word USER data area is required. After loading, typing

(PLNR)

starts Micro-PLANNER. Micro-PLANNER will then prompt for PLANNER expressions to evaluate by printing

THVAL: .

If the Pretty Printer is also loaded, the PLANNER data base may be dumped to a file by typing

(THDUMP FILE)

where FILE evaluates to a logical file number. Later the data base may be restored while using Micro-PLANNER by typing

$&(LOAD FILE) .

1.5.5. LISP Function Compiler.

The LISP function compiler, COMPILE, transforms LAMBDA expressions into machine code placed in the user-code, writable-I-space area. COMPILE gets from one to three parameters, e.g.

(COMPILE COMP-LIST DUMP-FLAG MASTER) .

The first parameter, COMP-LIST, is a list of 1) SYMBOLic atoms that are constantly bound to furctions, special forms, or macros; or of 2) sublists whose CAR is a property name, a SYMBOLic atom, found on the property lists of the sublist's remaining variables whose property values are user defined functions. The indicated functions, special forms, and macros have underlying S-expression definitions that COMPILE transforms into machine code added to the writable-I-space area. If COMPILE gets the optional, non-NIL, second parameter, DUMP-FLAG, COMPILE displays the intermediate tuples produced from the S-expressions which are converted into code. Any third parameter, MASTER, which is a master LINKER for the current area of code being generated, COMPILE uses in place of a new master LINKER that would ordinarily begin a fresh code area so that the code to be produced will continue the current code area. COMPILE prints a warning message about any constant bindings or properties which cannot be compiled.

COMPILE returns a list of any variables used freely without being declared fluid or having constant bindings. Variables are free when they are used by a functior without being declared as an accessable dummy argument of the function or a surrounding PROG special form. The function FLUID, included with the compiler, takes as a parameter a list of variables that FLUID will mark as fluid. The function UNFLUID, also included with the compiler, takes as a parameter, a list of variables from which UNFLUID will remove the fluid marking. EXCISE, a function of no arguments, removes all S-expression connections with the compiler that the garbage collector might reference, thus returning some S-expression space to general use.

2. Internal Configuration.

The PDP-11 LISP interpreter is modelled after the Wisconsin LISP UNIVAC 1110 interpreter. Each interpreter is written in assembly language to facilitate its optimization with respect to the architecture of its host. Both interpreters have been modularly organized to aid in their design and improvement. Standardized interfaces and data structures are used between most interpreter routines. Documentation is provided within the code listings for deviations from the standard interfaces. Both interpreters assume an operating system has been provided to handle system overhead chores. Moreover, the physical layout, the algorithms, and even many assembly labels used are, in general, similar. Understanding the workings of either interpreter should aid in the understanding of the other.

2.1. VOS Operating System calls.

Under VOS, the VLISP interpreter uses the "TRAP" instruction (1044XX) to perform input/output, to recover from errors and interrupts, and to do other miscellaneous system functions. CPU registers pass the parameters. In order to be compatible with the PDP-11/40, the operating system uses only one CPU register set and makes no attempt to charge register sets in PDP-11/45s from the starting register set. The operating system returns unaltered the contents of CPU registers not used for sending or receiving parameters. Any operating system that supports the following "TRAP" definitions and provides sufficient address space can house the PDP-11 LISP interpreter. The stand-alone operating system and the VOS emulator under DOS take advantage of the uniformity of the VOS interface. The label for each "TRAP" instruction offset below prececes its octal representation which is in parentheses. The value and action correspond to the "TRAP"s of the VOS control machine.

2.1.1. TRPTRP (0) - Simulate TRAP.

Offset TRPTRP (0) simulates any other TRAP. The low order byte of CPU register R5 passes the TRAP offset. The other CPU registers pass parameters in the normal manner according to the simulated TRAP.

2.1.2. READ (1) - Start Input of Line.

Offset READ (1) conditiors the input routines so that the next character will be transferred from the beginning of the next input line. Any unread characters from the previous line are lost. The end of line flag from the previous line is cleared. Register R0, which contains flags used by VOS, is cleared by the LISP interpreter before the call. Register R1 is used to specify a logical device, process, or port from which the next line will be obtained. If zero is used in R1, the default device assignment is used for input.

2.1.3. RDASC (2) - Read ASCII Character.

Offset RDASC (2) returns the next input character from the current input line in CPU register R0. Register R1 contains a non-zero flag. When all characters from the current line have already been read, zero is returned in register R1. The next line is not started until a "TRAP READ" is performed.

In the current system, register R2 contains a count of characters already received. The stand-alone operating system decrements the value returned ir CPU register R2 to backspace. Returning zero in CPU register R2 deletes the input line. More general operating systems need not attempt this kind of shortcut.

2.1.4. WRITE (5) - Send With No Carriage Controls.

The WRITE TRAP provides compatiblity with VOS. In the stand-alone systems, it performs no action. Under VOS, the WRITE TRAP signals the end of the current line of output characters, insuring message completion to receiving processes. CPU register R0, in which VOS passes flags, is cleared to zero before the "TRAP" by the LISP interpreter.

2.1.5. CRLF (6) - Send Line With Carriage Controls.

A CARRIAGE RETURN and LINE FEED are added to the current line of output. Then the "TRAP" performs the actions of "TRAP WRITE".

2.1.6. PRASC (7) - Send ASCII Character.

The character in register R0 is added to the current line for output. VOS uses the seven low order bits of register R0 and computes an even parity bit.

2.1.7. SYSPRT (020) - Change System Ports.

The logical port number specified by register R1 is used to temporarily change the standard I/O streams. If the upper byte of R1 is nonzero, the logical port specified is used for input. Otherwise the logical port specified by the lower byte is used for output.

2.1.8. SETRAP (024) - Prepare to Process Contingencies.

Register R0 contains the address at which the LISP interpreter wants to start processing contingencies. Attention interrupts, stack overflows, illegal instructions and I/O errors would all begin processing at the specified point.

2.1.9. ERINFO (032) - Get Status After Contingencies.

After a contigency, "TRAP ERINFO" obtains information about the contingency necessary for a restart. Thus uninterruptible operations can be resumed before an attention interrupt is processed. The LISP interpreter must ensure that an uninterruptible process did not cause the interrupt.

Upon return, CPU register R0 contains the virtual program counter (PC) location, register R1 contains the virtual processor status word (PS), and register R2 contains the error type in the low order byte. Attention interrupts return a negative error code in this byte while other types return positive codes.

2.2. Function Call Conventions.

The LISP interpreter code section consists of a collection of mostly independent subroutines. External routines, which the interpreted data may call directly, all have a common calling and exit convention. Thus individual routines may be added or modified without fear of affecting other sections of code. Internal subroutines, such as the garbage collector, which have different conventions, are documented within the LISP interpreter code listing. However, almost all subroutines follow the convention that the return address is on top of the control stack, which grows downward, pointed to by CPU register SP, R6.

2.2.1. On Entry.

On entry, external routires expect CPU registers R4, R5, and SP to be pointers. As noted above, SP, the hardware stack pointer, points to the control stack, which grows downward to lower unsigned addresses. On top of this inverted stack is a return address which may be accessed by the instruction

RTS PC .

CPU register R4 points to the top of the value stack, which grows upward. Register R4 points to the next free word on this stack. CPU register R5 points to the first parameter's location on the value stack. If the routine was called with no parameters, then R4 and R5 contain the same values. Otherwise, successive parameters occupy successively higher words on the control stack starting at R5's value and ending just below register R4's value. Data items in LISP contain pointers, which may be followed during garbage collection. The items on the value stack are also

pointers and hence the garbage collector marks the items referenced by the value stack to keep them from being reclaimed. All parameters passed to functions must have such protection and thus are placed on the value stack. Other addresses, such as return addresses, pointers into the stacks, or raw values (as opposed to the pointers to values) are stored on the control stack during evaluation. The items on the control stack are not referenced during garbage collections.

2.2.2. How to Call External Functions.

Two internal procedures facilitate subroutine entry and return. Before using the routines, any temporary data item pointers that may need protection from garbage collection are pushed onto the value stack. Next, the current value in register R4, the value stack top, is pushed onto the control stack, pointed to by register R6.

External subroutine calls use the internal subroutine ENTRY, externally named XNTRY. To use ENTRY a special LINKER node pointer is pushed onto the value stack. The LINKER node consists of a subroutine entry address and a pointer to a data item, such as a LAMBDA expression which is to be interpreted. The data item will be marked by the garbage collector to avoid reclamation. The subroutine entry address is not marked by the garbage collector. LINKER node usage permits one numerical address to have two simultaneous meanings, which the PDP-11/45 memory segmentation hardware permits. ENTRY must also be used for some internal subroutine calls which expect a LINKER node to be placed on the value stack. After the LINKER node, the parameters are pushed onto the value stack befor ENTRY is called.

A simplified entry procedure named ENTRY0, externally named XNTRY0, is used for calling acceptable subroutines. The parameters are simply pushed onto the value stack without any LINKER node. The address of the called subroutine is then put in CPU register R5 just before calling ENTRY0.

Both entry subroutines are then called using the jump subroutine instruction, JSR, using CPU register R5, i.e.

```
JSR %5,ENTRY0
```

or

```
JSR %5,ENTRY.
```

Both entry subroutines call the specified function in the conventional way. On exit, the stack pointers R4, R5, and R6 are restored to their values before the parameters and LINKER node were pushed onto the stack. The other registers may be used by the called procedure without having to save their values. CPU register R0 returns the pointer to the returned data item, the value of the called function. The calling routine must save any re-

gister values on the appropriate stack before beginning function calls.

2.2.3. Internal Subroutines.

The jump-subroutine instruction using CPU register R7, the program counter (PC)

```
JSR PC,SUBRTN ; Call subroutine
```

calls most internal subroutines. Parameters are transmitted in a manner peculiar to each subroutine. In general, CPU register R0 returns values.

2.2.3.1. Printing Subroutines.

Most of the printing subroutines expect just one parameter on the value stack. This parameter is popped from the value stack on return into R0. The value in R5 is unaffected.

2.2.3.2. Obtaining Data Nodes.

The procedure NODE, externally named NNODE, provides data nodes. CPU register R3 contains the type of data node required. CPU registers R0 and R1 or floating-point accumulator AC0, if needed, contain the value to be used in node construction. Additional entry points load CPU register R3 before entering the NODE routine. CPU register R0 returns a pointer to the node created. NODE saves only CPU registers R4 and R5. Calling NODE may cause a garbage collection.

2.2.3.3. Obtaining Node Types.

Small, externally available subroutines return the type of a given node in CPU register R3. Routines GETYPE, GETYP2, GETYP1, and GETYP0, externally named GTYPE, GTYPE2, GTYPE1, and GTYPE0, are used to obtain the types of nodes in R3, R2, R1, and R0, respectively. Only register R3 may be altered. Other subroutines that use node types assume the node type is in register R3.

2.2.3.4. Catching Error and Non-standard Returns.

Several procedures such as LISP, PROG, and ATTEMPT place restart points on the value and control stacks. These restart points provide stack reset positions after a non-standard return, the ERROR and GO procedures, and internal errors. The function UNWIND, externally named UNWND, finds the appropriate restarting point on the stacks. When UNWIND is called, CPU register R1 contains the return index and CPU register R0 contains an appropriate value, such as a GO label or RETURN value. After finding a match to the return index, the original procedure restarts immediately after the point where it established the restart point. The association list existing when the restart point was created

is also reestablished.

2.2.3.5. Internal List Manipulation.

Internal subroutines for manipulating the current association list, property list flags, and attribute-value pairs pass parameters and return values through registers R0 to R3.

2.3. Register Usage.

Although most registers have no fixed usages, register usage follows some general patterns. Registers R0 to R3 are used without being saved by subroutines, while registers R4 to R6 are normally restored after subroutine calls. The conventions follow:

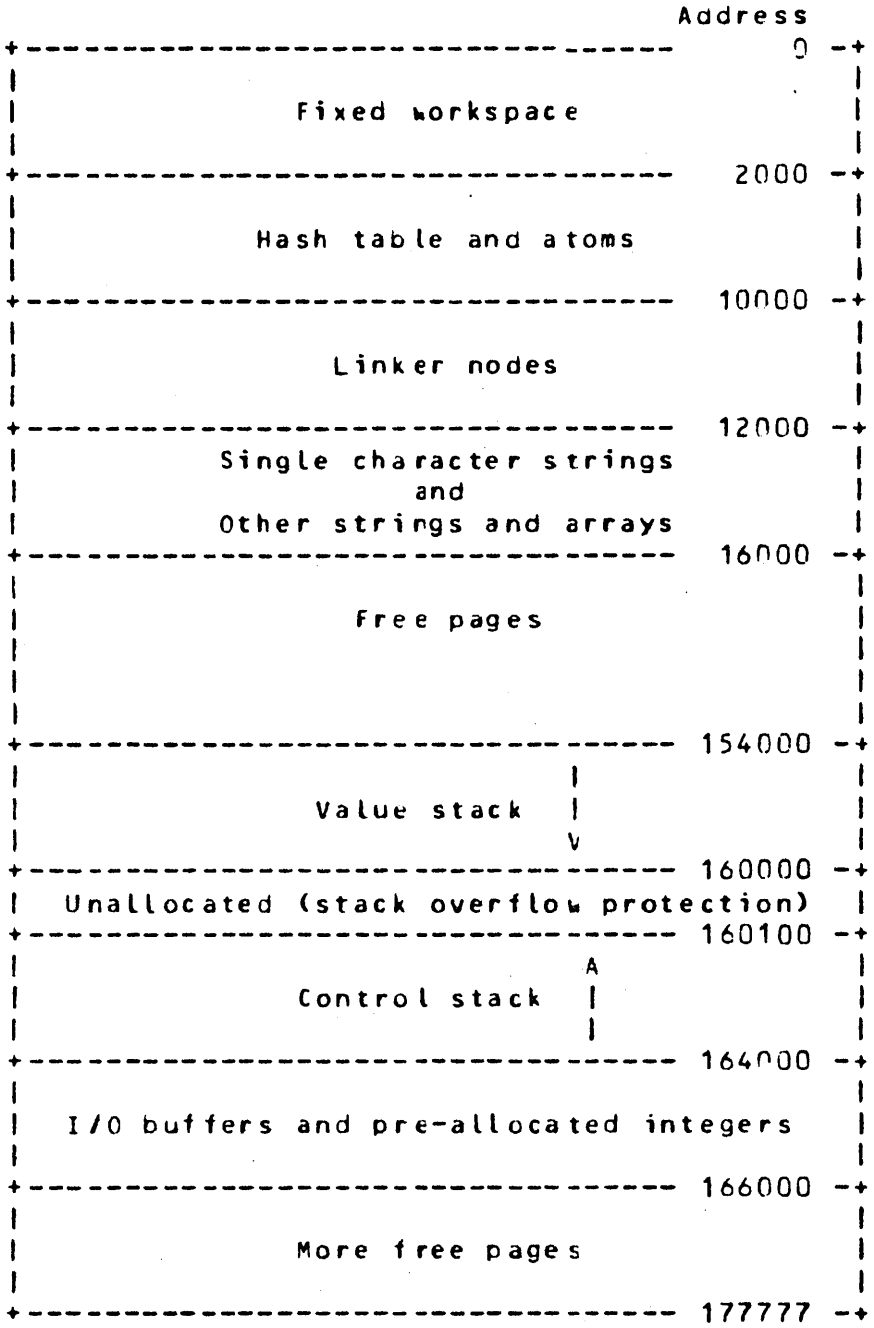
R0 = %0 is used to calculate and return values.
R1 = %1 is general purpose.
R2 = %2 is used as a loop counter.
R3 = %3 contains the type of a data item.
R4 = %4 points just beyond the top of the value stack.
R5 = %5 points to parameters within the value stack.
R6 = SP = %6 defines the hardware control stack top.
R7 = PC = %7 is the instruction counter.

2.4. Storage Allocation.

The user mode D-space area of storage is divided into equal size contiguous areas called pages. Data within each page has a uniform type. A page table records the current type within each page. Given a pointer to a data item, the page table is used to determine the type from the address. The pages are aligned on page boundary addresses that are multiples of the page size. Thus the high order bits of any pointer can be used as an index into the page table to determine the type. The numerical byte code for each type is included in parentheses in the description that follows. All of the types are even numbers to facilitate multiple branch instructions, e.g.

```
ADD %3,PC ; Branch according to type.
```

Figure 3 - Initial LISP data area layout.



2.4.1. SYSTEM and Stacks (-6).

The value and control stacks, I/O buffers, tables, error message string, and permanent addresses are located in SYSTEM pages. The control and value stacks expand to the lowermost and part of the top hardware segment. At least one block is left unallocated so that if either stack overflows, a hardware interrupt occurs.

2.4.2. Not Available (NA) (-4).

This type is reserved for pages used for non-standard purposes. This might include workspaces for other procedures that could be linked to the LISP interpreter code. Also pages that are not included in the hardware mapping are given type NA during the start-up procedure.

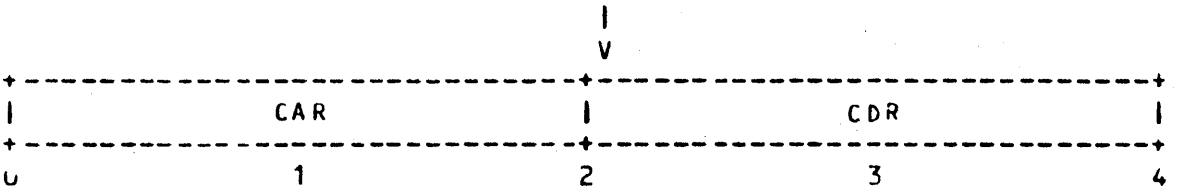
2.4.3. FREE (-2).

Pages that are available for conversion to other types when needed have type FREE. When the garbage collector reclaims an entire page it is given type FREE. Type FREE pages remain uninitialized until needed. A count is maintained of the number of FREE type pages. When an available FREE page is needed, the storage allocator searches the page table to compute the starting address of a FREE page.

2.4.4. CONSED Nodes (0).

CONSED nodes, the list connectives, are four bytes (two words) long and are aligned on two word boundaries. The low and high order words are pointers to the CAR and CDR of the node respectively. Pointers to CONSED nodes point to the high order (CDR) word. Taking advantage of the hardware decrement before addressing, both CAR and CDR can be reached directly without using the longer index addressing mode. During garbage collections, if a CONSED node is marked, its CAR and CDR are also marked.

Figure 4 - CONSED node.



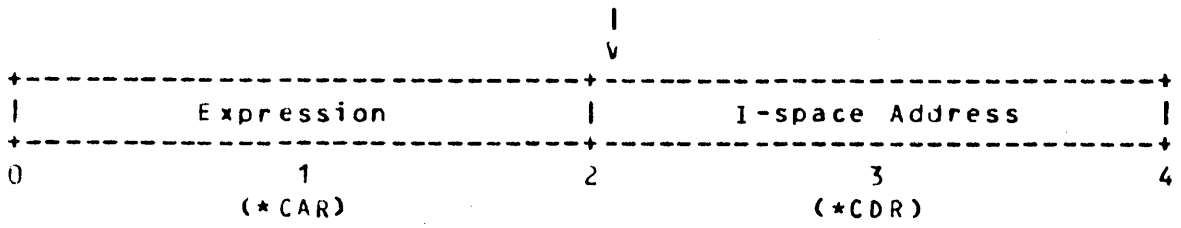
2.4.5. LINKER Nodes (2).

Two-word (four-byte) LINKER nodes are used to access functions and transmit auxiliary information. The high order word (*CDR) of a LINKER node is the starting address of a function whose code is in the memory-management-hardware instruction (I) space. This address should not be used as a pointer since the address specified may have a different meaning in the data space. The low order word (*CAR) of a LINKER node for a system-defined function points to a string giving the original name of the function. The *CAR of a LINKER node of a LAMBDA expression points to a list of the parameters given to LAMBDA to create the function. The *CAR of a node created by LAMBDA and FUNCTION points to a CONSED node whose CAR is the captured association list and whose CDR is a list of the parameters of the creating

LAMBDA call. The *CDR address of such a LINKER node specifies an address where the captured association list is established as the current one and the dummy arguments are given values from the value stack. On entry to this function, the *CAR of the LINKER node is placed on the value stack just below the first parameter. The *CAR of the LINKER node of an array points to the string containing the values of the array. The address given by the *CDR of such a LINKER node specifies whether the string contains pointers whose values must be marked during garbage collection. The *CAR of the LINKER node for the function ALIST points to the head of the association list.

The *CDR, I-space address, of a LINKER node determines if the associated routine involves a function of a special form. All functions have an unsigned I-space address greater than or equal to the I-space address of the system interpreter function, EVAL. Other I-space addresses specify special form routines. Both types of LINKER nodes are aligned on two-word (four-byte) boundaries. Function parameters are evaluated before being passed to the procedure. Special form and macro parameters are not evaluated before being passed.

Figure 5 - Function and special form LINKERs.

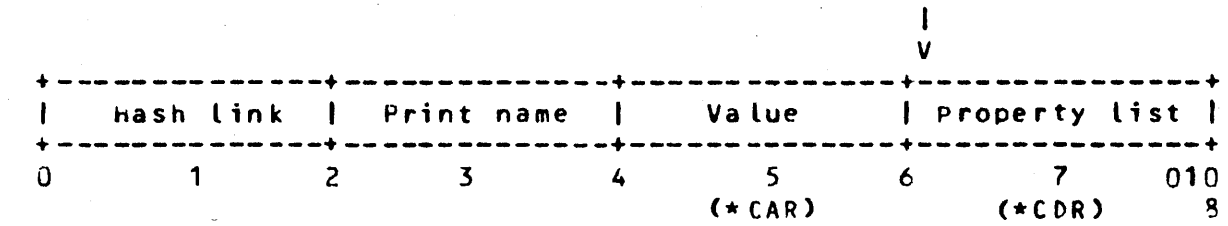


2.4.6. SYMBOL Nodes (4).

SYMBOL atoms, the named entities of LISP, are four words (eight bytes) long. The first, low order word is a hash link. The single character atoms embody the hash table bucket heads. The data initialization creates the single character atoms at the lowest unsigned addresses of the first SYMBOL atom page. The hash code is computed by adding the ASCII character bytes in the symbol name, truncating to the low order seven bits and multiplying by 8, i.e. algebraic shift left by 3, in order to find a bucket head in the hash table. The last hash link in a bucket is marked by a zero word. GENSYM atoms, which are not on the hash chains, have hash links that point to an integer index. The second word points to the ASCII string that gives the name of the atom. The third word, the *CAR of an atom, gives the constant binding of the atom. If the atom is not constantly bound, the third word is zero. If the third word is zero, a fluid binding of an atom may be placed on or retrieved from the association list. Each fluid binding on the association list is an atom and attribute pair. The fourth, high order word of a SYMBOL atom is the property list. A property list consists of flags, which are

other SYMBOL atoms, and attribute-value pairs in which the CAR of the pair is a SYMBOL atom and the CDR of the pair is the value or property. Pointers to the atom address the fourth word, the property list, which serves as the *CDR of the atom.

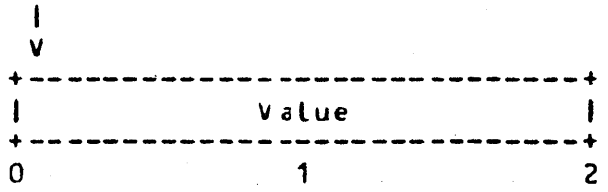
Figure 6 - SYMBOL atom node.



2.4.7. OCTAL (6).

OCTAL nodes are 16-bit words aligned on word boundaries. Although a sign may be specified on input, OCTAL nodes are printed as unsigned octal radix numbers followed by a "Q". Bits within the bytes at the beginning of each page of octal nodes serve as marking flags for garbage collection.

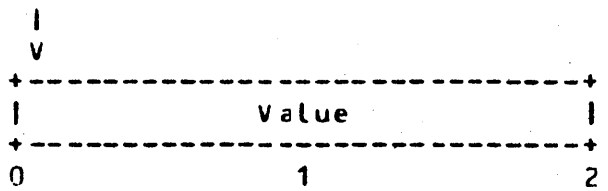
Figure 7 - OCTAL node.



2.4.8. Integer (INTGER) (010).

Integer nodes are signed, 16-bit, fixed-point, two's complement words aligned on word boundaries. Bits within the bytes at the beginning of each page of integer nodes serve as marking flags for garbage collection.

Figure 8 - Integer (INTGER) node.

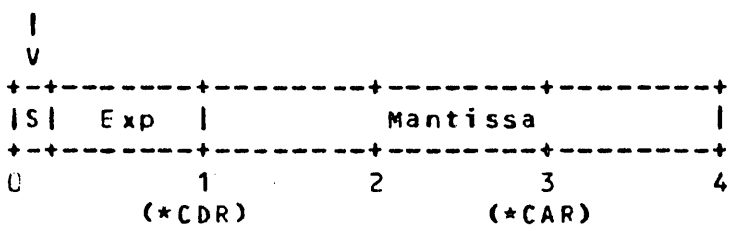


2.4.9. SINGLE precision (012).

Single-precision, sign-magnitude, floating-point nodes occupy 2 words (4 bytes). Bits within the bytes at the beginning

of each page of single-precision nodes serve as marking flags for garbage collection. *CDR anc most functions that use single-precision values as if they were fixed-point values without conversion reference the high-order word that contains the sign bit, 8 bits of biased exponent, and the most-significant bits of the mantissa. *CAR references the other word containing the least significant bits of the mantissa.

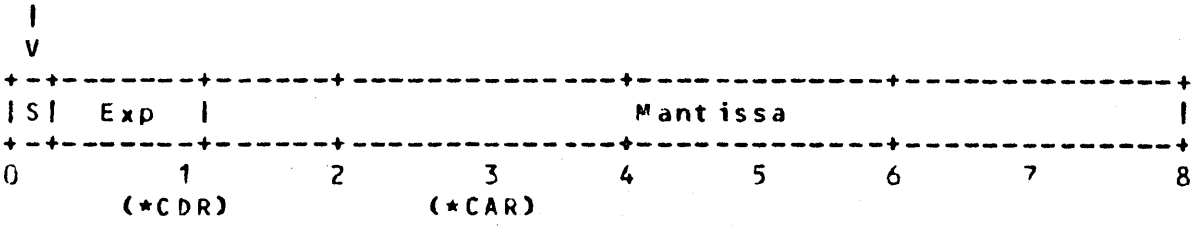
Figure 9 - SINGLE precision node.



2.4.10. DOUBLE precision (012 cr 014).

Double-precision, sign-magnitude, floating-point nodes occupy 4 words (8 bytes). Bits within the bytes at the beginning of each page of double-precisior nodes serve as marking flags for garbage collection. *CDR anc most functions that use double-precision values as fixed-point values without conversion reference the high-order word, which contains the sign bit, 8 bits of biased exponent, and the most significant bits of the mantissa. The *CAR references the second highest significance word containing part of the mantissa. The internal numerical type of double precision nodes is 014 (octal) if VLISP also supports single precision nodes, otherwise 012 (octal).

Figure 10 - DOUBLE precision node.

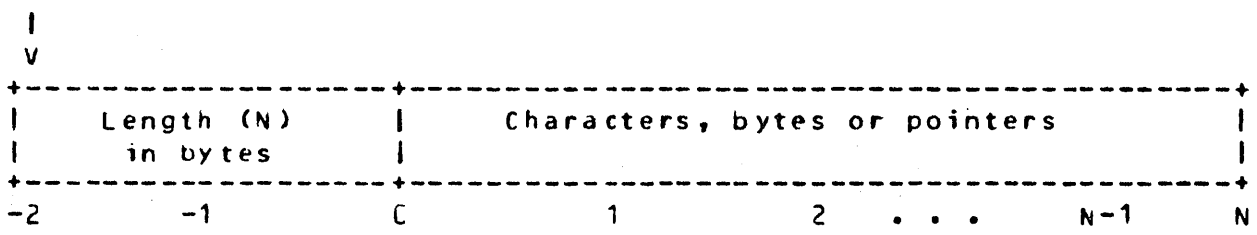


2.4.11. STRING and Array (012, 014, or 016).

Strings and arrays, that both have the same format, occupy the same page type. Pointers tc arrays or strings address a word that gives the length in bytes followed by string or array data. Strings and arrays must be less than 32K bytes long since the

high order bit of the length word is used by the garbage collector to mark strings and arrays. Strings consist of 7-bit ASCII characters in each byte. Context specifies array data, i.e. a special LINKER node's *CAR points to the array. Arrays of pointers, whose values must be marked during garbage collections, must have exactly one LINKER rode whose *CDR address is ARRAYA, the pointer array internal function, so that the garbage collector will mark the members of the array exactly once and will maintain pointer integrity. The starts of strings and arrays align on word boundaries, even when the preceding string length is odd. Strings and arrays may extend across page boundaries. The internal numeric type of strings and arrays is either 012, 014, or 016 (octal) if VLISP supports the 0, 1, or 2 type of floating-precision nodes, respectively.

Figure 11 - STRING or array nodes.



2.5. Garbage Collection.

Storage management and garbage collection differ greatly from those in wisconsin UNIVAC 1100 LISP.

2.5.1. The Deutch-Schorr-Waite Algorithm.

Each new data item created is stored in a node drawn from free storage lists. When a free storage list is exhausted, a FREE page is converted into a page of nodes of the requested type. Finally, when no FREE pages remain, the garbage collector is called to determine which nodes are no longer used to hold current values. These free nodes, which cannot be reached by any chain of pointers accessible to the user, are placed back onto the free storage chains. If an entire page consists of free nodes, the nodes in the page are removed from the free storage chain and their page reverts to type FREE.

The Deutch-Schorr-Waite algorithm underlies the marking method that only marks nodes still in use. Node marking starts from the hash table, the value stack, some unremovable atoms used as flags or LINKERs within the interpreter code, and any current pointers that will be incluced in the data item about to be generated. The Deutch-Schorr-Waite algorithm maintains a stack within the data by reversing the direction of the marked chain of pointers. It requires only a small fixed amount of additional storage for chain head pointers, which are kept in registers. Further, The Deutch-Schorr-Waite algorithm operates in linear

time with respect to the number of marked nodes. No other marking method can significantly improve upon linear time.

After the marking operation is completed, each page is swept for unmarked nodes. For each page, the page type is found to determine the method of marking used and current position in the free storage chain of a given type. The free storage chain for each type is kept in unsigned ascending order. Newly reclaimed nodes are placed in order on the appropriate chain. The free chain, current position pointers may be advanced when marked nodes are encountered. Also, the marking is removed from marked nodes. After sweeping each page, a count of the free nodes within the page is inspected to determine if the entire page is free. The sweeping algorithm operates in linear time with respect to the amount of storage.

Complications arise if the garbage collector is interrupted. While being marked, pointers do not necessarily give the expected value. Some marking is done by setting the low order bits of word pointers. If these pointers were used as word addresses, the odd-address hardware trap would occur. Moreover, some free storage chains may be temporarily disconnected during the sweeping procedure. The unallocated string chain, however, must remain intact to determine whether a given partition (slot) of a string page is either (1) on the unallocated string chain, (2) marked, or (3) allocated but unmarked. The partition's length is found in different positions within the slot accordingly. Similar problems may occur during node allocation. One solution is to disable interrupts during the critical periods. Unfortunately, disabling interrupts involves excessive overhead for such frequent operations as node allocation. Hardware interrupts could not be disabled for the duration of garbage collection during any simultaneous real time operations. Usually, garbage collection lasts beyond one second. Alternatively, a flag is set and cleared when entering and leaving critical areas. When an interrupt is intercepted, this flag is examined. If the flag is set, the operation procedes from the interrupt point to the point where interrupts can occur, the point at which the flag would be cleared. There, the normal processing is discontinued and the interrupt processing is completed. Note that the uninterruptible operations must not generate hardware interrupts themselves, for the system could not continue.

2.5.2. Free Storage Lists.

Each of the free storage chains for each node type is in unsigned ascending order. The end of each chain is indicated by a zero where the next link pointer is expected. Unlike allocated nodes, the chain links of each type always point to the unsigned low order word of the next slot.

For all node types except strings, all nodes are linked onto the free storage chain after their page is given the new type. The chain then consists of a forward linked list.

Within pages of strings and arrays, a chain of free slots is kept. The unsigned low order first word of each slot gives the link to the next slot. If the free slot consists of just two bytes (one word) then the low order bit is set, i.e. the pointer is odd. Following this convention, if the last unallocated slot is just two bytes, it contains the number one. Free slots longer than one word have zero in the low order bit of the first word. The second word of such a slot gives the slot's length. When a slot is added to the free storage chain it is immediately merged with any contiguous slots. The full length of the combined slot will then be available without waste.

2.5.3. Packing Storage.

Storage packing has not yet been implemented. Storage should not be packed after each garbage collection, but only upon request or garbage collection failure. There will probably be a need to implement this complex and time-consuming procedure.

After the LISP interpreter has been running for a long time, all of the pages probably will contain more or less permanently allocated nodes. At the same time, many of the pages probably will be mostly unallocated. Thus, although unused space is available, the garbage collector may eventually fail because it cannot allocate a new page for a type that densely populates its present pages. With fewer free pages available for recycling, the time consuming garbage collector will be called more often.

Packing storage consists of putting nodes of each type in as few pages as possible. For fixed-node-size page types, some pages would be marked to have their nodes placed in other pages of the same type. Pointers to these nodes must also be adjusted. For variable node sizes, the free slots must be removed from between allocated nodes by shifting the allocated nodes, preferably downward, and grouping the free slots into one large free slot at the end of the area. The greatest storage economy is obtained by also ensuring that pages with variable length nodes abut, so that allocated slots may extend across page boundaries. Of course, the pointers to variable length slots must also be adjusted. Nodes in the hash table and LINKER nodes and symbol flag nodes used by the system must not be moved since their positions are referenced by the LISP interpreter code. Moreover, references to moved data nodes must be altered in any compiled code.

2.6. Hindsight.

2.6.1. 32K.

The size of the data space, even using a virtual memory or additional core, is limited to 32K words. This is the largest number of words that can be directly addressed by a 16-bit word without modification. This restriction limits the absolute size of programs that may be interpreted by PDP-11 LISP. Limited additional program space can be obtained by compiling functions into the hardware supported I-space, but absolute limits on program size remain. Future implementations of PDP-11 LISP, working in a virtual environmert, could use 16-bit word pointers that must be modified before use, or 3 or 4 byte pointers to increase the effective address space.

2.6.2. Two Stacks.

Using two stacks, the value stack for pointers and the control stack for addresses and binary values, facilitated programming. However, having two stacks places restrictions on any larger virtual space version. Separate pointers and data areas must be maintained. If the stacks are allowed to overflow onto additional pages of virtual memory, each stack would need to be separately handled. Moreover, if stack sections were to be used as data, as in more advanced versions of LISP, both stacks would have to be manipulated, with double the overhead. Alternatives are to use a method whereby pointers may be distinguished from addresses and raw data on a single stack, or to eliminate the value stack as a separate contiguous area. With the latter alternative, the value stack would be kept among the CONSED nodes, thereby slowing acccesses into the value stack.

2.6.3. The Deutch-Schorr-Waite Algorithm.

The Deutch-Schorr-Waite algorithm, used by the garbage collector, has disadvantages as noted above. The process cannot be interrupted during garbage collection, an intolerable situation for some real time applications. Using other algorithms in virtual space, multiprocessing environments, simultaneous garbage collection can take place while processing continues. Furthermore, restarting after interrupts would be simplified.

3. Machine Code Generation.

User created machine code can be dynamically added to the LISP interpreter within machines whose memory management supports seperated I (instruction) anc D (data) spaces, in particular PDP-11/45s and PDP-11/70s. The operating system, such as VOS, must also provide for the dynamic expansion of the USER-mode I-space in units corresponding to full length hardware segments (020000 octal bytes). Using functions within LISP, pre-assembled routines of machine code can be added to the repertoire of LISP functions in order to perform slowly interpreted or non-standard actions such as system calls more efficiently. LISP LAMBDA expressions may be compiled into machine code in order to speed their execution, avoid unnecessary overhead, and allow the nodes originally occupied by the LAMBDA expression to return to general use, thus increasing the FREE storage space.

The user's machine code may reference S-expressions that are dynamically allocated by the LISP interpreter. Possibly a reference to an expression would be the only reference. To avoid garbage collection of references that are only known to the user's machine code, a table of offsets that point to the references is kept following the user's machine code groups in I-space. The garbage collector consults these tables during its marking phase. All S-expressiors thus referenced are marked as in-use to avoid reclamation. Storage packing routines would know which locations specify addresses to alter within USER-mode I-space when S-expressions are moved in D-space. If the user makes copies of the machine code, the table of offsets following the machine code specify which addresses must be reallocated by a later invocation of the LISP interpreter if the machine code is ever dynamically reloaded. LISP S-expressions written after the code specify how reloaded code must be altered to point to the reallocated S-expressions that the code references.

Reading locations within USER mode I-space by USER-mode programs cannot be done directly. Although the USER-mode instruction MTPI (Move To Previous Instruction space) can write into USER-mode I-space, the hardware design circumvents the USER-mode instruction MFPI (Move From Previous Instruction space) from reading USER-mode I-space by diverting the reference to the D-space. This unfortunate design was intended to support execute-only code, which no widespread operating system currently supports. Instead, the design has forced a system call to be added to operating systems to enable reading locations within USER-mode I-space.

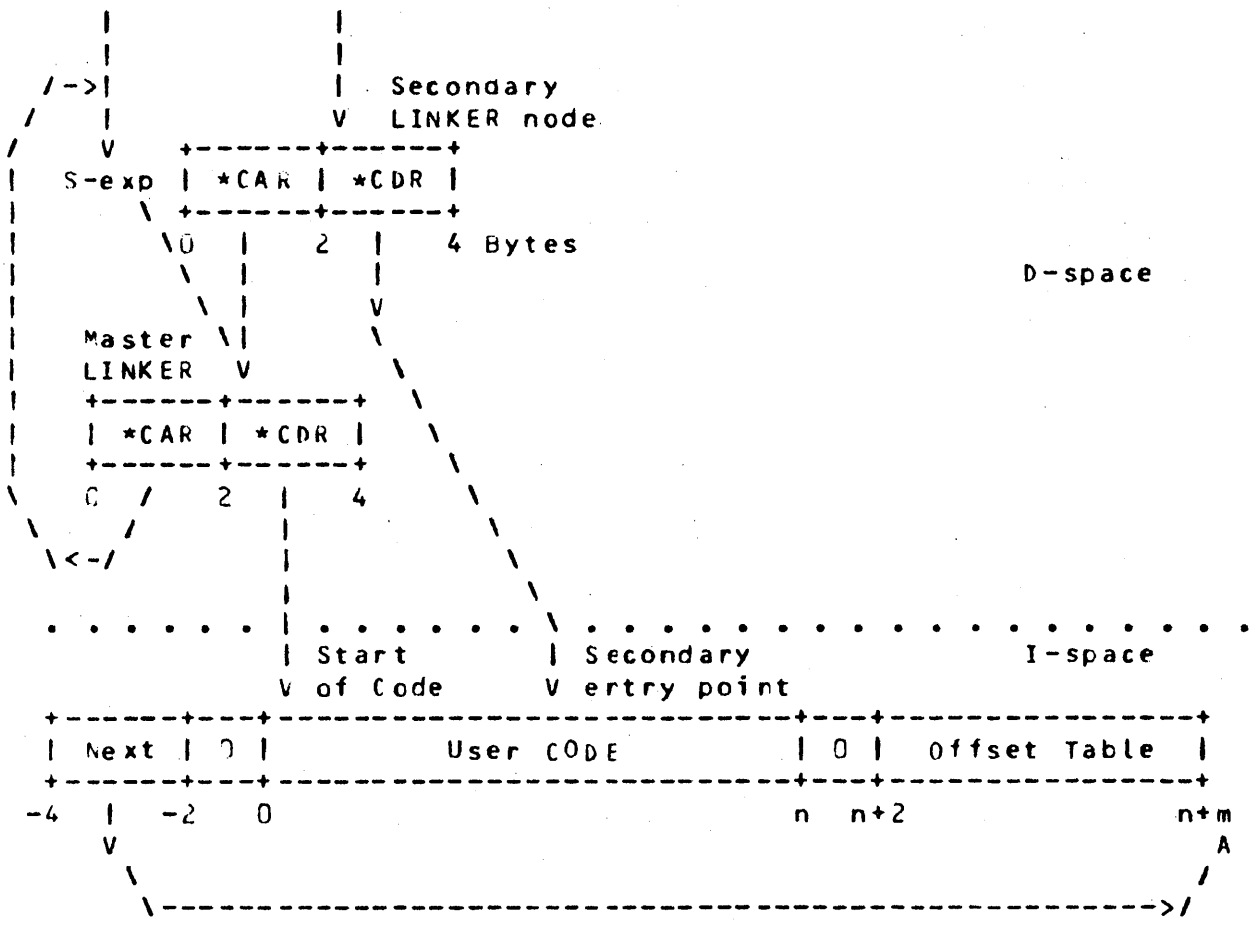
3.1. Manipulating the USER Instruction (I) Space.

Several functions definec in PDP-11 LISP manipulate the USER instruction (I) space that is not occupied by the LISP interpreter code. Although these I-space functions have names that match the names of Wisconsin UNIVAC 1110 LISP functions, their machine-dependent definitions are different. The code for the I-space functions is conditionally assembled with the LISP interpreter when the flag "CPLCPL" in the assembler source module "TRAPS" is set to one. When the code is not assembled the interpreter may occupy less than 4K words (020000 octal bytes), one hardware segment. With the I-space functions included, the LISP interpreter code resides in two hardware segments. This leaves a maximum of 6 hardware segments, 24K words (140000 octal bytes), for allocation to user code areas, depending upon the operating system.

The LISP interpreter manages the USER I-space as a forward linked chain of user code areas. Two words precede each user code area. The first word points just beyond the end of the contiguous user code area to the next area's pointer word. The second, flag word is normally zero. System programs such as the Pretty Printer and the S-expression editor examine the word preceding an address specified by a LINKER node. If the preceding word is zero, the start of a code area has probably been found. Hence, it is unwise to place any other zero word within user code such as a HALT instruction. Each user code area consists of two parts: the instructions and a table of offsets. The table of offsets, described below, has exactly one zero word that is used to mark the unsigned-lowest address within the table. Only the last user code area on the chain may be expanded or loaded.

System conventions should be followed for LINKER nodes that point to addresses within the user code areas. The *CDR address of one master LINKER node should specify the beginning of each user code area, the word preceded by a zero flag word. The *CAR of the master LINKER should point to an S-expression that is a formula that evaluates back to the master LINKER. The *CAR of other LINKER nodes that specify other addresses within the same user code area should point to the master LINKER node of the area. This convention facilitates dumping user code areas that may be loaded at a later invocation of LISP. During garbage collection before storage packing, any marking of secondary entry points to a code area would also lead to marking the master LINKER, which in turn could lead to marking the flag word preceding the code area. Thus any reference to a code area would keep the entire area from being reclaimed.

Figure 12 - Typical Structure of Pointers to User Code Area.



Descriptions of each I-space management function follow. *BEGIN creates a new area to receive code. *EXAM and *EMIT retreive and replace values in the USER-mode writable I-space. *ORG creates secondary entry points to a code area. DUMP, LOAD, and *DEPOSIT output and re-read code and related S-expressions.

3.1.1. *BEGIN New User Code Area.

*BEGIN creates a new area for user code and returns a master LINKER to it. If another I-space hardware segment is needed it is requested and integrated into the USER I-space chain. Any previous user data area under construction is finished by moving the previous table of offsets down to the previous end of instructions. The pointers to the ends of the areas are adjusted. The one parameter to *BEGIN is used as the *CAR of the master LINKER that *BEGIN returns:

(*BEGIN ARG)

3.1.2. *EXAMine a Word in I-space.

*EXAM returns an octal representation of a specified word in USER I-space. *EXAM may have from one to three parameters:

(*EXAM LINK OFFSET TABLE)

The first parameter, usually a LINKER node, possibly the only parameter, gives an address in the I-space. The second parameter, if given, provides a numerical offset from the address given by the first parameter. When the third parameter is given, it specifies an entry within the table of offsets. The third parameter, usually a negative number, is the offset of the table entry in bytes from the high address end of the offset table. The first parameter should be a master LINKER and the second zero in this case. The entry in the table of offsets determines an address among the instructions whose octal representation *EXAM returns. If the specified address lies within the LISP interpreter code, *EXAM returns NIL.

3.1.3. *EMIT a word to I-space.

*EMIT writes a value in a specified location of I-space. *EMIT may have from one to five parameters:

(*EMIT LINK OFFSET TABLE MODIFIER ARG)

If one parameter is given, usually a number, its value is added to the open user code area, the last area on the I-space chain created by *BEGIN. If two or more parameters are given, the last two parameters determine an offset and pointer. The value of the penultimate parameter, usually a pointer to a numeric node, modifies the last parameter, the address of some S-expression. This modified value replaces the value at the specified location. If exactly two parameters are given, the specified location is the next available location of the open user code area. *EMIT also expands the table of offsets by adding the offset to the next code location. Thus the S-expression given by the last parameter will henceforth be protected from garbage-collection reclamation. The table of offsets of the last user code area is kept at the extreme, unsigned-high-address end of the allocated I-space. The instructions and table of offsets in the last area grow toward one another. If *EMIT can not find enough unallocated space to add a new instruction word or offset table entry as requested by one or two parameters to *EMIT, *EMIT trys to expand the last USER I-space area by adding a new contiguous hardware segment to the existing user code area, updating the chain pointers to include the addition, and moving the table of offsets to the extreme high end of the new area. If the attempt to gather more I-space fails, the interpreter will call the operating system in error mode after sending the message

NO SPACE.

*EMIT uses any parameters given before the last two, the offset and pointer, like the parameters of *EXAM to specify a location in I-space. With three or more parameters, *EMIT expands neither the code area nor the offset table, but simply alters an existing word in a user code area. If *EMIT gets three or more parameters, the first parameter specifies an I-space location. With four or more parameters, the second parameter gives an offset from the first parameter. Lastly, with five parameters, the third parameter gives an offset from the high address end of the offset table. The offset table entry in turn specifies a location within user code to replace.

3.1.4. *ORiGinate a Secondary Entry Point.

*ORG creates a secondary LINKER node to a computed location within a user code area:

(*ORG LINK ARG1 . . . ARGn)

*ORG uses its first parameter, which should be the master LINKER node of the code area, as the *CAR of the secondary LINKER created. If no other parameters are given, the *CDR of the created LINKER points to the next location that might recieve code from *EMIT. Otherwise, *ORG uses the value of its second parameter as the *CDR of the created LINKER node. The value of any other parameter would additively modify the address specified by the second parameter.

3.1.5. *DEPOSIT User Code and LOAD S-expressions.

*DEPOSIT inputs S-expressions and code in DEC absolute loader format or UNIX a.out format from specified logical files. Since each operating system has its own conventions for opening and assigning logical names to files and devices, file and device opening and naming must occur before *DEPOSIT operates on a specified file. If LOAD has parameters

(LOAD ASCII-FILE BINARY-FILE)

the first parameter specifies an input file from which S-expressions are read in a READ-EVAL loop until the end of file is reached or a top level RETURN or ERROR function is evaluated. The last parameter specifies a file to be used later for inputting binary code by *DEPOSIT:

(*DEPOSIT ARG)

*DEPOSIT and (if LOAD gets no parameters) LOAD input binary code in DEC absolute loader format or UNIX a.out absolute loader format from the file specified by an invocation of LOAD with parameters as the last parameter. Under UNIX, the .data, .bss, relocation, and symbol table parts of the a.out load module should be empty. *DEPOSIT performs *BEGIN before initiating the input. If not enough space is available for the input code, the

LISP interpreter prints the message

NO SPACE

and calls the operating system in error mode. If the input format is incorrect or a checksum error is found, *DEPOSIT calls the interpreter routines for internal error handling. The input code image should contain irstructions followed by an offset table. *DEPOSIT creates the necessary code chain pointer and zero flag word. When *DEPOSIT finds the end of the code image as indicated by a transfer address record, *DEPOSIT closes the newly created user code area and returns a master LINKER to the start of the code. The *CAR of the created master LINKER is the parameter of *DEPOSIT, conventionally an S-expression that the interpreter can evaluate back to the master LINKER. If LOAD gets no parameters, subsequently evaluated S-expressions should amend the *CAR of the returned master LINKER node to point to a formula that evaluates back to the master LINKER.

3.1.6. DUMP User Code and Referenced S-expressions.

DUMP uses either two or three parameters to output a user code area in DEC absolute loader format or in UNIX a.out absolute loader format and to further invoke routines to handle each S-expression that is referenced by an address known to the table of offsets:

(DUMP MASTER FILE FUNC)

The first parameter must be a master LINKER of a user code area; otherwise, DUMP immediately returns NIL. If the second parameter is not NIL, the parameter is used as the logical name of a file to which an image of the user code area is sent in DEC absolute loader format. DUMP places the start (bottom) of code at location zero in the code image. Under DOS, DUMP produces no DOS communications directory (COMD) but does produce records shorter than 100 (octal) with a few NULL (zero) padding characters between records and longer padding before and after the image. S-expression references are changed to 160011 (octal) in the image to protect against improper reloading. The last record, when using DEC absolute loader format, signals a transfer address of one (000001), that normally indicates the transfer address is not to be used as a start address by a DEC absolute loader. Under UNIX, the a.out loader format produces contains only a .text part in separately-executable mode without relocation bits or a symbol table. Next, if the third DUMP parameter is given, DUMP checks to insure that the parameter is a function. DUMP calls the third parameter function once for each entry in the offset table. Three parameters are passed whose values may be used by *EMIT to recreate the S-expressions when reloaded. First is an octal node which gives the offset of the S-expression address from the start of the code area. Second is ar integer node whose value must be subtracted from the referenced pointer to make it conform to the standards of other pointers of its type. For example, a pointer

to the high order byte of an OCTAL node would need one (+1) subtracted from it to make the pointer word-addressable like normal OCTAL node pointers. Third is the pointer referenced in the standard format used for its type, that is, with any offset removed. Finally, DUMP returns the master LINKER node, the first parameter of DUMP.

3.2. Assembling Code.

Hand encoded assembly routines may be prepared for processing by the available assembler and link editor. A group of LISP S-expressions should also be prepared to command the LISP interpreter to dynamically install the load module output of the link editor and to redirect locations within the code to point to dynamically allocated storage areas. When the LISP system itself is link edited, a symbol table, preferably called LISP.STB, is created so that later global references within user code to fixed locations within the LISP system may be resolved by the link editor.

The following example, prepared for use with DEC's DOS MACRO assembler and LINK link editor, explains how to carry out this prodedure. A similar procedure under UNIX using the UNIX "as" assembler and "ld" link-editor could create a similar load module. Suppose a LISP function UMIN is desired that returns an unsigned minimum of an arbitrary number of integer parameters. If no parameters are supplied, minus one (-1=177777 octal), the largest unsigned two's complement integer, is returned. Such a function, UMIN, would be relatively lengthy and slow if written as a LAMBDA expression since unsigned comparisons are not (currently) directly supported by the LISP interpreter. A fast, machine-encoded UMIN would not need to create binding nodes and could assume the validity of parameters unlike interpreted LAMBDA expressions. A second function, ULESSP, an unsigned-less-than predicate, can also be defined beside the same code. Assume that the code below has been placed in a DOS file labeled "UMIN.PAL".

Figure 13 - Assembler Source for UMIN and ULESSP Example.

```
;
        .GLOBL  UMIN,ULESSP     ;Externalized definitions
        .GLOBL  TRU,NIL         ;External references
;
;       Find unsigned minimum among integer parameters.
;
UMIN:   MOV     (PC)+,R0        ;Load-immediate a pointer to -1
ADDRSS: .WORD   160011          ;Placeholder for pointer to -1
        BR      LABEL           ;Jump to end of loop
LOOP:   CMP     (R0),@-(R4)     ;Check arg from value stack top
        BLOS    LABEL           ;-> This arg is not smaller
        MOV     (R4),R0         ;Current arg is smaller
LABEL:  CMP     R4,R5           ;First arg reached?
        BHI     LOOP            ;No, -> more args to compare
        RTS     PC              ;R0 -> minimum arg
;
;       Unsigned-less-than predicate
;
ULESSP: MOV     #TRU,R0         ;Assume true = T
        CMP     @-(R4),@-(R4)   ;Is 2nd arg > 1st arg?
        BHI     RETURN          ;-> Yes
        MOV     #NIL,R0         ;No, return NIL for false
RETURN: RTS     PC              ;T or NIL is returned in R0
;
;       Table of offsets to dynamically allocated addresses
;
        .WORD   0               ;Marker for beginning of table
        .WORD   ADDRSS-UMIN     ;Offset of pointer from code start
        .END    UMIN            ;Any transfer address is ignored
```

The user code must be position-independent. Program counter (PC) relative references (mode 67) to the data space and to locations within the LISP interpreter code should not be made. In particular, subroutine calls to the LISP interpreter must be made in absolute ("@#") mode (37), rather than in the ubiquitous relative mode found in much assembly programming. However, references of the user code to itself, such as subroutine calls, should be relative. Storage packing routines may move the absolute locations of groups of code. The changes in location will only be reflected in the address portion (*CDR) of the LINKER nodes that reference the user code areas. Hence references from one user code area to another must only be make through LINKER nodes.

A table of offsets to references must be provided at the end of each user code area. The first word of the table, that must be provided even if the rest of the table is empty, is zero. Any entries that follow are offsets from the start of the code area to addresses within instructions of the I-space that reference S-expressions that must be dynamically allocated by the LISP interpreter. In the present example, the word at the label

"ADDRSS:" is identified in the table by the offset, "ADDRSS-UMIN". This MOV instruction operand is assembled as 160011 (octal) so that if the code is used before the proper dynamic storage allocation is complete, a hardware byte error trap will occur. Furthermore, the address 160011 (octal) in D-space has type SYSTEM. Hence, the garbage collector will not attempt to mark the location specified as an in-use S-expression. The word will eventually contain the address of an INTGER node whose contents are a two's complement minus one (-1 = 177777 octal). The D-space references to NIL and TRU are not included in the table since they have permanent locations that are externally defined in the symbol table "LISP.STB". The references to NIL and TRU can be resolved by the link editor before loading the user code.

An available program, "TRANSLATE", provided with UNIX ppLISP, can convert most of the syntax of DEC PAL assembler programs into the syntax of the UNIX "as" assembler. The above source code for the UMIN and LLESSP example could be translated by "TRANSLATE" or originally written with the syntax used by the "as" assembler. Under UNIX, the following shell command interpreter instructions could assemble the code, print an external name list, and resolve external references. In this example, the UNIX assembler source is named "umin.a" in the current directory, the lisp symbol table is named "/lib/lisp.stb", and the output produced is labeled "umin.o" in the current directory. The UNIX link-editor "ld" uses the squash option, "-s", to omit producing any symbol table or relocation bits.

Figure 14 - UNIX commands to assemble and resolve UMIN example.

```
: - Assemble source code
as umin.a
: - Produce name list from object module
: - in 3 columns on the line printer
nm a.out | pr -h "UMIN symbol table" -3 > /dev/lp
: - Link-edit removing symbol table and relocation bits
ld -s a.out /lib/lisp.stb
: - Rename load module
mv a.out umin.o
```

In the example, the user code could be assembled and linked in DOS BATCH mode by the following commands.

Figure 15 - DOS Commands to LINK and Assemble UMIN Example.

```
$RUN MACRO     ; Assemble
#SY:UMIN.OBJ,LP:<SY:UMIN.PAL
$RUN LINK      ; Link Edit with start of code at zero.
#SY:UMIN.LDA,LP:<SY:UMIN.OBJ,LISP.STB[1,1]/B:0/E
```

DOS includes a 16 (20 octal) word communications directory (COMD) as the first record of load modules. The COMD is normally loaded into core and then overwritten. In order to avoid interference with the dynamic code loader (*DEPOSIT) by the COMD, the user code loaded must be at least 16 (20 octal) words long, under DOS, so that the COMD may be completely overwritten. The "BOTTOM" switch, "/B:0", must be specified to the DOS link editor so that the virtual start (bottom) of the user code is at zero. The LISP loader (*DEPOSIT) expects the load module to be in DEC's PDP-11 absolute loader format, that the DOS link editor provides. The format includes a transfer address record at the end of the module. The LISP loader (*DEPOSIT) uses the transfer address record to signal the end of user code. Exactly one address offset table as described above must exist at the end of the combined link edited user code area. If several object modules are included, the offsets in the table must be computed from the beginning of the entire user code area, not from the start of each individual module. If padding words are needed to make the user code area at least 16 (20 octal) words long, the padding must precede the offset table.

In the example, the printed output of the DOS link editor should be consulted to find the offset of the secondary entry point, ULESSP, from the start of code, UMIN. The name list produced from the above example under UNIX by the "nm" processor lists the location of the seconcary entry point. In this case, the offset found, 22Q (octal in LISP notation), is used while preparing a set of LISP S-expressions to create bindings and allocate the S-expressions referenced by the code through the table of offsets. Assume the followirg S-expressions are placed in the DOS file "SY:UMIN.LSP" or the UNIX file "umin.lsp" in the current directory.

Figure 16 - S-expressions to Bind and Allocate UMIN Example.

```
?Comment - Create primary function binding and load code.
(CSETQ UMIN (*DEPOSIT 'UMIN))
?Comment - Load module could be placed here
?Comment - Create secondary function binding
(CSETQ ULESSP (*ORG UMIN UMIN 22Q))
?Comment - Install INTGER node for -1 using 1st offset, -2
(*EMIT UMIN 0Q -2 0 -1)
(PRINT "UMIN and ULESSP loaded")
```

The following commands would then: start LISP under DOS, associate logical file numbers to files, read the code and create pointers to it, and close the files.

Figure 17 - DOS Commands to Load UMIN and ULESSP Example.

```
>RUN LISP ; Invoke the interpreter from system file area
(OPEN "UMIN.LDA" NIL 4) ? Load module file
(OPEN "UMIN.LSP" NIL 5) ? S-expression file
(LOAD 5 4)              ? S-exp file closed as end reached
(CLOSE 4)               ? Close load module file
                          . . .
```

A similar sequence of UNIX commands can also read the code, create pointers to it, and resolve dynamic references.

Figure 18 - UNIX commands to load UMIN and ULESSP example.

```
: - Invoke LISP with prompts
lisp +
<(LAMBDA (FILE-NUM) ? Internal LAMBDA
          (LOAD "umin.lsp" FILE-NUM) ? Open, read-eval, close
          (CLOSE FILE-NUM))          ? Close binary input
 (OPEN "umin.o")>   ? Open secondary, binary input file
```

The assembled user code would then be ready to use.

This relatively short user code sequence could also be generated by the following sequence of S-expressions.

Figure 19 - S-expressions for Directly Generating UMIN Example.

```
(CSETQ UMIN (*BEGIN 'UMIN))
(*EMIT 012700Q)                 ? MOV (PC)+,R0
(*EMIT 0 -1)                    ? Location(-1) <Note 2 args>
(MAPC '(                        ? Generate the rest of UMIN
       000403Q                  ? BR LABEL
       021054Q                  ? LOOP: CMP (R0),@-(R4)
       101401Q                  ? BLOS LABEL
       011400Q                  ? MOV (R4),R0
       020405Q                  ? LABEL: CMP R4,R5
       101373Q                  ? BHI LOOP
       000207Q                  ? RTS PC
) *EMIT)                        ? One arg calls to *EMIT
(CSETQ ULESSP (*ORG UMIN))
(*EMIT (LIST                    ? Generate code for ULESSP
       012700Q                  ? MOV (PC)+,R0
       (RPLACA CQ 'T)           ? Octal value of address of T
       025454Q                  ? CMP @-(R4),@-(R4)
       101002Q                  ? BHI RETURN
       012700Q                  ? MOV (PC)+,R0
       (RPLACA CQ 'NIL)         ? Octal value of address of NIL
       000207Q                  ? RETURN: RTS PC
) *EMIT)                        ? One arg calls to *EMIT
(*BEGIN UMIN)                   ? Close code area
```

For longer sequences of code the direct generation method becomes impractical.

3.3. Compiling LISP S-expressions into Machine Code.

The LISP compiler, COMPILE, may convert LAMBDA expressions into machine code placed in the USER-mode, writable-I-space area of PDP-11/45s and PDP-11/70s. Compilable LAMBDA expressions may be constantly (globally) bound to variables, SYMBOLic atoms; used in a special form or macro defirition; or included as a property value. The LAMBDA function definitions may include macros, special forms, compile-time expressions, and LAMBDA and LAMDA (without a "B") expressions passed as internal functions to functionals. Sometimes, the user must provide additional information concerning variables -- in particular, whether the variables are constants; strictly formal parameters; or fluid variables accessible from the system association list, ALIST, the deep binding environment. The resulting code is faster and requires less D-space, but omits much of the validity checking of interpreted code.

3.3.1. Compiler Invocation.

The compiler must be loaded before invocation. Under DOS, if the file "SY:COMP[1,1]" contains a compiled version of the compiler, evaluating

```
(LOAD (OPEN "COMP"))
```

would define the compiler functions for use. Under UNIX, if the file "/lisp/comp" contains a compiled version of the compiler, ppLISP invoked with the shell command

```
lisp /lisp/comp +
```

loads the compiler and calls the LISP supervisor in prompt mode, or after invoking LISP, ppLISP could evaluate

```
(LOAD "/lisp/comp")
```

to load the compiler. The LAMBDA expression to be compiled should be loaded and tested before compilation. Since compiled code performs little of the syntax and semantics checking done on interpreted code, improperly formed functions may misbehave without warning after compilation. The compiler, COMPILE, gets from one to three parameters:

```
(COMPILE COMP-LIST DUMP-FLAG MASTER)
```

The first parameter, COMP-LIST, is a list of variables and sublists of variables. Each variable, SYMBOLic atom, in the list must either constantly (globally) bind a LAMBDA expression created by calling the special form LAMBDA, a special form created by DEFSPEC from a LAMBDA expression, or a macro created by DEFMAC from a LAMBDA expression. Each sublist, containing a least two variables, specifies a property, the CAR of each sublist, whose value is a LAMBDA expression of each property list of the SYMBOLic atoms in the remairder, CDR, of the sublist of the parameter, COMP-LIST. Although the first parameter, COMP-LIST, may not refer to LAMDA (without a "B") expressions, FUNCTION function expressions, that capture a binding environment, and fluidly bound definitions that the system association list, ALIST, maintains, compiled code can handle captured binding environments and fluid definitions. If the compiler gets a second, non-NIL parameter, DUMP-FLAG, the compiler sends a listing of intermediate code, pseudo-machine instructions, created from the S-expressions to be converted into machine code, to the current output file. If the compiler gets the third parameter, MASTER, the compiler may get a NIL secord parameter as a placeholder to avoid listing intermediate code. Without a third parameter, MASTER, the compiler begins a new area of writable I-space, of which the I-space address, *CDR, of a master LINKER indicates the start, and the compiler places the machine instructions contiguously into the I-space area. With the third parameter, MASTER, a master function LINKER for the code area just generated but not

finished, the compiler continues compilation into the current, I-space area. The compiler may place seperately compiled but mutually referencing functions in a contiguous area to insure correct dumping by the Pretty Printer. When the Pretty Printer outputs compiled code to a file, the variable binding the master LINKER for each compiled code area should precede any other variables bound to secondary entry points within each area in the first parameter, the dump list, of a single Pretty-Printer call.

After all compilations are complete, the function EXCISE with no parameters, defined with the compiler

(EXCISE)

removes the compiler, associatec flags, properties, and functions from garbage collection marking. EXCISE returns some space for use in new S-expressions and reduces the frequency and length of garbage collection.

3.3.2. Fluid Variables.

The compiler returns a list of variables that it found without a constant binding, without a fluid marking flag, and without a declaration as a formal parameter directly accessible to the current function environment. All free variables that the compiler encounters should either have a fluid marking or a constant binding. If a variable is not declared as a formal parameter in the argument lists of the nearest surrounding PROG, LAMBDA, or LAMDA expression, a variable is free. The compiler may treat a variable declared as a formal parameter of an outer expression, but strictly definec as free in an inner LAMBDA expression, as a formal parameter without needing a fluid marking. In particular, the compiler uses formal parameters freely referenced with an internal LAMBDA (or LAMDA) serving as the first member of a list in a calling sequence

<LAMBDA (FREE) . . . ((LAMBDA () . . . FREE . . .)) . . . >

and serving as the in-line function parameter of the system defined functionals (functions getting functions as parameters)

DUMP, INTO, LISP, MAP, MAPC, OBLIST, ONTO, and PROP

as formal parameters without needing a fluid marking. Thus the compiler compiling the function defined by

(LAMBDA (X) (PROG <Y> <INTO X (LAMBDA (Z) (LIST X Y Z))>))

would treat all of the variables X, Y, and Z as formal parameters even though X and Y are free within the innermost LAMBDA. The compiler allocates a run-time position on the interpreter value stack for formal-parameter variables without a fluid marking. The compiled code generated from the specialized internal LAMBDA expression used within some system functionals can directly find

the location of free variables used as formal parameters by knowing the number of subsequent values pushed onto the value stack. Compiled code referencing formal parameters without fluid marking creates no binding pairs on the system association list, ALIST. Indeed, the formal parameter variable need not be defined after compilation. The binding of fluid variables occurs as name-value pairs on the system association list, ALIST. All formal parameters of interpreted functions receive such fluid bindings. Like interpreter functions, compiled code calls the LISP interpreter to create fluid bindings. If the variable has a previous constant binding, the interpreter outputs a message

WARNING, (NAME . VALUE) BINDING HIDDEN

where (NAME . VALUE) is the attribute-value pair, since the interpreter references the previous constant binding before the newly created fluid binding. If compiled code does not treat a variable marked fluid as a formal parameter, compiled code calls the LISP interpreter to create, access, and alter fluid variable bindings. If SET or SETQ attempts to create a new fluid binding but cannot find any marker established by a LISP supervisor on the system association list, ALIST, such as during start-up loading under UNIX, the interpreter terminates in error (IOT under UNIX). If the interpreter attempts to access a fluid variable with no binding, the interpreter first generates an internal error -8, as if evaluating

(ERROR -8)

that a previously invoked ATTEMPT may use to restart evaluation. Otherwise, if no ATTEMPT catching error -8 is active, the interpreter outputs the message

WARNING, X IS UNBOUND

where X is the name of the fluid variable, solicits a replacement by outputting the prompt

Help:

and then reads and evaluates the replacement expression. If compiled code expects a variable to have a constant binding, the code will use the value in the constant binding part of the variable (*CAR) regardless of any other fluid bindings and of any undefined status of the variable. If the unbound constant variable was to have provided a function definition to a function call, usually because functions present during compilation have not been re-loaded with compiled code, the location referenced by the spurious function call will produce the cryptic message

WARNING [?OO] IS NOT A FUNCTION

will prompt for a new function by outputting

and will read, evaluate, and use the requested function in the compiled function call. Variables used by compiled code should be marked fluid if they may be unbound when compiled code is reloaded, if they are treated as free variables without constant bindings, or if the interpreter needs to find their binding on the stack. For example, the first parameter of SET or any variable referenced in an explicit call to EVAL, the interpreter, should be marked fluid. The function FLUID included with the compiler, puts a fluid marking on each variable member of the parameter, a list:

(FLUID LST)

The function UNFLUID, also included with the compiler, removes any fluid marking of each variable member of the parameter, a list.

(UNFLUID LST)

3.3.3. Compiling the Execution Sequence.

Compiled code limits the use of PROG labels and the RETURN function used within PROGs. In interpreted expressions, when calling the GO special form, the interpreter searches for the unevaluated GO parameter, an atomic symbol, within the parameter list of each surrounding PROG expression list until finding the parameter, which is a PROG label, or reaching a level of LISP supervision, regardless of the depth of nesting of function call construction. However, the compiler will correctly generate code for the GO special form only if a compiled PROG calls the GO special form at the top level or within nested calls of the COND, AND, OR, or DO special forms and provides the GO parameter as a label within the PROG body. The GO special form may also be nested within one call of the ATTEMPT special form as part of an ATTEMPT alternative. After compilation, a PROG no longer uses the atomic symbol, the PROG label, that served as a placeholder for GO searches. Thus a GO special form external to a compiled PROG may not access the location indicated by the former PROG label. The compiler converts RETURN function calls within PROG expressions into in-line machine code. A RETURN function call performed by a function called by a compiled PROG special form but external to the definition of the PROG body will not cause actions within the compiled PROG. Instead, the RETURN function produces a return from the nearest surrounding interpeted PROG special form or LISP supervisor call.

3.3.4. Compile-time Expressions.

The compiler evaluates some expressions while compiling, using the value obtained then, rather than generating compiled code to evaluate the expression. The function MANIFEST signals

to the compiler that the expression that the interpeter would evaluate to pass to MANIFEST as a parameter is to be evaluated by the compiler:

```
(MANIFEST EXP)
```

In particular, many function definitions within the DEBUG package use the MANIFEST function. Thus even though the DEBUG package could apply BREAK to a function used internally, the compiled DEBUG package uses the orignal cefinition of its internal functions without BREAK applied. This avoids using functions on which BREAK was applied for tracing. Otherwise, compiled code would reference the defining atomic symbol each time the code needed the function definition.

The compiler evaluates macro calls passing the unevaluated parameters. However, unlike the interpreter which then evaluates the expression returned by the macro call, the compiler compiles code for the value of the macro call. Thus macros which DEFMAC creates can create a new expression from their unevaluated parameters that can be interpreted or compiled at each invocation. For example, a macro that increments its parameter could be created by the following:

Figure 20 - DEFMAC ADD1MAC example.

```
(DEFMAC ADD1MAC ? Define macro
  (LAMBDA (ARG)  ? with one unevaluated parameter
          (LIST  ? that creates a new expression
           'SETQ ARG (LIST ? Replace parameter
                       'ADD1 ARG)))) ? as incremented
```

With the ADD1MAC macro defined, when the compiler encounters the S-expression

```
(ADD1MAC X)
```

the compiler generates code for the expression

```
(SETQ X (ADD1 X))
```

as if the latter expression were used instead of the former.

4. References.

Bell Laboratories, UNIX Programmer's Manual, Sixth Edition, Murray Hill, New Jersey, 1975.

Digital Equipment Corporation, The DOS/BATCH Handbook, DEC-11-ODBHA-A-D, Maynard, Massachusetts, April, 1974.

Digital Equipment Corporation, KT11-C Memory Management Unit Maintenance Manual, DEC-11-HKTCA-C-D, Maynard, Massachusetts, November, 1974.

Digital Equipment Corporation, PDP-11 Paper Tape Software Programming Handbook, Maynard, Massachusetts, 1973.

Digital Equipment Corporation, PDP-11 Peripherals Handbook, Maynard, Massachusetts, 1975.

Digital Equipment Corporation, PDP-11 Processor Handbook, Maynard, Massachusetts, 1974.

B. W. Kernighan, UNIX For Beginrers, Bell Laboratories, Murray Hill, New Jersey, 1974.

D. E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, pp. 417-420, Addison-Wesley Publishing Company, 1969.

W. M. Lay, D. L. Mills, M. V. Zelkowitz, Design of a Distributed Computer Netwcrk for Resource Sharing, AAIA Computer Network System Conference, Huntsville, Alabama, April, 1973.

W. M. Lay, D. L. Mills, M. V. Zelkowitz, Operating Systems Architecture for a Distributeo Computer Network, Proceedings of IEEE/ACM Conference on Trends and Applications of Mini-computer Networks, Gaithersburg, Marylanc, April, 1974.

J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, M. I. Levin, LISP 1.5 Programmer's Manual, The M. I. T. Press, 1962.

E. Norman, LISP, Academic Computing Center, 1210 West Dayton ST., Madison, wisconsin 53706, April, 1969.

E. Norman, Unpublished report on 1108 LISP implementation, Academic Computing Center, 1210 West Dayton St., Madison, Wisconsin 53706, 1974.

D. M. Ritche, UNIX Assembler Reference Manual, Bell Laboratories, Murray Hill, New Jersey, 1975.

D. M. Ritche, K. Thompson, The UNIX Time-Sharing System, Communications of the ACM, Vol. 17, pp. 365-375, July, 1974.

H. Schorr, W. M. Waite, An Efficent Machine-Independent Procedure for Garbage Collection in Various List Structures, Communications of the ACM, Vol. 10, pp. 501-506, August 1967.

G. J. Sussman, T. Winograd, Micro-Planner Reference Manual, PLNR.MEM 226, Stanford, California, April, 1971.

5. Appendices.

5.1. Available Operating Systems.

PDP-11 VLISP is available in several versions. The major differences between them are the operating systems and machines which house VLISP. The coding of the interpreter is nearly identical for all of the versions. The roster of operating systems follows in historical order.

5.1.1. Stand-Alone Systems.

In the absence of a reliable, available, operating system to develop VLISP, a rudimentary operating system has been used. The stand-alone operating system is a class project for a data concentrator that was modified and rewritten. The system is loaded into core by a bootstrapping process. Once in core it examines how much core is available (at least 16K and up to 32K words) and which communications device is present (DC11 or DL11). The system then continues using only what has been found. Programs may be loaded or printed out using either the console teletype, or one communicatiors device, or a combination of the two. A small debugging package is available to examine and alter absolute core locations with the console teletype. This permits patches to known bugs and trial corrections to problems with the interpreter code. Program patches should be made using the facilities of the LISP language.

The stand-alone system is available in several formats.

5.1.1.1. CIMSES - Canberra Magretic Tape System.

VLISP is available on magnetic tape cartridge used by CIMSES (Canberra Magnetic Tape Operating System). At present VLISP is kept on a separate cartridge by the author. Perhaps later, when a more finalized version is produced, VLISP will be included as a processor on the system tape.

5.1.1.2. PDP-11/45 with Disk.

VLISP is available on the disk cartridge of some machines and the fixed disk of others. The system is loaded and run with the appropriate disk loader.

5.1.1.3. PDP-11/40 with Disk.

VLISP is available on the cartridge disk of the PDP-11/40. An effort has been made to keep PDP-11 VLISP downward compatible with the PDP-11/40. However, the protection of separate instruction and data spaces is not provided on the PDP-11/40. Moreover,

the address space available for data on the PDP-11/40 is ultimately more restricted, even if virtual memory could be provided. Thus future versions of the operating system may not support VLISP on the PDP-11/40.

5.1.1.4. Paper Tape Software System.

A copy of the DEC program DUMPAB (Dump in Absolute Format) can be appended to the code. This would enable paper tape absolute versions of the system to be produced for systems without operative mass storage. Due to copyright restrictions, the program DUMPAB may not be transmitted to systems outside the University of Maryland.

5.1.2. Virtual Operating System (VOS).

The original intent was to write VLISP for an environment with virtual address space and cooperating processes. The interfaces of the VLISP interpreter have been designed to be compatible with the DCN/VOS (Distributed Computer Network/Virtual Operating System) being developed at the University of Maryland.

5.1.3. Disk Operating System (DOS).

A VOS emulator exists for use between the VLISP interpreter and DEC's Disk Operating System (DOS). The emulator intercepts the TRAP instructions given by the VLISP interpreter for I/O and other services. The emulator converts the interpreter requests into EMT instructions used by DOS. Buffers, link blocks, and filename blocks are maintained in the emulator for use by DOS. The emulator simulates the needed features of VOS for the VLISP interpreter while providing access to the DOS file structure.

5.1.4. Bell Laboratories' UNIX Operating System.

Bell Laboratories' UNIX operating system can support VLISP. Conversely, UNIX VLISP can access the powerful features of UNIX including system calls and command-interpreter, shell calls. The UNIX operating system may be extended to support a writable, per-process, I-space on PDP-11s with separated I and D space, so that VLISP may support compiled S-expressions. Another UNIX extension supports reading single lines of characters from files up to and including the next new line character, which improves the speed of VLISP character I/O.

5.2. Using the Operating Systems.

5.2.1. Bootstrapping.

After turning on a computer the contents of core may be unknown or unusable. A small procedure, a bootstrap, is initially used to start up whatever operating system is to be used. Hopefully, a hardware bootstrap will be available or the bootstrap will already be in core. If not, the bootstrap can be entered using the switches on the front of the machine. A listing of the CIMSES bootstrap is included at the end of the CIMSES system documentation. The 24K CIMSES bootstrap starts at 137720. For 16K core machines, the disk bootstrap starts at 77740, for 32K at 157700. The disk should be powered up after the system, then the run-load switch moved to the run position. Wait for the run light to go on (in less thar a minute). (Power down in reverse order.) The DEC paper tape software handbook contains the paper tape bootstrap and procedures. The tape and disk bootstraps are started by the following procedure:

A. Make sure the console teletype is online and the disk or tape reader is on.
B. Put the HALT-ENABLE key in the HALT (down) position.
C. Place the bootstrap start address in keys.
D. Press the load address key on the console.
E. Press the START key.
F. Put the HALT-ENABLE key in the ENABLE (up) position.
G. Press the continue (CONT) key.
H. If nothing happens, start over after checking the bootstrap for errors; otherwise, the loading operation can begin.

Alternatively an RK11 disk can be bootstrapped by the following procedure on PDP-11/45s without loading an in-core bootstrap program.

A. Put the HALT-ENABLE key in the HALT (down) position.
B. Put zero (0) in the keys and press the START key.
C. Put 777406 (Word Court Register) in the keys and press the load-address key.
D. Put 777000 (Negative of Word count) in the keys and lift the DEPOSIT key.
E. Put 777404 (Command Register) in the keys and press the load-address key.
F. Put 000005 (READ and GO) in the keys and lift the DEPOSIT key.
G. Press the EXAMINE key; bit 15 of the address display should be off.
H. Put the HALT-ENABLE key in the ENABLE (up) position.
I. Press the CONTINUE key.

5.2.2. Stand-Alone Systems.

The stand-alone systems differ only in the medium on which

they are housed and the method used to load them.

5.2.2.1. Loading and Running the Loader.

After the bootstrapping procedure, the following procedures will load and run the loacer.

5.2.2.2. Cartridge Disk Systems.

A. If the system types "READY TC DIAL" or "WAITING FOR CARRIER" some telephone connection must be made with the appropriate device, either DC11 or DL11. Any terminal may be called or even another computer.

B. If the system types "SELECT SPEED . . ." type "0" for 110 baud lines, "2" for 1200 baud lines, or "3" for 300 baud lines.

C. The system must be informed of the location of the program on disk. This is currently sector 5000 or 5100 on the cartridge, unit 0. To signal this type ALT-MODE A. The system will respond by querying U, F, and then D. After U type "0" for unit 0. After F and D type the starting sector location (5000).

D. To start the loading operation type ALT-MODE L.

E. If the system asks for a loader disk address use 16 (at present).

F. When the system asks for a loader address, type carriage return or any address higher than the highest address loaded, currently 44210.

G. When the system halts at an address near 22570, the VLISP interpreter has been loaded.

5.2.2.3. CIMSES - 24K Core.

A. Ensure the CIMSES 24K system tape is mounted on unit 1, the leftmost tape drive.

B. If the system is reacy to receive commands it will type a left bracket ("["). If not, try the bootstrap procedure.

C. Mount a tape containing the VLISP interpreter -- usually on unit 2.

D. Position the tape at the start of a copy of the VLISP interpreter. Usually pressing the rewind button is sufficient.

E. Type "RUN" followec by ar ALT-MODE to load the magnetic tape loader.

F. When the system queries "]U#" type "2" (if appropriate) giving the unit number which is positioned at the beginning of the interpreter.

G. When the system halts near location 22570, the VLISP interpreter has been loaded. Otherwise a loading error has occurred and the bootstrap procedure should be restarted.

5.2.2.4. Paper Tape Software Systems.

A. Load the paper tape absolute loader and modify it if DC11s and telephone lines are to be used instead of the console teletype.

B. Load a copy of the stand-alone VLISP system into another machine.

C. After loading, the VLISP interpreter halts. Note the address for later use.

D. Set 040004 in the switches of the sending machine, set the HALT switch, press LOAD ADDRESS, press START, set the ENABLE switch, and press CONTINUE. A modified copy of DUMPAB (dump in DEC absolute format) can be provided following the data area. DUMPAB will halt to wait for an address to begin dumping code. The code for DUMPAB is overwritten once the VLISP interpreter begins.

E. Connect the two machines by telephone. If using DC11s at 300 baud ensure that all error bits in their device status registers are off and that locations 774000 and 774004 both have octal 31 set.

F. Start the absolute loader in the receiving machine.

G. Start DUMPAB in the sending machine. Note the stack and device register queries have been preset to use a DC11. Only a dump start and stop location are needed.

H. In the sending machine, put 400 in the keys for the dump start location.

I. Press CONT; the machine should halt again.

J. Put 37000 in keys for last dumped address.

K. Press continue (CONT); the sending machine should start the transfer.

L. If the sending machine halts while the receiving machine does not, the VLISP interpreter has been loaded; otherwise try again from the beginning of the bootstrap.

M. Start the VLISP interpreter at the address where the sending machine halted.

5.2.2.5. Starting the VLISP interpreter.

After VLISP is loaded, the kernel stack pointer is set and the operating system halts. Patches can be made at this point. If DC11s are to be used at 110 baud instead of the preset 1200 baud, the device status register reset values located at 404 and 406 should be changed from 121 to 101 using the switches on the machine. If a different device, such as a DC11 instead of a DL11, is to be used, the receiving and transmitting status register addresses at locations 400 and 402 must be changed. After any such patches have been made, press continue (CONT) to restart the system and initialize the data area. The initialization code, which is used only once, is later overwritten by the user control stack. At the end of data initialization, a HALT instruction in user mode occurs which generates an interrupt.

The illegal user mode HALT interrupt is fielded by a small debugging procedure that is part of the stand-alone operating system. When the debugger starts, it sends a message and prints the top 16 words of the kernel system stack. The first word of each line is the starting address dumped. When an error interrupt occurs this stack contains:

A. A return address.
B. CPU registers R0 to R5 from register set 0.
C. Another return address.
D. The stack pointer for the previous mode.
E. The program counter (PC) of the interrupt.
F. The processor status word (PS) of the interrupt.

The values on the kernel stack are used when a restart is made.

The debugger accepts commands of the form:

OP1 OP2 CMD .

OP1 and OP2 are octal numbers of which only the last six digits are significant. If an error is made while typing a number, simply retype all six digits of the correct number. An unknown command will simply repeat the previous command. The second parameter may be omitted. The command letters are:

A - Restart using the current values on the stack. No parameters are needed.
B - Jump to location of OP1 resetting the kernel stack pointer to OP2.
C - Change the contents of location OP1 to the contents of OP2. The old and new values of location OP1 are displayed.
D - Dump OP2 locations starting at address OP1. Each line printed consists of an address followed by 8 dumped words.
E - Restart the VLISP interpreter at its error recovery point. No parameters are needed. The old PC and PS from the stack are saved for use by the interpreter. Thus if the VLISP interpreter is garbage collecting or doing some other uninterruptable operation, it may restart to complete the operation without irreparable damage to itself.

At this point any patches may be made using the debugger instead of the switches on the machine. After any patches are completed, the VLISP evaluation process is initiated by typing the command "A" to the debugger. The VLISP interpreter will then type a sign-on message and request an expression to evaluate by typing:

EVAL:

5.2.2.6. Changing I/O Paths.

The standard I/O paths may be altered by commands issued at the console. Three entities may send and receive character by character I/O. These entities and their logical device names are:

A - VLISP interpreter prccess.
B - Computer console teletype.
C - Auxillary serial I/O device (DC11 or DL11 modems).

The command

BELL LOGICAL-NAME-FROM LOGICAL-NAME-TO

issued at any time, including the middle of a line, at the console teletype, causes further output from the entity specified by LOGICAL-NAME-FROM to be sent tc the entity specified by LOGICAL-NAME-TO. Note that if the initial speed of the DC11, 1200 baud, is to be changed, a program patch must be made.

5.2.2.7. Typographical Error Ccrrection.

While typing an input lire characters may be corrected using the backspace character (BS), CONTROL/H, and then typing the correct charcter. Do not attempt to backspace beyond the beginning of a line or once the erd has been passed. The entire line may be deleted by typing the character cancel (CAN), CONTROL/X, before any other control character that will end the line. After a line has been sent to the process by typing carriage return (CR) or some other control character, CAN and BS have no effect on the line.

5.2.2.8. Stopping VLISP Under Stand-Alone Systems.

The process may be interrupted by typing the three character sequence:

BELL CHAR ENQ

(BELL is CONTROL/G and ENQ is CONTROL/E)

If the second character, CHAR, is also ENQ then the process may be stopped as is and the debug procedure called. Control may be return to the VLISP interpreter process to continue by giving the command "A" to the debugger. If CHAR is not ENQ then the process will complete any garbage collection and return to the latest level of supervision using CHAR as the error type code. In order to send a BELL to the process type two BELLs.

5.2.3. VOS.

VOS may be brought into core from disk by first bootstrapping the disk loader. Then the disk loader is used to bring in the VOS code. The computer may then be halted, any patches made, and then restarted at address zero (0). Next, the following procedure is used to load and start the VLISP interpreter.

A. The command language interpreter sends a period (.) in order to solicit the next command. Type a carriage return to end any current command. If after loading, the period does not appear, type the command "TT" ("TesT") to receive a test message. Typing CONTROL/E should interrupt any current processing and produce the command solicitatior, the period. If neither of these works, the system may need to be restarted or rebooted.

B. Type the command

```
OpenF 030C0 LISP OLD 0
```

in order to open the existing file "LISP" that contains the interpreter initialization procedure. The logical number, 03000, will be associated with this file. Note that only the upper byte of this number is significant. The fourth parameter, "0", specifies the drive that holds the file. Since zero is the default value of the fourth parameter it may be omitted.

C. Type the command

```
LinK 030C1 0 100C01
```

to map the segment 1 of the just opened file, 03000, to the default virtual address 0, with I and D space enabled with an execute only segment and to link to the initialization procedure just mapped at virtual location 0. This procedure assigns temporary data workspaces, initializes the data areas, and maps the VLISP interpreter code segment into the VLISP interpreter, which should send a sign-on message.

After the VOS VLISP interpreter initialization procedure has been started, the interpreter should send a sign-on message and proceed to request an expression to evaluate by typing:

```
EVAL:
```

Any patches should be made before loading, using the map segment (MP), display storage (DS) and alter storage (AS) commands. Except to make permanent patches and start the VLISP initialization procedure, the file "LISP" should not be used by the programmer. Inadvertently, the data initialization or code segments could be altered. Similarily, use of the logical file number, 03000, should be avoidec since VOS does not provide system file protection.

The process may be interrupted by typing the ENQ character. The process completes any uninterruptable operation and then links to the command language interpreter that solicits a command by typing a period (.). The top of the user stack is the error code that will be used by the process. The process may be restarted by the command "SP", which stops the command language interpreter and returns to the VLISP interpreter process.

If the VLISP interpreter must be restarted after operator intervention, stack overflow, or garbage collection failure, the old stacks and association list will be lost along with any SETQ bindings.

5.2.4. Disk Operating System (DOS).

The DOS-LISP interface has been developed and tested under DOS/BATCH version 6.

5.2.4.1. Getting DOS VLISP Started.

The following procedures load and start the DOS version of LISP, assuming that DOS has just been bootstrapped. If DOS is already running, only the last portion of the procedure may be needed. Note that DOS system commands must be typed in upper case and that only the first two letters of the command are significant.

A. After being bootstrapped, DOS should sign on with a version number and prompt for a command with "$". Note that the disk must not be write protected if the sign-on message is to appear. If another program is active type CONTROL/C followed by "KILL" in order to stop it.

B. Specify the date and time (24 hour clock) to the system by commands such as:

```
DATE 04-JUL-76
TIME 13:01:00
```

Files produced will be marked with given time and date.

C. Log in to the system by typing a command such as:

```
LOGIN 13,13
```

The numbers must specify a user group number and user number, the User Identification Code (UIC), between 10 and 376 in octal, known to the system file structure. New UIC numbers may be entered into the file system using the system program PIP (Peripheral Interchange Program). If a different UIC is desired, other than the one currently in use, type the command "FINISH" to log off the system before logging in under a new number.

D. Type the command:

```
RUN LISP
```

The above command brings in overlay segments used in code and data initialization, opens the primary input and output datasets, and sets values used by DOS. When the initialization is complete, the VLISP interpreter signs on and requests an expression to evaluate by typing:

```
EVAL:
```

5.2.4.2. Interrupting, Restarting, Killing DOS VLISP.

Once VLISP is running, the attention of the monitor may be obtained by simultaneously striking the CONTROL and "C" keys. The monitor responds by echoing, typing a period (.), and interrupting any current output. One of several one-line commands may then be given:

A. RESTART - Restart the interpreter at the interrupt point established by the program. This controls runaway

programs. Do not use "BEGIN" to restart DOS VLISP.

B. PRINT - Turn console output either off or back on again. This command, which is transparent to the program, can eliminate excessive output.

C. ASSIGN - Associate a logical name, for example:

ASSIGN SY:NEWFILLSP,4 .

This command specifies a logical port or dataset number between 3 and 10 that is to have the external name given. In the above example the logical port number 4 is associated with the dataset SY:NEWFIL.LSP on the system device, "SY:"; with file name, "NEWFIL"; and with file name extention, ".LSP", to denote a VLISP code source file. Further details and examples can be found in the DEC DOS manuals.

D. KILL - This command stops the current program in a tidy fashion. Files are closed and an orderly return is made to the monitor regardless of what the VLISP interpreter may have been doing. Once "KILL" is issued the program cannot be restarted.

After a system error message, e.g.

F345 001306 or A003 040676

the "KILL" or "RESTART" commands may also be given. Some system errors, such as stack overflows, are intercepted by the DOS-LISP interface, which prints some system stack values and returns directly to the VLISP interpreter.

5.2.4.3. Input and Output Datasets.

The VLISP interpreter refers to input/output files or datasets by logical number. The DOS-LISP interface provides link and filename blocks for logical numbers between 1 and 10. Logical numbers 1 and 2 are used for default input and output. DOS logical names, "CDI" and "CMO", are used in their link blocks respectively, so that VLISP may be used in batch mode. In interactive mode the device, "KB:", console keyboard, is used for default input and output. The remaining logically numbered files, 3 to 10, use, as defaults, the system device, "SY:"; extension, ".LSP", to denote LISP source files; default file names, "3" to "10"; and DOS logical names, "3" to "10". The datasets may be reassigned using the ASSIGN command or by a TRAP instruction issued by a LISP program. For example, the LISP expression

(TRAP 37Q 4 "SY:OLDFIL.LSP[1,1]")

which is equivalent to

(OPEN "SY:OLDFIL.LSP[1,1]" NIL 4)

will change the file name block to use "SY:", the system device;

"OLDFIL" as the file name; ".LSP" as the extension to denote a LISP source program; and User Identification Code (UIC) "[1,1]" to exactly specify the entry. The first TRAP function parameter gives the TRAP offset, 370 (octal), which specifies opening a file. The second parameter, in this example 4, specifies which logical file is to be altered. The third parameter of the function is a string in standard DOS command string syntax. The standard command string drop-out rules apply: the system device is assumed if no device is specified and the current user's UIC is used if none is specified.

5.2.5. The UNIX Operating System.

5.2.5.1. Getting UNIX Started.

The UNIX bootstrap procedure will read a larger bootstrap program that searches the file structure for a copy of the UNIX operating system whose name it obtains by prompting with an at-sign (@). Before using the larger bootstrap, place 173030 in the console switches so that the UNIX file structure may be checked in single-user mode. Usually typing "unix" followed by a carriage return will load a copy of the UNIX operating system. The UNIX system, if in single user mode, will print a sign-on message and prompt for input with a number-sign (#). Typing "date " followed by an 8-digit number giving the month, day, hour, and minute in pairs of digits such as

```
date 07041301
```

for 13:01 on July 4th corrects the system's idea of the time and date.

The file structures should each be checked for integrity by typing "icheck" and "dcheck" followed by a raw file structure device name for each file-structured device in use, such as

```
#icheck /dev/rrk0
```

to check platter zero of an RK11 disk. If the file structure checks report no anomalies, the system may then be safely used in multi-user mode by placing any other non-zero number in the keys, such as 141774, and pressing EOT (Control-D) on the console keyboard. Users may then log into the system by typing the appropriate identifier and password in response to the "login:" prompt.

5.2.5.2. Invoking UNIX VLISP.

Typing

```
lisp
```

to the shell command interpreter invokes UNIX VLISP in prompt

mode. In prompt mode, UNIX VLISP outputs a sign-on message and prompts for input by outputting

```
Eval:
```

The VLISP invocation command received by the shell may include file names or the options plus (+) or minus (-). The shell command interpreter may expand special characters within file name strings to create a list of files. VLISP loads the files in order, using the function LOAD, before signing-on or invoking a level of LISP supervision. Since the LISP supervisor handles most LISP error conditions, errors occurring during start-up loading usually cause an abort of VLISP (IOT). If the VLISP invocation gives file names, VLISP does not invoke a LISP supervisor until encountering either the option plus (+) for normal supervision with prompts and interrupt handling or minus (-) for silent supervision without prompts or interrupt handling. When VLISP enables interrupt handling, the delete key (DEL) asynchronous interrupt causes VLISP to generate an internal error -3, as if evaluating

```
(ERROR -3)
```

which will restart the LISP supervisor if not caught by a previously invoked ATTEMPT. Without interrupt handling, VLISP does not alter the previous asynchrorous interrupt status given during invocation, so that VLISP running as a background process with the minus (-) option will not be affected by asynchronous interrupts. In either instance, VLISP does not alter the status of the QUIT asynchronous interrupt. If UNIX VLISP gets the minus (-) invocation option, or if VLISP is a child process of the original invocation, VLISP suppresses the sign-on message, the expression to evaluate prompt "Eval:", and the "Value:" prefix so that processes may communicate in a non-interactive mode, such as using standard I/O through a pipe, e.g.

```
>commands lisp /lisp/pp - | lisp yourfiles? - > outfile &
```

Invoking VLISP without file names or options is equivalent to using just the plus (+), with-prompts option:

```
lisp +
```

5.3. Coding and Assembly.

5.3.1. Assembler Syntax Differences.

The PDP-11 VLISP interpreter is written in a modified version of PDP-11 Assembly Language (PAL). Modified PAL can be used with a cross assembler on the University of Maryland's UNIVAC 1100 series machines. Programs written in PAL can be transported to other PDP-11 installations. Moreover, optimization involving the use of addresses, as in the VLISP interpreter, would be difficult in higher level languages.

The University of Maryland's modified PAL is quite similar to the orginal DEC PAL. Although many of the features of the DEC MACRO assembly language were available in modified PAL, they are unused in order to maintain transportability. Unfortunately, some different features were used. Users outside the University of Maryland may have to program around them. These differences include:

.TITLE in Maryland PAL provides assembly listing headings only. In other versions, .TITLE also provides information to the LINK processor.

.EJECT in Maryland PAL has the same meaning as .PAGE in other assemblers. .EJECT and .PAGE continue the listing on the next page.

.ALIGN advances the current location counter to the next location that is a multiple of the power of two given by the parameter. .EVEN is equivalent to

```
.ALIGN 1
```

The .ALIGN directive was useful in developing the growing VLISP system. Its effect can be emulated by resetting the location counter, provided proper care is taken. For example, suppose that a previous label, "FLOOR:", were defined on a hardware segment boundary, a multiple of 020000 (octal), such as at the beginning of the code. The statement

```
.ALIGN 020-3 ; Align on hardware segment boundary
```

would align the assembly location counter (.) on the next segment boundary. Since (in octal)

```
020-3 = 015 and 020000 = 2**015
```

the location counter altering statement

```
. = .-FLOOR+020000-1/020000*020000+FLOOR
```

could replace the .ALIGN 015 statement.

.IF begins a section of code that is conditionally assembled. If the parameter to the .IF statement is false, the code following .IF is not assembled up to a matching statement

```
.ENDC ; End conditional assembly
```

that ends the conditional assembly area. Each assembler accepts a different syntax for the .IF statement. As an example, both the Maryland PAL cross assembler and DEC's MACRO assembler recognize the statement

```
.IF NE,CPLCPL ; Assemble only if compiler used
```

that is frequently used within the VLISP interpreter code. The code is assembled only if the label "CPLCPL" is not equal (NE) to zero. DEC's macro assembler also recognizes the statement if "NE" is replaced by "NZ", for not zero, while Maryland PAL does not. DEC's PAL-11S and PAL-11R assemblers would only recognize the equivalent statement

```
.IFNZ CPLCPL ; Assemble only if compiler used
```

that is also recognized by DEC's MACRO assembler. Bell Laboratories' UNIX assembler, "as", recognizes yet another, different, but equivalent statement

```
.if cplcpl / Assemble only if compiler used
```

to begin the conditional assembly area and the statement

```
.endif / End corditional assembly
```

to end the conditional assembly area. In order to reduce the difficulty in transforming code for different assemblers only the above formats for the .IF statement are used.

5.3.2. Conditional Assembly.

The assembler source code module "TRAPS" under DOS or "l0traps" under UNIX contains common definitions that are used with all of the assemblies. Several parameters defined by "TRAPS" may need to be changed depending upon the host configuration. The value "OBRSTV" (Output Buffer Reset Value) should be set to the column width (in octal) of the narrowest device used for primary output, usually the console keyboard. The values 110 (72), 120 (80), or 204 (132) may be used for KB33, LA30, or LA36 respectively. The flag "CPLCPL" is set to one (1 = on) or zero (0 = off) depending on whether or not, respectively, the compiled code functions are to be assembled as part of the interpreter. The compiled code functions should not be included with an interpreter for use with a PDP-11/40 or similar PDP-11s without memory-management-separated I and D spaces. The UNIX

operating system must be extended before using compiled code functions. The flag "PDP40" should be set if the code is usable on PDP-11/40s, that do not have separated I and D spaces. If "PDP40" is set, compiled coce should not be supported by not setting the flag "CPLCPL" and, under UNIX, the option "-i" for separated I and D spaces should not be used with the UNIX link-editor "ld".

Conditional assembly flags specify which kinds of floating-point arithmetic VLISP may support. If the host PDP-11/45 or PDP-11/70 has floating-point hardware, VLISP may support either single-precision or double-precision floating-point or both, depending on the settings of the flags FPPS and FPPD. To support no floating-point, VLISF has the flag NFPP set.

5.3.3. Assembly procedures.

Procedures for assembling each version of VLISP follow.

5.3.3.1. Stand-Alone Systems.

The stand-alone version of the VLISP interpreter and a small, in core, operating system are assembled together on the University of Maryland UNIVAC 1100 series machines by the following control cards. The source elements are assummed to reside in a file named "C.".

```
@SUSPEND . Divert the listing to a temporary file
@PDP*11.ASM,ICDS X,Y . Invoke the assembler
@ADD,P C.SVECS . Operating system workspace
@ADD,P C.TRAPS . Common values
@ADD,P C.SYS   . Operating system code
@ADD,P C.PLISP . VLISP interpreter code
       .ALIGN PAGBIT ; Align on 2000 byte boundary
@ADD,P C.WORKS . LISP fixed workspace and tables
@ADD,P C.ATOMS . SYMBOL, LINKER, and STRING initial data
@ADD,P C.STLISP . LISP data initialization code
@ADD,P C.URANUS . Operating system initialization code
       .END    DRIVER ; Start with system initialization
@RESUME,P . Print the listing efficiently
```

The load module, "Y", is then sent to the storage medium using the 1108 transmittion program PUNCH:

```
@PDP*11.PLNCH,CXT Y .
```

5.3.3.2. Virtual Operating System (VOS).

The VOS version of PDP-11 VLISP is assembled on the Maryland UNIVAC 1108 with the following commands:

```
@SUSPEND . Divert the listing to a print file
@PDP*11.ASM,ICDS X,C.V . Put load module in permanent file
        .TITLE  .LISP interpreter for VOS.
C       = . ;   Physical address of virtual origin of code
@ADD,P C.TRAPS . Common values
@ADD,P C.PLISP . LISP interpreter code
        .ALIGN  20-3 ;          Start data on segment boundary
B       = . ;   Physical address of virtual origin of data
@ADD,P C.WORKS . LISP fixed workspace and tables
@ADD,P C.ATOMS . SYMBOL, LINKER, and STRING initial data
        .ALIGN  20-3 ; Start LISP loader on segment boundary
@ADD,P C.LSPLD . VOS LISP loader creates mappings
@ADD,P C.STLISP . LISP data preprocessing code
@RESUME,P . Print the listing efficiently
```

The VOS VLISP load module C.V can then be transmitted to the VOS file system. Segments 3, 2, and 1 from the VOS file "LISP" should be mapped into hardware data segments 0, 1, and 2, respectively. The transmitted load module is then loaded into these segments. Segment 2 of the file "LISP", containing unpreprocessed initial data, is then mapped into USER hardware segment 0. The code (STLISP) is started at location 0 to preprocess the initial data. After a backup copy of the segments is made, VOS VLISP is ready for use as described above.

5.3.3.3. Disk Operating System (DOS).

The following DOS system commands, without the comments, construct the DOS VLISP interpreter, for the PDP-11/40 or PDP-11/45, assuming that the source is on the 9-track magnetic tape device, "MT:". The commands below are written in batch mode for clarity; however, the sequence is more safely performed in interactive DOS mode by someone quite familiar with DOS. Over 150 pages of 132 column wide output may be produced. If the available printers do not support 132 column print width, the statement

```
.NLIST TTM
```

must be deleted from the file TRAPS.MAC using the system program EDIT. The amount of output may be greatly reduced by including the no-list switches (/NL/NL:SYM) on MACRO output. If the primary keyboard is used in upper-case-only mode, the assembly line

```
.ENABLE LC ; Use lower case characters
```

should be commented out of the source code module "TRAPS.MAC".

```
$JOB MAKELISP[1,1]
$MESSAGE Mount VLISP source MAGTAPE
$MESSAGE When ready type CONTINUE
$WAIT
$RUN PIP ; Replace LP: by KB: if no line printer configured
#LP:<MT:[*,*]/DI ; Multiple copies are on tape
#SY:TRAPS.MAC<MT:TRAPS.MAC ; Common values
#SY:E.PAL<MT:E.PAL ; An enc card
#SY:LISPSY.MAC<MT:LISPSY.MAC ; Main DOS-LISP interface
#SY:LISPEX.MAC<MT:LISPEX.MAC ; XAP labels, storage allocator
#SY:LISPIN.MAC<MT:LISPIN.MAC ; Interface initialization
#SY:PLISP.PAL<MT:PLISP.PAL ; Interpreter code
#SY:STLISP.PAL<MT:STLISP.PAL ; Data preprocessing code
#SY:WORKS.PAL<MT:WORKS.PAL ; Fixed LISP workspace and tables
#SY:ATOMS.PAL<MT:ATOMS.PAL ; SYMBOL LINKER STRING data
$RUN MACRO
#SY:LISPSY,LP:<SY:TRAPS,LISPSY,E
#SY:LISPEX,LP:<SY:TRAPS,LISPEX,E
#SY:LISP,LP:<SY:TRAPS,PLISP,WORKS,ATOMS,STLISP,E
#SY:LISPIN,LP:<SY:TRAPS,LISPIN
$RUN LINK ; Create load module and symbol table
#SY:LISP[1,1],LP:,SY:LISP[1,1]<SY:LISP,LISPEX,ODT/OD
#LISP,LISPIN/T:110000/E
$RUN PIP ; Remove the scraps
#SY:E.PAL,TRAPS.MAC,LISP.OBJ/DE
#SY:PLISP.PAL,STLISP.PAL,WORKS.PAL,ATOMS.PAL/DE
#SY:LISPSY.MAC,LISPSY.OBJ/DE
#SY:LISPEX.MAC,LISPEX.OBJ/DE
#SY:LISPIN.MAC,LISPIN.OBJ/DE
#SY:[1,1]<MT:*.LSP ; LISP system programs available to all
#MT:/RU
$MESSAGE VLISP is now ready to go
$FINISH
```

If ODT (On-line debugging) is not desired, ODT/OD may be omitted from the commands to LINK and the top of code lowered to octal 74000 by using the switch, "/T:74000". Note that the top of code specified to LINK should lie on a VLISP page boundary, for example octal 70000, 74000, or 100000.

5.3.3.4. UNIX Operating System.

The following UNIX shell commands when interpreted cause UNIX VLISP to be assembled and installed, create necessary directories, and compile the LISP software assuming that UNIX has been altered to support compiled code. A later appendix (Modifying the UNIX Operating System) gives instructions for installing the UNIX improvements reeded tc support compiled code and provide reading up to the new-line character. The module "l0traps" should be edited to reflect the current configuration. The VLISP source must have already been read into the current directory, usually using the "tp" tape handler, and the LISP system software must have been put into a brother directory accessed as "../l". If compiled code is not available place the system software directly into a new directory called "/lisp" with write protection set. The current user should be either "bin" or "root".

```
: - Assemble interpreter code and workspace.
: - The assembled modules are
: - l0traps - Conditional assembly flags and definitions
: - l1top   - Top of code locaticn
: - l2stlisp - Start up procedure
: - l3garb  - Garbage collector and utilities
: - l4spch  - System special forms and basic functions
: - l5syst  - Operating system interface
: - l6ic    - I/O routines
: - l7comp  - Compiler rcutines
: - l8mapc  - List handlers
: - l9arth  - Arithmetic functions
: - labot   - Bottom of .text
: - lbworks - .data workspace
as l[0-b]* ; mv a.out text
: - Assemble initial atom definitions
as l0traps lcatoms ; mv a.out atoms
: - Link-edit leaving only the symbol table
: - No "-i" option if PDP40 flag set
ld -i -x text atoms /lib/liba.a
: - Install program for users
mv a.out /usr/bin/lisp
: - Extract symbol table
 ../symtab /usr/bin/lisp /lib/lisp.stb
: - Protect the code
chmod 755 /usr/bin/lisp
: - Make a directory for compiled LISP software
mkdir /lisp
: - Shell command file ccmpiles all LISP software
 ../l/cmpall
: - Protect compiled software, directory, and symbol table
chmod 644 /lisp/* /lisp /lib/lisp.stb
```

VLISP will then be ready for use.

5.4. Distribution.

The preferred medium of distribution is 9-track, odd-parity magtape recorded at either 800-NRZ frames per inch (FPI) or 1600-phase-encoded FPI by DEC's DOS system program PIP (Peripheral Interchange Program) or by the UNIX magnetic tape handler "tp". The files on the DOS version include assembler source code, object modules, and load modules for the VLISP interpreter designed for the Virtual Operating System/Distributed Computer Network (VOS/DCN) ceveloped at the University of Maryland, and a VOS emulator to use VLISP with DEC's DOS. LISP code for a Pretty Printer, a LISP S-expression editor, a debug package, and PLANNER are also included. Normally, copies of the DOS files are placed under UICs [1,1], [13,13], and [13,31] in order to minimize the possible effects of tape errors. The files recorded represent a current version working under DOS.

The UNIX "tp" version will include multiple copies of the VLISP source code modules. The tape includes a source code version for use with the DEC PAL assembler; a source code version for use with the UNIX "as" assembler; "c" source code and compiled version of a program, TRANS, to translate DEC PAL source code modules into UNIX "as" source code modules; S-expression versions of the LISP software including a Pretty Printer, an S-expression editor, a debug package, micro-PLANNER, a LISP function compiler, and a package of LISP operating system calling functions; text editor instructions to alter the UNIX operating system modules; "nroff" format synopses for use with the "man" processor; and other miscellaneous bits and pieces. The tape should not include usable compiled versions since the flags in "l0traps" will have to be altered to conform to the current configuration.

The University of Maryland's UNIVAC 1108 computer may produce other distribution media. Backup copies of the assembler source code for the VLISP interpreter, the VOS emulator for DOS, and the stand alone operating system which also emulates VOS, ready for assembly by the Maryland PAL cross assembler together with the LISP system programs, may be recorded on 9 or 7 track magnetic tape in one of the UNIVAC 1108 supported formats. Blocked card image magnetic tapes with cdd parity may be encoded in 7-track-BCD or 9-track-EBCDIC. Even-parity, 7-track tapes should not be produced since the tape hardware truncates any physical record with a zero frame, the value produced by a BCD encodeo ampersand (&). BCD anc EBCDIC translation also lose the upper/lower case qualities of the programs. The normally lower case variable names used by PLANNER must be preceded by a double exclamation sign (!!) in code tc be read by the VLISP interpreter to distinguish them from upper case variables when either BCD or EBCDIC translation is to be used. Unless otherwise requested, the physical records of blocked card image tapes each contain 720 characters that represent nine (9) 80-character-card images. The approximately 12,000 card images are recorded several times, each copy being followed by a file mark and the last copy followed by

multiple file marks, the logical end of tape. Tapes are normally produced at 800-NRZ-FPI. 1600-phase-encoded-FPI 9-track magnetic tapes and 200 or 556 FPI 7-track tapes are also possible. Since the PDP-11 assembler source code kept on the UNIVAC 1108 is intended for use with the cross assembler, some modification, mostly syntactic, may be needed before use with other assemblers, such as the DOS MACRO assembler.

Tape requests should specify the medium desired including format, number of tracks, density, parity, and any encoding method. Permissable variations on these tape parameters should also be specified to allow alternate methods to be used in case of hardware failures. Usually a listing of one copy of the tape's contents and basic documentation are included.

The material is copyrighted and may only be copied, used, transmitted, or altered as allowed by a copyright license. The license is intended to protect the system from unauthorized commercial exploitation. Requestors may prepare a suitable license for signature or a license will be created for them. Generally, the license permits the system to be copied, used, transmitted or altered provided that the copyright notice is included on all copies and versions created by the licensee.

5.5. Known Problems.

Some bugs still remain.

5.5.1. READing Floating-point Numbers.

The READ and TOKEN functions cannot construct floating-point numbers. As a temporary fix, a readmacro for the percent (%) character is included with the Pretty Printer routines. Before reading floating-point numbers, the readmacro must be defined, such as by loading the Pretty Printer. Once the readmacro has been defined, floating-point numbers must be preceded by a percent sign. For example,

%3.3 %0.7 %33D %52.12345E-22 %33333.333d+25 or %0.0

may be used to input floating-pcint numbers. In the absence of floating-point read routines, PRIN2 prefixes floating-point number output with a percent sign (%) so that the output may be re-read. Some day, the input scanner, which was written in the era of fixed-point-only numbers, will be re-written. Users may then define "%" as an ignored character so that old programs may be used without further difficulties.

5.5.2. Problems with the DOS Version.

When using the VOS emulator on DCS some problems may occur involving the interface.

5.5.2.1. Attention Interrupt and Free Storage Lists.

The system may not properly restart at the latest level of supervision as described above if free storage lists are being manipulated. This may occur during garbage collection or when allocating strings or array space.

5.5.2.2. Too Many Open Files.

Although communications packets have been provided for ten files, if VLISP is LINKed close to the DOS operating system buffer area only a few files may be open at once. When the allowed number of open files is exceeded, the DOS monitor will loop, probably hunting for non-existant buffer space. The system must then be rebooted: it cannot be restarted from the console teletype. This condition is especially likely in BATCH mode, since the system must allocate extra buffers for the BATCH I/O files.

5.5.2.3. Unsuccessful Storage Allocation Looping.

After an unsuccessful attempt to allocate storage following garbage collection the system will attempt to restart by resetting the stacks and the association list to their initial values. If this recovery is unsuccessful, it will nonetheless be attempted again. The only ways to halt this loop are through operator intervention, output file overflow, or exceeding a time limit. Each attempted recovery produces a register dump followed by the message

NO SPACE

if again unsuccessful.

5.5.2.4. Random Disk I/O.

Although code has been included to support random address I/O to contiguous disk files, it has not been debugged. Unexpected results may occur when using random I/O.

5.5.3. Problems with the UNIX Version.

The UNIX version of VLISP is not without its faults.

5.5.3.1. Number of Output Columns.

When UNIX VLISP is used in multi-user mode, different terminals with different column wicths may be used and the printer may have a different column wicth from the user terminal. Since many terminals wrap excess characters around to the next line, VLISP may send output using the largest available column width without difficulty in many configurations. However, if terminals are used that truncate the excess characters of a line, the output column width must be adjusted. The Pretty Printer includes a function PP-SHORT getting one parameter, a new column width, that may be called to alter the maximum output column width:

(PP-SHORT COL)

5.5.3.2. Floating Point Simulation.

The UNIX floating point simulator will not work with programs that use separated I anc D spaces. It may be possible to fix the simulator by altering it to use the new "mfpit" "sys" call. Use of the simulator for PDP-11/40 compatible code has not been fully tested.

5.5.4. Problems using PDP-11/4Cs.

Althougn VLISP may run or a PDP-11/40 host, VLISP will not have enough space to implement many modest scale programs. VLISP has not been fully tested for use with PDP-11/40 hosts.

5.6. Modifying the UNIX Operating System.

Two improvements can be added to the UNIX operating system that will extend the capabilities of LISP. The first improvement allows LISP to support compiled code on PDP-11s with memory management that support separated Instruction (I) and Data (D) spaces, such as the PDP-11/45. With the improvement, each process may maintain a writable I-space area following the protected, .text section of I-space, which processes share. Internally, UNIX maintains a writable I-space as part of the swappable process image following the process control section, the data section, and the stack area after a call to a new system call, "obreak" (overlay break). The new "obreak" system call has a similar syntax to the "break" system call. The lowest address not to be included in the requested area is passed in CPU register R0 to "obreak". If the address is within a hardware segment allocated to protected code (.text), "obreak" creates an empty writable, I-space, overlay area, the initial state after an "exec" system call. The "fork" operating system call replicates any writable I-space area, giving each process its own copy. If the writable I-space area requested of "obreak" cannot be allocated, "obreak" sets the error bit (C) on return. If the address specified falls within a hardware segment that contains protected code (.text) of the I-space area, "obreak" deallocates any writable I-space area to the initial empty state. If the specified address lies outside the hardware segments devoted to protected, .text code but above or below the highest address of any previously allocated writable I-space, "obreak" expands or contracts the writable I-space (overlay) area appropriately.

The code extension implementing "obreak" adds very little overhead. The UNIX operating system does not execute most of the code implementing the extension until an explicit "obreak" call. An additional parameter to the internal routine "estabur", which checks that requested data areas may be allocated and allocates allowable user .text, .data, stack, and obreak configurations to hardware registers, is the only routinely used interface between the extension and the rest of the operating system. In turn, the UNIX operating system calls "estabur" usually about once per switch between users, a relatively infrequent event compared to total processing. Within "estabur", user processes not using "obreak" perform less than twenty additional instructions per call, a negligible number. Thus "obreak" does not appreciably slow the UNIX operating system. In terms of space, in all, implementing "obreak" adds only a few hundred instructions to the UNIX operating system, a minor space requirement. The "obreak" extension is compatible with UNIX used on PDP-11/40s. On a PDP-11/40, "obreak" returns with an error status, since PDP-11/40s do not support separated I and D spaces.

"Mfpit" is a companion extension to "obreak". Although PDP-11s that support separated I and D spaces will permit writing into I-space with the "MTPI" (Move To Previous I-space) instruction, provided that memory management protection is disabled, such PDP-11s divert attempts by the "MFPI" (Move From Previous I-space) instruction ir USER mode from reading the USER mode instruction space while separated spaces are enabled. This "protection" was devised to enable execute-only code. Unfortunately, the protection cannot be directly overwritten. Thus the "mfpit" UNIX system call must be used to read USER-mode I-space. "Mfpit" returns, in CPU register R0, the contents of the USER-mode, I-space, word location passed to "mfpit" in CPU register R0. If the specified location is protected from reading or if it does not align on a word boundary, "Mfpit" returns -1, 177777 (octal), but does not set the "C" (carry) error bit of the PS (Processor Status) register. The extension for "mfpit" to the UNIX operating system entails a one-assignment-statement "c" function that calls "fiuword" within the operating system. "Mfpit" executes less than 100 machine instructions and executes them only when called.

A further extension, "readnl", rectifies an omission of the UNIX "read" system call by permitting line-at-a-time input in addition to the buffer-at-a-time input provided by "read". "Readnl" perform like "read" except when inputting characters from file systems of block structured devices and from pipes, which the file system of a block structured device, the system device, supports. The syntax of the "readnl" call is identical to the syntax of the "read" UNIX operating system call. When invoked to input characters from a file system, "readnl" does not return characters beyond the first new-line, line-feed character (012 octal) encountered. Like "read", "readnl" returns a count of the characters returned. Thus "readnl" inputs characters as if they were read from a terminal, one line at a time, without requiring that more characters than are needed be buffered. A more disconcerting situation occurs when using "read" with pipes, since backward (or forward) seeks are nct allowed. If two processes want to read varying-lergth character data from a pipe at once, each process must input data one byte per call using "read" so that each process will get the data intended for it. This involves excessive system overheac that using "readnl" can avoid by doing line-at-a-time input using the new-line character as an end-of-line marker.

The "readnl" extension is economical. When using "read", the "readnl" extension adds less than five additional machine instructions per data block. When reading from file systems, a call of "readnl" is only slightly slower than a corresponding call to "read" since "readnl" compares each character to new-line. The "readnl" extension should require less than 100 machine instructions of core space. In return for the expense of incorporating the "readnl" extension, "readnl" provides a more uniform method of character data handling by UNIX.

Each of the extensions tc the UNIX operating system are distributed as files of UNIX text editor "ed" commands produced by the UNIX file difference processor "diff". The UNIX operating system source code files to be changed are found under the UNIX directories

/usr/sys

and

/usr/sys/ken

as distributed by Bell Laboratories in the sixth edition. The names of the files distributed with ppLISP suffixed by ".e" correspond to names suffixed with ".c" in the UNIX system source code directories. For example, if the system source file

rdwri.c

were to be changed by the editor commancs in the file

rdwri.e

with both files in the current directory, the shell command interpreter line

(cat rdwri.e ; echc "w") | ed - rdwri.c

would implement the changes. Naturally the file protection mode should permit changes to the system source file. After the source code has been changed, a new UNIX configuration should be made following the instructicns distributed with UNIX. A comparative listing of the changes need for each file follow as produced by the UNIX file difference processor "diff". The listings consist of line numbers in the cld and new versions followed by the old and new lines to be changed or added.

Changing the stack size parameters in "param.h" allows ppLISP to allocate the longest possible stack within the last hardware segment, leaving one block at the bottom unallocated to enable stack overflow detection.

Figure 21 - Parameter changes to "/usr/sys/param.h".

```
11,12c11,12
- #define SSIZE 20        /* initial stack size (*64 bytes) */
- #define SINCR 20        /* increment of stack (*64 bytes) */
***
+ #define SSIZE 22        /* initial stack size (*64 bytes) */
+ #define SINCR 21        /* increment of stack (*64 bytes) */
```

Both the "obreak" and "readnl" extensions require changes to "user.h". Each extension adds new variables to the end of the user per process data area.

Figure 22 - Add to "/usr/sys/user.h" per-process variables.

```
54a55,59        : After u.u_intflg
+       char u_nlflag;  /* Flag for reading only to
+                        * the next new line char */
+       int u_osize;    /* Overlay, writable I-space
+                        * segment size.
+                        */
```

Changes to "sys1.c" embody most of the "obreak" system call extension. The changes also implement the "mfpit" extension with a single function.

Figure 23 - Add "Obreak" system call in "/usr/sys/ken/sys1.c".

```
123c123,129
-       if(estabur(ts, ds, SSIZE, sep))
***
+ /*
+  *    Include a null writable I-space length.  RLK
+  */
+       if(estabur(ts, ds, SSIZE, sep, 0))
+ /*
+  *    if(estabur(ts, ds, SSIZE, sep))
+  */
145c151,157
-       estabur(0, ds, 0, 0);
***
+ /*
+  *    Includes a null overlay segment argument.  RLK
+  */
+       estabur(0, ds, 0, 0, 0);
+ /*
+  *    estabur(0, ds, 0, 0);
+  */
159c171,179
-       estabur(u.u_tsize, u.u_csize, u.u_ssize, u.u_sep);
***
+ /*
+  *    Reset size of overlay, writable I-space.  RLK
+  */
+       u.u_osize = 0;
+       estabur(u.u_tsize, u.u_csize, u.u_ssize, u.u_sep,
+               u.u_osize);
+ /*
```

```
+   *     estabur(u.u_tsize, u.u_csize, u.u_ssize, u.u_sep);
+   */
381,382c401,414
-       n =+ USIZE+u.u_ssize;
-       if(estabur(u.u_tsize, u.u_dsize+d, u.u_ssize, u.u_sep))
***
+ /*
+   *     Include overlay segment in total size computation.  RLK
+   */
+       n =+ USIZE+u.u_ssize+u.u_osize;
+ /*
+   *     n =+ USIZE+u.u_ssize;
+   *
+   *     Include overlay, writable I-space in size checking.  RLK
+  .*/
+       if(estabur(u.u_tsize, u.u_dsize+d, u.u_ssize, u.u_sep,
+                u.u_osize))
+ /*
+   *     if(estabur(u.u_tsize, u.u_dsize+d, u.u_ssize, u.u_sep))
+   */
387c419,422
-       a = u.u_procp->p_addr + n - u.u_ssize;
***
+       a = u.u_procp->p_addr + n - u.u_ssize - u.u_osize;
+ /*
+   *     a = u.u_procp->p_addr + n - u.u_ssize;
+   */
389c424,430
-       n = u.u_ssize;
***
+ /*
+   *     Include size of overlay area when moving.  RLK
+   */
+       n = u.u_ssize + u.u_osize;
+ /*
+   *     n = u.u_ssize;
+   */
400c441,448
-       n = u.u_ssize;
***
+ /*
+   *     Include the overlay segment with the stack
+   *     when moving them up.  RLK
+   */
+       n = u.u_ssize + u.u_osize;
+ /*
+   *     n = u.u_ssize;
+   */
407a456,498     : Put at end
+ /*
+   *     OBREAK system call.
+   *     Expand the writable I-space, overlay area which follows
+   *     the stack segment.
+   *     R0 contains some address within the highest segment to be
```

```
+  *     allocated.
+  *
+  *     Added by RLK.
+  */
+ obreak()
+ {
+       register a, n, d;
+       a = u.u_ar0[R0];
+       n = ((a + 63) >> 6) & 01777;
+       if(n==0 && a<0) n = 02000;
+       n =- nseg(u.u_tsize) * 128;
+       if(n<0) n = 0;
+       d = n - u.u_osize;
+       if(estabur(u.u_tsize, u.u_dsize, u.u_ssize, u.u_sep, n))
+               return;
+       a = u.u_procp->p_size;
+       expand(a + d);
+       u.u_osize = n;
+       if(d>0) {
+               a =+ u.u_procp->p_addr;
+               while(d--) clearseg(a++);
+       }
+ }
+ /*
+  *     MFPIT System call.
+  *     The contents of the locations in USER I-space given by R0
+  *     are returned in R0.
+  *     Because the PDP-11/45 was designed to support execute
+  *     only code, USER I-space may not be read with MFPI by USER
+  *     programs, even though MTPI may write directly.
+  *     To overcome this annoyance, the system reads USER I-space
+  *     while in KERNEL mode.
+  *     Added by RLK.
+  */
+ mfpit()
+ {
+       u.u_ar0[R0] = fuiword(u.u_ar0[R0]);
+ }
```

Changes to the core dumping routines in "sig.c" cause any writable I-space area to be dumped following the normal dump. Thus system debugging software is not affected by the writable I-space extensions.

Figure 24 - Core dumping changes in "/usr/sys/ken/sig.c".

```
190c190,197
-       register s, *ip;
***
+ /*
+  * More variables needed with writable I-space overlay areas.
+  */
+       register s, *ip, i;
+       int a;
+ /*
+  *     register s, *ip;
+  */
213,214c220,232
-               s = u.u_procp->p_size - USIZE;
-               estabur(0, s, 0, 0);
***
+ /*
+  *     First write data and stack segments.  RLK
+  */
+               s = u.u_dsize + u.u_ssize;
+ /*
+  *             s = u.u_procp->p_size - USIZE;
+  *
+  *     Include the writable I-space in the area.  RLK
+  */
+               estabur(0, s, 0, u.u_sep, u.u_osize);
+ /*
+  *             estabur(0, s, 0, 0);
+  */
218a237,253     : After writei(ip);
+ /*
+  *     Copy any writable I-space onto the data and stack area,
+  *     then write it at end of file.  RLK
+  *     The following if statement was added.
+  */
+               if(a = u.u_osize) {
+                       i = u.u_procp->p_addr + USIZE;
+                       while(a--) {
+                               copyseg(i+s, i);
+                               i++;
+                       }
+                       i = u.u_osize;
+                       estabur(0, i, 0, 0, 0);
+                       u.u_base = 0;
+                       u.u_count = i * 64;
+                       writei(ip);
```

```
+               }
239c274,281
-       if(estabur(u.u_tsize, u.u_dsize, u.u_ssize+si, u.u_sep))
***
+ /*
+  *     Include writable I-space in size checking.  RLK
+  */
+       if(estabur(u.u_tsize, u.u_dsize, u.u_ssize+si, u.u_sep,
+               u.u_osize))
+ /*
+  *     if(estabur(u.u_tsize, u.u_dsize, u.u_ssize+si, u.u_sep))
+  */
243c285,291
-       for(i=u.u_ssize; i; i--) {
***
+ /*
+  *     Move both stack and writable I-space up.  RLK
+  */
+       for(i=u.u_ssize+u.u_osize; i; i--) {
+ /*
+  *     for(i=u.u_ssize; i; i--) {
+  */
```

Changes to "main.c" enable memory management hardware registers to be set up to handle the writable I-space allocated by "obreak".

Figure 25 - Change "/usr/sys/ken/main.c" to allocate I-space.

```
128c128,134
-               estabur(0, 1, 0, 0);
***
+ /*
+  *     Include null overlay segment size RLK
+  */
+               estabur(0, 1, 0, 0, 0);
+ /*
+  *             estabur(0, 1, 0, 0);
+  */
182c188,195
- estabur(nt, nd, ns, sep)
***
+ /*
+  *     The additional argument gives overlay segment size
+  *     for writable I-space.  RLK
+  */
+ estabur(nt, nd, ns, sep, no)
+ /*
+  * estabur(nt, nd, ns, sep)
+  */
189c202,208
-               if(nseg(nt) > 8 || nseg(nd)+nseg(ns) > 8)
```

```
***
+ /*
+  *     Include size of writable I-space.  RLK
+  */
+       if(nseg(nt) + nseg(no) > 8 || nseg(nd)+nseg(ns) > 8)
+ /*
+  *             if(nseg(nt) > 8 || nseg(nd)+nseg(ns) > 8)
+  */
192c211,217
-               if(nseg(nt)+nseg(nd)+nseg(ns) > 8)
***
+ /*
+  *     Include size of writable I-space.  RLK
+  */
+               if(no || nseg(nt)+nseg(nd)+nseg(ns) > 8)
+ /*
+  *             if(nseg(nt)+nseg(nd)+nseg(ns) > 8)
+  */
194c219,225
-       if(nt+nd+ns+USIZE > maxmem)
***
+ /*
+  *     Include size of writable I-space.  RLK
+  */
+       if(nt+nd+ns+no+USIZE > maxmem)
+ /*
+  *     if(nt+nd+ns+USIZE > maxmem)
+  */
209a241,260     : After   if(sep)
+ /*
+  *     Set up prototypes for writable I-space.  RLK
+  *     The following is added code.
+  */
+       {
+       a = nd+ns+USIZE;
+       while (no >= 128) {
+               *dp++ = (127 << 8) | RW;
+               *ap++ = a;
+               a =+ 128;
+               no =- 128;
+               }
+       /* Finish last partial segment of writable I-space */
+       if (no) {
+               *dp++ = ((no - 1) << 8) | RW;
+               *ap++ = a;
+       }
+ /*
+  *     end of added code.
+  */
213a265,268     : Just before   a = USIZE;
+ /*
+  *     This parenthesis added for balance.
+  */
+       }
```

Changes to "text.c" add an additional parameter to the "estabur", establish-user-register, function call.

Figure 26 - Fix "estabur" call in "/usr/sys/ken/text.c".

```
117c117,123
-       estabur(0, ts, 0, 0);
***
+ /*
+   *     Include argument for null writable I-space segment.  RLK
+   */
+       estabur(0, ts, 0, 0, 0);
+ /*
+   *     estabur(0, ts, 0, 0);
+   */
```

Changes to "sysent.c" add new entry point to the system call table for the "obreak", "mfpit", and "readnl" extensions.

Figure 27 - Extensions to "/usr/sys/ken/sysent.c" table.

```
40c40
-       0, &nosys,                      /* 27 = x */
***
+       0, &obreak,                     /* 27 = obreak */
46c46
-       0, &nosys,                      /* 33 = x */
***
+       0, &mfpit,                      /* 33 = mfpit */
66c66
-       0, &nosys,                      /* 53 = x */
***
+       2, &readnl,                     /* 53 = readnl */
```

Changes to "rdwri.c" implement the "readnl" system call extension.

Figure 28 - Extensions to "/usr/sys/ken/rdwri.c" for "readnl".

```
11a12,14        : After   #incluce "../systm.h"
+ /* Added macro call to support readnl. RLK */
+ #include "../file.h"
+
34a38,41        : After   if((ip->i_mode&IFMT) == IFCHR) {
+       /* Turn off the read-up-to new-line flag
+        * when using interruptable character devices.
+        * Line of code added by RLK */
+               u.u_nlflag = 0;
170a178,199     : After   cp = bp->b_addr + o;
+
+ /* When the flag is set by readnl, search the data about
+  * to be transferred for a new-line, line-feed character.
+  * If found adjust the byte counts requested
+  * to include nothing beyond the new-line character.
+  * "if" statement added by RLK */
+
+       if (u.u_nlflag) {
+               t = n;
+               while (t--) if (*cp++ == 012) {
+       /* the current number of bytes yet to transfer */
+                       n =- t;
+       /* Lower originally requested number of bytes */
+                       u.u_arg[1] =- u.u_count-n;
+       /* Lower total number of bytes yet to transfer */
+                       u.u_count = n;
+                       break;
+               }
+       /* Recalculate the transfer start address */
+               cp = bp->b_addr + o;
+       }
+
195a225,243     : Put at end
+
+ /*
+  * READNL is an added system call which acts like "read"
+  * except that it does not transfer any characters beyond
+  * the new-line, line-feed character on a single call
+  * to block devices.  This provides line at a time input.
+  * Function added by RLK.
+  */
+
+ readnl()
+ {
+       /* set flag used by iomove RLK */
+       u.u_nlflag++;
+       rdwr(FREAD);
```

```
+       /* Reset flag here.
+        * This flag manipulation depends on having
+        * uninterruptable calls to block devices */
+       u.u_nlflag = 0;
+ }
```

5.7. Alphabetical Function Syncpsis

ADD1 - increment argument.

(ADD1 X) - Adds 1 to the parameter X. If the parameter has floating-point type, ADD1 returns the same type. Otherwise, ADD1 returns an integer.

ALIST - return system Association LIST.

(ALIST) - Obtains the current system association list. The system association list starts at the *CAR of the function linker.

AMB - AMBiguity function.

(AMB X1 . . . Xn) - Returns a random selection from an arbitrarily long parameter list.

AND - evaluate arguments while true.

(AND EXP1 . . . EXPn) - Special form; sequentially evaluates its parameters until done or a parameter evaluates to NIL (false). AND returns the value of the last evaluated parameter.

APPEND - create a new list from argument lists.

(APPEND X Y) - Creates a new list by CONSing the members of the first list onto the second list. APPEND makes a copy of the first list while using the secord list as is. If the first parameter is NIL, APPEND returns the second parameter. If the second parameter is NIL, APPEND creates a copy of the first parameter.

ARRAY - create an ARRAY.

(ARRAY SIZE TYPE) - Creates a function that can access or alter the elements of an array of length SIZE. If the created function receives one parameter, a fixed-point number, the created function returns the array member indexed. If the created function receives a second parameter, whose type matches that of the array, the created function returns the second parameter and retains its value in the array member referenced by the first parameter, a fixed-point number. The parameters of ARRAY, SIZE and TYPE, should be fixed-poirt numbers. If ARRAY does not get the optional TYPE parameter, ARRAY produces an array of pointers by default.

TYPE	MEMBERS	RANGE
0	Pointer	Any S-expression
1	Logical	T (true) or NIL (false)
2	Binary	0 or 1
3	Signed byte	-128 to 127
4	Unsigned byte	0 to 255
5	16-bit word	-32768 to 32767
6	2-word single precision floating point	
7	4-word double precision floating point	

VLISP supports all of the floating point array types only if VLISP supports at least one type of floating point arithmetic.

ARRAYL - ARRAY Length predicate.

(ARRAYL ARR) - Returns the logical length of its parameter ARR, if it is an array. Otherwise the predicate returns NIL (false). The length of a logical or binary array (a bit array) is rounded up to the least multiple of 8 greater than or equal to the length given as the first parameter of ARRAY when creating the array access function.

ARRAYP - ARRAY type Predicate.

(ARRAYP ARR) - Returns the type, an integer, of its parameter ARR, if it is an array. Otherwise the ARRAYP predicate returns NIL (false).

ASSOC - search an ASSOCiation list.

(ASSOC ITEM LST COUNT) - Returns the COUNTth occurrence in the second parameter, LST, a list, of a CONSed pair whose CAR in EQUAL to the first parameter, ITEM. If the third parameter, COUNT, the count, is omitted, 1 is used.

ATOM - ATOM predicate.

(ATOM X) - Returns T (true) if the parameter X has an atomic type. Otherwise, ATOM returns NIL (false) when the parameter is a CONSed node.

ATSYMB - create ATomic SYMEol.

(ATSYMB X) - Finds or creates, if needed, an atomic symbol specified by the parameter X. If the parameter is a SYMBOLic atom, ATSYMB returns it, regardless of whether or not the hash table, OBLIST, contains it. Otherwise, ATSYMB converts the parameter to internal type STRING and searches for an atom with this print name in the hash lists, OBLIST, used by the READ and TOKEN routines. After not finding an atom with the print name, ATSYMB creates one and enters it into the hash tables.

ATTEMPT - catch errors after ATTEMPTing evaluation.

```
(ATTEMPT EXP
         [N1 E1-1 . . . E1-n]
                . . .
         [Nm Em-1 . . . Em-nm]
         ) -
```

Special form; evaluates the first parameter, EXP, and returns its value if no system errors have occurred. However, if an error does occur while evaluating EXP, the interpreter examines the other arguments for a list whose CAR is a number whose value matches the error type. If the interpreter finds a match, ATTEMPT evaluates the remaining expressions in the CDR of the list. ATTEMPT returns the value of the last expression evaluated.

BACKSP - BACKSPace the READ routine buffer pointer.

(BACKSP) - Returns T (true) and prepares the READ, TOKEN, or READCH function to read the previous character in the READ buffer if the buffer pointer was not set to read the first character. Otherwise BACKSP returns NIL (false) and leaves the buffer pointer where it was.

BREAK - intercept functions before application.

(BREAK ATM NEWFN) - Functional; uses the second parameter, NEWFN, a function, in place of the function or special form that is constantly (globally) bound to the first parameter, ATM, a SYMBOLic atom. When called, the new function, NEWFN, bound to ATM receives at least two parameters. The first parameter is the atom ATM whose binding BREAK altered. The second parameter is the function LINKER bound originally to the atom ATM. All other parameters follow as they would have been passed to the original unbroken function. If BREAK acts on an atom bound to a special form, the third parameter passed to the new, intercepting function, NEWFN, is a list of the unevaluated parameters intended for the original special form.

C . . . R - find CARs and CDRs.

(C . . . A . . . D . . . R ARG) - Returns the pointer derived by recursively taking CARs and CDRs of the parameter ARG, a CONSED node, a dotted pair. The CAR of a dotted pair is the first part as printed, the lefthand side. The CDR of a dotted pair is the second part as printed, that which follows a dot in a simple dotted pair, the righthand side. For example, if ARG were the dotted pair

(X . Y)

then

(CAR ARG) = (CAR '(X . Y)) = X

and

(CDR ARG) = (CDR '(X . Y)) = Y .

The input routines READ and TOKEN bind any atom whose name consists of the character, "C", followed by an arbitrary number of "A"s and "D"s, ended by an "R" to a composition function of CARs and CDRs. The order of evaluation is from right to left. For example, evaluating the S-expression

(CADADR EXP)

is equivalent to evaluating

(CAR (CDR (CAR (CDR EXP)))) .

If the interpreter attempts to follow the pointer into an atomic object, anything but a dotted pair, while evaluating a CAR-CDR-chain function call, the interpreter prints a warning message.

CLEARBUFF - reset input BUFfer for new line.

(CLEARBUFF FILE) - Resets the irput buffer to input a new line on the next call to READ, TOKEN, or READCH. If CLEARBUFF gets the optional parameter FILE, a fixed-point number, subsequent input will come from the logical file number specified. The operating system must have previously provided an external meaning to the internal, logical file-number created by OPEN or other system calls such as PIPE under UNIX. A NIL parameter returns input to the standard, default file. CLEARBUFF saves the parameter as the constant (global) binding of the atom *CLEARBUFF.

CLOSE - CLOSE logical file number.

(CLOSE FILE) - Closes the interral, fixed-point, logical file-number specified by its parameter FILE. If CLOSE get a NIL parameter, the operating system closes the standard, default input. If successful, CLOSE returns NIL. Otherwise, CLOSE returns the integer error number returned by the UNIX operating system in R0.

COMPLEMENT - logical one's COMPLEMENT negation.

(COMPLEMENT X) - Computes the octal, logical one's complement of the parameter X. COMPLEMENT uses the high-order, most-significant word of a floating point parameter as a fixed-point value.

COMPRESS - COMPRESS list irto a noce.

(COMPRESS LST) - Uses the TOKEN routines (the scanner) to convert a list of single character elements into an appropriate atomic node. The list elements may be either single character atoms, single character strings, or fixed point numerical ASCII values. The TOKEN routine examines the syntax of the character to determine the type of node to create. User defined readmacro characters cause no special actions. Thus readmacros may use COMPRESS without escaping the user readmacro characters in the input list.

COND - CONDitionally evaluate arguments.

```
(COND [EXP1 E1-1 . . . E1-n1]
                . . .
      [EXPm Em-1 . . . Em-nm]
       ) -
```

Special form; expects parameters that are lists of at least one expression. At least one parameter, a list, must be given. COND evaluates the CAR of each list until one returns a non-NIL (true) value or until it has evaluated the CAR of every parameter. At the first instance of a true value, COND sequentially evaluates the remaining expressions, if any, in the CDR of the current parameter, a list. COND returns the value of the last expression evaluated.

CONS - create a CONSolidated pair.

(CONS X Y) - Creates a new CONSED node of two pointers, its CAR, X, and its CDR, Y, the left and right hand sides respectively, called a dotted pair.

(X . Y)

CSET - create a new Constant (global) binding.

(CSET ATM EXP) - Function; creates or replaces a constant binding on the first parameter, a SYMBOLic-atom variable, ATM, using the second parameter, EXP. Any old constant (global) binding disappears. Any fluid bindings on the system association list, ALIST, become hidden, since the interpreter checks the constant binding cell (*CAR ATM) before searching the ALIST for fluid bindings.

CSETQ - Quote the first argument to CSET.

(CSETQ NAME EXP) - Special form; serves as an abbreviation for

(CSET 'NAME EXP)

since CSETQ only evaluates the second parameter to change the constant binding of the first parameter, NAME, as given. Thus, unlike CSET, in this example the binding of NAME itself would be changed, instead of any variable that might have been bound to NAME.

CURRCOL - determine CURRent COLumn in output buffer.

(CURRCOL) - Returns an integer that represents the next column in the output composition buffer that will receive a character.

DEFINE - establish a list of constant bindings.

(DEFINE LST) - Applies CSETQ to each sublist of the parameter LST, a list, constantly binding the first member of each sublist, a SYMBOLic-atom variable, to the value of the second member of the sublist. DEFINE creates a list of the variables that received bindings.

DEFMAC - DEFine a MACro special form.

(DEFMAC NAME FUNC) - Special form; constantly (globally) binds a macro special form creates from the evaluated second parameter, FUNC, a function LINKER, to the unevaluated variable which is the first parameter, NAME. The created macro passes the unevaluated parameters received, if any, to the original function LINKER, FUNC. The interpreter then evaluates the results. Thus the created MACRO can pre-process unevaluated parameters into a new S-expression that the interpreter finally evaluates.

DEFSPEC - DEFine SPECial form.

(DEFSPEC NAME FUNC) - Special form; constantly (globally) binds a special form either given as the constant binding of a variable, the second parameter, FUNC, or else creates from the evaluated second parameter, FUNC, a function LINKER to the unevaluated first parameter, NAME. If the unevaluated second parameter was already bound to a special form or MACRO, DEFSPEC performs a renaming so that the first parameter variable will have the same meaning as the second parameter. Otherwise, the created special form returned as the value of the second argument of DEFSPEC will pass any parameters it receives to the function, FUNC, unevaluated by the interpreter.

DELIM - specify input scanner DELIMiters.

(DELIM STR FLG) - Converts the first parameter, STR, into a single character of internal type STRING. If the optional second parameter, FLG, is given, the character specified by STR has its delimiter status changed. A NIL (false) second parameter removes delimiter status. Any other second parameter (true) turns delimiter status on. Regardless of the presence of the second parameter, DELIM returns the previous delimiter status of the character, T (true) for on and NIL (false) for off. READ and TOKEN use delimiter characters to terminate the input scanner's creation of a name. A SYMBOLic atom name being created does not include a non-escaped delimiter character unless no previous characters have been read. If a delimiter character is not also a readmacro character, the scanner will return it as a single character SYMBOLic atom when encounterec initially. The string scanner and READCH ignore delimiter status.

DIFFERENCE - compute DIFFERENCE of arguments.

(DIFFERENCE X Y) - Subtracts the second parameter, Y, from the first parameter, X. The result uses the greater precision of its two parameters. DIFFERENCE returns integer fixed-point results if neither parameter has floating type. The single-character-atom "-" is a synonym for DIFFERENCE.

DO - unconditional evaluation special form.

(DO EXP1 . . . EXPn) - Special form; sequentially evaluates its arguments. DO returns the value of the last parameter evaluation and discards all other results of evaluation.

DOUBLE - convert to DOUBLE precision floating.

(DOUBLE ARG) - Converts the parameter ARG into a double-precision floating-point value (four 16-bit words). VLISP defines DOUBLE only when supporting both double and single precision floating-point-number types.

DUMP - output compiled code and pointers.

(DUMP LINK FILE FUNC) - Returns NIL if the first parameter, LINK, is not a master LINKER whose I-space address (*CDR LINK) points to the beginning of a compiled code area. Otherwise, DUMP sends a binary image of the compiled code area to the internal, logical file-number specified by the second fixed-point-number parameter, FILE. VOS and the DOS emulator of VOS VLISP use DEC absolute loader format. UNIX VLISP uses a.out load module format. DUMP performs an implicit *BEGIN to compact the code area. If FILE is NIL, DUMP inhibits the binary output phase. Next, if DUMP receives the optional third parameter, FUNC, a function of three arguments, DUMP applies FUNC to each pointer offset at the end of the compiled code area. The first argument of FUNC gets the octal offset from the start of code of the specified pointer. The second gets that integer which must be added to a normal pointer to produce the pointer giver in the code. The third gets the normal pointer derived from the given pointer. Lastly, DUMP returns the master LINKER given as the first parameter, LINK.

ENTIER - round down to next whole integer.

(ENTIER X) - Returns an integer node whose value is the greatest whole, signed number less than or equal to the parameter X after performing any conversion needec for a floating parameter. If the converted value cannot be represented by a signed, 16-bit integer, ENTIER returns integer zero. VLISP defines ENTIER only if supporting floating point.

EQ - test pointer EQuality.

(EQ X Y) - Returns T (true) if the two parameters are the same, otherwise NIL (false). EQ returns NIL when comparing two different nodes even though they may have the same value.

EQUAL - test arguments for congruerce.

(EQUAL X Y) - Returns NIL (false) only if its two parameters, X and Y, cannot be made congruent; otherwise, EQUAL returns T (true). EQUAL converts two numerical parameters to the type of greatest common precision befcre testing for equality of value. It tests two string parameters character by character. EQUAL recursively descends two CONSED ncde parameters to see if both the CARs and CDRs are also EQUAL. The right recursive descent may loop if presented with two corgruent, circular lists. EQUAL tests other node types for pointer equality.

ERASE - remove atom constant binding and property list.

(ERASE LST) - Expects a parameter which is a list of SYMBOLic atoms. ERASE sets the constart (global) binding cell (*CAR) of each atom to its undefined state and sets each atom's property list to NIL, the initial state. Any hidden bindings to these atoms on the system association list, ALIST, will reappear since the constant binding cell is undefined. Since user readmacro definitions are kept on the property list of the associated-single-character atom, ERASE will remove such readmacros as a side-effect.

ERROR - generate LISP internal ERROR condition.

(ERROR NUM) - Simulates the LISP internal ERROR condition given by the fixed-point-number parameter NUM. If NUM is omitted, ERROR produces a type 0 error. Some non-positive error numbers have special meanings to the VLISP interpreter. These reserved error types will cause specialized actions if not caught by a previously invoked ATTEMPT:

0 - System errors
-1 - RETURN value
-2 - GO label
-3 - Asychronous interrupts
-4 - Previous error type was not caught
-8 - Unbound variable or pointer array element
-9 - Bad array index
-10 - Floating point exceptions
-11 - End of file.

During the initial file loading of the UNIX VLISP startup procedure, most uncaught errors cause a premature error termination (IOT) of the UNIX VLISP interpreter.

EVAL - interpret argument.

(EVAL ARG) - Calls the VLISP interpreter to evaluate the parameter ARG. If ARG is an atcm with internal type SYMBOL, EVAL first checks tne constant bindirg cell (*CAR). Upon finding a non-zero pointer to a node, a cefined reference, in the constant binding cell, EVAL returns this pointer as the value. If the SYMBOLic atom has no constant binding, EVAL searches the system association list, ALIST, for a binding CONSED pair whose CAR is the atom and whose CDR is the value of the ATOM, a non-zero pointer. If EVAL still cannot find a value for the atom, VLISP prints a warning message and queries the user for a value to use. If the parameter ARG of EVAL is a CONSED node, EVAL assumes the parameter heads a list. EVAL checks if the CAR of the list is a SYMBOLic atom which is constantly bound to a special form LINKER by comparing the I-space address of any such LINKER (*CDR) with the I-space address of EVAL. If the unsigned, I-space address of the LINKER is less than that of EVAL, the LINKER specifies a special form. Upon finding that the CAR of the parameter ARG, in this case a list, is constantly bound to a special form, EVAL calls the special form using the remaining members of the parameter (CDR ARG) as parameters tc the special form without further evaluation. However, if EVAL finds that the parameter ARG, a list, is not a special form call, EVAL recursively evaluates the first member (CAR) of the parameter, a list, and checks that the returned value is a functior LINKER. If not, EVAL prints a warning message and queries the user for a new function LINKER to use. by recursion, EVAL evaluates any remaining members of its parameter (CDR ARG) and passes their values as parameters to a call of the previously obtained function. EVAL returns all other types of nodes, namely LINKERs, numbers, and STRINGs, used as the parameter ARG, without further evaluation.

EXEC - EXECute program in place of intepreter.

(EXEC ARG0 . . . ARGn) - Calls STRING to convert all of its parameters, including lists of characters, into strings followed by a zero byte, the format used by the UNIX sys EXEC call. EXEC creates an integer array of poirters to the start of data of each string, which it passed as the second parameter of the system call. The first parameter is also passed as the file name to the sys call. If the call returns, EXEC returns the integer error number. No LISP system error is generated. Only UNIX VLISP defines EXEC.

EXPLODE - create list from print name.

(EXPLODE ARG) - Uses the PRIN1 cutput routines to create a list of single-character, SYMBOLic atoms that represent the characters that PRIN1 would use to print the parameter ARG.

EXPLODE2 - create list from print name with escapes.

(EXPLODE2 ARG) - Uses the PRIN2 output routines to create a list of single-character atoms that represent the characters that PRIN2 would use to print the parameter ARG in a format with escapes so that the READ or TOKEN routines could recreate the printed object.

FIXP - FIXed-point Predicate.

(FIXP X) - Returns T (true) if the parameter X is a fixed point number (octal or integer); otherwise, NIL (false). VLISP defines FIXP only if supporting floating point.

FLAG - put FLAG on atom property list.

(FLAG ATM FLG) - Puts the flag given by the second parameter, FLG, a SYMBOLic atom, on the property list (*CDR) of the first parameter, ATM, another SYMBOLic atom.

FLOAT - convert to a FLOATing type.

(FLOAT X) - Returns a floating-point number by converting the parameter to floating-point type, using single precision if available, otherwise double precision. If the parameter has floating type, FLOAT returns it as is. VLISP only defines FLOAT if supporting floating point.

FLOATP - FLOATing-point Predicate.

(FLOATP X) - Returns T (true) if the parameter X has floating-point type; otherwise, NIL (false). VLISP only defines FLOATP if supporting floating point.

FORK - spawn a child process.

(FORK) - Creates a child process, a copy of the current process, by calling the UNIX operating system, returns the integer process identification (PID) of the child process to the parent process, and returns NIL (false) to the child process. Only UNIX VLISP defines the FORK predicate.

FUNCTION - create function that captures the ALIST.

(FUNCTION FUNC) - Creates a new function from its parameter FUNC, which captures the current system association list, ALIST, that maintains the status of fluid binding pairs. When this new function is invoked, the captured ALIST, which contains the binding environment during the creation of the function, is temporarily re-established for the duration of the function call. The created function then calls the old function parameter FUNC in this new environment with the parameters passed to the created function.

GENSYM - GENerate a temporary atomic SYMbol.

(GENSYM ATM) - Creates a new atomic symbol that is not on the hash lists, the OBLIST. If the caller provides the parameter ATM, its print name is used as the print name of the newly created symbol. If the caller provides no parameter, GENSYM uses the SYMBOLic atom G. The created atom will be different from any previous atom. When the atom is printed, its print name will be followed by a colon (:) and a unique integer. Since the atom is not on the hash lists, READ and TOKEN cannot directly access the name, even when its name, as printed, is input. Instead an atom will be created on the hash lists, OBLIST, for the input name. Unlike atoms on the hash lists, when an atom created by GENSYM is no longer explicitly referenced, its space may be reclaimed.

GET - obtain property from atom property list.

(GET ATM PRP) - Obtains the property specified by the second parameter, PRP, a SYMBOLic atom, from the property list of the first parameter, ATM, another atom. If ATM is not typed SYMBOL or CONSED, or if the property list (*CDR) of ATM does not contain the property given by PRP, GET returns NIL.

GO - GO to PROG label.

(GO LABEL) - Special form; continues evaluation with the next expression following the given label, LABEL, in the most recent PROG. If the most recent PROG does not use the GO parameter LABEL as a label, the interpreter recursively searches in the next most recent PROG for LAEEL until reaching a level of LISP supervision. If a level of LISP supervision intercepts the label search, the interpreter prints an error message and restarts the LISP supervisor.

GREATERP - GREATER than Predicate.

(GREATERP X Y) - Returns T (true) if the first parameter, X, is greater than the second parameter, Y. Otherwise, GREATERP returns NIL (false). The comparison is signed, i.e. positive values are greater than negative ones. In interpreters supporting floating point, if the type of the parameters differ and at least one parameter has floating type, GREATERP converts the parameter of lesser precision to the type of the parameter with greater precision before making a comparison.

IFFLAG - FLAG existence predicate.

(IFFLAG ATM FLG) - Returns T (true) if the property list (*CDR) of the first parameter, ATM, a SYMBOLic atom, contains the flag given by the second parameter, FLG, another SYMBOLic atom, as a member. Otherwise, IFFLAG returns NIL (false).

IFTYPE - 1100 LISP internal node TYPE predicate.

(IFTYPE NODE TYPE) - Returns T (true) if the type of the first parameter, NODE, has an internal type that corresponds to the Wisconsin UNIVAC 1100 LISP internal type specified by the second parameter, TYPE, a fixed-poirt number. The 1100 LISP internal types used by IFTYPE differ from the internal types used by VLISP.

1100 type	Internal name	VLISP type
0	CONSED	0
1	INTGER	010
2	OCTAL	6
3	SINGLE	012
4	SYSTEM	-2, -4, and -6
5	Compiled	Not ir data space
6	LINKER	2
7	SYMBOL	4
8	STRING	012+[2* number of floating types]
(5)	DOUBLE	012+[2 if SINGLE used, else 0]

INDEX - recursively apply function to CARs.

(INDEX LST END FUNC) - Functional; applies the third parameter, FUNC, a function of two arguments, recursively to each element of the first parameter, LST, a list, and the value of subsequent calls to the remaining members of LST. When applying FUNC to the last element of the parameter LST, INDEX passes the second parameter, END, as the second parameter to FUNC. Thus if

LST = (X1 X2 . . . Xn)

then the call is equivalent to

(FUNC 'X1 (FUNC 'X2 . . . (FUNC 'Xn END) . . .))

INTO - list of values of function application to CARs.

(INTO LST FUNC) - Functional; creates a list of the values resulting from applying the second parameter, FUNC, a function of one argument, to each member, successive CAR, of the first parameter, LST, a list. MAPCAR is a synonym.

LAMBDA - create function.

(LAMBDA ARG-LIST EXP1 . . . EXPr) - Special form; creates a function that uses the first parameter, ARG-LIST, a list of arguments, as arguments of the created function. The argument list need not be a true list since the rightmost CDR of the list need not be NIL, which ordinarily specifies the end of a list. The members of the argument list that will act as variables must be atoms with internal type SYMBOL. When called, the created function binds its arguments, the members of ARG-LIST, to the values passed as parameters in the function call by adding CONSED node pairs to the beginning of the system association list, ALIST. The CAR of these binding pairs consists of the argument name as given by a member of ARG-LIST, and the CDR consists of the respective value passed as a parameter to the function call. The new binding obscures any previous binding on the ALIST with the same variable name for the duration of the function evaluation. If the end of ARG-LIST, the rightmost CDR, is NIL, i.e. the list has the form

ARG-LIST = (X1 X2 . . . Xn)

then the number of parameters passed to the function must be the same as the number of arguments given in ARG-LIST. If the argument list is NIL, the degenerate case, then calls may pass no parameters to the created function. If the end of ARG-LIST, i.e. rightmost CDR, is not NIL, then it must be a SYMBOLic atom to which the created function binds a list of any parameters passed which remain after the created function has bound the other variables. For example, if

ARG-LIST = (X Y . Z)

during a function call, the created function would bind X and Y to the first two parameters of the function call, create a list of any remaining parameters, and then bind that list to the last "list" variable, Z. Calls to the created function must provide sufficient parameters for each variable exclusive of any "list" variable. If ARG-LIST consists of a single SYMBOLic-atom variable, the degenerate case of "list" variables, for example, if ARG-LIST is the SYMBOLic atom Z, then during each call, the created function makes a list of any parameters passed and binds that list to the solitary "list" variable Z. After the created function has bound any parameters passed to its variables, the function evaluates the other parameters of LAMBDA sequentially in the new binding environmert. After evaluating the last expression, the created function restores the system association list, ALIST, to its state at function entry, thus restoring the original binding environment with any previously obscured bindings. The created function returns the value of the last expression evaluated.

LAMDA - apply FUNCTION to LAMbDA expression.

(LAMDA ARG-LIST EXP1 . . . EXPn) - Special form; serves as a shorthand for the function FUNCTION applied to the LAMBDA expression specified by the parameters of LAMDA. When called, the function created by LAMDA installs the binding environment captured when LAMDA was evaluated, binds any variables to the parameters of the created function call, evaluates the remaining expressions of LAMDA, reinstates the original binding environment in effect before the created function call, and returns the value of the last expression evaluated.

LEFTSHIFT - SHIFT LEFT for positive counts.

(LEFTSHIFT X COUNT) - Returns the two's complement arithmetically-shifted octal representation of the first parameter, X, a fixed-point number, using the second parameter, COUNT, a signed, fixed-point number. If COUNT is positive, LEFTSHIFT performs a left arithmetic shift with zero fill entering from the right into the least significant bits. If the second parameter is negative, the first parameter, X, is right circularly shifted as a 16-bit value. Otherwise, given a zero count, LEFTSHIFT creates an octal node of the first parameter value. LEFTSHIFT uses the most significant word of floating-point parameters as is without converting to fixed-point-number type.

LENGTH - count LENGTH of list.

(LENGTH LST) - Returns an integer count of the number of members, CARs, of the parameter LST. LENGTH repetitively performs *CDRs on the parameter LST until NIL is found, which represents the end of a list in correct format, or until the count overflows, which produces a system error condition.

LESSP - LESS than Predicate.

(LESSP X Y) - Returns T (true) if the first parameter, X, is less than the second parameter, Y; otherwise, LESSP returns NIL (false). The comparison is signed, i.e. negative values are less than positive ones. In VLISP interpreters supporting floating point, if either one of the parameters has floating-point type, LESSP converts the parameters to the type with greater precision before making a comparison.

LISP - LISP supervisor.

(LISP READ-FUNC) - Iteratively prints the results of evaluating the expression obtained by its parameter READ-FUNC, a function of no arguments. The LISP supervisor prefixes the returned value with

Value:

except under UNIX in child processes of the original LISP invocation or if LISP is invoked with "-" as a parameter of the call from the shell. If the (LISP) call does not supply a parameter READ-FUNC, the interpreter supplies a default S-expression-READing function. At each call of the default READing function by the LISP supervisor, the default function resets the input buffer, resets to use the standard input, resets to use the standard output, sends a prompt for the user on the standard output saying

Eval:

and calls READ to obtain the next S-expression from the standard input as the value of the default-READing-function call. In those cases in which the LISP supervisor does not use the "Value:" prefix, the default READing function does not print the "Eval:" prompt either. The LISP supervision handles any errors that are not caught by ATTEMPT-special-form calls by printing a warning message and restarting the READing, EVALing, and value-printing sequence. The supervisor may be exited by using the RETURN function or by providing an end of file condition on the standard input with UNIX EOT (control/D). The LISP supervisor call returns any value of the RETURN function or NIL if no value is provided. At the end of the start-up procedure, the VLISP interpreter invokes a level of the LISP supervisor with the default expression-obtaining function which converses with the user. The VLISP interpreter prints any non-NIL and non-fixed-point-numeric value RETURNed by the top level of LISP supervision and uses the value as the UNIX exit status with NIL converted to zero.

LIST - create a LIST from arguments.

(LIST ARG1 . . . ARGn) - Creates a list from any parameters. If the call provides no parameters, LIST returns the empty list, NIL. For example, evaluating

(LIST 'W 'X 'Y 'Z)

produces the list

(W X Y Z)

which is a shorthand used by the interpreter for the CONSED, dotted-pair expression,

(W . (X . (Y . (Z . NIL)))) .

LOAD - LOAD definitions from file.

(LOAD ASCII-FILE BINARY-FILE) - Repetitively reads and evaluates S-expressions from the the file specified by the first parameter, ASCII-FILE, a fixed-point logical file number, until reaching end of file or evaluating a RETURN function call. LOAD closes the file if under UNIX and returns the logical file number used. Under UNIX, the first parameter may alternatively specify an external file name, which the interpreter will open. Under UNIX, if the LISP interpreter invocation provides file names, the interpreter LOADs them before invcking a level of LISP supervision. A parameter "+" must then be explicitly used to produce a sign-on line and invoke the LISP supervisor. The parameter "-" could also be used to invoke the LISP supervisor without prompts under UNIX. If the call provides the second parameter, BINARY-FILE, an internal, fixed-point, logical rame, LOAD saves the parameter as the constant binding of the atom *LOAD for use with the next *DEPOSIT call. Otherwise, LOAD constantly binds *LOAD with the logical file number computed for ASCII-FILE. The binary file number specifies a file containing binary machine code in load module format to be installed in I-space later by *DEPOSIT.

LOGAND - bitwise LOGical AND.

(LOGAND ARG1 . . . ARGn) - Returns a 16-bit octal representation of the bitwise logical AND of any parameters. LOGAND uses the high-order, most-significant word of floating-point-number parameters as a fixed-point value. If the call to LOGAND provides no parameters, LOGAND returns octal negative 1, 177777Q, all bits on (true).

LOGOR - bitwise LOGical OR.

(LOGOR ARG1 . . . ARGn) - Returrs a 16-bit octal representation of the bitwise logical OR cf any parameters. LOGOR uses the high-order, most-significant word of floating-point-number parameters as a fixed-point value. If the call to LOGOR provides no parameters, LOGOR returns octal zero, 0Q, all bits off (false).

LOGXOR - bitwise LOGical eXclusive OR.

(LOGXOR ARG1 . . . ARGn) - Returns a 16-bit octal representation of the bitwise logical exclusive OR cf any parameters. LOGXOR uses the high-order, most-significant word of floating-point-number parameters as a fixed-pcint value. If the call to LOGXOR provides no parameters, LOGXOR returns octal zero, 0Q, all bits off (false).

MANIFEST - signal compile time computation.

(MANIFEST ARG) - When interpreted returns the parameter ARG as value. VLISP defines MANIFEST for use with potentially compileable functions to signal to the compiler that the parameter is to be evaluated at compile time instead of being evaluated by the compiled code.

MAP - apply function to each final segment.

(MAP LST FUNC) - Functional; applies the second parameter, FUNC, a function of one argument, to each final segment of the first parameter, LST, a list. The final segments are the successive, non-NIL CDRs of a list. Thus if LST is NIL, the degenerate case, it has no final segments to which to apply to FUNC. MAP always returns NIL.

MAPC - apply function to all members of a list.

(MAPC LST FUNC) - Functional; applies the second parameter, FUNC, a function of one argument, to each member (CAR) of the first parameter, LST, a list. MAPC always returns NIL.

MAPCAR - synonym for INTO.

(MAPCAR LST FUNC) - Functional; performs the same as INTO.

MAPLIST - synonym for ONTO.

(MAPLIST LST FUNC) - Functional; performs the same as ONTO.

MEMBER - MEMBER of list predicate.

(MEMBER ITEM LST) - Searches the second parameter, LST, a list, for the first congruent occurrence of the first parameter, ITEM, using EQUAL to test for congruerce. If found, MEMBER returns the first final segment of LST whose CAR coincides with ITEM. Otherwise, MEMbER returns NIL (false).

MINUS - arithmetic negation.

(MINUS X) - Returns the signed magnitude negation with the same type as a floating-point parameter X; otherwise, the integer, two's-complement, arithmetic negation of the parameter X.

MINUSP - negative number Predicate.

(MINUSP X) - Returns T (true) if the high-order, sign bit of its numerical parameter X is on, i.e. the parameter is negative; otherwise, NIL (false).

NCONC - CONCatenate two lists.

(NCONC X Y) - Returns the concatenation of the two parameters, X and Y, lists, formed by altering the end (rightmost CDR) of the first parameter, X, so that the end becomes the second parameter, Y. If either parameter is NIL, NCONC returns the other.

NOT - logical NOT predicate.

(NOT ARG) - Returns T (true) if the parameter ARG is NIL (false); otherwise, NIL. NULL is a synorym.

NTH - count to the NTH final segment.

(NTH LST COUNT) - Returns the final segment, CDR, of the first parameter, LST, a list, specified by the second parameter, COUNT, a fixed-point number. If COUNT is positive, NTH counts from the left (the head) of the list. If COUNT is negative, NTH counts from the right (the tail) of the list. Otherwise, if COUNT is zero, NTH returns LST as is. If the absolute value of COUNT exceeds the length of LST, NTH returns NIL.

NULL - NULL argument predicate.

(NULL ARG) - Returns T (true) if the parameter ARG is NIL (false); otherwise, NIL. NOT is a synonym.

NUMBERP - NUMBER type Predicate.

(NUMBERP X) - Returns T (true) if the parameter X has a numeric internal type, octal, integer, or floating-point. Otherwise, NUMBERP returns NIL (false).

OBLIST - apply function to members of the OBject LIST.

(OBLIST FUNC) - Functional; applies the parameter FUNC, a function of one argument, to each SYMBOLic atom which is on the hash lists used by the READ and TOKEN routines. OBLIST returns NIL as its value. If the call omits the parameter, OBLIST uses a default function that prints each SYMBOLic atom on the current output starting each bucket, the object list divisions that the hash values reference, on a new line.

ONDEX - recursively apply function to CDRs.

(ONDEX LST END FUNC) - Functional; applies the third parameter, FUNC, a function of two arguments, recursively to each final segment, CDR, of the first parameter, LST, a list. When applying FUNC to the last final segment, penultimate CDR, of LST, INDEX uses the second parameter, END, as the second parameter of FUNC. Thus if

LST = (X1 X2 . . . Xn)

the call is equivalent to

(FUNC LST (FUNC (CDR LST) . . . (FUNC '(Xn) END) . . .)) .

ONTO - list of values of function application to list CDRs.

(ONTO LST FUNC) - Functional; creates a list of the values resulting from applying the second parameter, FUNC, a function of one argument, to each final segment (successive non-NIL CDR) of the first parameter, LST, a list. MAPLIST is a synonym.

OPEN - prepare to use external file.

(OPEN ARG MODE NUM) - Returns an integer that can be used internally by CLEARBUFF, TERPRI, LOAD, and CLOSE to specify the external name given by the first parameter, ARG, a string or SYMBOLic atom. The optional second parameter, MODE, a number, specific to the host operating system, is zero if not given or if NIL. The optional third parameter, NUM, used only with the DOS operating system, forces DOS to return that integer as the logical file number. An unknown external name (first parameter) causes a system error.

OR - evaluate arguments until true.

(OR EXP1 . . . EXPn) - Special form; sequentially evaluates its parameters until a parameter evaluates non-NIL (true) or no unevaluated parameters remain. OR returns the value of the last evaluated parameter.

PIPE - create UNIX PIPE.

(PIPE) - Returns a CONSED node (dotted pair) of two integers whose CAR and CDR (left and right) specify read and write logical, internal file-numbers used by CLEARBUFF and TERPRI, respectively, to communicate arbitrarily among the future offspring of the current process and itself. Only UNIX VLISP defines PIPE.

PLENGTH - Print LENGTH court.

(PLENGTH ARG) - Returns an integer that represents the number of characters which would be used by PRIN1 to print the parameter ARG without escapes or line feeds.

PLENGTH2 - Print LENGTH with escapes.

(PLENGTH2 ARG) - Returns an integer that represents the number of characters which would be used by PRIN2 to print the parameter ARG without line feeds but with any escapes which would be needed by READ to re-read the output of PRIN2 as input.

PLIMIT - manipulate Print routine LIMITs.

(PLIMIT ARG) - Returns a CONSed node (dotted pair) of integers that represent the maximum print depth and length limits of lists. When passed, the optional parameter ARG, a dotted pair of integers in the same format as that returned, changes the respective print limits. While composing output, the print routines use ampersand (&) in place of sublists which exceed the depth limit and use two hyphens (--) in place of the CDRs of sublists which exceed the length limit.

PLUS - sum parameters.

(PLUS ARG1 . . . ARGn) - Sums the parameters from left to right, converting either the next parameter or the current subtotal to the type of the one with higher precision if either has floating-point type. PLUS does not check for addition overflow when adding two fixed-point values. The value returned has the type of the highest precision parameter used. If all of the parameters have fixed-point type, PLUS returns an integer total. If the PLUS call gives no parameters, PLUS returns integer zero (0), the empty total. The single-character-atom "+" is a synonym for PLUS.

PRINT - compose object for PRINTing and send.

(PRINT ARG) - Composes an external representation of the parameter ARG in the output buffer; sends the entire contents of the output buffer to the current output logical-file given by *TERPRI; and prepares the output buffer to compose a new line of output. Whenever PRINT fills the output buffer, it sends the buffer and continues composition at the beginning of a new line.

PRIN1 - compose object for PRINting.

(PRIN1 ARG COL) - Composes the external representation of the first parameter, ARG, in the output buffer starting at the column given by the optional second parameter, COL, a fixed-point number. Skipped columns that have not previously received a character contain blanks (ASCII spaces). PRIN1 replaces an omitted second parameter, COL, with the current output column. It sends the contents of the output buffer to the current output file when the length of the external representation requires positions beyond the end of the output buffer and continues composition at the beginning of an empty buffer.

PRIN2 - compose re-readable output.

(PRIN2 ARG COL) - Composes an external representation of the first parameter, ARG, in the output buffer in a format that READ could use to reconstruct a congruent object. PRIN2 places the most recently defined escape character, which is initially exclamation point (!), before characters with readmacro or delimiter status used in SYMBOL atom print names and before SYMBOLic atom names whose first character is a number (0-9). It surrounds strings with the most recently defined string delimiter character, which is initially couble-quotes ("), and doubles any instance of a string delimiter character within strings. It starts composing in the columr specified by the optional second parameter, COL, a fixed-point number, in lieu of composing into the next available column. Skipped columns that have not previously received a character contain blanks (ASCII space). PRIN2 sends the contents of the output buffer to the current output file when the length of the external representation requires positions beyond the end of the output buffer and continues composition at the beginning of ar empty buffer. It cannot compose re-readable external representations for function LINKERs, the interpreter workspace, and stacks. The print routines compose the "unprintable" object within square brackets ([]) with either the name of system defined function LINKERs, the LAMBDA parameter list of user defined function LINKERs, the bytes as characters of short arrays (less than 128 bytes), or a question mark (?) preceding an octal number for long arrays and parts of the interpreter workspace and stacks.

PROG - PROGram special form.

```
(PROG ARG-LIST
  LAB EXP1
        . . .
      EXPn) -
```

Special form; places binding pairs on the system association list, ALIST, for each member (CAR) of the first parameter, ARG-LIST, a list of arguments consisting of SYMBOLic atoms and sublists. PROG binds each member that is a SYMBOLic atom, a variable, to NIL of the dummy argument list, ARG-LIST. The CAR of each sublist of the argument list is also a SYMBOLic atom which PROG binds to the value obtained by evaluating the second member of the sublist, the CADR. If ARG-LIST is NIL (empty), PROG places no new bindings on the association list. After PROG binds any arguments, PROG seqtentially evaluates any remaining, non-atomic parameters until either evaluating the GO special form, evaluating the RETURN furction, or reaching the end of the parameter list. The unevaluated atomic parameters are labels for the GO special form. After evaluating the GO special form, PROG restarts the sequential evaluation with any parameter following the PROG label used as the unevaluated parameter of GO. If PROG evaluates the RETURN function, PROG uses the value of any RETURN function parameter as the value of the PROG call and ceases se-

quential evaluation of further PROG parameters. If PROG evaluates a RETURN function call without parameters or if it exausts the supply of parameters to evaluate, it ceases and returns NIL as value. In any case, as PROG returns, it restores the system association list, ALIST, the old binding environment existing before the PROG call.

PROP - obtain PROPerty list pair.

(PROP ATM PRP FUNC) - Functional; returns any property binding pair on the property list of the first parameter, ATM, a SYMBOLic atom. The property binding pair consists of a CONSED node (dotted pair) whose left part (CAR) is the second parameter, PRP, a SYMBOLic atom, the property name, and whose right part (CDR) is the current binding (value) of the property. If ATM has no appropriate property binding, PROP returns a value by calling its third parameter, FUNC, a function of no arguments.

PUT - PUT property binding on property list.

(PUT ATM PRP ARG) - Replaces the property value of the property name, PRP, a SYMBOLic atom, on the property list of the first parameter, ATM, another SYMBOLic atom, with the third parameter, ARG. If the specified property does not exist on the property list, PUT creates a property binding pair. It returns the first parameter, ATM, whose property list it modified.

QUOTE - use argument as is.

(QUOTE ARG) - Special form; returns its parameter ARG as is. Since special forms receive parameters without prior evaluation, QUOTE returns its parameter ARG without evaluation. The READ function recognizes the single quotation mark (') followed by an S-expression as a shorthand for a list of the SYMBOLic atom "QUOTE" and the S-expression. For example, if READ encounters the characters

'(A B C)

it produces the list

(QUOTE (A B C)) .

QUOTIENT - divide arguments.

(QUOTIENT X Y) - Returns the quotient of dividing the first parameter, X, by the second parameter, Y. If either parameter has floating-point-numeric type, QUOTIENT converts the parameter of lesser precision to the type of the other before dividing. Otherwise, QUOTIENT returns an integer whose value is the number-theoretic quotient. The single-character-atom "/" is a synonym for QUOTIENT.

READ - create S-expressions from input characters.

(READ) - Returns an S-expression created from input characters starting at the current input buffer position. Upon encountering a list opening character ("(", "[", "<", or "{"), READ recursively calls itself to obtain members of a list expression. After encountering a list closing character (")", "]", ">", or "}"), READ completes each sublist under construction until matching a corresponding list opening character. READ creates a CONSED node of the expressions before and after a period (.). It ignores excess list close characters and any characters after the comment character, initially question mark (?), up to the next non-printing ASCII character, such as a new-line character (012), which delimits any token being scanned. When READ encounters user-defined readmacro characters, it uses the value of a call to the associated readmacro. Otherwise, it calls the scanner, TOKEN, to return the next item scanned in the input buffer. For example, the S-expressions READ creates from the characters

[<A 'B> (C . D) NIL]

or

((A . ((QUOTE . (B . NIL)) . NIL)) . ((C . D) . (NIL . NIL)))

or

({A 'B} . <C . D> [)

are congruent. Whenever READ reaches the end of the input buffer, it calls the operating system to obtain more ASCII characters from the current input file.

READCH - READ a single-CHaracter atom.

(READCH) - Returns a single-character, SYMBOLic atom which represents the next character in the input buffer, regardless of any delimiter or readmacro status of the character. If READCH finds no further characters in the input buffer, READCH calls the operating system for another line of ASCII characters.

READMAC - manipulate character READMACro status.

(READMAC CHAR ARG) - Returns the existing readmacro status of the character specified by the first parameter, CHAR, which STRING, called by READMAC, converts irto a single character string. If the character is not a readmacrc character, READMAC returns NIL (false). If the character specifies a user defined readmacro, READMAC returns the function LINKER that the user established to be called by READ whenever READ encounters the character while looking for the start of a new token. Otherwise, READMAC returns a pseudo-function LINKER used by a system defined readmacro, e.g. the question mark (?), comment character, or the single quotation mark ('), QUOTE S-expression character. If READMAC gets the second, optional parameter, ARG, READMAC establishes a new readmacro status following the same rules used to return the old readmacro.

REMAINDER - REMAINDER after division.

(REMAINDER X Y) - Returns the number-theoretic remainder of dividing the first parameter, X, by the second parameter, Y, when both parameters are fixed-point numbers.

REMOB - REMove OBject from hash lists.

(REMOB ATM) - Searches the apprcpriate hash list for the parameter ATM, a SYMBOLic atom. If REMOB finds the atom and the user created the atom as opposed to the atom existing during sign-on, REMOB removes the parameter from the hash list and returns the parameter. REMOB also accepts a user-code-area, function master-LINKER as the parameter ARG. If REMOB had not previously marked the user-code area specified by the I-space address of the master-LINKER as unused, REMOB marks the code area as unused so that any points referenced by the code area may be reclaimed and returns the parameter, the function master-LINKER. Otherwise, REMOB returns NIL (false).

REMOBP - REMoveable OBject Predicate.

(REMOBP ARG) - Returns T (true) if the garbage collector could potentially reclaim the parameter ARG, i.e. the user defined the object after invoking LISP. Otherwise, REMOBP returns NIL (false).

REMPROP - REMove PROPerty from property list.

(REMPROP ATM PRP) - Removes any property binding pair indicated by the second parameter, PRP, a SYMBOLic atom, from the property list (*CDR) of the first parameter, ATM, another SYMBOLic atom. REMPROP returns the first parameter, ATM, whose property list REMPROP altered.

REQUEST - output query for S-expression input to evaluate.

(REQUEST ARG) - Forces output of the parameter ARG to the current output file without a carriage return and then returns the evaluation of the next S-expression read from the current input.

RETURN - RETURN to caller.

(RETURN ARG) - Returns the most current invocation of the PROG special form, the LOAD function, or the LISP supervisor to their caller. The caller receives any optional parameter ARG as the value of a call of PROG or LISP. If RETURN has no parameter, RETURN returns NIL. When RETURN leaves the top level of LISP supervision, the interpreter may exit back to the operating system using any RETURN parameter as status.

REVERSE - create REVERSEd list.

(REVERSE LST) - Creates a new list whose elements, CARs, are the elements of the parameter LST, a list, in reverse order. If the parameter LST is NIL, the empty list, REVERSE returns NIL.

RPLACA - RePLACe CAR.

(RPLACA ARG ITEM) - Replaces the lefthand side, *CAR, of the first parameter, ARG, usually a list, with the second parameter, ITEM. RPLACA returns the altered first parameter, ARG. In order to preserve system integrity, the first parameter of RPLACA should not be an integer or string node.

RPLACD - RePLACe CDR.

(RPLACD ARG ITEM) - Replaces the righthand side, *CDR, of the first parameter, ARG, usually a list, with the second parameter, ITEM. RPLACD returns the altered first parameter, ARG. In order to preserve system integrity, the first parameter, of RPLACD should not be an integer or string node.

SET - change fluid binding.

(SET ATM EXP) - Replaces any previously existing constant (global) binding given by a non-zero pointer in the constant binding cell (*CAR) of the first parameter, ATM, a SYMBOLic atom, a variable, with the second parameter, EXP. If ATM has no constant binding, SET searches the system association list, ALIST, for a binding dotted-pair whose lefthand side (CAR) is the first parameter, ATM, and whose righthand side, CDR, is the previous fluid binding value that SET will replace with EXP. If SET can find no binding pair for ATM, it inserts a new binding pair, consisting of a CONSED node (dotted pair) whose CAR is ATM and whose CDR is EXP, on the current system association list just below a marker, the atom LISP, added to the list by the most current level of LISP supervision. Any such binding disappears as the current level of LISP supervision exits. If SET cannot find any marker to use in the latter case, usually because a level of LISP supervision is not in effect during LOADing at start-up under UNIX, the LISP interpreter exits in error mode (IOT).

SETCOL - SET next COLumn to read input.

(SETCOL COL) - Sets the input routines, READ, READCH, or TOKEN, to obtain the next characters from the input-buffer column indicated by the parameter COL, a fixed-point number.

SETQ - Quote the first argument of SET.

(SETQ NAME EXP) - Special form; serves as an abbreviation for

(SET 'NAME EXP)

since SETQ only evaluates the second parameter, EXP. SETQ fluidly binds the first parameter, NAME, a SYMBOLic atom, a variable, as is, without evaluation. Thus, in the example, SETQ alters the fluid binding of NAME rather than any variable which could have been bound to NAME.

SH - UNIX SHell.

(SH ARG) - Invokes the UNIX shell, the operating system command interpreter, with an implicit "-c" option, using any optional, given parameter ARG, which SH converts to a string followed by a zero byte, as a shell command line. If SH gets no parameter, SH invokes the UNIX shell without options or parameters, so that the shell will read commands from the current standard input file. While SH waits for the shell command interpreter to finish, SH ignores the standard (DEL) and quit (CNTR-SHIFT-L or CNTR-Backslash) asynchronous interrupts. While waiting, SH absorbs, without notification, any other offspring created by forking that terminate concurrently. SH returns an octal representation of the status word that the operating system returns in CPU register R1, %1. Only UNIX VLISP defines SH.

SINGLE - convert to SINGLE precision floating.

(SINGLE ARG) - Converts the parameter ARG into a single-precision, floating-point value (two 16-bit words). VLISP defines DOUBLE only when defining both double and single precision floating-point-number types.

SPACE - set vertical output SPACE count.

(SPACE ARG) - Sets the number of vertical spaces, line feeds (012), given by the parameter ARG, a fixed-point number, that will precede the next output from the output composition buffer. If ARG is zero, SPACE outputs a carriage return (015) and no line feed, which will allow many output devices to overprint the current line with the next. If ARG is large (greater than 64) or negative, SPACE outputs an ASCII form feed (014) instead of any line feeds, which causes many output devices to perform top of form actions. Alternatively, ARG may be NIL, which causes the next line of output to be sent without carriage control characters. Thus SPACE with a NIL parameter may be used to send a prompt without advancing to a new line. SPACE sends any control characters to the current output file immediately.

STACK - STACK list as arguments to function call.

(STACK LST) - Special form; evaluates the members (CARs) of the parameter LST, a list, and passes the values as parameters to the most immediate, surrounding, furction call being constructed. If LST is NIL (the empty list) STACK passes no new parameters to the function call being constructed. For example, evaluating the function call

(FN A 'B 'C D)

is equivalent to evaluating the function call

(FN A 'B (STACK (LIST 'C D)))

that uses a STACK invocation.

STRING - convert to STRING internal type.

(STRING ARG) - Converts the parameter ARG into internal type STRING. If ARG already has type STRING, STRING returns it. STRING uses the print name of SYMBOLic atoms, exclusive of any GENSYM number. It produces the printed representation of all other internal types and converts the characters into a string.

SUBST - SUBSTitute one item for another in S-expression.

(SUBST NEW OLD EXP) - Returns a copy of the third parameter, EXP, an S-expression without cycles, with all occurrences that are congruent (EQUAL) to the second parameter, OLD, altered recursively to the first parameter, NEW. The returned S-expression creates new CONSED nodes only for subexpression which have been altered.

SUB1 - decrement argument.

(SUB1 X) - Subtracts 1 from the parameter X. If X has floating-point type, SUB1 returns the same type. Otherwise, SUB1 returns an integer.

SYS - call UNIX operating SYStem.

(SYS X ARG1 . . . ARGn) - calls the UNIX operating system by constucting an indirect "sys" instruction call using the parameter X, which SYS converts into a fixed-point number, as the offset, low order byte, by bitwise, logically ORing the values. SYS passes any remaining parameters as parameters following the "sys" call, after appropriate conversions. It passes the values of fixed-point numbers, pointers to floating-point numbers, the texts of arrays, and the I-space addresses of other function LINKERs. If needed, SYS converts strings, the print names of SYMBOLic atoms, and lists, whose members SYS assumes to specify single characters, into strings which have a zero (null) byte, the delimiter for strings passed to the UNIX operating system. SYS also places the last two values computed from the parameters into registers, R1 and R0. SYS returns an integer representation of the value returned in CPU register R0 by the operating system. Only UNIX VLISP defines SYS.

TERPRI - TERminate and send PRInt buffer.

(TERPRI FILE) - Sends any output in the printing composition buffer to the logical, internal file-number given by the parameter FILE, a fixed-point number obtained from the operating system as the value of OPEN or similar function calls under UNIX. If TERPRI gets no parameter, TERPRI sends the composition buffer contents to the current output file. If FILE is NIL, the current output is sent to the standard output file. TERPRI saves the current parameter FILE, as the constant (global) binding of the SYMBOLic atom, *TERPRI, to redefine the current output for calls to PRINT, PRIN1 and PRIN2. The LISP supervisor resets the current output file to the standard output file before printing values.

TIME - TIME in clock ticks.

(TIME) - Returns an octal representation of the low order word of the current time measured in system clock ticks by the operating system. The SYS or TRAP function may obtain the high-order word of the time under UNIX or DOS, respectively.

TIMES - multiply arguments.

(TIMES ARG1 . . . ARGn) - Multiplies the parameters from left to right, converting either the next parameter or the current subproduct to the type of the one with higher precision if either has floating-point type. If VLISP supports double-precision floating-point and the product of two fixed-point values, with signs, cannot be represented by a 16-bit, signed, fixed-point number, TIMES converts the subproduct to double precision to avoid losing information. If VLISP supports single-precision but not double-precision, TIMES converts a fixed-point product that overflows into a single-precision value. If VLISP does not support floating-point arithmetic, TIMES uses the low order word of all products, even if multiplication overflows. If the TIMES function gets no parameters, TIMES returns integer one (1), the empty product. The single-character-atom "*" is a synonym for TIMES.

TOKEN - scan next input TOKEN.

(TOKEN) - Scans and creates an atom from the next position in the input buffer. TOKEN ignores the readmacro status of most characters. It skips ASCII blanks and commas before starting the scan and converts any alpha characters which are not in the standard case into the corresponcing characters in the other case. If VLISP uses lower case characters for system defined atom names, TOKEN will convert upper case characters in SYMBOLic names into lower case characters as they are scanned. TOKEN terminates scanning after encountering a character with delimiter status and positions the read routine input buffer pointer so that the delimiter character will be read first by the next call to READ, TOKEN, or READCH. When TOKEN encounters an escape character, initially exclamation point (!), while scanning any atom other than a string, TOKEN uses the following character as is, regardless of any delimiter or readmacro status of the character. The result of scanning an atom which possessed an escaped character is a SYMBOLic atom, even if the format of what was read is otherwise that of a number. When TOKEN encounters a string delimiter character, initially double quotes ("), in the first unskipped position, TOKEN creates a string using the characters as they appear, regardless of readmacro or delimiter status, using each pair of string delimiter characters as a single string delimiter character, until TOKEN finds an unpaired string delimiter character which ends the string scan. When TOKEN scans the list opening or closing characters as the first unskipped character of the input buffer scan or any other character with delimiter status on which it does not perform exceptional actions, it returns the corresponding single-character atom and positions the input buffer point to read the following character on the next call to READ, TOKEN, or READCH. If TOKEN has not completed scanning but has reached the end of the input buffer, it obtains a new line of input from the current input file.

TRAP - call DOS or VOS operating system.

(TRAP ARG0 ARG1 . . . ARGn) - Calls the Virtual Operating System (VOS) or DEC's Disk Operating System (DOS) through the VOS emulator, performing an indirect system call using the TRAP instruction offset given by the first parameter, ARG0, a fixed-point number. TRAP converts any remaining parameters into values which it places in CPU registers R0 to R4 before the indirect call. It converts numbers to their values, obtains the starting address of array data when given a function linker to an array, uses the start of data of strings and the print name of SYMBOLic atoms, and the CAR pointer of CONSED nodes. It converts NIL into a zero value. TRAP returns a CONSED node (dotted pair) of octal representation of the values the operating system returned in CPU registers, R0 and R1. Only DOS and VOS VLISP define TRAP.

UNBREAK - UNdo the BREAK function binding.

(UNBREAK ATM) - Recreates any constant function or special form constant (global) binding of the parameter ATM, a SYMBOLic atom, a variable, which existed before a prior BREAK call, with ATM as the first parameter of the BREAK call. If ATM has no binding created by a BREAK call, UNBREAK changes nothing. UNBREAK returns ATM with any changed binding.

UNFLAG - remove FLAG from property list.

(UNFLAG ATM FLG) - Removes the second parameter, FLG, a SYMBOLic atom, from the property list of the first parameter, ATM, another SYMBOLic atom. If the property list does not contain the flag, UNFLAG makes no changes. UNFLAG returns the first parameter, ATM.

WAIT - WAIT for concurrent process termination.

(WAIT) - Pauses if the current process has active children, usually created by the FORK predicate, and no child has terminated whose remnants still exist. Upon finding an existent, terminated child, WAIT removes the remnants of the child process and returns a CONSED node (dotted pair) of two integers returned by the UNIX operating system in registers R0 and R1, which give the child's process identification number (PID) and termination status word, as the CAR, lefthand side, and CDR, righthand side, respectively. If the current process has no children, WAIT generates an internal-type-zero (C) error. Only UNIX VLISP defines WAIT.

ZEROP - ZERO Predicate.

(ZEROP X) - Returns T (true) if the high-order word of its numerical parameter X is zero; otherwise NIL (false). If the high-order word of floating-point values is zero, floating-point hardware treats the value as zero. Fixed-point values consist of the high-order word.

*BEGIN - BEGIN new area for compiled code.

(*BEGIN ARG) - Returns a master LINKER whose I-space address begins a new area which may receive compiled code and whose *CDR (lefthand side) is the parameter ARG. By convention, ARG should be an S-expression which evaluates back to the master LINKER that *BEGIN creates. VLISP defines *BEGIN only if supporting compiled LISP code.

*CAR - unrestricted CAR.

(*CAR ARG) - Finds the unrestricted CAR (lefthand side) of the parameter ARG. The *CAR of CONSED nodes is the same as the CAR; of LINKERs, the associated pointer; of SYMBOLic atoms, any constant (global) binding or a pointer whose value is zero if the SYMBOLic atom has no constant binding. The *CAR of strings is a word consisting of the first two bytes; of floating-point numbers, the second word; and of fixed-point numbers, the value, each returned as a value instead of a pointer to the value. Such values should not be retained while any other node is allocated, since garbage collection may be misled.

*CDR - unrestricted CDR.

(*CDR ARG) - Finds the unrestricted CDR (righthand side) of the parameter ARG using the value indicated as a pointer. The *CDR of CONSED nodes is the same as the CDR; of SYMBOLic atoms, it is the property list. The *CDR of numbers (the high-order word) and of strings (the byte length), and of LINKERS (the I-space address), each returned as values rather than pointers to values, should not be retained while allocating any other node, since garbage collection may be misled.

*CHAIN - obtain definition of CAR-CDR chain function.

(*CHAIN ATM) - Returns the defining string of any CAR-CDR chain function which is constantly bound to the parameter ATM, a SYMBOLic atom; otherwise NIL (false). *CHAIN calls *CAR to obtain any constant binding of ATM.

*DEF - obtain LAMBDA function definition.

(*DEF ATM) - Returns the LAMBDA parameters (in a list) which were used to create the function constantly (globally) bound to the parameter ATM, a SYMBOLic atom; otherwise NIL (false). *DEF calls *CAR to obtain any constart binding of ATM.

*DEPOSIT - create master LINKER for binary input.

(*DEPOSIT ARG) - Returns a master LINKER whose I-space address specifies an area of user code read from the file given by the constant binding of *LOAD, a previously-opened, logical file-number given as the last parameter of the most recent LOAD function call. The file read is in DEC absolute or a.out format depending on whether VOS, possibly emulated by DOS, or UNIX is the host operating system, respectively. The lefthand side (*CAR) of the created master LINKER is the parameter ARG, which conventionally is an S-expression which evaluates back to the master LINKER. VLISP defines *DEPOSIT only if supporting compiled LISP code.

*EMIT - install value into writable I-space.

(*EMIT MASTER POINTER POINTER-OFFSET ARG-OFFSET ARG) - Places the last parameter, ARG, a pointer to a node as modified by the penultimate parameter, ARG-OFFSET, a fixed-point number, into a location determined by adding the optional parameter, POINTER-OFFSET, a fixed-point number, to the pointer given by the table of pointers location, the parameter POINTER, a fixed point number, from the start of the user I-space area indicated by the *CDR of the first, optional parameter, MASTER, a master function LINKER. The parameters may be omitted in the order third, second, first, and fourth, which are POINTER-OFFSET, POINTER, MASTER, and ARG-OFFSET, respectively. If *EMIT gets only the last two parameters, *EMIT places ARG as modified by ARG-OFFSET into the next available location for generated code, and places an entry which references the modified parameter into the table of offsets at the end of the user code area. If *EMIT gets only the last parameter, ARG, a fixed-point number in this case, it places the value of the number into the next available I-space generated code location. *EMIT always returns NIL. VLISP only defines *EMIT if supporting compiled LISP code.

*EPT - obtain location from Entry Point Table.

(*EPT X) - Returns the function LINKER or pseudo-function LINKER counted by the parameter X, a fixed-point number, from the beginning of the entry point table, which starts in the first page of function LINKERs. The entry point table begins with the pseudo-function linkers used by system readmacros. If VLISP supports compiled code, the readmacro pseudo-function LINKERs precede pseudo-functions giving I-space addresses used by compiled code and constants within the system workspace. The system function LINKERs follow the pseudo-function LINKERs.

*EXAM - obtain value from user code area.

(*EXAM MASTER POINTER POINTER-OFFSET) - Returns an octal representation of the value at the user-code-area I-space address referenced by the second parameter, POINTER, a fixed-point number which indicates a pointer address, modified by the optional third parameter, POINTER-OFFSET, another fixed-point number, found by adding the offset found from the end of the table of pointers at the end of the user code area indicated by the first parameter, MASTER, a master LINKER, to the I-space address given by the master LINKER. If *EXAM gets only the first parameter, MASTER, it returns an octal representation of the value at the address referenced by MASTER. If the calculated address is not in the user code area, *EXAM returns NIL (false).

*MACRO - obtain MACRO defirition.

(*MACRO ATM) - Returns the function LINKER which DEFMAC used to create a macro special form and constantly (globally) bind it to the parameter ATM, a SYMBOLic atom. *MACRO calls *CAR to obtain any constant binding of ATM. If *MACRO finds no appropriate binding, it returns NIL (false).

*ORG - return LINKER to next available code location.

(*ORG ARG0 ARG1 . . . ARGn) - Returns a function LINKER whose I-space address is the next available location for compiled code and whose lefthand side (*CAR) is the first parameter, ARG0, an S-expression which conventionally is the master LINKER to the current code area. If *ORG receives additional, optional parameters, it sums the parameters and uses the total as the I-space address of the returned function LINKER. VLISP defines *ORG only if supporting compiled LISP code.

*REVERSE - REVERSE list without generating new nodes.

(*REVERSE LST) - Returns a list by altering the righthand sides (CDRs) of the parameter LST, a list, whose members (CARs) *REVERSE returns in reversed order. If LST is NIL, *REVERSE returns NIL, the empty list. *REVERSE generates no new nodes while reversing LST.

*SPEC - obtain SPECial form definition.

(*SPEC ATM) - Returns the function LINKER which DEFSPEC used to create a special form and constantly (globally) bind it to the parameter ATM, a SYMBOLic atom. *SPEC calls *CAR to obtain any constant binding of the parameter ATM. If *SPEC finds no appropriate binding, it returns NIL (false).

CORRECTION TO TR-546

The name "VLISP" should be changed throughout to ULISP. A version of LISP known as VLISP was released in France in 1976. The author apologizes for the inadvertent name duplication.

COMPUTER VISION LABORATORY

Image Analysis
Picture Processing

301-454-4526

ULISP DISTRIBUTION INFORMATION

ULISP can be supported by PDP-11s with memory management, i.e. 11/40, 11/45, and 11/70, using either the UNIX operating systems or DOS. In order to support LISP compiled code under the UNIX operating system, at least 80K words of primary memory should be available and the UNIX operating system will need some modification. More information is available in the manual:

> ULISP for PDP-11s with Memory Management, TR-546,
> Robert L. Kirby
> Computer Science Center
> University of Maryland
> College Park, Maryland 20742
> June, 1977.

If you want a copy of ULISP, please send:

1) A check for $75.00 (US) payable to the Computer Science Center, University of Maryland (or purchase order) for the distribution costs (no warranty or service is implied);

2) A signed copy of the ULISP copyright license which will be returned to you with my signature;

3) Choice(s) of operating system (DOS or UNIX) which will support ULISP;

4) Specifications of the density of the 9-track tape (800 or 1600 FPI) and format (UNIX "tp" format or DOS-PIP format) which will be sent containing two copies of:
   a) a load module version of ULISP,
   b) the ULISP source code,
   c) LISP software,
   d) if the UNIX operating system is to be used, UNIX system modification instructions and short manuals;

5) A description of each configuration which will support ULISP.

The description will be used to create an appropriate ULISP load module. The description should include:
   a) the number of words of primary memory,
   b) the processor model (/40, /45, etc),
   c) the availability of a floating point processor,
   d) the print width (in columns) of terminals. (Give the narrowest print width of terminals which will not wrap-around when sent characters beyond the last column.)

COMPUTER VISION LABORATORY

Image Analysis
Picture Processing

301-454-4526

ULISP COPYRIGHT LICENSE

I grant the licensee, (name and address)

_____

_____

_____

_____

permission to use, copy, and modify my ULISP, LISP interpreter, ULISP related software, and documentation for use by the licensee and for distribution to other ULISP copyright licensees provided that:

1) The copyright notice

COPYRIGHT 1978, Robert L. Kirby

is conspicuously placed on all copies and versions including physical media used for transmission (such as magnetic tapes) and within copies of source code;

2) The interactive-mode sign-on message of the ULISP interpreter continues to include the copyright notice;

3) Copies and versions are transmitted only to the licensee or to other ULISP copyright licensees;

4) If the ULISP version for the UNIX operating system, which contains modified UNIX software, is requested, the licensee maintains a UNIX license agreement with Western Electric Corporation; and

5) A responsible agent of the licensee has acknowledged agreement to these conditions.

For the licensee:

Robert L. Kirby
Computer Science Center
University of Maryland
College Park, Maryland 20742

Dated: _____