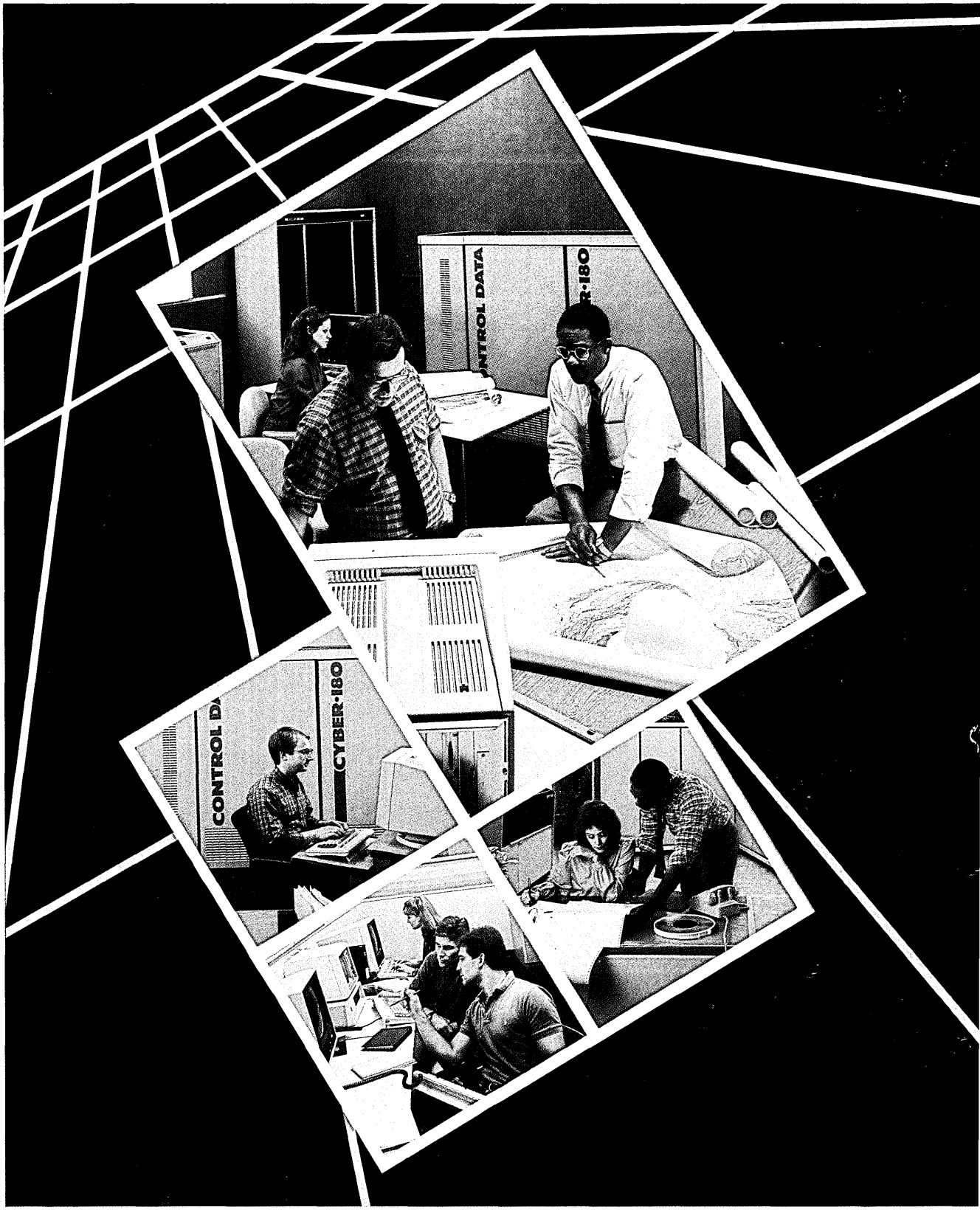
LISP for NOS/VE
Language Definition

CONTROL DATA



Usage Supplement

60486213

Instructions for Updating This Manual

December 1989

About This Packet

Title:	LISP for NOS/VE Language Definition Usage Supplement
Publication number:	60486213
Revision:	B
NOS/VE Version:	1.5.1
PSR Level:	739

About This Revision

This revision describes the support of the load-entry function. This function allows you to improve the loading speed while building large applications. This feature is described in chapter 27, Loading Speed.

Instructions

To update this manual, make the following changes:

Remove	Insert
Title Page thru 7	Title Page thru 9
1-1/1-2	1-1/1-2
25-3/25-4	25-3/25-4
25-7	25-7
---	Blue Fly Sheet, Chapter 27
---	27-1 thru 27-5
E-11/E-12	E-11/E-12
Index-1 thru Index-3	Index-1 thru Index-3
Mailer/Comment Sheet	Mailer/Comment Sheet

LISP for NOS/VE
Language Definition

Usage Supplement

Publication Number 60486213

Manual History

Revisions	System Version/ PSR Level	Product Version	Date
01	1.1.2/630	1.0	March 1985
02	1.1.3/644	1.1	October 1985
03	1.1.4/649	1.2	January 1986
04	1.2.1/664	1.3	September 1986
05	1.2.2/678	1.4	April 1987
06	1.2.3/688	1.5	September 1987
07	1.3.1/700	1.6	April 1988
A	1.4.1/716	1.7	December 1988
B	1.5.1/739	1.8	December 1989

Revision B documents LISP for NOS/VE at release level 1.5.1 and at PSR level 739. This manual was published in December 1989.

This revision describes the support of the load-entry function. This function allows you to improve the loading speed while building large applications. This feature is described in chapter 27, Loading Speed.

Contents

Figures

Tables

About This Manual

List Processing (LISP) for NOS/VE is a full implementation of the Common LISP language dialect defined by the Carnegie-Mellon University Spice LISP project. CONTROL DATA® LISP is implemented from the description of the Spice project results given in the commercial textbook *Common LISP, The Language*. CDC® LISP uses this manual (referred to throughout this book as *Common LISP*) as the basis for its usage manual with permission of Digital Press.

Within this manual, NOS/VE LISP is referenced as LISP, and the Common LISP language as Common LISP.

Acknowledgments

This document is based on *Common LISP, The Language*, written by Guy L. Steele, Jr., published by Digital Press (Billerica, Massachusetts), copyright 1984 by Digital Equipment Corporation. The original work constitutes the sole specification for the Common LISP language, and any departures from that specification are the responsibility of CDC.

Audience

This manual and *Common LISP* constitute the reference text for application programmers familiar with Common LISP or another LISP dialect. We presume you have read *Common LISP* and are familiar with the NOS/VE operating system.

Organization

This manual is organized for use as a reference supplement to *Common LISP*. The chapters in this manual have the same numbers and the section titles are the same as in *Common LISP* when possible. The page number where each corresponding discussion in *Common LISP* begins is indicated in parentheses next to the titles in this manual.

Conventions

This manual uses the same notational conventions as *Common LISP*, except for the use of typefaces to define syntax. The following notational conventions are unique to this manual.

UPPERCASE	Terms other than those in LISP forms appear in uppercase to depict names of commands, functions, parameters, and their abbreviations. Names of nonLISP variables, files, and system constants also are shown in uppercase within text.
lowercase	For consistency with *Common LISP*, required terms (function names and so forth) in forms appear in lowercase.
italics	Within command formats, italics indicates optional parameters. For example, in the following format of the LISP command, all parameters are optional: **LISP** *INPUT=file reference* *OUTPUT=file reference* *STATUS=status variable* Within text, italics indicates the title of a book. For example, *Common LISP*.
computer font	Indicates examples.
(abbreviations)	Recognized abbreviations for parameter keyword names in NOS/VE command parameter descriptions are indicated in parentheses.
numbers	All numbers are base 10 unless otherwise noted.

Additional Related Manuals

The Related Manuals section, appendix B, shows you which manuals you should be familiar with, and which manuals you might want to read following this one. In addition, several commercial tutorials on LISP are available, including:

- *LISP* (Second Edition by Patrick Henry Winston and Berthold Klause Paul Horn, copyright 1984 by Addison-Wesley Publishing Company, Reading, Massachusetts.) This book uses the Common LISP dialect.

Ordering Printed Manuals

Control Data printed manuals are available through Control Data sales offices or by sending an order to:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

When ordering a manual, please specify the complete manual title and the publication number. For example, if you are ordering this manual, specify LISP for NOS/VE Language Definition Usage Supplement, publication number 60486213.

Submitting Comments

The last page of this manual is a comment sheet. Please use it to give us your opinion of this manual's usability, to suggest specific improvements, and to report technical or typographical errors. If the comment sheet has already been used, you can mail your comments to:

Control Data Corporation
Technical Publications
5101 Patrick Henry Drive
Santa Clara, CA 95054

Please indicate whether you would like a written response.

If you have access to SOLVER, the Control Data online facility for reporting problems, you can use it to submit your comments. You should specify LI8 when SOLVER prompts you for a product identifier.

In Case You Need Assistance

Control Data's CYBER Software Support maintains a hotline to assist you if you have trouble using our products. If you need help beyond that provided in the documentation or find that the product does not perform as described, call us at one of the following numbers and a support analyst will work with you.

From the USA and Canada: (800) 345-9903

From other countries: (612) 851-4131

The preceding numbers are for help on product usage. Address questions about the physical packaging and/or distribution of printed manuals to Literature and Distribution Services at the following address:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

or you can call (612) 292-2101. If you are a Control Data employee, call CONTROLNET® 243-2100 or (612) 292-2100.

Introduction 1

Introduction 1

This chapter supplements chapter 1 of *Common LISP*. The LISP command and ve-command function unique to LISP are introduced.

Errors (5)

LISP signals all errors that *Common LISP* requires to be signaled. Many additional errors are also signaled. Signaling an error causes the debugger to be entered if there is no handler for the error. See chapter 26 for a description of the debugger's capabilities.

Entering LISP

Use the following command to enter LISP:

LISP
INPUT=file reference
OUTPUT=file reference
STATUS=status variable

Parameters:

INPUT (I)

NOS/VE file containing valid LISP input statements. If you omit this parameter, the file $INPUT is used, and you are prompted for input at your terminal.

OUTPUT (O)

NOS/VE file to receive LISP output values or diagnostic messages. If you omit this parameter, the file $OUTPUT is used, and output appears at your terminal.

STATUS

See the NOS/VE System Usage manual for a description of the use of this optional parameter.

LISP responds to the LISP command with the message:

```
Lisp/VE 1.8 89284
```

The Julian date is the date the LISP system was generated. The message is followed by the currently-defined NOS/VE input prompt (usually a question mark).

The INPUT and OUTPUT parameters supplied on the call to LISP provide files to which the two-way-stream *terminal-io* can connect. For example, if you need to enter LISP, read some startup commands, and then connect input/output to a terminal, you can write an SCL procedure containing the following commands:

```
DELFC $OUTPUT $LOCAL.OUTPUT
CREFC $OUTPUT $NULL
CREFC $INPUT TEST_INPUT_EXAMPLE
LISP
```

The TEST_INPUT_EXAMPLE file contains the LISP commands to execute. To eliminate extraneous output to the terminal, the last LISP command in the TEST_INPUT_EXAMPLE file must be the following:

```
(progn
  (ve-command "CREFC $OUTPUT $LOCAL.OUTPUT")
  (ve-command "DELFC $INPUT TEST_INPUT_EXAMPLE")
  (values))
```

Using NOS/VE or Other Software From Within LISP

You can use any NOS/VE command, or NOS/VE software that can be started with a NOS/VE command, from within LISP. To start and use other software or issue a NOS/VE command, use the following function:

```
(ve-command string)
```

Parameters:

string

Any string containing a valid NOS/VE command and its parameters, enclosed in quotation marks ("), or any valid form that evaluates to such a string. The LISP syntax for strings requires quotation marks, rather than the apostrophes used within a NOS/VE command.

Example:

```
(ve-command "EDIF MY_SOURCE I=COMMAND")
```

or

```
(setq a "ATTACH_FILE FILE=$USER.theorem_prover")
(ve-command a)
```

When you use this function, LISP submits the string to the NOS/VE command interpreter. If the command executes other software, LISP is pushed down on the job stack and subsequent dialog occurs with the executed software, such as an editor. When you leave that software, the job stack is pushed back up, and execution of LISP resumes.

LISP returns a NIL value after a normal return from ve-command execution, including any command that detaches the job; an abnormal return produces a non-NIL value containing an informative message.

Leaving LISP

Use either of the following functions to leave LISP:

```
(exit)
```

or

```
(quit)
```

You cannot omit the parentheses when you type (quit) or (exit).

If you use the NAM (255x Network Processing Unit) network user-break-2 character or the NOS/VE terminate_break_character, you abort LISP execution. The NAM user-break-1 character or the NOS/VE pause_break_character can be used to interrupt and discard unwanted input and output.

Leaving LISP Temporarily

If you wish to leave an interactive LISP session temporarily, and return to it at a later time (anytime before the next system deadstart), you may do so by detaching the job. With the job detached, you are free to do whatever other work you choose (and even logoff). When you are ready to resume the LISP session, just reattach the job and resume the LISP command. Resuming LISP is preferable to a (quit), which ends your LISP session irretrievably, in two ways:

1. The environment that existed when the job was detached is preserved.
2. Reentering LISP in this way is considerably faster than initiating a new LISP session.

An example of how to detach and attach a job follows:

Type in:

```
<NCC>d
```

<NCC> is your Network Command Character (usually % if you use the CDCNET Communications Network).

A new login and a message such as the following is displayed, containing the number of the detached job:

```
You have the following detached jobs:
$0855_7777_AAI_7654
```

For further reference, write down the last four digits of the above message preceded by a $, in this case $7654. When you are ready to reactivate the LISP session, use the NOS/VE command ATTACH_JOB or ATTJ and the job number you wrote down earlier. For example:

```
ATTJ $7654
```

The following prompt appears:

```
Job has been reconnected to this terminal
*Suspended - 1*
p/
```

Respond to this prompt by typing:

```
RESUME_COMMAND
```

or

```
RESC
```

and your LISP session is now reactivated.

If you use the Control Data Distributed Communications Network (CDCNET), you can also switch between your LISP session and other tasks by establishing multiple connections.

For example, suppose you create a connection by using the CDCNET command CREATE_CONNECTION or CREC:

```
%CREC SYS1 LISPVE
```

and enter LISP. Then, you can create another connection as follows:

```
%CREC SYS2 LISPVE
```

and log in.

For more information on how to access services, such as NOS/VE and NOS, through CDCNET, refer to the CDCNET Access Guide.

Data Types 2

Data Types 2

This chapter supplements chapter 2 of *Common LISP*. LISP implementation of data types is described.

LISP stores every data object as a LISP-object. A LISP-object contains:

- The type of the data (such as integer, character, or array).
- The actual data or a pointer to the location of the actual data.

Data Type Support (11)

LISP supports the following four array data types, specified through the :element-type keyword of the make-array function:

- General (arrays of LISP-objects created without an :element-type keyword argument, or with the :element-type keyword of T).
- Character (character string arrays created with an :element-type keyword argument of character).
- Short-float (floating-point number arrays created with an :element-type keyword argument of float).
- Bit (single-bit boolean variable arrays created with an :element-type keyword argument of bit).

Integers (13)

LISP uses two's-complement for internal representation. The internal radix used is 2; the external radix used is 10. Fixnums are stored in LISP-objects and accessed directly; fixnum use is faster than use of floating-point numbers.

LISP supports fixnum integers between −80000000 hexadecimal (−2147483648 decimal) and 7FFFFFFF hexadecimal (2147483647 decimal), inclusive. This restricted range permits a fixnum integer to fit into a LISP-object. The integer −0 does not exist as an entity distinct from +0.

LISP supports the bignum infinite-magnitude integer.

Floating-Point Numbers (16)

Short-format (short-float) floating-point numbers use the representation of a signed-magnitude fraction. These 64-bit floating-point numbers consist of a 1-bit sign, a 1-bit exponent sign, a 48-bit mantissa and a 14-bit exponent. The binary point is implied to the left of the mantissa. Approximate precision is 13 decimal digits. The number −0.0 is not distinguished from +0.0.

LISP supports short-float numbers between B00080000000 hexadecimal and 4FFFFFFFFFFF hexadecimal, inclusive. The smallest positive value is 3000800000000000 hexadecimal. The smallest negative value is CFFFFFFFFFFFFFFF hexadecimal. The following table provides the decimal equivalents:

Hexadecimal Number	Decimal Equivalent
B00080000000	$-0.4787488730476 \times 10^{-1233}$
4FFFFFFFFFFF	$0.5221944407065 \times 10^{1233}$
3000800000000000	$0.4787488730476 \times 10^{-1233}$
CFFFFFFFFFFFFFFF	$-0.5221944407065 \times 10^{1233}$

LISP supports only one format for floating-point numbers, which is as described above. LISP does not support 128-bit floating-point numbers. All of the formats (short, single, double, and long) are identical.

Floating-point numbers are stored as LISP-objects with pointers to the actual numbers; floating-point use is slower than fixnum use.

Characters (20)

LISP supports the Common LISP definition of character data types, as noted in the following subsections and in chapter 13.

Standard Characters (20)

LISP uses the following definitions for semi-standard Common LISP characters:

Common LISP Character	ASCII Character
#\Backspace	BS
#\Linefeed	LF
#\Page	FF
#\Return	CR
#\Rubout	DEL
#\Space	space
#\Tab	HT

Line Divisions (21)

LISP uses the ASCII US character for the Common LISP #\newline character. This is compatible with CDC network software and allows use of that software's terminal-dependent output formatting features. The sequences #\newline #\return or #\return #\newline produce output effects dependent on the terminal you use and on the network's definition of that terminal.

Non-standard Characters (23)

LISP does not support non-standard characters.

Character Attributes (23)

LISP does not support the non-zero font attribute or the non-zero bits attribute. The char-font-limit constant is 1. The char-bits-limit constant is 1.

Lists and Conses (26)

LISP does not use the equivalent of endp to test for the end of a list. LISP does not signal an error when a list is terminated by a non-NIL atom.

Overlap, Inclusion, and Disjointness of Types (33)

LISP has no extensions to the types number or array that exclude them as subtypes of type common.

Scope and Extent 3

Scope and Extent 3

This chapter supplements chapter 3 of *Common LISP*. LISP support of the concepts of scope and extent are described. The Glossary appendix contains definitions useful when reading this chapter.

Support of Extent (36)

If an entity has indefinite extent, LISP destroys the entity when reference is no longer possible.

LISP does not support multiprogramming or multiprocessing.

Type Specifiers 4

Type Specifiers 4

This chapter supplements chapter 4 of *Common LISP*. LISP support of type specifiers is described.

Type Specifiers That Specialize (45)

LISP supports only array specializations. You can specify the following specialized data types through the :element-type keyword of the make-array function:

- Character (created with a keyword argument of character); this is a specialized representation of arrays of characters of the data type character.
- Floating-point (created with a keyword argument of float); this is a specialized representation of arrays of short-float numbers of the data type float.
- Boolean (created with a keyword argument of bit); this is a specialized representation of arrays of boolean variables of the data type bit.

General arrays are created by omitting the :element-type keyword or by specifying the :element-type keyword with an argument of T. Such arrays are nonspecialized and have the data type T.

Program Structure 5

Program Structure 5

This chapter supplements chapter 5 of *Common LISP*. LISP support of program structures is described.

Forms (54)

The LISP evaluator has no extensions.

LISP signals an error for an attempt to evaluate an array or other invalid form.

Special Forms (56)

Appendix E lists all predefined special forms that LISP supports. Some special forms are implemented as macros within LISP, as indicated in the appendix.

Macros (57)

No LISP macro expansions contain data objects not considered to be forms in Common LISP. Some LISP macros have expansions that contain LISP-defined special forms.

Predicates 6

Predicates 6

This chapter supplements chapter 6 of *Common LISP*. LISP support of predicates is described.

Equality Predicates (77)

For the eq function, fixnum and character instances can be true. The following statement evaluations occur:

Statement	Value Returned
(eq 3 3)	T
(eq 3.0 3.0)	NIL
(eq #c(3 -4) #c(3 -4))	NIL
(eq '(a . b) '(a . b))	NIL
(eq #\A #\A)	T
(eq "Foo" "Foo")	NIL

For the eql function, the following statement evaluations occur:

Statement	Value Returned
(eql '(a . b) '(a . b))	NIL
(eql 0.0 -0.0)	T
(eql "Foo" "Foo")	NIL

Control Structure 7

Control Structure 7

This chapter supplements chapter 7 of *Common LISP*. LISP support of control structures is described.

Indefinite Iteration (121)

LISP has no extensions to the Common LISP syntax of the loop function.

Multiple Values (133)

LISP does not limit the number of multiple values that can be received by a special form.

Macros 8

Macros 8

This chapter supplements chapter 8 of *Common LISP*. LISP support of macros is described.

Macro Support (143)

LISP must encounter a macro definition before that macro is first used. In interpreted code, a macro is expanded each time it is encountered.

Declarations 9

Declarations 9

This chapter supplements chapter 9 of *Common LISP*. LISP support of declarations is described.

Declaration Syntax (153)

LISP allows the full set of Common LISP declarations in the declare special form.

Declaration Specifiers (157)

LISP provides no additional declaration specifiers.

The interpreter ignores all declarations except special.

The compiler ignores the following declaration specifiers:

ftype
function
compilation-speed

The compiler does not ignore the following declaration specifiers:

type
inline
notinline
ignore
optimize
declaration
space

For the optimize declaration, the qualities recognized are speed and safety. The values are relative. For example, specifying a value of 3 for both speed and safety has the same meaning as specifying a value of 1 for both speed and safety. It does not cause a trade-off between speed and safety. However, the following declaration:

```
(declare (optimize (speed 3) (safety 1)))
```

does optimize for speed at the expense of safety.

If you use an optimize declaration specifying a relatively high value for space, LISP ignores the inline declaration.

Symbols 10

Symbols 10

This chapter supplements chapter 10 of *Common LISP*. LISP support of symbols is described.

Creating Symbols (168)

The LISP make-symbol function installs a string in a symbol's print-name component that is the given print-name string. The string is not copied to a read-only area.

Packages 11

Packages 11

This chapter supplements chapter 11 of *Common LISP*. LISP support of packages is described.

Package Support (171)

LISP supports packages as described in *Common LISP*.

Modules (188)

For the require function, the value of the *module-file-translations* variable is an association list (alist) holding a mapping of module names to files.

The default value of *module-file-translations* is NIL. The default value of *modules-file* is the file $USER.LISP_MODULES.

If *module-file-translations* is NIL, the value of *modules-file* is the location of a file whose first expression is an alist to which *module-file-translations* is set. When the pathname argument to the require function is NIL or not provided, the *module-file-translations* variable is used to locate the files needed by the module. For example, if the file $USER.LISP_MODULES holds the following list:

```
(("foo" "$user.foo_file")
 ("goo" "$user.goo_file"))
```

and if the value of *modules-file* is "$USER.LISP_MODULES"; and you evaluate the following:

```
(require "foo")
```

then the *module-file-translations* variable is set to:

```
(("foo" "$user.foo_file")
 ("goo" "$user.goo_file"))
```

and the file $USER.FOO_FILE is loaded. If FOO had not been found in the *module-file-translations* variable, the require function would have attempted to load the file FOO.

Numbers 12

Numbers 12

This chapter supplements chapter 12 of *Common LISP*. LISP support of numbers is described.

Comparisons on Numbers (196)

For the max and min functions, LISP returns the argument in its current format. (LISP has only one floating-point format.)

Irrational and Transcendental Functions (203)

LISP uses the NOS/VE Common Math Library for these functions. For more information on Math Library functions, see the Math Library Usage manual.

Type Conversions and Component Extractions on Numbers (214)

All floats are normalized; they have precision 64 and radix 2.

Logical Operations on Numbers (220)

LISP uses two's-complement for representation when performing the integer-length computation.

Implementation Parameters (231)

The constants are defined, but because LISP provides only one floating-point format, the following constants are set to 0.0:

least-negative-double-float
least-negative-long-float
least-negative-short-float
least-negative-single-float

least-positive-double-float
least-positive-long-float
least-positive-short-float
least-positive-single-float

Characters 13

Characters 13

This chapter supplements chapter 13 of *Common LISP*. LISP support of characters is described.

LISP characters use standard 7-bit ASCII character codes. Characters are held directly in LISP-objects.

Character Attributes (233)

LISP does not support the font or bits attributes. The char-font-limit constant is 1. The char-bits-limit constant is 1.

Character Control-Bit Functions (243)

All of the following LISP constants are zero:

char-control-bit
char-hyper-bit
char-meta-bit
char-super-bit

Sequences 14

Sequences 14

This chapter supplements chapter 14 of *Common LISP*. LISP support of sequences is described.

LISP supports every sequence function listed in *Common LISP*.

Simple Sequence Functions (247)

A type defined via the deftype function is not permissible for the type argument of make-sequence.

Lists 15

Lists 15

This chapter supplements chapter 15 of *Common LISP*. LISP support of lists is described.

Lists (264)

LISP supports lists as described in *Common LISP*.

Hash Tables 16

Hash Tables 16

This chapter supplements chapter 16 of *Common LISP*. LISP support of hash tables is described.

Hash Table Support (282)

LISP supports hash tables.

Hash Table Functions (283)

The rehash-size argument has a default value of 100.

The rehash-threshold argument has a default value of NIL.

Arrays 17

Arrays 17

This chapter supplements chapter 17 of *Common LISP*. LISP support of arrays is described.

LISP supports arrays of up to 65,000 dimensions.

Array Creation (286)

LISP supports the following four array data types, specified through the :element-type keyword of the make-array function:

- General (arrays of LISP-objects created without an :element-type keyword argument; the :element-type parameter cannot have a value of T for general arrays).
- Character (character string arrays created with an :element-type keyword argument of character).
- Short-float (floating-point number arrays created with an :element-type keyword argument of float).
- Bit (single-bit boolean variable arrays created with an :element-type keyword argument of bit).

Fill Pointers (295)

vector-push

It is an error for new-element not to be the correct type for the vector.

vector-push-extend

The optional argument extension defaults to the size of the vector.

Strings 18

Strings 18

This chapter supplements chapter 18 of *Common LISP*. LISP support of strings is described.

String Access (299)

The LISP char and schar functions execute at the same speed.

Structures 19

Structures 19

This chapter supplements chapter 19 of *Common LISP*. LISP support of structures is described.

Defstruct Options (311)

If a type is specified for a slot, it must be the same as, or a subset of, the type specified in the included structure. If it is a strict subtype, LISP does not check assignments for errors.

LISP specifies an array to represent the structure if the type option is not specified.

The Evaluator 20

The Evaluator 20

This chapter supplements chapter 20 of *Common LISP*. The LISP evaluator is described.

The LISP evaluator is a recursive interpreter, performing each step as it is encountered. Forms are evaluated from left to right. Macros are expanded each time encountered.

Run-Time Evaluation of Forms (321)

The example on pages 323 and 324 works as follows:

```
(defvar *hooklevel* 0)
(declare (special *ncalls*))
(setq *ncalls* 0)
(defun hook (x)
    (let (( *evalhook* 'eval-hook-function)) (eval x)))

(defun eval-hook-function (form &optional env)
    (let ((*hooklevel* (+ *hooklevel* 1)))
        (setq *ncalls*  (+ *ncalls* 1))
        (format *trace-output* "~%~V@TForm: ~S"
            (* *hooklevel* 2) form)
    (let ((values (multiple-value-list
                (evalhook form
                    #'eval-hook-function
                    nil
                    env))))
        (format *trace-output* "~%~V@TValue: { ~S~}"
            (* *hooklevel* 2) values)
        (values-list values))))
```

The above routines display the following:

```
(hook '(cons (floor *print-base* 3) 'b))
Form:  (EVAL X)
    Form:  X
    Value:  (CONS (FLOOR *PRINT-BASE* 3) (QUOTE B))
    Form:   (CONS (FLOOR *PRINT-BASE* 3) (QUOTE B))
        Form:  *PRINT-BASE*
        Value:  10
        Form:   3
        Value:  3
    Value:  3 1
    Form:   (QUOTE B)
    Value:  B
Value:  (3 . B)
Value:  (3 . B)
```

The Top-Level Loop (324)

The top-level loop in LISP requires input in one or more continued lines and uses the user's currently specified terminal prompting character for each line. The value resulting from evaluation of the last-entered form always appears on a separate line. If multiple values are returned, each one appears on a separate line beginning on the first line after the last entered form.

You can view the top-level loop as the bottom of LISP's binding stack. As each occurrence of a form is encountered and evaluated, it pushes down any prior values bound to the same variables in the stack. This is most meaningful when recursion occurs. An error in LISP causes the debugger to be entered, if there is no handler for the error.

The prompt ? is printed whenever the top level loop requires input. Only one prompt is printed for each form. This differs from the previous versions of LISP where a prompt was printed at the start of each input line.

Streams 21

Streams 21

This chapter supplements chapter 21 of *Common LISP*. LISP support of streams is described.

Standard Streams (327)

LISP stream special variables have the following values:

```
*standard-input*     #<STREAM TO NIL>
*standard-output*    #<STREAM TO NIL>
```

NIL specifies that the streams are not directly connected to files.

terminal-io (328)

If other users need to use your LISP application without directly interacting with LISP, you can use an SCL procedure as shown in Entering LISP, chapter 1.

Input/Output 22

Input/Output 22

This chapter supplements chapter 22 of *Common LISP*. LISP support of input and output is described.

Input Functions (374)

LISP supports character and binary streams inputs.

Input From Binary Streams (382)

LISP allows input from binary streams. It supports the read-byte function.

For the read-byte function, the number of bits to transfer is determined in the same way as in the write-byte function. LISP transfers the appropriate number of bits from the stream to the integer. If the :element-type of the stream is a signed quantity, the highest bit is extended.

Output Functions (382)

LISP supports character and binary streams outputs.

Output to Character Streams (382)

If you print something and do not see it displayed immediately at the terminal, the reason may be the operating system's buffering of output. The finish-output function can be used to cause the buffer to be flushed. For example, the following function displays the question: What do you want?

```
(defun prompter ()
    (print "What do you want? ")
    (finish-output)
    (read))
```

Calling finish-output is necessary in order to force the question to be displayed and read the user's response from the same line.

Output to Binary Streams (385)

LISP allows output to binary streams. It supports the write-byte function.

For the write-byte function, LISP looks at the :element-type of the stream and calculates the number of bits to send to the stream. It then transfers the appropriate number of the low order bits of the integer to the stream.

Formatted Output to Character Streams (385)

Common LISP, in describing the ~F and ~E formats says that when rounding up and rounding down would produce printed values equidistant from the scaled value of arg, the implementation is free to use either one. LISP rounds down in this situation.

File System Interface 23

File System Interface 23

This chapter supplements chapter 23 of *Common LISP*. The LISP interface with the NOS/VE file system is described. For more information on the NOS/VE file system, see the NOS/VE System Usage manual.

File Names (409)

The filenames that are accepted by LISP pathnames are any valid NOS/VE file path. The definition for a filename is:

:family_name.user_name.catalog.file_name.cycle_reference.file_position

where catalog is specified as:

name.name. ... .name

where each name is a subcatalog in a permanent file catalog hierarchy.

Pathnames (410)

The components of a LISP pathname consists of six required components and two additional components unique to NOS/VE (family_name and file_position). The contents of each of the components are described as follows:

Host — The name of the file system, that is, NOS/VE.

Device — This slot contains a keyword that describes the nature of the file reference. If the reference begins with the family_name, the device component contains :family_name. If the file reference is absolute (such as, beginning with '.' as in .lisp_maintenance), then the device component contains :absolute. For a file reference that is relative (such as, a filename or a reference that starts with a catalog name such as $SYSTEM), the device component contains NIL.

NOTE

The value of this component is assigned by the system; it should not be assigned by the user.

Family_name — This component contains a string containing the family_name, if present; otherwise, it contains NIL.

Directory — If there are catalog names in the file reference, this component contains an array whose elements are strings corresponding to the individual catalog names. If the file reference contained .lisp.test, the directory component contains a two-element array whose first element is the string 'lisp' and whose second element is the string 'test'. If the file reference contains no catalogs, this component is NIL.

Name — This component contains a string that corresponds to the name of the file if a filename is given in the reference. If no name is specified, this component is NIL.

Version

If an explicit cycle number is specified, this component contains an integer corresponding to that cycle. If one of the special designators for file cycles is used, the appropriate keyword from the following list is placed in the version component:

Designator	Keyword
$HIGH	:newest
$LOW	:oldest
$NEXT	:next

If there is no cycle specified, this component is NIL.

File_position

If a file position is specified in the file reference, the correct keyword from the following list appears in the file_position component:

Designator	Keyword
ASIS	:asis
BOI	:boi
EOI	:eoi

If no file_position is specified, this component is NIL.

There is no provision for a :wild keyword as NOS/VE does not provide a wildcard facility.

Opening and Closing Files (418)

All LISP input/output functions use standard NOS/VE files. For example:

```
(open "$USER.filename")
```

This statement opens the permanent NOS/VE file named filename. The open function file reference parameter must be a namestring.

LISP does not override access modes assigned at the operating system level. An action or assignment within LISP which conflicts with the assigned NOS/VE access modes is not detected when LISP opens a file. For example:

```
(ve-command "ATTACH_FILE FILE=$USER.filename ACCESS_MODE=READ")
(setq an-output-stream (open "filename" :direction :output))
```

As requested by the ATTACH_FILE command, the named file is attached by NOS/VE as a read-only file. However, the open function gives a conflicting file direction (:output). This error goes undetected until a LISP action invokes an output function, such as:

```
(setq a-string "Please enter a form")
(print a-string an-output-stream)
```

LISP supports the open function keywords as described in *Common LISP*, with the exceptions mentioned in the following paragraphs.

:direction

To open an existing file for direction :output or :io, either use $NEXT in the filename or specify :if-exists to be something other than :error (which is the default).

:element-type

If :element-type is not string-char or character, the file opens as a segment access file. The file is random access, and it can be read or written using read-byte or write-byte. You can also use the file-position function on such files.

The byte sizes supported in the specification of :element-type are:

(:signed-byte n)
where n=1, 2, 3, 4, 5, 6, 7, 8, 16, 24, or 32.

(:unsigned-byte n)
where n=1, 2, 3, 4, 5, 6, 7, 8, 16, or 24.

(mod n)
where n=2, 4, 8, 16, 32, 64, 128, 256, 65536, or 16777216.

If you want to read or write segment access files with read-char and write-char, use the :element-type of (:unsigned-byte 8).

:if-exists

If :if-exists is :rename the file's new name is the same as the old name with _old appended. It is an error if both of the files already exist.

In Common LISP, the description of the open function with :direction :io and :if-exists :overwrite states that the file is not truncated back to length zero when it is opened. LISP, however, truncates the file back to its current position when the file is closed. For example, if you create a file ZX containing the following:

```
(1 2 3)
(4 5 6)
(7 8 9)
```

and then enter LISP and type the following:

```
(let ((str (open "zx" :direction :io :if-exists :overwrite)))
   (print (file-length str))
   (close str))
(let ((str (open "zx" :direction :io :if-exists :overwrite)))
   (print (file-length str))
   (close str))
```

the first length printed is 63. The second length printed is 28 because the file was truncated as a result of the first close.

Renaming, Deleting, and Other File Operations (423)

rename-file

You cannot rename NOS/VE files in the $LOCAL catalog. You can only rename permanent files using the function rename-file.

delete-file

The delete-file function deletes a file immediately. For a permanent file, if cycle is not specified, the lowest cycle is deleted immediately. Because NOS/VE does not allow deletion of an open file, using the delete-file function with an argument that is an open stream associated with a file is not successful. An attempt to delete a nonexistent file is also not successful.

file-write-date

For the file-write-date function, LISP returns the date of the file's last modification.

file-author

The file-author function returns the user information field of the file's directory entry.

file-position

Binary files are allocated a page at a time. EOF (and therefore the :end designation for file position) extends past the number of bytes actually written to the file unless the number of bytes written is a multiple of the system page size. For example, if the system page size is 8192 and 8000 bytes are written, the following form:

```
(file-position file-stream :end)
```

sets the position within file-stream to be 8192. In the preceding example, end-of-file is 8192 although only 8000 bytes are written.

For the file-position function of an :element-type of a (:signed-byte 8) segment-access file, performing a single read-char or write-char operation does not increase the file position by more than one.

Loading Files (426)

load

The *default-pathname-defaults* variable allows you to establish defaults for the components of a pathname. If you do a SET_WORKING_CATALOG and then specify a relative pathname for the filename parameter of the load function, the loaded file is the same file referenced by a NOS/VE command using the same relative path name. If the filename is a string instead of a pathname, the string is treated as a full path name with no defaults taken from *default-pathname-defaults*.

You can reset the value of *default-pathname-defaults* to cause pathname components to default to values of your choice. For example, if you do:

```
(setq *default-pathname-defaults* (pathname ".abc.def.ghi"))
```

then the following form:

```
(merge-pathnames "xyz")
```

returns the following:

```
#.(pathname ".abc.def.xyz")
```

Accessing Directories (427)

LISP supports the directory function. The ve-command function and the NOS/VE command DISPLAY_CATALOG can also be used for this operation.

The argument to the directory function must specify a directory, or it is an error. The :wild keyword cannot be used in the pathname argument.

Errors 24

Errors 24

This chapter supplements chapter 24 of *Common LISP*. LISP handling of errors is described.

LISP signals an error for the first encountered incorrect argument of a form. If there is no handler for the error, the debugger is entered.

Appendix D lists all diagnostic messages generated by LISP. LISP does not use the SCL STATUS variable for diagnostic messages.

General Error-Signalling Functions (429)

LISP error message indentation is uniform.

The warn function advances to a new line before and after output; the name of the function calling warn does not appear.

Specialized Error-Signalling Forms and Macros (433)

The check-type macro does not issue messages in a form dependent on the recognition of a particular form.

Debugging Tools (440)

The debugging tools supported by LISP are described in chapter 26, Error Handling and Debugging.

Miscellaneous Features 25

Miscellaneous Features 25

This chapter supplements chapter 25 of *Common LISP*. The remaining features of LISP are described.

The Compiler (438)

LISP supports compilation.

Several features that speed up compiled code slow down interpreted code. For instance, a macro is much more efficient when used in compiled code, where it is only expanded once at compile time; in interpreted code, it must be expanded each time it is encountered.

The LISP/VE compiler deletes and creates some files, whose names begin with the characters LIF$, in the $LOCAL catalog. To avoid damage to your $LOCAL files by the LISP/VE compiler, you are advised to not create files whose names have LIF$ as the first four characters.

compile (439)

The compile function compiles a lambda expression into object code. The format is:

compile name &optional definition [Function]

If definition is supplied, it should be a lambda-expression; if it is not supplied, then name should be a symbol with a definition that is a lambda expression. If name is a non-NIL symbol, then the compiled function object is installed as the global function definition of the symbol, and the symbol is returned; if name is NIL, then the compiled function itself is returned.

compile-file (439)

This function compiles the contents of the file specified as input-pathname and writes the compiled code to a NOS/VE object library whose name is specified by the :output-file argument. The result returned by compile-file is NIL. For arguments :output-file and :error-file, it is an error to specify an existing permanent file; it is not an error to specify the next cycle of an existing permanent file. The format of compile-file is:

compile-file &optional input-pathname &key :module-name [Function]
:output-file :error-file
:errors-to-terminal :load

input-pathname

This should be a LISP source file; its contents are compiled. If supplied, the input-pathname must be a valid file specifier acceptable to OPEN. If not supplied, the compiler prompts the user for the name. The input-pathname must not be the same as any name accessible in your program attribute libraries.

An error can occur if you compile a function and load it; then modify the function to have a different number of arguments but the same function name, and recompile and load it.

:module-name

This module-name is used as the name of the load module. The module-name must not be the same as the name of any of the other arguments, the pname of any symbol naming a function to be compiled, and any name accessible in your program attribute libraries. The module-name can consist of 1 to 20 upper case characters not including a dash (-), but including:

- letters
- digits
- #
- @
- _

If the module name is not supplied, the compiler prompts you for the name. An error can occur if multiple compilations are done in one LISP session, reusing the same module name.

The use of the term module has no relation with the *module* variable or the require and provide functions.

:output-file

The :output-file argument is used to specify an output pathname for the object library that the compiler produces. This file can be loaded using the load function or by specifying T for the :load argument of compile-file. If the output-file is specified, it must be a valid file specifier to OPEN. The default is LISP_LGO. An error can occur if multiple compilations are done in one LISP session, reusing the same output file name.

:error-file

Error messages for any of the errors detected by the compiler are written to :error-file if :errors-to-terminal is NIL. If the error-file is specified, it must be T, NIL, or a valid file specifier acceptable to OPEN. If the error-file is specified as NIL or not specified, then no error-file is generated. If the error-file is specified as T, then the file name is constructed by adding the characters _ERRORS to the input-pathnames' true name.

:errors-to-terminal

If T is specified, error messages are sent to the terminal; if NIL is specified, error messages are written to the error-file. The default is T.

:load

If T is specified, the compiled code is loaded; if NIL is specified, the compiled code is not loaded. The default is NIL.

disassemble (439)

The disassemble function takes a symbol with a lambda expression as a function definition and produces a CYBIL listing. The result of the disassemble function is the location of such a listing.

The disassemble function does not accept an already compiled function as an argument. It is an error to pass a function object or a lambda-expression to the disassemble function.

compile time environment

Certain features of LISP that occur automatically in the interpreter are not included in compiled code unless explicitly requested by the user. These features alter the environment in which compiled code is loaded. For example, a DEFSTRUCT in the interpreter creates setf-methods to set the structure's slot values. Each method is associated with the slot-accessor function for that slot. Consider this example:

```
(defstruct ship size speed)

(setq s (make-ship :size 100 :speed 20))

(setf (ship-speed s) 30)
```

When this DEFSTRUCT is executed by the interpreter, setf-method's are created for SHIP-SIZE and SHIP-SPEED, thus altering the interpreter environment for all ensuing processing. This alteration is not made when compiled code is loaded (unless the user requests it). A subsequent call to SETF with one of these slot-accessors returns an error signaling that a bad location specifier has been used for SETF, since there is no setf-method for the slot-accessor in the current environment.

To specify that the environment be altered to include the setf-methods for slot-accessors, execute the DEFSTRUCT within an EVAL-WHEN that specifies all changes in environment be made during compilation, loading, and evaluation, as in:

```
(eval-when (compile load eval)
    (defstruct ship size speed)
)
```

This form of EVAL-WHEN can be used whenever it is desirable to have all changes in environment implicit in compiled code be made when the code is used.

Debugging Tools (440)

For the room function, the degree of memory compaction of the internal data types and the format of the printed information are as follows:

Memory Compaction	Format
Conses allocated	Integer
Conses reclaimed	Integer
Array blocks allocated	Integer
Array blocks reclaimed	Integer
Array bytes allocated	Integer

The optional argument for the room function is ignored; thus, LISP prints the same information whether the room function is called with the argument or without it.

$save-lisp

This function allows you to save a LISP workspace between terminal sessions (however, a workspace created under Lisp/VE 1.7 cannot be used under Lisp/VE 1.8). $save-lisp has the form:

```
($save-lisp)
```

$save-lisp creates an executable NOS/VE file called $LOCAL.LISP_BASE_SYSTEM_SPACES, containing the state of the LISP system. This file can be made permanent for subsequent sessions.

$LOCAL.LISP.BASE_SYSTEM_SPACES does not contain any of your loaded compiled code. If you want to use the compiled code, the code has to be reloaded.

Please remember that $LOCAL.LISP_BASE_SYSTEM_SPACES will not execute properly under future versions of LISP. To execute this file, use the following NOS/VE commands:

```
SET_PROGRAM_ATTRIBUTES ADD_LIBRARY=$SYSTEM.LISP.BOUND_PRODUCT
EXET LISP_BASE_SYSTEM_SPACES SP=LIP$LISP_SYSTEM TEL=FATAL
```

You can include these commands in your user prolog file for convenience.

Substituting for the ed Function (442)

LISP does not have an internal editor. The ed function calls EDIT_FILE, the full screen editor of NOS/VE from within LISP. To do this, enter the function:

```
(ed "EDIT_FILE")
```

or

```
(ve-command "EDIT_FILE FILE=filename INPUT=COMMAND")
```

The argument to the ed function is required and must be a string specifying the file to edit.

LISP returns a NIL value after a normal return from ve-command execution; an abnormal return produces a value other than NIL containing an informative message.

More information about the NOS/VE EDIT_FILE command and the full screen editor can be found in the NOS/VE File Editor Tutorial/Usage manual.

Files edited with the full screen editor can be subsequently loaded by LISP using the load function. For example:

```
(load "EDIT_FILE")
```

The argument to the load function is a NOS/VE file reference (not a Common LISP pathname) and must be a namestring, enclosed in quotation marks.

You can debug your program the same way with any system-supplied editor available at your site. A convenient way to work is to enter code into a text file, which you then load into the LISP system using the load function.

Miscellaneous Debugging Tools

The time macro and the describe, inspect, and dribble functions are miscellaneous debugging tools that allow access to information about timing the evaluation of LISP expressions, internal storage management, program objects, and program input and output. These tools are described in the following paragraphs.

The Time Macro

time form [Macro]

Time is a macro which evaluates form and returns what form returns. However, as a side effect, timing data are printed to the stream that is the value of *trace-output*.

```
? (time (car '(i o)))
Evaluation times (in microseconds)
   JOB time     :  6814
   MONITOR time :  9872
   REAL time    :   8.01690000000000E+0005
I
```

The time macro is only accurate to within 50,000 microseconds. The job and monitor times are obtained from the operating system function:

```
PMP$GET_TASK_CP_TIME
```

and real time is obtained from operating system function:

```
PMP$GET_MICROSECOND_CLOCK
```

The Describe Function

describe object [Function]

The describe function prints, to the stream in the variable *standard-output*, information about the object. Sometimes it describes something that it finds inside something else; such recursive descriptions are indented appropriately. For instance, describe of a symbol exhibits the symbol's value, its definition, and each of its properties. Describe of a floating-point number exhibits its internal representation in a way that is useful for tracking down round-off errors. The describe function returns no values.

The Inspect Function

inspect object [Function]

The inspect function is an interactive version of describe. It allows you to examine and modify data structures.

The Dribble Function

dribble pathname [Function]

(dribble pathname) rebinds *standard-input* and *standard-output* to send a record of the input/output interaction to a file named by pathname. The primary purpose of this is to create a readable record of an interactive session. Because of a NOS/VE limitation, only the output actually gets recorded on the file.

(dribble) terminates the recording of input and output and closes the dribble file.

Environment Inquiries (443)

LISP provides several environment inquiry functions that identify the NOS/VE software and CDC's computer hardware. These functions are described below.

Other Environment Inquiries (447)

LISP returns the following values for functions in this section:

Function	Value	Comments
lisp-implementation-type	"LISP/VE"	Obtained from released code.
lisp-implementation-version	"Lisp/VE 1.8 89286"	Where 89286 is the Julian date on which the LISP system was built.
long-site-name	"CONTROL-DATA-CORPORATION-SITE"	Obtained from released code.
machine-instance	A string	The string contains the CYBER 180 serial number known to NOS/VE.
machine-type	A string	The string contains the CYBER 180 model type known to NOS/VE.
machine-version	"CDC CYBER 800 series"	Obtained from released code.
short-site-name	"CDC-SITE"	Obtained from released code.
software-type	"NOS/VE"	Obtained from released code.
software-version	A string	The string contains the operating system and product set levels.
features	(:CDC :VE :CYBER)	Obtained from released code.

All symbols placed on the *features* list must be in the keyword package.

Error Handling and Debugging 26

Error Handling and Debugging 26

This chapter discusses the handling of errors in and debugging of LISP programs. Topics discussed include defining errors (what is an error and what can you do with it), the LISP representation of errors, and how to find and fix unexpected errors. The error handling package includes:

- A set of error functions that allow detection and, where possible, correction of errors that you anticipate.
- A step facility that allows you to examine the control flow of a program one step at a time.
- A trace facility that allows you to watch the calling pattern of a specified set of functions.
- A debugger that allows you to examine and, where possible, correct the system state interactively when an error occurs.
- A set of debugging tools which allow access to information about timing the evaluation of Lisp expressions, internal storage management, program objects, and program input and output.

The following sections describe these capabilities.

Error Processing

The following description of error processing in LISP extends the current Common Lisp capabilities in this area. This debugger attempts to follow proposals now being considered by the ANSI Standard Committee.

Conditions

A condition is a LISP-object used to represent an exceptional situation which arises during execution of a program. The situation discussed here is the error: a condition which results from an incorrect program or incorrect data. Errors are not the only types of conditions however. Storage conditions are examples of serious conditions that are not errors. For example, the control stack may legitimately overflow without a program being in error.

Some types of conditions are predefined by the system. The predefined condition types are described in the section Predefined Condition Types. Condition types form an inheritance hierarchy in which each type has one parent type and each type can be the parent of any number of children. All types of conditions are subtypes of the predefined type condition; therefore, (typep c 'condition) is true only if c is a condition. All conditions inherit the properties of the condition type.

You are free to define any additional condition types necessary for a particular application. Condition types are defined using the define-condition function. Creating a condition object of a specified type is accomplished using the make-condition function. These functions are described in detail in this chapter.

Once a condition is created, it is common to signal it. To signal a condition means to attempt to locate and invoke a function, called a handler, which is specifically designed to deal with the type of situation represented by the condition. When a condition is signaled, handlers that may be appropriate are tried in a predefined order until one decides to handle the condition or until no more handlers are found. A condition is said to have been handled if a handler performs a non-local transfer of control to exit the signaling process.

Although non-local transfers may be accomplished using traditional Lisp mechanisms such as catch and throw, block and return, or tagbody and go, the condition system also provides a structured method for proceeding a condition. Proceeding a condition means resuming execution of the program that signaled the condition from some prespecified point. The use of structured primitives for proceeding allows a more integrated relationship between the user program and the interactive debugger.

It is not necessary that all conditions be handled. Some conditions are trivial enough that a failure to handle them may be disregarded. Other serious conditions must be handled in order to assure correct program behavior. If a serious condition is signaled but no handler is found, the interactive debugger is invoked so that the user may examine the state of the program and, in some cases, continue execution. An error is a serious condition.

It is usually useful to report a condition to the user or a log file of some sort. When the printer is invoked on a condition while *print-escape* is NIL, its report function is invoked. In particular, this means that an expression like (format t "~A" condition) invokes condition's report function. The report function is specified at the time the condition type is defined.

Defining Conditions and Creating Condition Objects

The define-condition macro is used to define a new condition type. The make-condition function is used to create an instance of a given type.

The Define-Condition Macro

define-condition name parent-type [keyword value]* &rest slots [Macro]

The define-condition macro defines a new condition type with the given name. Parent-type is the name of the super type. Slots is a sequence of symbol slot names or lists of slot names and default values, with the same syntax as the slots of a defstruct. For example:

```
(define-condition bad-food-color error food (color 'green))
```

creates a new type of condition called bad-food-color. Its parent type is error so it would inherit all the slots of error.

The slots specific to this condition are food and color. The color slot is initialized to the value green each time a bad-food-color condition is created unless some other value is supplied at the time of creation.

Slot accessor functions are generated automatically for each slot name that is unique to this condition type. If a slot name is specified which is also a slot name of one of name's ancestors, only one slot is created in the condition object. However, if a default is specified for this slot in the definition of name, it will override any ancestors' defaults.

The keyword and value pairs are:

:CONC-NAME symbol-or-string

The conc-name keyword specifies a prefix for the slot accessor functions of a condition. The default behavior for generating the accessor functions is to use the name of the new type, name, followed by a hyphen and the name of the slot. For example:

```
(define-condition incredibly-obscure-error obscure-error
    :conc-name ioe-
    date message)
```

defines a new condition type called incredibly-obscure-error with slots of date and message. The accessor functions are ioe-date and ioe-message rather than the default accessor functions incredibly-obscure-error-date and incredibly-obscure-error-message.

:REPORT-FUNCTION expression

Expression should be a suitable argument to the function special form, either a symbol or a lambda expression. It designates a function of two arguments, a condition and a stream, which prints the condition to the stream when *print-escape* is NIL.

:REPORT form

A short form of :report-function to cover two common cases. If form is a string, this is equivalent to:

```
:report-function
    (lambda (ignore stream) (write-string form stream))
```

Otherwise, if form is not a string, it is equivalent to:

```
:report-function
    (lambda (condition *standard-output*) form)
```

In the latter case, form describes how to print objects of the type being defined. It should send output to *standard-output*. The condition being printed is bound to a variable called condition during execution of the form.

:HANDLE form

Form is an expression to be used as the body of a default handler for this type of condition. Form may refer to a variable called condition; this is bound to the condition being handled during the execution of form.

NOTE

It is an error to specify both :REPORT-FUNCTION and :REPORT in the same define-condition. If neither is specified, the report method is inherited from a parent type.

The following examples define condition types. First, a condition called machine-error is defined which inherits from error:

```
(define-condition machine-error error
    :report (format t "There is a problem with ~A."
                    (machine-error-machine-name condition))
    machine-name)
```

The following defines a new error condition (a subtype of machine-error) for use when machines are not available:

```
(define-condition machine-not-available-error machine-error
    :report (format t "The machine ~A is not available."
                    (machine-error-machine-name condition)))
```

And finally, the following defines a still more specific condition, built upon machine-not-available-error which provides a default for machine-name but does not provide any new slots:

```
(define-condition my-favorite-machine-not-available-error
                  machine-not-available-error
    (machine-name "CDC CYBER 830"))
```

This gives the machine-name slot a default initialization. Since no :report clause was given, the format information supplied in the definition of machine-not-available-error is used if a condition of this type is printed while *print-escape* is NIL.

The Make-Condition Function

make-condition type &rest slot-initializations [Function]

The make-condition function calls the appropriate constructor function for the given type, passing along the given slot initializations to the constructor, and returning an instantiated condition. Slot-initializations are given in alternating keyword and value pairs, such as:

```
(make-condition 'bad-food-color :food my-food :color my-color)
```

Invoking the Signal and Debug Facilities

When a condition object is created, the most common operation to be performed upon it is to signal it. Signaling a condition means that the system tries to locate the most appropriate handler for the condition and invoke that handler. Signaling is done by invoking the signal function. Handlers are located according to the following rules:

- Check for locally bound handlers.
- If no appropriate bound handler is found, check for a default handler: first for the signaled type and then for each of its ancestors' types.
- If a handler is found, it is called. In some circumstances (to be described later), the handler may decline by simply returning without performing a non-local transfer of control. In such cases, the search for an appropriate handler is picked up where it left off, as if the called handler was not present.

- If no bound handler or default handler is found, or if all handlers which were found decline, signaling returns the condition which was signaled.

A handler is a function of one argument, the condition to be handled. The handler may inspect the condition to be sure it is interested in handling it. After inspecting the condition, the handler must take one of the following actions:

- It may decline to handle the condition by simply returning. When this happens, the returned values are ignored and the effect is the same as if the handler had been invisible to the mechanism seeking to find a handler. The next handler in line is tried, or if there is no such handler, the condition is returned.
- It may perform some non-local transfer of control using go, return, throw, abort, invoke-proceed-case, or a defined proceed function (described in the section Proceeding a Condition).
- It may signal another condition.
- It may invoke the interactive debugger.

It was stated above that the first step in signaling a condition is to look for a locally bound handler. Handlers are locally bound using the special form condition-bind.

condition-bind bindings &rest forms [Special Form]

Condition-bind executes forms in a dynamic context where the given local handler bindings are in effect. Bindings is a list of binding elements where each element takes the form:

```
(type handler)
```

Type may be a condition type or a list of condition types. Handler should evaluate to a function to be used to handle conditions of type(s) during execution of forms.

There are three common situations in which the signal facility is invoked: during the execution of the error, cerror, and warn functions. The debugger can also be invoked in these situations, as well as during the execution of the break function. Signaling is accomplished by calling the signal function. The debugger is invoked by calling the debug function.

The Signal Function

signal datum &rest arguments [Function]

The signal function searches for a handler for a condition. If a handler is found which handles the condition, control does not return to signal. If the condition is not handled, signal returns the condition object it was attempting to handle.

If datum is a condition, then that condition is used directly. In this case, it is an error for arguments to be non-NIL.

If datum is a condition type, then the condition used is the result of doing:

```
(apply #'make-condition datum arguments)
```

If datum is a string, then the condition used is the result of doing:

```
make-condition 'simple-condition
    :format-string datum
    :format-arguments arguments)
```

The Debug Function

debug condition [Function]

The debug function enters the debugger directly.

If the special variable *abort-debug* has the value t, condition is printed out and control is returned to the top level Lisp.

If *abort-debug* is NIL, condition is printed out and the interactive debugger is entered.

The Error Function

error datum &rest arguments [Function]

The error function invokes the signal facility on a condition. If the condition is not handled, the debugger is invoked. Control never returns to error from the debugger which means that error never returns to its calling program.

If datum is a condition, then that condition is used directly. In this case, it is an error for arguments to be non-NIL.

If datum is a condition type, then the condition used is the result of doing:

```
(apply #'make-condition datum arguments)
```

If datum is a string, then the condition used is the result of doing:

```
(make-condition 'simple-error
    :format-string datum
    :format-arguments arguments)
```

The Cerror Function

cerror proceed-format-string datum &rest arguments [Function]

The cerror function invokes the signal facility on a condition. If the condition is not handled, the debugger is called. While signaling is going on, and while in the debugger if it is reached, it is possible to proceed this condition using the function proceed. The value returned by cerror is the condition which was signaled.

If datum is a condition, then that condition is used directly. In this case, arguments are used only with the proceed-format-string and are not used to initialize datum.

If datum is a condition type, then the condition used is the result of doing:

```
(apply #'make-condition datum arguments)
```

If datum is a string, then the condition used is the result of doing:

```
(make-condition 'simple-error
    :format-string datum
    :format-arguments arguments).
```

The proceed-format-string must be a string. Note that if datum is not a string, then the format arguments used by the proceed-format-string are still the arguments (in the keyword format as specified). In this case, some care may be necessary to set up the proceed-format-string correctly. The format operator ~* may be particularly useful in this situation.

The Warn Function

warn datum &rest arguments [Function]

The warn function invokes the signal facility on a condition. If the condition is not handled, the text of the warning is output to the stream that is the value of *error-output* and, if the global variable *break-on-warnings* is true, the debugger is entered. In this case, warn returns only if proceed is done from the debugger. The value returned is the condition which was signaled.

If datum is a condition, then that condition is used directly. In this case, it is an error for arguments to be non-NIL.

If datum is a condition type, then the condition used is the result of doing:

```
(apply #'make-condition datum arguments)
```

If datum is a string, then the condition used is the result of doing:

```
(make-condition 'simple-warning
    :format-string datum
    :format-arguments arguments)
```

The Break Function

break &optional datum &rest arguments [Function]

The break function directly enters the debugger with a condition without trying to invoke the signal facility. Executing the function proceed while in the debugger causes a return from break. The value returned is the condition that was used.

If datum is a condition, then that condition is used directly. In this case, it is an error for arguments to be non-NIL. If datum is a condition type, then the condition used is the result of doing:

```
(apply #'make-condition datum arguments)
```

If datum is a string, then the condition used is the result of doing:

```
(make-condition 'simple-condition
    :format-string datum
    :format-arguments arguments)
```

Proceeding a Condition

Conditions are proceeded by error handlers or, in the cases of the warn, break, and cerror functions, you can proceed from the debugger. Proceeding a condition means passing control to some prespecified point in the program and resuming execution from that point. There are two special forms which are used to set up these points: proceed-case and condition-case. There is also a macro, catch-error-abort which is similar to proceed-case but is used only in a specific context. These forms are described below.

The Proceed-Case Form

proceed-case form &rest clauses [Special Form]

The proceed-case form passes control to some prespecified point in the program and resumes execution from that point.

Form is evaluated in a dynamic context where clauses specify points to which control may be transferred in the event that a condition is signaled. If form runs to completion without signaling a condition, all values it returns are simply returned by proceed-case. Otherwise, a handler may transfer control to one of the clauses. A proceed-case-clause has the form:

```
(proceed-function-name arglist [keyword value]* [body-form]*)
```

Each proceed case clause defines a proceed case structure. Proceed case structures are objects which behave similarly to catch tags. They serve as points to which control may be transferred by error handlers or by the debugger. In the following discussion, proceed case refers to a proceed case structure. The special form proceed-case is referred to as the proceed case form.

When control is transferred to a clause, body-form(s) is evaluated and any values returned by the last form is returned by the proceed-case form.

Proceed-function-name may be NIL, it may be the name of a defined proceed function (see define-proceed-function), or it may be any symbol which can be an argument to invoke-proceed-case.

Arglist is a list of variables to be bound during the execution of body-forms. Arglist may be NIL if you don't care about any of the arguments; otherwise, the variables must be compatible with the arguments of the proceed function or the values passed by invoke-proceed-case. The first variable is always the condition that was signaled.

The valid keyword and value pairs are:

:TEST function

Function is a function of one argument, the condition, which must return true for this clause to be visible to handlers. The function should be in a form which is acceptable as an argument to function.

:CONDITION type

Shorthand for the common case of :test in which the type of a condition is being tested to determine the visibility of the handler. The following two forms are equivalent:

```
:condition foo

:test (lambda (condition) (typep condition 'foo))
```

:REPORT-FUNCTION expression

Expression must be an appropriate argument to function and should designate a function of two arguments, a proceed case and a stream. The function should print a message which summarizes the action that proceed case will take.

:REPORT form

This is a shorthand for two important special cases of :report-function. If form is a string, then this is the same as:

```
:report-function
    (lambda (ignore stream) (write-string form stream))
```

If form is not a string, this is the same as:

```
:report-function
    (lambda (condition *standard-output*) form)
```

In the latter case, form must send output to *standard-output* and should summarize the action that this proceed case clause will take.

NOTE

Only one of :test or :condition and only one of :report or :report-function may be specified.

Keyword values specified in the proceed case clause override any defaults given in the definition of a defined proceed function.

If proceed-function-name is NIL, it is an error if report information is not supplied. Otherwise, default report information is generated if necessary using the proceed function name.

When the printer is invoked on a proceed case while *print-escape* is NIL, the report function for that structure is invoked. In particular this means that an expression like (format t "~A" proceed-case) invokes proceed case's report function.

The following example demonstrates the use of the proceed case form; the function break can be defined as follows:

```
(defun break (datum &rest arguments)
    (proceed-case
        (debug (cond ((typep datum 'condition) datum)
                     ((symbolp datum)
                      (apply #'make-condition datum arguments))
                     ((stringp datum)
                      (make-condition 'simple-condition
                         :format-string datum
                         :form-arguments arguments))
                     (t (error "Bad argument to Break: ~S" datum))))
    (proceed (condition)
        :test (lambda (ignore) t)
        :report "Return from Break."
        condition)))
```

The following are auxiliary functions to be used with the proceed-case form:

proceed-case-name proceed-case [Function]

The proceed-case-name function returns the name of proceed-case or NIL if it is not named.

compute-proceed-cases [Function]

The compute-proceed-cases function returns a list of proceed cases which are available in the current dynamic extent.

The list which results from a call to compute-proceed-cases is ordered so that the innermost (that is, most recently established) proceed cases are nearer the head of the list.

find-proceed-case name [Function]

The find-proceed-case function searches for a proceed case structure with name as its proceed-function-name and which is in the current dynamic extent.

If name is a defined proceed function name, then the most recently established proceed case with that name is returned. NIL is returned if no such proceed case is found.

If name is a proceed case, then it is simply returned unless it is not currently valid for use. In that case, NIL is returned.

invoke-proceed-case proceed-case condition &rest values [Function]

The invoke-proceed-case function transfers control to the given proceed-case, passing values as arguments. Proceed-case must be a proceed case structure or the name of a defined proceed function which is valid in the current dynamic context. If the argument is not valid, an error is signaled.

define-proceed-function name [keyword value]* &rest variables [Special Form]

The define-proceed-function special form defines a function called name which will proceed a condition. The proceed function takes a required argument of a condition and optional arguments which are given by variables.

The variable condition is bound to the condition object so that it is accessible during the initialization of the optional arguments.

Each element of variables is either:

1. variable-name
2. (variable-name initial-value)

If initial-value is not supplied, it defaults to NIL.

Keyword and value pairs are the same as those which are defined for clauses of the proceed case form: :TEST, :CONDITION, :REPORT-FUNCTION, and :REPORT.

The following examples demonstrate some possible proceed functions which might be useful in conjunction with a bad-food-color error:

```
(define-condition bad-food-color error food (color 'green))

(define-proceed-function use-food
    :report "Use another food."
    (food (read-typed-object 'food "Food to use instead: ")))

(define-proceed-function use-color
    :report "Change the food's color."
    (color (read-typed-object 'food "Color to make the food: ")))
```

The following sample condition handler uses the use-food function defined above. Notice that, although the define-proceed-function only specifies one variable, food, the invocation of the use-food function can take two variables, a condition and food. Condition is an optional argument. If it is not provided, it defaults to NIL.

```
(defun maybe-use-water (condition)
   (if (eq (bad-food-color-food condition) 'milk)
       (use-food condition 'water)))
```

The handler defined above might be associated with the bad-food-color condition using condition-bind:

```
(condition-bind ((bad-food-color #'maybe-use-water))
    (some-computation))
```

If during the execution of some-computation a bad-food-color condition is signaled, the handler maybe-use-water inspects the condition and, if the food specified in the condition is milk, it invokes the proceed function use-food.

If a named proceed function is invoked in a context in which there is no active proceed case by that name, the proceed function simply returns NIL. So, for example, in the following pair of handlers, the first is equivalent to the second except that it is less efficient:

```
#'(lambda (condition)
    (cond ((find-proceed-case 'use-food condition)
           (use-food condition 'chocolate))
          ((find-proceed-case 'use-color condition)
           (use-color condition 'orange))))
```

```
#'(lambda (condition)
    (use-food condition 'chocolate)
    (use-color condition 'orange))
```

The Condition-Case Form

condition-case form &rest clauses [Special Form]

The condition-case executes the given form. Each clause has the form:

```
(type [var] . body)
```

If a condition is signaled during the execution of form which is not handled by an intervening handler, and if there is an appropriate clause for that condition, that is, one for which (typep condition type) is true, then control is transferred to the body of the relevant clause. Var is bound to the condition which was signaled. If no condition is signaled, the values returned from the execution of form are returned by condition-case.

If var is not needed, it may be omitted.

Type may also be a list of types, in which case it catches conditions of any of the specified types.

The following example demonstrates the use of condition-case:

```
(condition-case (open *the-file* :direction :input)
    (file-error (condition)
        (format t "~&Open failed: ~A~%" condition)))
```

The Catch-Error-Abort Macro

catch-error-abort print-form &body forms [Macro]

The catch-error-abort macro sets up a proceed-case context for the proceed function error-abort (described later in this chapter).

If no error-abort is done while executing forms, all values returned by the last form in forms are returned. If an error-abort transfers control to this catch-error-abort, two values are returned: NIL and the condition which was given to error-abort (or NIL if none was given).

catch-error-abort could be defined by:

```
(defmacro catch-error-abort (print-form &rest forms)
    `(proceed-case (progn ,@forms)
       (error-abort (condition)
          :report ,print-form
          :test (lambda (ignore) t)
          (values nil condition))))
```

The Error-Abort Function

error-abort &optional condition [Function]

The error-abort function transfers control to the innermost catch-error-abort form, causing it to return NIL immediately.

It is not usually useful to specify condition. This is because the default test for error-abort unconditionally returns true and all catch-error-abort forms are therefore likely to be visible. The only such forms which might not be visible are those which override the default test. In that rare case, specifying condition might make a difference.

error-abort could be defined as:

```
(define-proceed-function error-abort
    :report "Abort."
    :test (lambda (ignore) t))
```

The Ignore-Errors Macro

ignore-errors & body forms [Macro]

The ignore-errors macro executes its body in a context which handles errors of type error by returning control to this form. If no error is signaled, any values returned by the last form are returned by ignore-errors. Otherwise NIL is returned. This is the same as:

```
(condition-case (progn forms)
    (error () nil))
```

Specialized Error-Signalling Macros

The check-type, assert, etypecase, ctypecase, ecase, and ccase macros are designed to make it convenient for you to insert error checks into code. The following descriptions are brief summaries of the information in *Common LISP, The Language.* For a complete description, refer to that source.

The Check-Type Macro

check-type place typespec &optional string [Macro]

The check-type macro signals an error if the contents of place are not of the desired type. If you continue from this error, you are asked for a new value; check-type stores the new value in place and starts over, checking the type of the new value and signaling another error if it is still not of the desired type. Subforms of place can be evaluated multiple times because of the implicit loop generated. Check-type returns NIL.

The Assert Macro

assert test-form [({place}*) [string {arg}*]] [Macro]

The assert macro signals an error if the value of test-form is NIL. Continuing from this error allows you to alter the values of some variables, and assert then starts over, evaluating test-form again. The assert macro returns NIL.

The Etypecase Macro

etypecase keyform {(type {form}*)}* [Macro]

This control construct is similar to typecase, but no explicit otherwise or t clause is permitted. If no clause is satisfied, etypecase signals an error. You cannot continue from this error. The name of this function stands for "exhaustive type case".

The Ctypecase Macro

ctypecase keyplace {(type {form}*)}* [Macro]

This control construct is similar to typecase, but no explicit otherwise or t clause is permitted. The keyplace must be a generalized variable reference acceptable to setf. If no clause is satisfied, ctypecase signals an error. Continuing from this error causes ctypecase to accept a new value from you, store it into keyplace, and start over, making the type tests again. Subforms of keyplace can be evaluated multiple times. The name of this function stands for "continuable exhaustive type case".

The Ecase Macro

ecase keyform {({({key}*) | key} {form}*)}* [Macro]

This control construct is similar to case, but no explicit otherwise or t clause is permitted. If no clause is satisfied, ecase signals an error. You cannot continue from this error. The name of this function stands for "exhaustive case".

The Ccase Macro

ccase keyform {({({key}*) | key} {form}*)}* [Macro]

This control construct is similar to case, but no explicit otherwise or t clause is permitted. The keyplace must be a generalized variable reference acceptable to setf. If no clause is satisfied, ccase signals an error. Continuing from this error causes ccase to accept a new value from you, store it into keyplace, and start over, making the clause tests again. Subforms of keyplace may be evaluated multiple times. The name of this function stands for "continuable exhaustive case".

Predefined Condition Types

Predefined condition types are supplied to the user. The default condition type for signal and break is simple-condition, for warning is simple-warning and for error and cerror is simple-error. The predefined condition types are described as follows:

Condition [Type]

All types of conditions, whether error or non-error must inherit from this type.

Warning [Type]

All types of warnings should inherit from this type. This is a subtype of condition.

Serious-condition [Type]

Any condition, whether error or non-error, which should enter the debugger when signaled but not handled, should inherit from this type. This is a subtype of condition.

Error [Type]

All types of error conditions inherit from this condition. This is a subtype of condition.

Simple-condition [Type]

Conditions signaled by signal or break when given a format string as a first argument are of this type. This is a subtype of condition. The init keywords :FORMAT-STRING and :FORMAT-ARGUMENTS are supported.

Simple-warning [Type]

Conditions signaled by warn when given a format string as a first argument are of this type. This is a subtype of warning. The init keywords :FORMAT-STRING and :FORMAT-ARGUMENTS are supported.

Simple-error [Type]

Conditions signaled by error and cerror when given a format string as a first argument are of this type. This is a subtype of error. The init keywords :FORMAT-STRING and :FORMAT-ARGUMENTS are supported.

Stepping

step form [Macro]

Step is a macro which allows you to interactively single-step through the evaluation of forms. When step is turned on, each successive sub-form is evaluated by a read-eval-print loop. Before each cycle of the loop, information about the current step is printed out, and the stepper enters a break where you have available several alternatives about how to proceed. These are shown in table 26-1.

If step is called with the argument t, single step evaluation is turned on globally. Any subsequent forms which are read from the terminal (or a file) are evaluated using single step mode until the stepper is turned off.

If step is called with the argument NIL, single step evaluation is turned off globally.

If step is called with the argument function-name+ (one or more function names), single step evaluation is turned on each time one of the specified functions is invoked, but is off during all other evaluations.

If step is called with any expression (cons object) as its argument, that expression is evaluated in single step mode.

Table 26-1. Step Commands

Command Name	Action
N (next)	Evaluate current expression in step mode.
S (skip)	Evaluate current expression without stepping.
M (macro)	Step through a macroexpansion.
Q (quit)	Turn stepper off and finish evaluation.
P (print)	Print the current expression.
B (break)	Enter debugger break loop.
E (eval)	Prompt for an arbitrary expression which is evaluated in the current environment.
H (help)	Print the list of available commands.
R (return)	Prompt for an arbitrary value to return as the result of evaluating the current expression.
A (abort)	Throw to the top level, abandoning the current computation.

Tracing

When a function is traced, invoking that function causes information about the call, the arguments, and the eventually returned values to be printed to the stream that is the value of *trace-output*. The value of *trace-output* can be initialized by setting it to an open stream or by using the Common LISP command with-open-file. The default stream for *trace-output* is *standard-output*. To trace a function, the trace macro is used. To stop tracing of a function, the untrace macro is used.

The Trace Macro

trace {function-name | (function-name option-list)}* [Macro]

The trace macro is used to trace a function.

Calling trace with no arguments returns the list of currently traced functions. For example, if there are currently two functions being traced, foo and bar, the call and response is:

```
> (trace)
(foo bar)
```

Calling trace with one or more function names causes those functions to be traced and to be added to the list of traced functions. As an example of the information provided by tracing a function, given the following defined functions:

```
(defun testfun (number1 number2)
    (cond ((> number1 0)
           (highest number1 number2))
          (t 0)))

(defun highest (num1 num2)
    (if (>= num1 num2)
        num1
        num2))
```

if the functions testfun and highest are traced:

```
>(trace testfun highest)
```

then invoking testfun:

```
>(testfun 3 5)
```

produces the following output:

```
 1: (TESTFUN 3 5)
    2: (HIGHEST 3 5)
    2: returned 5
 1: returned 5
5
```

Trace also allows a number of options to be associated with a traced function. These options are described below:

The syntax for trace is:

```
(trace function-name*)   or

(trace function-with-option-list*).
```

The syntax for function-with-option-list is:

```
(function-name  {:keyword value}+).
```

An example of a complex trace invocation is:

```
(trace (highest :wherein testfun :print ("From testfun")))
```

The optional keywords and values are:

:condition

Specifies a form to evaluate at each entry to the function. If the form evaluates to true, the normal trace information is printed. If it evaluates to false, nothing is printed.

:break

Specifies a form to evaluate at each entry to the function. If the form evaluates to true, a break occurs and the system enters the debugger. If it evaluates to false, execution continues normally.

:break-after

Specifies a form to evaluate before exiting from a function. If the form evaluates to true, a break occurs and the system enters the debugger. If it evaluates to false, execution continues normally.

:break-all

Specifies a form to be used as an argument to both the break and break-after options.

:wherein

Specifies a function name or a list of function names. The traced function only prints its normal trace information when it is called by one of the specified functions. When it is called from any other function, no trace occurs.

To illustrate, in the example above if the call to trace is:

```
(trace (highest :wherein testfun))
```

then invoking testfun: (testfun 3 5), produces:

```
  1: (HIGHEST 3 5)
      1: returned 5
5
```

However, if highest is invoked directly: (highest 3 5), no trace information is printed, since the call did not occur from within the testfun function.

:print

Specifies a list of forms to evaluate and display at each entry to the function.

:print-after

Specifies a list of forms to evaluate and display at each exit from the function.

:print-all

Specifies a list of forms to be used both as an argument to both the print and print-after options.

Tracing a function which is already traced does not have a noticeable effect unless there are options associated with the trace. Only options specified by the new trace are in effect; any old options are cancelled.

The Untrace Macro

untrace function-name [Macro]

The untrace macro specifies that the function or functions be no longer traced.

Calling untrace with no arguments causes all currently traced functions to be no longer traced.

Calling untrace with one or more function names causes those functions to be no longer traced.

Untracing a function that is not currently being traced displays a message but does not affect the function or any traced functions.

Debugging

LISP enters the debugger when a condition is signaled that is not handled by a condition handler, when a break occurs, or when the function user-break-2 is invoked. The entry point to the debugger is through the debug function.

The debugger has two main capabilities:

- Assisting you in examining the state of the system at the time of an error or call to the debugger. You must be able to examine the calling stack and variable values.
- Allowing you to modify the state of the system and/or continue execution in cases where this is feasible. This gives you the ability to interactively "try out" hypotheses about what is wrong.

The Debug Function

debug condition [Function]

The debug function enters the debugger directly.

If the special variable *abort-debug* has the value t, condition is printed out and control is returned to the top level Lisp.

If *abort-debug* is NIL, condition is printed out and the interactive debugger is entered.

The Debugger Commands

Common Lisp does not completely define the functionality of an interactive debugger. The following paragraphs describe a set of debugger commands which provide the ability to examine and modify the system state. The commands and their abbreviations are shown in table 26-2.

Table 26-2. Debugger Commands

Command Name	Command Abbreviations
Abort-debugger	abort a
Backtrace	bact bt
Backtrace-all	baca ba
Backtrace-full	bacf bf
Change-argument	chaa ca
Change-variable	chav cv
Evaluate-expression	evale eval e
Help	help h
Go-to-frame	goto go g
Make-invisible-to-backtrace	makib invis i
Make-visible-to-backtrace	makvb vis v
Move-bottom	movb bottom b
Move-down	movd down d
Move-down-all	movda da
Move-top	movt top t
Move-up	movu up u
Move-up-all	movua ua
Print-arguments	pria pa
Print-condition	pric pc
Print-frame	prif pf
Print-full-frame	priff pff
Print-variables	priv pv
Proceed-debugger	proc proceed p
Proceed-with-returned-value	prowrv rv
Search-frame	search s
Throw-to-a-tag	thrtt throw tt
Quit-debugger	quit q

Control Commands

The debugger control commands are as follows:

Quit-debugger

Leave the current debugger computation and return control to the last read-eval-print loop. This may be the top level of Lisp, or some other read-eval-print loop, such as an earlier invocation of the debugger.

Abort-debugger

Returns control to Lisp command level.

Proceed-debugger

Attempts to continue execution of the program.

Proceed-with-returned-value &optional frame-name value

The program continues executing with value used as the return value of the function frame-name. If frame-name or value is not supplied, you are prompted for it. Frame-name is restricted to be the name of an interpreted function currently on the stack. You cannot return from a compiled function in this way. Appropriate functions can be determined by looking for frames labeled as "blocks" in the debugger's backtrace (described below).

Throw-to-tag &optional tag value

Throws value to tag. If the arguments are not supplied, you are prompted for them. Attempting to throw to a non-existent or illegal tag causes an error.

Stack Examination Commands

These commands allow you to examine the calling sequence of functions. Normally, you are not interested in system functions, and these are not visible. However, they can be examined by using the backtrace-all command. The commands make-invisible-to-backtrace and make-visible-to-backtrace allow you to make any function invisible or visible to the normal backtrace command. The stack examination commands are as follows:

Make-invisible-to-backtrace &rest names

Makes the functions specified by names invisible to the backtrace function.

Make-visible-to-backtrace &rest names

Makes the functions names visible to the backtrace function.

Backtrace &optional frame-number

Prints names of visible functions on the stack, starting with frame-number. If frame-number is not specified, it defaults to the current frame. For example:

```
debug1? bt
      primt #30 : BREAK
      block #19 : FUNCTION2
      block #4 : FUNCTION1
```

indicates that the debugger was entered while the program was executing the primitive function break which was called from within the function function2, which was in turn called from function function1. Primitive functions are system functions or compiled functions and are indicated in the backtrace by the primt heading, while interpreted functions are indicated by the block heading.

Backtrace-full &optional frame-number

Displays the name of the function, argument names and values, and variable names and values for each frame on the stack starting with frame-number. If frame-number is not specified, it defaults to the current frame. For example, given the following function definitions:

```
(defun test1 (z)
   (let ((local-var (+ z 5)))
   (test2 local-var z)))
(defun test2 (var1 var2)
   (break)
   (+ var1 var2))
```

and the function call: (test1 5), the full backtrace is:

```
debug1? bf
   primt #30 : BREAK
   block #19 : TEST2
      Arguments:
           VAR1:  10
           VAR2:  5
   block #4 : TEST1
      Arguments:
           Z:  5
      Local variables:
           LOCAL-VAR:  10
```

Arguments and variables are not shown for the function which caused entry to the debugger (in this case, break). The names of arguments for primitive frames are not known; therefore they are referred to by number only. For example, if test2 was a compiled function, the print-out for that frame would be:

```
primt #19:  TEST2
   Arguments:
      ARG #1: 10
      ARG #2: 5
```

Backtrace-all &optional frame-number

Displays the names of all active functions starting with frame-number on the stack, including those on the invisible function list. If frame-number is not specified, it defaults to the current frame. For example, with the same function definitions shown above, the backtrace-all command would produce the following output:

```
debug1? ba
   primt #30 : BREAK
   primt #26 : BLOCK
   block #19 : TEST2
   primt #12 : LET
   primt #8 : BLOCK
   block #4 : TEST1
```

Frame Movement Commands

Several commands allow you to move to any frame on the stack. Moving to a frame makes that frame the debugger's current frame. The frame movement commands are as follows:

Go to frame &optional frame-number

Makes the frame associated with frame-number the current frame. You are prompted for a number if it is not supplied.

Move down &optional n

Moves down the stack n frames, or one frame if n is not supplied.

Move down all &optional n

Moves down the stack n frames, including 'invisible' frames. If n is not specified, it defaults to 1.

Move up &optional n

Moves up the stack n frames or to the next frame if n is not specified.

Move up all &optional n

Moves up the stack n frames, including 'invisible' frames. If n is not specified, it defaults to 1.

Move top

Moves to the top of the stack. Normally the functions called from within the debugger are not visible to you.

Move bottom

Moves to the bottom of the stack.

Search-frame &optional function-name

Attempts to find a frame for function-name. Function-name can be a string or symbol. If no symbol or string is supplied, you are prompted to supply one. Search sets the current frame to the highest level frame which satisfies the search.

Print Commands

The print commands used while in the debugger are as follows:

Print condition

Prints the original message that was given when the debugger was invoked.

Print frame &optional frame-number

Prints the name of the frame at frame-number. If frame-number is not supplied, it defaults to the current frame.

Print arguments &optional frame-number

Prints the argument names and values of the frame specified by frame-number. If frame-number is not supplied, it defaults to the current frame.

Print full frame &optional frame-number

Prints information about the frame specified by frame-number: the frame name, its argument names and values, and its local variable names and values. If frame-number is not specified, it defaults to the current frame.

Print variables &optional frame-number

Prints the local variable names and values of the frame at frame-number or, if frame-number isn't specified, the current frame.

Modification Commands

Several commands allow you to modify values in the stack and either recompute or continue with the modified values. The modification commands are as follows:

Change-argument &optional n value

Allows you to change the value of the nth argument of the current frame to value. If n and value are not supplied, you are prompted for them. This command is included primarily because you don't always know the names of the arguments to system functions or primitive functions and can't easily use the change-variable command which requires a name.

Change-variable &optional variable-name value

Allows you to change the value of the named variable variable-name to value. If the arguments are not supplied, you are prompted for them. Any special or lexical variable can be modified.

Miscellaneous Commands

A few miscellaneous commands can be used while in the debugger. They are as follows:

Evaluate-expression &optional expression frame-number

The expression is evaluated in the context of the frame specified by frame-number. If frame-number is not specified, it defaults to the current frame. If no expression is supplied, you are prompted for it.

Help &optional option

Prints out help information on debugger commands. If option is not supplied, a list of debugger commands and their abbreviations is printed. If option is the word 'full', all commands are printed with a description of their functionality. If option is the name or abbreviation of a specific command, a description of that command is printed.

Loading Speed 27

Loading Speed 27

This chapter discusses improving the loading speed of large LISP applications.

When you are creating a large LISP application, the goal is to save the state of the LISP application after it is initialized so that the loading is faster. Do this by following these steps:

1. Load the application.
2. Initialize the application.
3. Save the state of the LISP system at that point.

By following these steps when the application is built, loading is faster because you do not need to repeat the initialization. You can build your application in this way by using the load-entry function, in conjunction with the $save-lisp function described in chapter 25, Miscellaneous Features.

The following section describes the load-entry function.

The Load-Entry Function

The load-entry function allows you to improve the loading speed while building large applications. The format of the function is:

load-entry object-library entry-point-name [Function]

The load-entry function executes the entry point in the object library specified. Both arguments of the function must be strings; you must capitalize the letters in the entry-point-name argument.

The compiler produces an object library holding one module. Multiple libraries can be combined to produce a single library. Such libraries can be loaded using the load function.

The compiler divides its loading activity into three parts. The load function first executes a procedure that has the same name as the MODULE on the library. Then, this procedure calls two more routines. These routines are named the same as the MODULE name with $SLOW and $FAST appended to the end.

For example, if you compile a file with the :module-name of "MYMOD", the object library module MYMOD contains three procedures called MYMOD, MYMOD$FAST, and MYMOD$SLOW:

- The procedure MYMOD just calls MYMOD$FAST and MYMOD$SLOW.
- The procedure MYMOD$FAST loads the parts that the $save-lisp function cannot preserve. Addresses of compiled code in the symbol table cannot be saved by $save-lisp because in the CYBER 180 architecture those addresses are not valid when the $save-lisp image is later executed. MYMOD$FAST is responsible for populating the symbol table with addresses of compiled code.
- The procedure MYMOD$SLOW loads the parts that the $save-lisp function can preserve. Since MYMOD$SLOW will probably use functions initialized by MYMOD$FAST, MYMOD$SLOW cannot be executed prior to MYMOD$FAST. You should never use the load-entry function to load MYMOD$SLOW.

To save the work done by the initialization of the application, you may want to load and initialize large chunks of code and then perform the $save-lisp function. When you reenter LISP, you then only need to initialize the pieces not saveable by $save-lisp. This can be done by either loading the files again or by just executing the $FAST procedures. You can use the load-entry function to execute the $FAST procedures.

When loading an application, keep in mind the following:

- If a file is changed and recompiled, you must create a new $save-lisp image. Also, you must recreate the $save-lisp image whenever a new version of LISP is installed.
- LISP currently does no timestamp checking to ensure that the core image and the object libraries match. You can, however, create an object library that holds all the loaded code and the result of $save-lisp on the same library.
- When compiling the application, you should use valid names for the :module-name argument on every call to the compile-file function. For more information about valid names, see chapter 25, Miscellaneous Features. If a module name is invalid, the compiler renames the module; you must use this new name as the second argument of the load-entry function. LISP prints a compiler warning message with the new name assigned to the module. When in doubt, use the NOS/VE command DISPLAY_OBJECT_LIBRARY with DISPLAY_OPTION = ENTRY_POINT to find the module name that the compiler assigned.

Example of Using the Load-Entry Function

Assume that the source code for your LISP application is in two files named FILEA and FILEB, and that the start-application function initializes the application to be ready to use. You have two choices for building and invoking the application:

1. The slower method:
   a. Compile the application.
   b. Invoke the application by loading the compiled code and calling the start-application function.
2. The faster method:
   a. Compile the application, by calling the start-application and $save-lisp functions.
   b. Invoke the application by executing the file created by $save-lisp and using load-entry to complete the loading.

The following example illustrates the faster method:

```
collect_text filea
(DEFUN AFUN (X)
  (FORMAT T "~% AFUN was called with arg= ~S" X)
  (FORCE-OUTPUT))
**


collect_text fileb
(DEFUN BFUN (X)
  (FORMAT T "~% BFUN was called with arg= ~S" X)
  (FORCE-OUTPUT))
(DEFUN START-APPLICATION ()
  (FORMAT T "~% Starting a 60 Second initialization ...") (FORCE-OUTPUT)
  (SLEEP 60) (SETQ *PRINT-BASE* 16)
  (FORMAT T "~% Initialization is finally finished. ") (FORCE-OUTPUT))
;   The (SLEEP 60) is just used to illustrate a lengthy
;   initialization function.
**


collect_text build
(compile-file "filea" :module-name "aamod" :output-file "liba" :load t)
(compile-file "fileb" :module-name "BBMOD" :output-file "libb" :load t)
(start-application)
($save-lisp)
(exit)
**
lisp i=build

creol
addm liba
comm libb
comm $local.lisp_base_system_spaces
genl mylib
quit

"   NOTE: Before the following steps, check the output from LISP
"   compilations to find the names that LISP assigned your modules.
"   Be sure to use those (if they differ from the names you specified),
"   as the second argument of the calls to lisp::load-entry.
"


collect_text in
(lisp::load-entry "mylib" "MN$1$FAST")
(lisp::load-entry "mylib" "MN$32$FAST")
(AFUN 15) (FORCE-OUTPUT)
; Notice that the 60 seconds for initialization does not occur, but
;   the installation done before IS in effect... as you can see by
;   the fact that the result of (AFUN 15) prints as F instead of 15
;   (initialization had set *print-radix* to 16).
(quit)
**

exet mylib l=$system.lisp.bound_product sp=lip$lisp_system p='i=in'
```

Appendixes

Glossary A

This appendix defines terms used in Common LISP specifications. There is no corresponding chapter in *Common LISP*.

A

Array

A multidimensional collection of data elements. Each element is accessed using unique positional descriptors called indices.

Atom

A general term for a symbol, number, string, or array. Anything that is not a cons.

B

Binding

(1) The LISP-object currently associated with a symbol. (2) The process of associating a LISP-object with a symbol.

Bound Symbol

A symbol that is associated with a LISP-object. A bound symbol that can be evaluated because it is currently associated with a value.

C

CAR

The first portion of a cons cell. The CAR contains a LISP-object.

CDR

The portion of a cons cell not included in the CAR. The CDR contains a LISP-object.

Cons Cell

The fundamental structure of data storage. A cons cell is an object having two components called the CAR and CDR.

Constant

A symbol whose binding does not change.

D

Dotted List

A list whose final CDR is not NIL.

Dotted Pair

A cons cell construct whose CDR is not a cons cell.

Dynamic Extent

When an entity can be referenced any time between its establishment and when it completes or is terminated. Entities with dynamic extent obey stacking rules paralleling the nested executions of their establishing constructs.

Dynamic Scoping

Having indefinite scope and dynamic extent.

E

Element

The basic unit of data within a list. An element can be another list (including the empty list NIL), a cons cell, an atom, or an array. Any LISP-object can be an element.

Environment

The present state of the LISP system. The environment includes all bindings of LISP-objects.

Evaluation

The process of determining the value of a LISP-object.

Extent

See Dynamic Extent and Indefinite Extent.

F

Form

The fundamental entity of LISP syntax. A LISP-object meant to be evaluated. When evaluated, forms produce values and side effects. There are three types of forms: self-evaluating (such as numbers), symbols, and lists.

Function

An instance of an algorithm. Functions accept zero or more LISP-objects as arguments and produce a LISP-object as a result.

G

Garbage Collection

Process of reclaiming LISP-objects that have been discarded by LISP.

I

Indefinite Extent

When an entity exists as long as it is possible to reference it. Compare to Dynamic Extent.

Indefinite Scoping

Scoping that is not lexical. References can occur anywhere within a program.

L

Lambda Notation

(1) A method of defining a function in-place. The function definition is temporary and does not exist outside of the form in which it appears. (2) A function type within LISP.

Lexical Scoping

When a variable must appear textually within a function. Embedded lambda expressions do not effect the scope of variables.

LISP-Object

A general term referring to any LISP data item.

List

The basic unit of data grouping within LISP, and the most common data type. A list is a cons.

M

Macro

Mechanism that replaces one list with another.

N

NIL

The empty list, designated by (). The empty list contains an infinite number of empty lists. NIL is used to represent logical falsehood.

P

Package

Group of logically related LISP-objects. Packages provide restricted access to secure objects and allow name hiding. In effect, a package is a subspace within a LISP workspace. Access to objects within a package is under the control of the package.

Primitive Function

A function that is built into LISP. Primitive functions are associated with a LISP symbol.

Print Name

A string holding the external representation of a symbol; for example, the characters displayed on a user's terminal screen. Sometimes referred to as pname.

Property List

Traditionally, a list that holds user-defined attributes of a symbol. A globally accessible LISP-object associated with each symbol, and sometimes referred to as plist.

Pseudo Function

A function executed for side effects and not for the value returned.

Q

Quote

A special form that returns its input without evaluation. Also, a syntax that allows symbols to be manipulated without evaluation.

R

Reader

The portion of the LISP evaluator code that processes input for correct syntax, and so forth. The reader collects input characters into a printed representation of a LISP-object builds the object, and returns its value.

Recursion

The process of invoking a function from within that function. Recursion is closely related to mathematical induction.

S

S-Expression

A synonym for symbolic expression and LISP-object.

Scope

See Indefinite Scoping or Lexical Scoping.

Semantics

The meaning of a syntactically correct statement. LISP has semantic rules which are used to decide whether functions can be applied to arguments.

Side Effects

When a function causes a change in the LISP environment that remains in effect after the function completes and the effect is not returned as an explicit result.

Special Form

A form that does not have its arguments automatically evaluated.

String

A finite ordered sequence of characters. Under NOS/VE, a string cannot exceed 256 characters.

Symbol

A fundamental data type. A symbol is associated with a value, a print name, a property list, a function definition, and a package.

Syntax

Rules defining whether a statement is well formed.

T

True List

A list whose final CDR is NIL.

V

Value

(1) The LISP-object bound to a symbol. (2) The LISP-object returned by evaluating a function.

Variable

A symbol with an associated value.

Related Manuals B

Table B-1 lists all manuals that are referenced in this manual or that contain background information. A complete list of NOS/VE manuals is given in the NOS/VE System Usage manual. If your site has installed the online manuals, you can find an abstract for each NOS/VE manual in the online System Information manual. To access this manual, enter:

```
/explain
```

Ordering Printed Manuals

You can order Control Data manuals through Control Data sales offices or through:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

Table B-1. Related Manuals

Manual Title	Publication Number	Online Title
LISP Manuals:		
Common LISP, The Language	60486201	
LISP Language Definition Usage Supplement	60486213	
Background Material (Access as Needed):		
NOS/VE System Usage	60464014	
NOS/VE Commands and Functions	60464018	SCL
Additional References:		
NOS/VE File Editor Tutorial/Usage	60464015	
CDCNET Access Guide	60463830	CDCNET_ACCESS
NOS/VE Object Code Management	60464413	OCM
Math Library for NOS/VE	60486513	

Character Set C

This appendix defines the ASCII character set as used by NOS/VE software and LISP. There is no corresponding chapter in *Common LISP*.

NOS/VE supports the American National Standards Institute (ANSI) standard ASCII character set (ANSI X3.17-1977). NOS/VE represents each 7-bit ASCII code in an 8-bit byte. The 7 bits are right-justified in each byte. For ASCII characters, the leftmost bit is always zero.

In addition to the 128 ASCII characters, NOS/VE allows use of the leftmost bit in an 8-bit byte for 256 characters. The use and interpretation of the additional 128 characters is user-defined.

LISP uses ASCII characters as described in chapter 22 of *Common LISP*. For your convenience, the following table indicates implementation-dependent #\ definitions.

Table C-1. ASCII Character Set Table

Decimal	ASCII Code Hexa-decimal	Octal	Graphic or Mnemonic	ASCII Name or Meaning	LISP Definition
000	00	000	NUL	Null	
001	01	001	SOH	Start of heading	
002	02	002	STX	Start of text	
003	03	003	ETX	End of text	
004	04	004	EOT	End of transmission	
005	05	005	ENQ	Enquiry	
006	06	006	ACK	Acknowledge	
007	07	007	BEL	Bell	
008	08	010	BS	Backspace	#\Backspace
009	09	011	HT	Horizontal tabulation	#\Tab
010	0A	012	LF	Line feed	#\Linefeed
011	0B	013	VT	Vertical tabulation	
012	0C	014	FF	Form feed	#\Page
013	0D	015	CR	Carriage return	#\Return
014	0E	016	SO	Shift out	
015	0F	017	SI	Shift in	
016	10	020	DLE	Data link escape	
017	11	021	DC1	Device control 1	
018	12	022	DC2	Device control 2	
019	13	023	DC3	Device control 3	
020	14	024	DC4	Device control 4	
021	15	025	NAK	Negative acknowledge	
022	16	026	SYN	Synchronous idle	
023	17	027	ETB	End of transmission block	

(Continued)

Table C-1. ASCII Character Set Table *(Continued)*

Decimal	ASCII Code Hexa-decimal	Octal	Graphic or Mnemonic	ASCII Name or Meaning	LISP Definition
024	18	030	CAN	Cancel	
025	19	031	EM	End of medium	
026	1A	032	SUB	Substitute	
027	1B	033	ESC	Escape	
028	1C	034	FS	File separator	
029	1D	035	GS	Group separator	
030	1E	036	RS	Record separator	
031	1F	037	US	Unit separator	#\Newline
032	20	040	SP	Space	#\Space
033	21	041	!	Exclamation point	
034	22	042	"	Quotation marks	
035	23	043	#	Number sign	
036	24	044	$	Dollar sign	
037	25	045	%	Percent sign	
038	26	046	&	Ampersand	
030	27	047	'	Apostrophe	
040	28	050	(	Opening parenthesis	
041	29	051	)	Closing parenthesis	
042	2A	052	*	Asterisk	
043	2B	053	+	Plus	
044	2C	054	,	Comma	
045	2D	055	-	Hyphen	
046	2E	056	.	Period	
047	2F	057	/	Slant	
048	30	060	0	Zero	
049	31	061	1	One	
050	32	062	2	Two	
051	33	063	3	Three	
052	34	064	4	Four	
053	35	065	5	Five	
054	36	066	6	Six	
055	37	067	7	Seven	
056	38	070	8	Eight	
057	39	071	9	Nine	
058	3A	072	:	Colon	
059	3B	073	;	Semicolon	
060	3C	074	<	Less than	
061	3D	075	=	Equals	
062	3E	076	>	Greater than	
063	3F	077	?	Question mark	

(Continued)

Table C-1. ASCII Character Set Table *(Continued)*

Decimal	ASCII Code Hexa-decimal	Octal	Graphic or Mnemonic	ASCII Name or Meaning	LISP Definition
064	40	100	@	Commercial at	
065	41	101	A	Uppercase A	
066	42	102	B	Uppercase B	
067	43	103	C	Uppercase C	
068	44	104	D	Uppercase D	
069	45	105	E	Uppercase E	
070	46	106	F	Uppercase F	
071	47	107	G	Uppercase G	
072	48	110	H	Uppercase H	
073	49	111	I	Uppercase I	
074	4A	112	J	Uppercase J	
075	4B	113	K	Uppercase K	
076	4C	114	L	Uppercase L	
077	4D	115	M	Uppercase M	
078	4E	116	N	Uppercase N	
079	4F	117	O	Uppercase O	
080	50	120	P	Uppercase P	
081	51	121	Q	Uppercase Q	
082	52	122	R	Uppercase R	
083	53	123	S	Uppercase S	
084	54	124	T	Uppercase T	
085	55	125	U	Uppercase U	
086	56	126	V	Uppercase V	
087	57	127	W	Uppercase W	
088	58	130	X	Uppercase X	
089	59	131	Y	Uppercase Y	
090	5A	132	Z	Uppercase Z	
091	5B	133	[	Opening bracket	
092	5C	134	\	Reverse slant	
093	5D	135	]	Closing bracket	
094	5E	136	^	Circumflex	
095	5F	137	_	Underline	
096	60	140	`	Grave accent	
097	61	141	a	Lowercase a	
098	62	142	b	Lowercase b	
099	63	143	c	Lowercase c	
100	64	144	d	Lowercase d	
101	65	145	e	Lowercase e	
102	66	146	f	Lowercase f	
103	67	147	g	Lowercase g	

(Continued)

Table C-1. ASCII Character Set Table *(Continued)*

Decimal	ASCII Code Hexa-decimal	Octal	Graphic or Mnemonic	ASCII Name or Meaning	LISP Definition
104	68	150	h	Lowercase h	
105	69	151	i	Lowercase i	
106	6A	152	j	Lowercase j	
107	6B	153	k	Lowercase k	
108	6C	154	l	Lowercase l	
109	6D	155	m	Lowercase m	
110	6E	156	n	Lowercase n	
111	6F	157	o	Lowercase o	
112	70	160	p	Lowercase p	
113	71	161	q	Lowercase q	
114	72	162	r	Lowercase r	
115	73	163	s	Lowercase s	
116	74	164	t	Lowercase t	
117	75	165	u	Lowercase u	
118	76	166	v	Lowercase v	
119	77	167	w	Lowercase w	
120	78	170	x	Lowercase x	
121	79	171	y	Lowercase y	
122	7A	172	z	Lowercase z	
123	7B	173	{	Opening brace	
124	7C	174	|	Vertical line	
125	7D	175	}	Closing brace	
126	7E	176	~	Tilde	
127	7F	177	DEL	Delete	#\Rubout

Diagnostic Messages D

This appendix describes all diagnostic messages issued by LISP. There is no corresponding chapter in *Common LISP.*

LISP sends the diagnostic messages described in this appendix to the output (O=) file specified in the command to enter LISP. The output file also receives information summarizing such things as variable bindings in effect when the error was detected.

The NOS/VE $ERRORS file name function is not used.

No errors will abort LISP, except stack overflow.

Where ~S or ~A appears in the messages that follow, LISP fills in the appropriate information. See the description of the control-string of the format function, starting on page 385 of *Common LISP*, for the meaning of the information.

Errors

The following are possible errors that can be generated by LISP/VE.

Apply of ~S not understood as a location for Setf.

Description: You cannot specify the directive ~S as the location argument symbol in a setf macro call.

User Action: Redesign your program.

Argument is not a cons. Argument encountered is xxxxxxxx

Description: The form being evaluated requires a cons cell for the argument indicated by xxxxxxxx.

User Action: Correct the argument; check the argument for a syntax error or the form for incorrect placement. If the argument is another form, check the value it returns.

Argument is not a character. Argument encountered is xxxxxxxx

Description: The form being evaluated requires a character for the argument indicated by xxxxxxxx.

User Action: Correct the argument; check the argument for a syntax error or the form for incorrect placement. If the argument is another form, check the value it returns.

Argument is not a list. Argument encountered is xxxxxxxx

Description: The form being evaluated requires a list for the argument indicated by xxxxxxxx.

User Action: Correct the argument; check the argument for a syntax error or the form for incorrect placement. If the argument is another form, check the value it returns.

Argument is not a number. Argument encountered is xxxxxxxx

Description: The form being evaluated requires a number for the argument indicated by xxxxxxxx.

User Action: Correct the argument; check the argument for a syntax error or the form for incorrect placement. If the argument is another form, check the value it returns.

Argument is not a positive number or zero. Argument encountered is xxxxxxxx

Description: The form being evaluated returns a negative number or a nonnumeric value for the argument indicated by xxxxxxxx. The form requires a positive or zero number.

User Action: Correct the argument; check the argument for a syntax error or the form for incorrect placement. If the argument is another form, check the value it returns.

Argument is not a positive integer. Argument encountered is xxxxxxxx

Description: The form being evaluated returns a negative number, a zero, or a nonnumeric value for the argument indicated by xxxxxxxx. The form requires a positive number.

User Action: Correct the argument; check the argument for a syntax error or the form for incorrect placement. If the argument is another form, check the value it returns.

Argument is not a primitive. Argument encountered is xxxxxxxx

Description: The form being evaluated contains another form or a value as the argument indicated by xxxxxxxx. The form requires that argument to be a Common LISP primitive.

User Action: Correct the argument; check the argument for a syntax error or the form for incorrect placement. If the argument is another form, check the value it returns.

Argument is not a proper list. Argument encountered is xxxxxxxx

Description: The form being evaluated requires a list for the argument indicated by xxxxxxxx. The argument is recognizable as a list but is improperly structured and might be infinitely recursive. The final CDR of the list is not NIL.

User Action: Correct the argument; check the list structure pointers.

Argument is not a readtable. Argument encountered is xxxxxxxx

Description: The form being evaluated requires a read table for the argument indicated by xxxxxxxx.

User Action: Correct the argument; check the argument for a syntax error or the form for incorrect placement.

Argument is not a stream. Argument encountered is xxxxxxxx

Description: The form being evaluated requires a stream name for the argument indicated by xxxxxxxx.

User Action: Correct the argument; check the argument for a syntax error or the form for incorrect placement.

Argument is not a string. Argument encountered is xxxxxxxx

Description: The form being evaluated requires a string for the argument indicated by xxxxxxxx.

User Action: Correct the argument; check the argument for a syntax error or the form for incorrect placement. One or both quotation marks might be missing.

Argument is not a symbol. Argument encountered is xxxxxxxx

Description: The form being evaluated requires a symbol for the argument indicated by xxxxxxxx.

User Action: Correct the argument; check the argument for a syntax error or the form for incorrect placement. The argument might begin with an unneeded apostrophe or might need to be quoted.

Argument is not an array. Argument encountered is xxxxxxxx

Description: The form being evaluated requires an array for the argument indicated by xxxxxxxx.

User Action: Correct the argument; check the argument for a syntax error or the form for incorrect placement.

Argument is not an integer. Argument encountered is xxxxxxxx

Description: The form being evaluated requires an integer for the argument indicated by xxxxxxxx.

User Action: Correct the argument; check the argument for a syntax error or the form for incorrect placement.

Argument is not real. Argument encountered is xxxxxxxx

Description: The form being evaluated requires a real number for the argument indicated by xxxxxxxx.

User Action: Correct the argument; check the argument for a syntax error or the form for incorrect placement. A decimal point might be missing.

Argument list is poorly formed. Argument list encountered is xxxxxxxx

Description: The arguments specified do not follow the rules of Common LISP.

User Action: Check the arguments specified to be sure they are within the bounds of the function. Reenter the argument list correctly.

Arguments are contradictory.

Description: A conflict exists in the arguments specified.

User Action: Check to see if an argument is out of bounds for the function used.

Argument must be an integer.

Description: The form being evaluated requires an integer for the argument.

User Action: Change the argument to an integer.

Arguments must be integers.

Description: The form being evaluated requires integers for the arguments.

User Action: Change the arguments to integers.

Arithmetic overflow was encountered.

Description: Evaluation of the current form (usually a function from the NOS/VE Common Math Library) stopped because the form's value cannot be properly calculated or returned.

User Action: Examine the data used by the form and correct it if possible.

Array index not recognized. Array index encountered is nnnnn

Description: The form being evaluated requires a valid integer within the array bounds for an array index. The value represented by nnnnn was found instead.

User Action: Check the index for a typographical error or transposition of index values. Check the original definition of the array's index for an error. Ensure that the index specified is within bounds for the array.

Array space is full.

Description: The number of arrays LISP can handle is determined by the data types used. The space available for arrays is full and your program attempted to define or extend an array.

User Action: Simplify your program's data use to reduce memory usage. Check for infinite recursion.

ASH, shift argument is not a fixnum.

Description: The shift argument to ASH must be a fixnum.

User Action: Change the shift argument to a fixnum.

Attempt to replacd or replaca nil is encountered.

Description: You cannot perform this operation.

User Action: Rewrite your program so that it does not use replaca or replacd on NIL.

Bad &rest or &body arg in ~S. errloc

Description: The value indicated by errloc identifies the unrecognized argument.

User Action: Replace or remove the argument.

Further Information: See *Common LISP*, page 60.

CAR only works on a CONS or NIL.

Description: The argument to CAR must be a CONS or NIL.

User Action: Change the argument to be a CONS or NIL.

CDR only works on a CONS or NIL.

Description: The argument to CDR must be a CONS or NIL.

User Action: Change the argument to be a CONS or NIL.

Comma used outside of backquote.

Description: This is incorrect syntax. Commas are allowed only within a backquoted form.

User Action: Correctly place the comma.

Cons space is full.

Description: The number of conses LISP allows depends on all of the data types used by the program. That space is full and you attempted to define another cons.

User Action: Simplify your program's data use to reduce memory usage.

COUNT must be a fixnum.

Description: The COUNT argument to ASH must be a fixnum.

User Action: Change the argument to a fixnum.

Division by zero.

Description: Division by zero is not allowed.

User Action: Correct the divisor so that it is not zero.

Dotted arglist after &AUX in ~S.

Description: You cannot use a dotted list as an argument in this position.

User Action: Change the list or reposition the argument.

Dotted arglist after &KEY in ~S.

Description: You cannot use a dotted list as an argument in this position.

User Action: Change the list or reposition the argument.

Dotted arglist terminator after &rest arg in ~S.

Description: You cannot use a dotted list as an argument in this position.

User Action: Change the list or reposition the argument.

Dotted list on stream is poorly formed.

Description: The reader found zero or more LISP objects after a dot.

User Action: Check for a typographical error in the list.

Dual wildcard mode not implemented for ABBREV.

Description: The current version of LISP does not support more than one wildcard matching character in an abbreviation.

User Action: Restate the abbreviation.

File already exists.

Description: The open function attempted to create a file that already exists.

User Action: Create another file.

File cannot be opened. xxxxxxxx

Description: The open function attempted to process the file identified by the namestring xxxxxxxx. That file is not attached to the LISP job, or is attached without a needed permission. For example, the file might be attached with only read permission and open attempted to open the file for output or for input and output.

User Action: Use the NOS/VE ATTACH_FILE SCL command to reattach the file properly.

Further Information: See the NOS/VE System Usage manual.

File does not exist or is attached in write mode.

Description: The open function attempted to process a file that does not exist, or the file is open in write mode.

User Action: If the file does not exist, you must create the file. If the file is open in write mode, close the file before processing it again.

File system resources exceeded.

Description: The total number of files NOS/VE allows you to have at the same time has been exceeded.

User Action: See the NOS/VE System Usage manual.

Function is not defined. fffff

Description: LISP has reserved the function name indicated by fffff for future implementation of an intrinsic function.

User Action: Check to see if a typographical error occurred in entering the function's name, or if the defun entry that defined the function contained an error. If you are referencing a user-defined function, rename that function.

Function is not recognized.

Description: The form being evaluated must be a valid function; LISP found the entity indicated by fffff. LISP does not have an intrinsic function and cannot find a user-defined one by that name.

User Action: Check to see if a typographical error occurred in entering the function's name, or if the defun entry that defined the function contained an error.

GCD - argument not an integer.

Description: Argument to GCD must be an integer.

User Action: Change the argument to an integer.

Go to unseened compiled tag.

Description: Go was called with an unseen tag.

User Action: Correct the program to make the matching tag lexically visible to the point of the GO.

Illegal or ill-formed DEFSETF for ~S.

Description: The format directive cannot be evaluated because the defsetf macro it references is improperly defined.

User Action: Check that the body of a complex form defsetf is correctly specified.

Illegal or ill-formed &whole arg in ~S.

Description: The format directive cannot be evaluated because of the &whole argument.

User Action: Check the function body for the proper use of the corresponding parameter.

Illegal character name encountered in reading a #\.

Description: A #\ construct can only contain a name of (string-upcase name) and the name must have the syntax of a symbol.

User Action: Check that the name you specified has a defined character object.

Illegal sharp-sign syntax.

Description: LISP does not support the # construct you specified.

User Action: Check that a font number does not appear after the #. Check for use of an unimplemented feature.

Illegal stuff after &rest arg in Define-Modify-Macro.

Description: You can specify only a symbol after an &rest lambda-list keyword. You might have omitted a subsequent lambda-list keyword.

User Action: See *Common LISP*, page 60. Reenter the macro form without extra trailing information.

Improper bounds for string comparison.

Description: The referenced strings cannot be compared within the bounds specified.

User Action: Check that you correctly specified the bounds.

Index is out of bounds.

Description: An array index is out of bounds.

User Action: Ensure that the index specified is within the array bounds.

Index must be a nonnegative integer.

Description: The array index must be a nonnegative integer.

User Action: Correct the index of the array.

Initial closing parenthesis encountered in stream zzzzzzzz

Description: This results from unbalanced parentheses. If a form with an extra closing parenthesis is entered, then following evaluation of that form, the read function finds an initial closing parenthesis.

User Action: Delete any extra closing parentheses.

Invalid argument to :direction keyword.

Description: The argument to :direction keyword is invalid. Could also be signaled by a poorly formed keyword argument to the open function. For example:

```
(open <filename> extra-arg :<keyword> :<keyword-value>)
```

User Action: See *Common LISP*, page 418, for valid keywords.

Invalid argument to :if-does-not-exist keyword is invalid.

Description: The argument to :if-does-not-exist keyword is invalid. Could also be signaled by a poorly formed keyword argument to the open function. For example:

```
(open <filename> extra-arg :<keyword> :<keyword-value>)
```

User Action: See *Common LISP*, page 418, for valid keywords.

Invalid argument to :if-exists keyword.

Description: The argument to :if-exists keyword is invalid. Could also be signaled by a poorly formed keyword argument to the open function. For example:

```
(open <filename> extra-arg :<keyword> :<keyword-value>)
```

User Action: See *Common LISP*, page 420, for valid keywords.

ISQRT: ~S argument must be a nonnegative integer.

Description: The argument to ISQRT must be a nonnegative integer. You are raising a rational number to a bignum power. Raising a rational number to a bignum power requires a large amount of cpu time and storage to calculate and store the result.

User Action: Use a nonnegative integer as the argument to ISQRT.

LCM: argument not an integer.

Description: Argument to LCM must be an integer.

User Action: Change the argument to an integer.

Macro ~S cannot be called with ~S args.

Description: You cannot nest these directives.

User Action: Redesign your program.

No bits attributes in character objects.

Description: LISP character objects do not have bit attributes.

User Action: Redesign your program.

Non-symbol &rest arg in definition of ~S.

Description: You can specify only a symbol after an &rest lambda-list keyword. You might have omitted a subsequent lambda-list keyword.

User Action: See *Common LISP*, page 60.

Non-symbol variable name in ~S.

Description: Variable names referenced by these directives must be valid LISP symbols.

User Action: Check for a syntax error. Properly define the variable name as a symbol.

Odd number of args to PSETF.

Description: The psetf macro requires an even number of arguments.

User Action: Check the form for a missing argument or a misplaced parenthesis.

Odd number of args to setf.

Description: The setf macro requires an even number of arguments.

User Action: Check the form for a missing argument or misplaced parenthesis. Reenter the macro form correctly.

Odd-list-length property list in REMF.

Description: Property lists must contain an even number of elements. The one used in the remf macro form does not meet this requirement.

User Action: Correct the list content.

Only one new-value variable allowed in DEFSETF.

Description: You specified more than one such variable in a defsetf macro form. Check for a misplaced parenthesis.

User Action: Respecify the form without extra variables.

Pathname specified is too long.

Description: The argument (usually a string) given to the open function has too many characters for a NOS/VE filename. The NOS/VE System Command Language allows a maximum of 31 characters.

User Action: Correct the argument to contain no more than 31 characters.

Poorly formed function encountered. Function is xxxxxxxx

Description: The form in the function position of the input statement is not recognized.

User Action: Ensure that the CAR of the statement is lambda and that the lambda list is properly constructed.

Poorly formed plist encountered. Plist is xxxxxxxx

Description: The property list identified as xxxxxxxx does not have the correct structure for the use made of it in the form currently being evaluated. The list might have an odd number of elements (the number of elements must always be even).

User Action: Count the elements in the property list. Correct the property list structure. Check to be sure that symbol-plist is not modified by setf.

POSITION argument must be non-negative.

Description: The POSITION argument must be a fixnum.

User Action: Change the argument to a fixnum.

Read macro context error encountered on stream zzzzzzzz

Description: A syntax error probably occurred.

User Action: Check for a missing backquote (`).

Redundant &optional flag in varlist of ~S.

Description: More than one &optional lambda-list keyword exists in the form. The beginning of the next form might be missing.

User Action: Delete the extra lambda-list keyword and any related symbol. See *Common LISP*, page 60.

Return from unseened compiled block.

Description: Call to return-from specified on block.

User Action: Correct the program to make a block construct with the specified name lexically enclose the occurrence of the return-from.

SIZE argument must be non-negative.

Description: The SIZE argument must be a fixnum.

User Action: Change the argument to a fixnum.

Size is out of bounds.

Description: The SIZE argument is out of bounds.

User Action: Correct the argument to ensure that the size is within bounds.

Space for real numbers is exhausted.

Description: The number of real numbers LISP can handle is determined by all the data types used. The space available for real numbers is full and your program attempted to define one.

User Action: Simplify your program's data use to reduce memory usage. Reduce the number of real numbers used. Check for infinite recursion.

Space for streams is exhausted.

Description: The number of streams LISP can handle is determined by all the data types used. The space available for streams is full and your program attempted to define one.

User Action: Simplify your program's data use to reduce memory usage. Reduce the number of streams used. Check for infinite recursion.

Space for symbols is exhausted.

Description: The number of symbols LISP can handle is determined by all the data types used. The space available for symbols is full and your program attempted to define one.

User Action: Simplify your program's data use to reduce memory usage. Reduce the number of symbols used. Check for infinite recursion.

Space for the stack is exhausted.

Description: The number of stack entries LISP can handle is determined by all the data types used. The space available for entries is full and your program attempted to add one.

User Action: Simplify your program's data use to reduce memory usage. Reduce the number of forms used. Check for infinite recursion.

Stray &ALLOW-OTHER-KEYS in arglist of ~S.

Description: The &allow-other-keys lambda-list keyword must follow all other symbols after the &key lambda-list keyword and must precede subsequent lambda-list keywords.

User Action: Reorder the arguments in the form. See *Common LISP* page 60.

Stream does not exist.

Description: The stream argument given to the open function is a non-existent stream.

User Action: Change the code to create the correct stream before you use it as an argument.

Stream is not recognized. fffff

Description: The form being evaluated requires a valid stream name where fffff was used. LISP does not recognize fffff as the name of a defined stream.

User Action: Check for an omitted or incorrect function call to define the stream.

Symbol does not have a global value.

Description: The symbol used does not have a global value.

User Action: Check for a typographical error.

Symbol is not defined. fffff

Description: The symbol indicated by fffff exists but has no value defined to LISP.

User Action: Check the entered form for a possible typographical error. Correct the form if necessary, or define the symbol to LISP before reentering the form.

The lists of keys and data are of unequal length.

Description: These lists must contain the same number of elements. An element might have been omitted or entered twice.

User Action: Correct the lists.

Too few argument forms to a SHIFTF.

Description: The shiftf macro requires an argument for at least one place form and for a new value.

User Action: Check for an omitted argument or a misplaced parenthesis.

Unexpected end-of-stream encountered on stream zzzzzzzz

Description: You attempted to input an incomplete LISP object. The load function could not match an opening parenthesis (( ) with a closing parenthesis before the end of information occurred on the stream indicated as zzzzzzzzz. The file you attempted to load is either incomplete or contains a syntax error.

User Action: Check for a missing closing parenthesis. Correct the file and reload it.

Unexpected go encountered.

Description: You used go outside of a tagbody.

User Action: Enclose go within a tagbody.

Unexpected return encountered.

Description: You entered the return function when you were not within the named block. The return function can only work from within the named block.

User Action: Check for a typographical error in the block name.

Unfound catch-tag in compiled code.

Description: A throw was done for which there was no suitable catch tag.

User Action: Check for a typographical error.

Unpaired item in keyword portion of macro call.

Description: Each keyword parameter must have a corresponding symbol. You might have omitted a keyword or symbol.

User Action: Correct the macro form.

Unreadable object encountered in stream.

Description: An entity that is not a valid LISP object was found in the file being read.

User Action: Ensure that the stream is associated with the correct file. You might be reading a binary file. Check that the file is not damaged.

User break encountered.

Description: One of the break conditions identified to NOS/VE for your terminal was detected.

User Action: Depends on the cause of the break. This message is informative only.

Wrong number of arguments encountered in form xxxxxxxx

Description: There are too many or too few arguments in the form indicated by xxxxxxxx.

User Action: Check for misplaced parentheses.

~A is not a reasonable value for *Print-Base*.

Description: The value referenced by the directive is outside the range permitted for the radix currently defined as *print-base*.

User Action: Check for a nondecimal digit (possibly a hexadecimal digit) in the value. The default for *print-base* is 10; to use a nondecimal number, you must change *print-base*.

~S Bad clause in CASE.

Description: One of the clauses is not a proper LISP form.

User Action: Check for an omitted or extra argument, or for a misplaced parenthesis.

~S Macro too short to be legal.

Description: The full form of the directive was used but at least one of the required parameters cannot be found.

User Action: Check for a missing comma.

~S Macro-name not a symbol.

Description: The argument found must be a valid LISP symbol.

User Action: Check the macro name for a typographical error.

~S can't be converted to type ~S.

Description: You cannot nest these forms.

User Action: Redesign your program.

~S cannot be coerced to a string.

Description: The value referenced by the directive cannot be used in a context that evaluates to a string.

User Action: Redesign your program.

~S illegal atomic form for get-setf-method.

Description: The form referenced in the get-setf-method function must be a generalized variable (a list cons).

User Action: Check that the form is not a number, an array, or a string.

~S is a bad thing in a DO varlist.

Description: The form referenced by the directive produces a do loop with potentially dangerous consequences.

User Action: Check for incorrect nesting or potential binding problems. Check that setq does not change the var argument within the loop.

~S is a bad type specifier for sequence functions.

Description: LISP does not recognize the form referenced by the directive as a valid type specifier.

User Action: Check for a typographical error.

~S is a bad type specifier for sequences.

Description: LISP does not recognize the form referenced by the directive as a valid type specifier.

User Action: Check for a typographical error.

~S is a malformed property list.

Description: Property lists must contain an even number of items. Each property object must have a unique indicator symbol.

User Action: Check for a missing item. Check that the correct object is specified as a property list. Check for an indicator symbol that is used twice.

~S is an ill-formed do.

Description: The object referenced by the directive does not conform to the requirements of a Common LISP do macro.

User Action: Check for a missing argument or a misplaced parenthesis.

~S is an illegal N for SETF of NTH.

Description: The argument referenced by the directive as n is a negative integer or a noninteger.

User Action: Correct the n argument. Check for a hexadecimal digit used in a decimal integer.

~S is an illegal size for MAKE-LIST.

Description: The size argument must be a nonnegative integer.

User Action: Check for a noninteger used as the size argument.

~S is not a known location specifier for setf.

Description: The form referenced by the directive as the setf place argument does not access a LISP data object.

User Action: Check for a typographical error in the argument.

~S is not a list.

Description: The argument referenced by the directive must be a true list.

User Action: Check that the object is not a dotted list. Check for a typographical error in the symbol.

~S is not a sequence.

Description: The argument referenced by the directive is not recognized by LISP as a valid sequence. A sequence must be a true list or a vector.

User Action: Check that the object is not a dotted list. Check for a typographical error in the symbol.

~S is not of type (mod 16).

Description: The OP argument to BOOLE is not valid.

User Action: Correct the argument.

~S is too large an index for SETF of NTH.

Description: The argument referenced by the directive as n is either equal to or greater than the length of the list.

User Action: Check for a hexadecimal digit in a decimal integer.

~S is too short to be a legal do.

Description: The form referenced by the directive as a do macro does not contain enough arguments to define a functional do loop. At least one of the optional arguments must be present.

User Action: Redesign the loop.

~S is too short to be a legal dotimes.

Description: The form referenced by the directive as a dotimes macro does not contain enough arguments to define a functional do loop. At least one of the optional arguments must be present.

User Action: Redesign the loop.

~S is too short to be a legal dolist.

Description: The form referenced by the directive as a dolist macro does not contain enough arguments to define a functional do loop. At least one of the optional arguments must be present.

User Action: Check for a missing declaration or statement argument.

~S: invalid output type specification.

Description: The argument referenced by the directive cannot be used as an output type specification.

User Action: Check for a missing argument before the argument indicated.

Errors Encountered Less Frequently

The following are possible errors that can be generated by user code. You need to inspect your program to correct the errors described below. Note that error messages from the compiler that are prefixed with WARNING are just informative; not fatal to the compilation. A warning is something suspicious in the code that probably means some form of loss, but that may be ignored if you understand and accept the possible consequences.

A package named ~S already exists.

Argument can't be zero.

Argument is excluded from the domain of ATAN.

Argument is excluded from the domain of ATANH.

Argument must be a non-complex number.

Argument must be a non-negative, non-complex number.

Arguments must be numbers.

Array values must be a list.

Bad argument to ~S: ~S

Bad argument, ~A, for RANDOM-STATE.

Bad clause in CCASE - ~S.

Bad clause in CTYPECASE - ~S.

Bad option in included slot spec: ~S.

Base can't be 1 in this case.

Bit vector is longer than specified length #~A*~A

Bogus function type: ~S

Bogus item on *BENV* list.

Cannot find description of structure ~S to use for inclusion.

Cannot generate object for form:

Can't coerce ~S to type ~S.

Comma not inside a backquote.

Compilation source holds no function definitions.

Complex arguments not allowed.

Complex numbers cannot have complex components.

Complex numbers can't be coerced into floats.

Constant ~S being redefined.

Control string is not a string

Debug requires a condition as its argument.

Deftype ~S cannot be called with ~S args.

Denominator cannot be zero.

Dimensions argument to #A not a non-negative integer: ~S

Dispatch character already exists.

Dot context error.

Ecase key must be one of ~S

End of file encountered after reading a colon.

End-of-file after escape character.

End-of-file inside dispatch character.

Escape character appeared after #*

Escape character appears in number.

Etypecase key must be one of these types: ~S

Flaming PPrint death

For two argument case, args must be non-complex.

Form not a cybil-atomic:

Frame ~S not found FIND-LEXICAL-BOUNDARY.

Garbage in dispatch vector - ~S

Illegal action ~s

Illegal atomic form to eval: ~S

Illegal complex number format.

Illegal complex-number syntax.

Illegal digits ~S for radix ~S

Illegal element given for ~ bitvector #~A*~A

Illegal sharp character ~S

Illegal stuff in lambda list of Define-Modify-Macro.

Illegal terminating character after a colon, ~S

Ill-formed condition bindings: ~A

I'm not yet implemented.

Initial contents for #A is inconsistent with ~

Internal error in floating point reader.

Internal error - validation of boolean incorrect: ~S

Internal error - validation of fixnum incorrect: ~S

INTERNAL- assoc-list not nil

Log of zero undefined.

Meaningless bit name in character name: ~A

Meaningless character name ~A

More than one object follows . in list.

No dimensions argument to #A.

No dispatch function defined for ~S.

No dispatch table for dispatch char.

No frame found on stack with name ~S.

No non-whitespace characters in number.

Non-integer label #~S=

Non-integer label #~S#

Non-list following #S

Non-list following #S: ~S

Non-positive argument, ~A, to RANDOM.

Non-symbol variable name in ~S.

Not a condition type: ~S

Nothing appears after . in list.

Nothing appears before . in list.

Numerator and denominator must be integers.

Object is not labelled #~S#

Odd number of setf-args to SETF.

Package ~S not found.

Prepare-for-fast-read-char might GC stream.

Real part of power of zero must be strictly positive.

Redundant bit name in character name: ~A

Strange file-position ~s.

Strange version ~s.

Structure type is not a symbol: ~S

Symbol following #: contains a # : ~S

Symbol ~S not found in the ~A package.

T or OTHERWISE clause is not permitted in CCASE.

T or OTHERWISE clause is not permitted in CTYPECASE.

T or Otherwise clause is not permitted in ECASE.

T or Otherwise clause is not permitted in ETYPECASE.

The Defstruct option :NAMED takes no arguments.

The Defstruct option ~S cannot be used with 0 arguments.

THE expected type ~A and got type ~A.

The file ~S does not exist.

The type ~S does not inherit from CONDITION.

The ~S structure does not have a default constructor.

There's junk in this string: ~S.

There's no digits in this string: ~S

This is a bad thing for a directory name: ~S

Too many colons after ~S:

Too many colons in ~S

Too many dots.

Undefined read-macro character ~S

Unexpected end-of-file encountered.

Unmatched right parenthesis.

Unreadable object encountered in stream.

Use-lisp-reader needs to be compiled

Vector longer than specified length: #~S~S

Wrong number of arguments to CATCH-ERROR-ABORT.

Wrong type argument, ~A, to RANDOM.

You have to give a little bit for non-zero #* bit-vectors.

Zero invalid as a divisor.

&REST keyword is missing.

~A is an ill-formed function object for ~A

~A not a valid number for *read-base*.

~A should be a function object.

~S -- Bad clause in CASE.

~S -- Deftype form too short to be legal.

~S -- Illegal type specifier to TYPEP.

~S -- Type-name not a symbol.

~S -- ill-formed keyword arg in ~S.

~S -- non-symbol variable name in arglist of ~S.

~S already compiled.

~S has illegal definition.

~S is a bad :TYPE for Defstruct.

~S is a lexical closure. Cannot compile it alone.

~S is an ill-formed function object for ~S

~S is an illegal :Test for hash tables.

~S is an inappropriate type of object for coerce.

~S is an unknown Defstruct option.

~S is neither a symbol nor a list of symbols.

~S is not a defined structure type.

~S is not a dispatch char.

~S is not a sequence.

~S is not a subtype of SEQUENCE.

~S is not a symbol.

~S is not accessible in the ~A package.

~S is not of type

~S is not of type integer.

~S not allowed in Define-Modify-Macro lambda list.

~S not declared or bound, assuming special.

~S should be a function object.

~S should be a list or nil.

~S should have been a function object.

~S: index too large.

~S: index too small.

~S: invalid output type specifier.

~s is not a frame boundary type.

~s is not a primitive frame.

,. after backquote in ~S

,. after dot in ~S

,@ after backquote in ~S

,@ after dot in ~S

Index of LISP Symbols E

This appendix lists all functions, macros, special forms, special variables, and constants supported by LISP. There is no corresponding chapter in *Common LISP*.

This appendix lists the page in *Common LISP* of the primary description for each LISP symbol. The symbols are listed alphabetically, in ASCII collating sequence order.

NOTE

There is no guarantee that only functions listed in this index are defined in the LISP system. There may be some Common LISP functions which are partially implemented, but not yet ready for use. There may also be some functions that are part of LISP and are never intended for your use. These are not supported and may not be accessible in later versions of the product.

Symbol	Type	Common LISP Page	Notes or LISP Page
check-type	macro	433	26-13
cis	function	207	
clear-input	function	380	
clear-output	function	384	
close	function	332	
clrhash	function	285	
code-char	function	240	
coerce	function	51	
commonp	function	76	
compile	function	438	
compile-file	function	439	
compiled-function-p	function	76	
compiler-let	special form	112	
complex	function	220	
complexp	function	75	
compute-proceed-cases	function		26-10
concatenate	function	249	
cond	macro	116	
condition-bind	special form		26-5
condition-case	special form		26-12
conjugate	function	201	
cons	function	266	
consp	function	74	
constantp	function	324	
copy-alist	function	268	
copy-list	function	268	
copy-readtable	function	361	
copy-seq	function	248	
copy-symbol	function	169	
copy-tree	function	269	
cos	function	105	
cosh	function	209	
count	function	257	
count-if	function	257	
count-if-not	function	257	
ctypecase	macro	436	26-14

Symbol	Type	Common LISP Page	Notes or LISP Page
member-if-not	function	275	
merge	function	260	
merge-pathnames	function	415	
min	function	198	
minusp	function	196	
mismatch	function	257	
mod	function	217	
modules	variable	188	
most-negative-double-float	constant	232	
most-negative-fixnum	constant	231	
most-negative-long-float	constant	232	
most-negative-short-float	constant	231	
most-negative-single-float	constant	232	
most-positive-double-float	constant	232	
most-positive-fixnum	constant	231	
most-positive-long-float	constant	232	
most-positive-short-float	constant	231	
most-positive-single-float	constant	232	
multiple-value-bind	macro	136	
multiple-value-call	special form	135	
multiple-value-list	macro	135	
multiple-value-prog1	macro	136	
multiple-value-setq	macro	136	
multiple-values-limit	constant	135	
name-char	function	243	
namestring	function	417	
nbutlast	function	269	
nconc	function	269	
nil	constant	72	
nintersection	function	277	
ninth	function	266	
not	function	82	
notany	function	250	
notevery	function	250	
notinline	declaration	159	
nreconc	function	269	
nreverse	function	248	
nset-difference	function	278	
nset-exclusive-or	function	278	
nstring-capitalize	function	304	

Symbol	Type	Common LISP Page	Notes or LISP Page
pi	constant	209	
plusp	function	196	
pop	macro	271	
position	function	257	
position-if	function	257	
position-if-not	function	256	
pprint	function	383	
prin1	function	383	
prin1-to-string	function	383	
princ	function	383	
princ-to-string	function	383	
print	function	383	
print-array	variable	373	
print-base	variable	371	
print-case	variable	372	
print-circle	variable	371	
print-escape	variable	370	
print-gensym	variable	372	
print-length	variable	372	
print-level	variable	372	
print-pretty	variable	371	
print-radix	variable	371	
probe-file	function	424	
proceed-case	special form		26-8
proceed-case-name	function		26-10
proclaim	function	156	
prog	macro	131	
prog*	macro	131	
prog1	macro	109	
prog2	macro	109	
progn	special form	109	
progv	special form	112	
provide	function	188	
psetf	macro	97	
psetq	macro	92	
push	macro	269	
pushnew	macro	270	

Symbol	Type	Common LISP Page	Notes or LISP Page
query-io	variable	328	
quit	function		1-3
quote	special form	86	
random	function	228	
random-state	variable	230	
random-state-p	function	231	
rassoc	function	281	
rassoc-if	function	281	
rassoc-if-not	function	281	
rational	function	214	
rationalize	function	214	
rationalp	function	74	
read	function	375	
read-base	variable	344	
read-byte	function	382	
read-char	function	379	
read-char-no-hang	function	380	
read-default-float-format	variable	375	
read-delimited-list	function	377	
read-from-string	function	380	
read-line	function	378	
read-preserving-whitespace	function	376	
read-suppress	variable	345	
readtable	variable	361	
readtablep	function	361	
realpart	function	220	
reduce	function	251	
rem	function	217	
remf	macro	167	
remhash	function	284	
remove	function	253	
remove-duplicates	function	254	
remove-if	function	253	
remove-if-not	function	253	
remprop	function	166	
rename-file	function	423	
rename-package	function	184	
replace	function	252	

Symbol	Type	Common LISP Page	Notes or LISP Page
require	function	188	
rest	function	266	
return	macro	120	
return-from	special form	120	
revappend	function	269	
reverse	function	248	
room	function	442	
rotatef	macro	99	
round	function	215	
rplaca	function	272	
rplacd	function	272	
$save-lisp	function		25-4
sbit	function	293	
scale-float	function	218	
schar	function	300	
search	function	258	
second	function	266	
set	function	92	
set-char-bit	function	244	
set-difference	function	278	
set-dispatch-macro-character	function	364	
set-exclusive-or	function	278	
set-macro-character	function	362	
set-syntax-from-char	function	361	
setf	macro	94	
setq	special form	91	
seventh	function	266	
shadow	function	185	
shadowing-import	function	186	
shiftf	macro	97	
short-float-epsilon	constant	232	
short-float-negative-epsilon	constant	232	
short-site-name	function	448	
signal	function		26-5
signum	function	206	
simple-bit-vector-p	function	76	
simple-string-p	function	75	
simple-vector-p	function	75	
sin	function	207	
single-float-epsilon	constant	232	
single-float-negative-epsilon	constant	232	

Symbol	Type	Common LISP Page	Notes or LISP Page
subsetp	function	279	
subst	function	273	
subst-if	function	273	
subst-if-not	function	273	
substitute	function	255	
substitute-if	function	255	
substitute-if-not	function	255	
subtypep	function	72	
svref	function	291	
sxhash	function	285	
symbol-function	function	90	
symbol-name	function	168	
symbol-package	function	170	
symbol-plist	function	166	
symbol-value	function	90	
symbolp	function	73	
t	constant	72	
tagbody	special form	130	
tailp	function	275	
tan	function	105	
tanh	function	209	
tenth	function	266	
terminal-io	variable	328	
terpri	function	384	
the	special form	161	
third	function	266	
throw	special form	142	
time	macro	441	25-5
trace	macro	440	26-17
trace-output	variable	328	
tree-equal	function	264	
truename	function	413	
truncate	function	215	
type	declaration	158	
type-of	function	52	
typecase	macro	118	
typep	function	72	

Tautology Proving Example F

This appendix contains an example of LISP use. There is no corresponding chapter in *Common LISP*.

The sample LISP statement file shown in figure F-1 is a tautology proving program called theorem-prover, written to illustrate LISP features. It is not intended to teach a specific LISP programming style. This program uses a Gentzen implication algorithm.

```
; The basic data structures are the LHS (lefthand side) and the RHS
; (righthand side).  These represent the respective sides of an
; implication.
;
; The RHS is a list representing a disjunction of clause.
; The LHS is a list representing a conjunction of clauses.
; The goal is to find something on the LHS which is also on the RHS.
;
; First, the input clause is placed on the RHS.  A clause is extracted
; from either the LHS or the RHS.  The operator of the clause
; is examined.  One of several productions are applied to the RHS and
; the LHS to produce an equivalent simpler form.
;
; This process is repeated until either the intersection
; of the RHS and LHS is not empty, or until all clauses are simplified.
;
; For example:
;
;             ==> P -> (- Q -> - (P -> Q))   reduces to
;           P ==> ( - Q -> - (P -> Q))       reduces to
;     P ^ - Q ==> - (P -> Q)                 reduces to
;P ^ (P -> Q) ==> Q                          split into
;     P ==> P,Q    P,Q ==> Q                 simplifies to
;
;        *             *
;
;     ------------------------------------------------
;
; The theorem-prover function is the read eval print loop.
; The formula is read in from the terminal.  It is then reduced
; and the result is printed.
;
(DEFUN THEOREM-PROVER
  NIL
    (LET ((FORMULA NIL))
      (TAGBODY A (PRINC " TP?") (FINISH-OUTPUT) (SETQ FORMULA (READ))
                 (IF (MEMBER FORMULA '(END BYE QUIT STOP HALT EXIT))
                     (GO B))
                 (THEOREM-PROVER-PRINTER (REDUCE NIL (LIST FORMULA)))
                 (GO A)
               B (PRINT "Thank you")))
    )
```

Figure F-1. Theorem-Prover Code

(Continued)

(Continued)

```
;  The reduce function is the workhorse of the theorem prover.
;
;  Arguments - LHS the lefthand side if the GENTZEN implication
;              RHS the righthand side of the GENTZEN implication
;
; The data (theorem) is represented as a list.  The list is in prefix notation.
; If a sublist is itself a list, then it is a candidate for simplification.
; The algorithm used takes the first possible simplification on the left.
; If none exists on the left, it checks the righthand side.  If none exists
; there, it checks for trivial validation.
;
; Reduce returns NIL if the statement is valid; otherwise, it returns
; a list whose CAR is the lefthand side that did not resolve
; and the list's CDR is the righthand side.
;
(DEFUN REDUCE
  (LHS RHS)
   (COND ((NOT-SIMPLIFIED LHS) (REDUCE-LHS LHS RHS))
         ((NOT-SIMPLIFIED RHS) (REDUCE-RHS LHS RHS))
         (T (CHECK-SIMPLE-CASE LHS RHS))))

(DEFUN REDUCE-LHS
  (LHS RHS)
   (REDUCE-LHS2 (GET-REDUCTION LHS) (REMOVE-REDUCTION LHS) RHS))

(DEFUN REDUCE-LHS2
  (REDUCTION LHS RHS)
   (APPLY (GET 'LHS (CAR REDUCTION)) (LIST (CDR REDUCTION) LHS RHS)))

; REDUCE-RHS retrieves the subtheorem to be reduced, extracts the
; subtheorem from the lefthand side and does the reduction
;
(DEFUN REDUCE-RHS
  (LHS RHS)
   (REDUCE-RHS2 (GET-REDUCTION RHS) LHS (REMOVE-REDUCTION RHS)))

(DEFUN REDUCE-RHS2
  (REDUCTION LHS RHS)
   (APPLY (GET 'RHS (CAR REDUCTION)) (LIST (CDR REDUCTION) LHS RHS)))
```

Figure F-1. Theorem-Prover Code

(Continued)

(Continued)

```
; The setf function stores the intelligence of the system with symbol-plist.
; As each operator is detected in REDUCE-LHS, the information on how to
; process the information is retrieved from this plist.  To add more
; operators, just add the code here with the operator name as the plist
; indicator.  See REDUCE-LHS2 for how the properties are executed.
;
; Arguments - ARG is a list of arguments for this operation. The operator must
;             indicate how long the list is to be.
;             LHS is the lefthand side with what is being simplified removed.
;             RHS is the righthand side of the implication.
;
(SETF (SYMBOL-PLIST 'LHS)
'(NOT
  (LAMBDA (ARG LHS RHS)
    (COND ((MEMBER (CAR ARG) LHS) NIL)
          (T (REDUCE LHS (CONS (CAR ARG) RHS)))))
 AND
 (LAMBDA (ARGS LHS RHS)
    (COND ((MEMBER (CAR ARGS) RHS) NIL)
          ((MEMBER (CADR ARGS) RHS) NIL)
          (T (REDUCE (APPEND ARGS LHS) RHS))))
 IMPLIES
 (LAMBDA (ARGS LHS RHS)
    (COND
     ((MEMBER (CAR ARGS) (CONS (CADR ARGS) LHS)) NIL)
     ((MEMBER (CADR ARGS) (CONS (CAR ARGS) RHS)) NIL)
     (T (OR (REDUCE LHS (CONS (CAR ARGS) RHS))
            (REDUCE (CONS (CADR ARGS) LHS) RHS)))))
 IF
 (LAMBDA (ARGS LHS RHS)
    (REDUCE
     (CONS (LIST 'AND
                 (LIST 'IMPLIES (CAR ARGS) (CADR ARGS))
                 (LIST 'IMPLIES (LIST 'NOT (CAR ARGS)) (CADDR ARGS)))
           LHS)
     RHS))
 OR
 (LAMBDA (ARGS LHS RHS)
    (COND
     ((MEMBER (CAR ARGS) RHS) NIL)
     ((MEMBER (CADR ARGS) RHS) NIL)
     (T (OR (REDUCE (CONS (CAR ARGS) LHS) RHS)
            (REDUCE (CONS (CADR ARGS) LHS) RHS))))))
)
```

Figure F-1. Theorem-Prover Code

(Continued)

(Continued)

```
; The following code is the complement of LHS (see above.)

(SETF (SYMBOL-PLIST 'RHS)
'(NOT
  (LAMBDA (ARG LHS RHS)
    (COND ((MEMBER (CAR ARG) RHS) NIL)
          (T (REDUCE (CONS (CAR ARG) LHS) RHS))))
 AND
 (LAMBDA (ARGS LHS RHS)
    (COND
     ((MEMBER (CAR ARGS) LHS) NIL)
     ((MEMBER (CADR ARGS) LHS) NIL)
     (T (OR (REDUCE LHS (CONS (CAR ARGS) RHS))
            (REDUCE LHS (CONS (CADR ARGS) RHS))))))
 IMPLIES
 (LAMBDA (ARGS LHS RHS)
    (COND
     ((EQ (CAR ARGS) (CADR ARGS)) NIL)
     ((MEMBER (CAR ARGS) RHS) NIL)
     ((MEMBER (CADR ARGS) LHS) NIL)
     (T (REDUCE (CONS (CAR ARGS) LHS) (CONS (CADR ARGS) RHS)))))
 IF
 (LAMBDA (ARGS LHS RHS)
    (REDUCE
     LHS
     (CONS (LIST 'AND
                 (LIST 'IMPLIES (CAR ARGS) (CADR ARGS))
                 (LIST 'IMPLIES (LIST 'NOT (CAR ARGS)) (CADDR ARGS)))
           RHS)))
 OR
 (LAMBDA (ARGS LHS RHS)
    (COND ((MEMBER (CAR ARGS) LHS) NIL)
          ((MEMBER (CADR ARGS) LHS) NIL)
          (T (REDUCE LHS (APPEND ARGS RHS))))))
)

; The function below checks for success when no other reductions can
; be made.  All failures must end here.  Individual
; operator processing can detect success earlier, however.
;
(DEFUN CHECK-SIMPLE-CASE
  (LHS RHS)
    (COND ((INTERSECT LHS RHS) NIL)
          (T (CONS LHS RHS))))

(DEFUN GET-REDUCTION
  (LST)
   (COND ((NULL LST) NIL)
         ((LISTP (CAR LST)) (CAR LST))
         (T (GET-REDUCTION (CDR LST)))))
```

Figure F-1. Theorem-Prover Code

(Continued)

(Continued)

```
; Do a set intersection to determine if anything on the right appears
; on the left.  Right represents the disjunction and left a
; conjunction.  Therefore if anything on the left is also on the
; right, the theorem is valid.
;
(DEFUN INTERSECT
  (SET1 SET2)
      (REMALL NIL
           (MAPCAR
                   (LAMBDA (X) (COND ((MEMBER X SET2) X) (T NIL)))
                   SET1)))

;
; Not-simplified returns T if there exists an expression that can be
; simplified; otherwise, it returns NIL.
; Arguments - LST is a list representing one side of the implication; any
;             element that is a list can be simplified.
;
(DEFUN NOT-SIMPLIFIED
  (LST)
   (COND ((NULL LST) NIL)
         ((LISTP (CAR LST)) T)
         (T (NOT-SIMPLIFIED (CDR LST)))))

(DEFUN REMALL
  (ELEMENT LST)
   (COND ((NULL LST) NIL)
         ((EQ ELEMENT (CAR LST)) (REMALL ELEMENT (CDR LST)))
         (T (CONS (CAR LST) (REMALL ELEMENT (CDR LST))))))

(DEFUN REMOVE-REDUCTION
  (LST)
   (COND ((NULL LST) NIL)
         ((LISTP (CAR LST)) (CDR LST))
         (T (CONS (CAR LST) (REMOVE-REDUCTION (CDR LST))))))
```

Figure F-1. Theorem-Prover Code

(Continued)

(Continued)

```
; The following decodes the result of REDUCE and prints knowledgeable
; information.  NIL means success.  A list means failure.
; The CAR is the lefthand side. The CADR is the RHS.
;
(DEFUN THEOREM-PROVER-PRINTER
  (RESULT)
   (COND
    ((NULL RESULT) (PRINC " --VALID--") (TERPRI))
    (T (PRINC " --INVALID--")
       (TERPRI)
       (PRINC " LEFT-HAND-SIDE  -->")
       (PRINC (CAR RESULT))
       (TERPRI)
       (PRINC " RIGHT-HAND-SIDE --> ")
       (PRINC (CADR RESULT))
       (TERPRI)
       (TERPRI))))
```

Figure F-1. Theorem-Prover Code

Using Theorem-Prover

To execute theorem-prover, type:

```
(theorem-prover)
```

Respond to each TP?? prompt with a logical function in prefix normal form. The theorem-prover recognizes the logical operators OR, AND, IMPLIES, NOT, and IF. For example, you can enter the clause

```
(a => b) => (-a OR b)
```

by typing

```
(implies (implies a b) (OR (NOT a) b))
```

which returns

```
--VALID--
```

To stop the theorem-prover, type

```
end
```

Index

Index

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Comments (continued from other side)

Please fold on dotted line;
seal edges with tape only.

FOLD

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
First-Class Mail Permit No. 8241 Minneapolis, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CONTROL DATA
Technical Publications
SVLF45
5101 Patrick Henry Drive
Santa Clara, CA 95054-1111

LISP for NOS/VE Language Definition Usage Supplement 60486213 B

We would like your comments on this manual to help us improve it. Please take a few minutes to fill out this form.

Who are you?

- ☐ Manager
- ☐ Systems analyst or programmer
- ☐ Applications programmer
- ☐ Operator
- ☐ Other _____

How do you use this manual?

- ☐ As an overview
- ☐ To learn the product or system
- ☐ For comprehensive reference
- ☐ For quick look-up
- ☐ Other _____

What programming languages do you use? _____

_____

How do you like this manual? Answer the questions that apply.

Yes	Somewhat	No	
☐	☐	☐	Does it tell you what you need to know about the topic?
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the order of topics logical?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Are there enough examples?
☐	☐	☐	Are the examples helpful? (☐ Too simple? ☐ Too complex?)
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Is the manual easy to read (print size, page layout, and so on)?
☐	☐	☐	Do you use this manual frequently?

Comments? If applicable, note page and paragraph. Use other side if needed.

Check here if you want a reply: ☐

Name _____ Company _____

Address _____ Date _____

_____ Phone _____

Please send program listing and output if applicable to your comment.


CONTROL DATA