LISP for NOS/VE Language Definition

CONTROL DATA

Usage Supplement

Preliminary

60486213

LISP for NOS/VE Language Definition

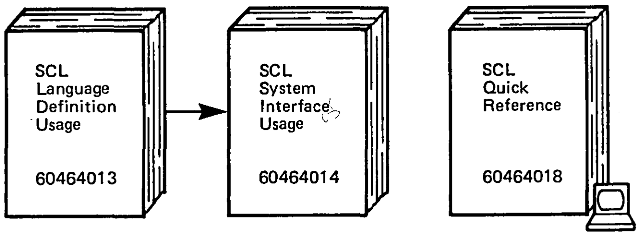
Usage Supplement

Preliminary

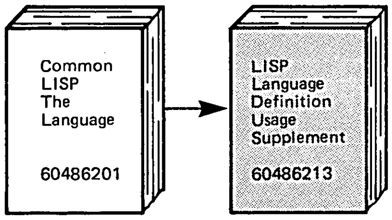
Publication Number 60486213

Related Manuals

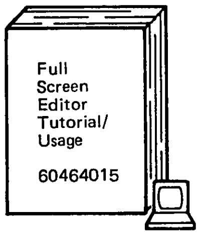
Background Material (Access as Needed):



LISP Manual Set:



Additional References:



→ indicates the recommended reading sequence

🖥 means available online

Manual History

This manual is revision 01, printed March 1985. It reflects the release of LISP under NOS/VE Version 1.1.2 at PSR level 630.

Contents

About This Manual

List Processing (LISP) for NOS/VE is a partial 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.

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.

We gratefully acknowledge the work of the Carnegie-Mellon University implementation team, especially Scott E. Fahlman, who has allowed Control Data to use their resources.

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.

LISP for NOS/VE is a subset of Common LISP that provides you with a working base to write typical applications.

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

For consistency with other NOS/VE manuals, 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.

(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 diagram on page 2 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, A Gentle Introduction to Symbolic Computation (David S. Touretsky, copyright 1984 by Harper & Row Publishers, 10 East 53rd Street, New York, New York 10022.) This book uses a version of the MacLISP dialect.

- 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 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

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
Publications and Graphics Division
P.O. Box 3492
Sunnyvale, California 94088-3492

Please indicate whether you would like a written response.

Introduction 1

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

LISP Command Summary

Command	Page
LISP	1-1
INPUT=input file reference	
OUTPUT=output file reference	
STATUS=status variable	(Optional)

Introduction 1

Errors (5)

LISP signals all errors that Common LISP requires to be signalled. All detectable errors are also signalled. Most signalled errors are fatal to current evaluation (none are fatal to execution of LISP.) An error is signalled with a diagnostic message, beginning with the characters

```
    --LISP ERROR--
```

If you try to use partially implemented LISP features, LISP produces additional informative messages in a different format.

Overview of Syntax (9)

Colons can be used in keywords but cannot be used to indicate membership in a package.

Entering LISP

Use the following NOS/VE System Command Language (SCL) command to enter LISP:

```
    LISP
        INPUT=input file reference
        OUTPUT=output file reference
        STATUS=status variable                                          (Optional)
```

Parameters:

INPUT (I)

NOS/VE file containing valid LISP input statements. If you omit this parameter, the local 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 local file $OUTPUT is used and output appears at your terminal.

STATUS

See the SCL Language Definition Usage manual for a description of the use of this optional parameter.

LISP responds to the LISP command with the message:

```
    Welcome to LISP.  lisp-implementation-version
```

and the currently defined NOS/VE input prompt (usually a question mark.) The string lisp-implementation-version is the value of the function by that name, as defined when LISP was installed on your system.

Using SCL or Other Software From Within LISP

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

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

Parameters:

string

Any string containing a valid SCL 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 an SCL command.

Example:

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

When you use this function, LISP submits the string to the SCL 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 value other than NIL and an informative message as a side effect.

Leaving LISP

Use either of the following functions to leave LISP:

```
    (exit)
or
    (quit)
```

If you omit the parentheses when you type QUIT, the message:

```
    "To exit LISP/VE, type (QUIT) or (EXIT)."
```

appears.

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 output.

Data Types 2

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

Data Types 2

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 does not support the quotient of two integers as a ratio or support Cartesian complex numbers.

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 a keyword argument, or with the :element-type keyword of T)

character (character string arrays created with a keyword argument of CHARACTER)

short-float (floating-point number arrays created with a keyword argument of FLOAT)

bit (single-bit boolean variable arrays created with a keyword argument of BIT)

LISP does not support hash tables or user-defined packages.

Integers (13)

LISP uses two's-complement for internal representation. The internal radix used is 2; the external radix used is 10. Integers are stored in LISP-objects and accessed directly; integer 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 does not have a bignum infinite-magnitude integer.

Ratios (15)

LISP does not support rational numbers in ratio form.

Floating-Point Numbers (16)

Short-format (short-float) floating-point numbers use the immediate 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 14 decimal digits. The number -0.0 is not distinguished from +0.0.

LISP supports short-float numbers between B00080000000 hexadecimal (-0.47874887304761 x 10^{-1233} decimal) and 4FFFFFFFFFFF hexadecimal (0.52219444070657 x 10^{1233} decimal), inclusive. The smallest positive value is 3000800000000000 hexadecimal (0.47874887304761 x 10^{-1233} decimal). The smallest negative value is CFFFFFFFFFFFFFFF hexadecimal (-0.52219444070657 x 10^{1233} decimal).

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

LISP single-format (single-float) numbers are not identical to short-float numbers. LISP does not support single-format (single-float), long-format (long-float), and double-format (double-float) floating-point numbers.

Characters (20)

LISP supports the Common LISP definition of character data types, except 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 these characters.

Character Attributes (23)

LISP does not support the font or bits attributes. It does not have the char-bits-limit constant, and the char-font-limit constant is always 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.

Vectors (29)

No significant difference in efficiency exists between using a vector and using a one-dimensional array in LISP. In LISP, a vector is a one-dimensional array.

Hash Tables (31)

LISP does not support hash tables.

Packages (31)

LISP does not support packages. See chapter 11.

Pathnames (31)

LISP does not support pathnames.

Random States (31)

LISP does not support random states.

Structures (31)

LISP does not support structures.

Overlap, Inclusion, and Disjointedness of Types (33)

In LISP, the types short-float and single-float are not identical. The types single-float, double-float, and long-float do not exist.

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

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.

Scope and Extent

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. LISP does not support lexical closures, so a function does not save the binding of lexically scoped variables. For this reason, the compose function on page 37 of Common LISP does not work properly.

Type Specifiers 4

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

Type Specifiers 4

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.

LISP does not use the list-format name complex. The complex data type is not supported.

Type Specifiers That Abbreviate (48)

LISP does not use the following list-format names:

single-float
double-float
long-float
rational

The single-float, double-float, long-float, and rational data types are not supported.

Defining New Type Identifiers (50)

You cannot define new type identifiers. LISP does not have the deftype macro.

Determining the Type of an Object (52)

The LISP type-of function does not return the list-format name for any data type.

Program Structure 5

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

Program Structure 5

Forms (54)

The LISP evaluator has no extensions. Anything other than a valid form signals an error.

Special Forms (56)

LISP does not have the following special forms:

```
compiler-let
eval-when
function
macrolet
progv
```

Appendix D 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 macros contain data objects not considered to be forms in Common LISP. Some LISP macros have expansions that contain LISP-defined special forms.

Appendix D lists all predefined macros that LISP supports.

Functions (59)

Appendix D lists all predefined functions that LISP supports.

Lambda-Expressions (59)

LISP does not have the following constants.

```
lambda-list-keywords
lambda-parameters-limit
```

Top-Level Forms (66)

LISP does not have a compiler. There are no forms which LISP does not recognize at levels other than the top level.

Defining Named Functions (67)

LISP does not have the defconstant macro.

Control of Time of Evaluation (69)

You cannot control the time of evaluation. Immediate evaluation occurs for all forms entered through the input file (forms entered through the load function are evaluated when encountered.) LISP does not have the eval-when special form.

Predicates

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

Predicates 6

Data Type Predicates (72)

LISP is a subset of Common LISP in all three categories of data type predicates:

General
Specific
Equality

General Type Predicates (72)

LISP does not have the subtypep function.

Specific Data Type Predicates (73)

LISP does not have the following functions:

bit-vector-p
commonp
compiled-function-p
complexp
packagep

rationalp
simple-bit-vector-p
simple-vector-p
simple-string-p

Equality Predicates (77)

For the eq function, fixnum and character instances can be true. LISP does not have a compiler, so no collapsed constants can exist. The following statement evaluations occur:

Statement	Value Returned
(eq 3 3)	T (true)
(eql 3.0 3.0)	NIL (false)
(eq #c(3 -4) #c(3 -4))	NIL (false) and a diagnostic; complex numbers are not supported
(eq ´(a . b) ´(a . b))	NIL (false)
(eq #\A #\A)	T (true)
(eql "Foo" "Foo")	NIL (false)

For the eql function, the following statement evaluations occur:

Statement	Value Returned
(eql ´(a . b) ´(a . b))	NIL (false)
(eql 0.0 -0.0)	T (true)
(eql "Foo" "Foo")	NIL (false)

Control Structure 7

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

Control Structure 7

Constant and Variable Reference (86)

LISP does not have the function special-form-p or the special form function.

Generalized Variables (93)

The LISP setf macro does not recognize place parameter function call forms with the following function names as the first element:

```
apply
bit
char
char-bit
documentation

elt
fill-pointer
gethash
ldb
mask-field

sbit
schar
string-char
subseq
svref
```

The LISP setf macro does not support Common LISP structures.

Function Invocation (107)

LISP does not have the call-arguments-limit constant.

Establishing New Variable Bindings (110)

LISP does not have the following special forms:

```
compiler-let
macrolet
progv
```

Multiple Values (133)

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

Constructs for Handling Multiple Values (133)

LISP does not have the multiple-value-setq macro or the multiple-values-limit constant.

Macros

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

Macros 8

Macro Support (143)

LISP must encounter a macro definition before that macro is first used. A macro is expanded each time it is encountered.

Macro Definition (144)

LISP does not support the macro call for lexical environments; lexically scoped entities are seen within the body of the expansion function.

LISP does not allow the optional env parameter in the macroexpand or macroexpand-1 functions; lexical closures are not supported.

LISP does not have the *macroexpand-hook* special variable.

Declarations

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

Declarations

Declaration Syntax (153)

LISP allows only a subset of Common LISP declarations in the declare special form. The only valid declaration specifier is special. (The car portion of the decl-spec parameter can only contain special.)

Declaration Specifiers (157)

LISP does not have the following declaration specifiers:

declaration
ftype
function
ignore

inline
notinline
optimize
type

LISP provides no additional declaration specifiers.

Symbols 10

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

Symbols

The Property List (163)

The LISP getf and remf macros and get-properties function do not recognize place parameter function call forms with the following function names as the first element:

```
apply
bit
char
char-bit
documentation

elt
fill-pointer
gethash
ldb
mask-field

sbit
schar
string-char
subseq
svref
```

The LISP getf and remf macros and get-properties function do not support Common LISP structures.

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.

LISP does not have the following functions:

```
copy-symbol
gentemp
```

Packages

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

Packages 11

Package Support (171)

Ignore all of chapter 11 of Common LISP, except as noted in this chapter of this manual. LISP does not support user-defined packages and system-defined packages do not yet exist.

Translating Strings to Symbols (174)

The LISP reader accepts symbol names that start with a colon; package-name:symbol-name is the standard notation for symbols within packages. You can use EQ to find symbol names beginning with a colon. You can create code using keywords (which are symbol names beginning with a colon) and use it with little change when LISP supports packages.

Package System Functions, Macros, and Variables (182)

LISP does not have the following special variable:

```
*package*
```

LISP does not have the following functions:

```
export
find-package
find-all-symbols
find-symbol
import

in-package
list-all-packages
make-package
rename-package
package-name

package-nicknames
package-shadowing-symbols
package-use-list
package-used-by-list
shadow

shadowing-import
unexport
unintern
unuse-package
use-package
```

LISP does not have the following macros:

```
do-all-symbols
do-external-symbols
do-symbols
```

Modules (188)

LISP does not support modules. It does not have the special variable *modules* or the provide and require functions.

intern Function (184)

The LISP intern function has the form:

```
intern string
```

The optional package name parameter is not supported.

Numbers 12

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

Numbers 12

Precision, Contagion, and Coercion (193)

LISP processes numerical expressions from left to right.

Comparisons on Numbers (196)

For the max and min functions, LISP returns the argument in its current format (there is only one LISP floating-point format, and LISP does not support rational numbers.)

Arithmetic Operations (199)

The following forms are equivalent in LISP:

```
(1+ x) and (+ x 1)
(1- x) and (- x 1)
```

LISP does not have the following macros:

```
decf
incf
```

LISP does not have the conjugate function.

Irrational and Transcendental Functions (203)

LISP uses the NOS/VE Common Math Library for these functions.

Trigonometric and Related Functions (205)

LISP does not have the following functions:

```
asin    cis
asinh   cos
acos    cosh
acosh   sinh
atan    tan

atanh   tanh
abs     phase
```

LISP does not have the constant pi.

Type Conversions and Component Extractions on Numbers (214)

LISP does not have the following functions:

```
complex
decode-float
denominator
float-digits
float-precision
```

float-radix
imagpart
integer-decode-float
numerator
rational

rationalize
realpart
scale-float

Logical Operations on Numbers (220)

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

LISP does not have the following functions:

ash
boole
logand
logandc1
logandc2

logbitp
logcount
logeqv
logior
lognand

lognor
lognot
logorc1
logorc2
logtest

logxor

LISP does not have the following constants:

boole-and
boole-andc1
boole-andc2
boole-clr
boole-c1

boole-c2
boole-eqv
boole-ior
boole-nand
boole-nor

boole-orc1
boole-orc2
boole-set
boole-xor
boole-1

boole-2

Byte Manipulation Functions (225)

LISP does not support byte manipulation. It does not have the following functions:

```
byte
byte-position
byte-size
deposit-field
dpb

ldb
ldb-test
mask-field
```

Random Numbers (228)

LISP does not support random numbers. It does not have the *random-state* special variable or the following functions:

```
make-random-state
random
random-state-p
```

Implementation Parameters (231)

LISP does not support these parameters. It does not have the following constants:

```
double-float-epsilon
double-float-negative-epsilon
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

long-float-epsilon
long-float-negative-epsilon
most-negative-fixnum
most-negative-double-float
most-negative-long-float

most-negative-short-float
most-negative-single-float
most-positive-fixnum
most-positive-double-float
most-positive-long-float

most-positive-short-float
most-positive-single-float
short-float-epsilon
short-float-negative-epsilon
single-float-epsilon

single-float-negative-epsilon
```

Characters 13

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

Characters

LISP characters use standard 7-bit ASCII character codes. Characters are held directly in LISP-objects. In some Common LISP implementations, character-objects hold special attributes (such as font, bits, graphic, meta, super, or hyper) in addition to the ASCII code for the character. LISP does not support these special attributes.

Character Attributes (233)

LISP does not support the font or bits attributes. It does not have the char-bits-limit constant, and the char-font-limit constant is always 1.

Predicates on Characters (234)

LISP does not have the following functions:

```
graphic-char-p
standard-char-p
```

Character Conversions (241)

LISP does not have the following function:

```
name-char
```

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

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

Sequences

Simple Sequence Functions (247)

LISP does not have the following functions:

```
elt
make-sequence
subseq
```

Concatenating, Mapping, and Reducing Sequences (249)

LISP does not have the following functions:

```
concatenate
every
notany
notevery
reduce

some
```

Modifying Sequences (252)

LISP does not have the following functions:

```
delete
delete-duplicates
delete-if
delete-if-not
fill

remove
remove-duplicates
remove-if
remove-if-not
substitute

substitute-if
substitute-if-not
```

Searching Sequences for Items (256)

LISP does not have the following functions:

```
count
count-if
count-if-not
find
find-if

find-if-not
mismatch
position-if
position-if-not
search
```

Sorting and Merging (258)

LISP does not support the sorting or merging of sequences. It does not have the following functions:

```
merge
sort
stable-sort
```

Lists 15

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

Lists 15

Lists (264)

LISP does not have the following function:

    endp

The LISP push and pop macros do not recognize place parameter function call forms with the following function names as the first element:

    apply
    aref
    bit
    char
    char-bit

    documentation
    elt
    fill-pointer
    gethash
    ldb

    mask-field
    sbit
    schar
    string-char
    subseq

    svref

The LISP push and pop macros do not support Common LISP structures.

LISP does not have the pushnew macro.

Using Lists as Sets (275)

LISP does not have the following functions:

    nset-difference
    nset-exclusive-or
    set-difference
    set-exclusive-or

The following LISP functions do not recognize the :key parameter:

    intersection
    nintersection
    nunion
    union

Hash Tables 16

This chapter supplements chapter 16 of <u>Common LISP</u>. LISP support of hash tables is described.

Hash Tables 16

Hash Table Support (282)

Ignore all of chapter 16 of Common LISP. LISP does not support hash tables.

Hash Table Functions (283)

LISP does not have the following functions:

```
clrhash
gethash
hash-table-count
hash-table-p
make-hash-table

maphash
remhash
```

Primitive Hash Function (285)

LISP does not have the following function:

```
sxhash
```

Arrays 17

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

Arrays

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 a keyword argument; the :element-type parameter cannot have a value of T for general arrays)
- character (character string arrays created with a keyword argument of CHARACTER)
- short-float (floating-point number arrays created with a keyword argument of FLOAT)
- bit (single-bit boolean variable arrays created with a keyword argument of BIT)

The LISP make-array function does not recognize the :displaced-to or :displaced-index-offset parameters. Displaced arrays are not supported.

LISP does not have the following constants:

```
array-rank-limit
array-total-size-limit
array-dimension-limit
```

LISP does not have the vector function.

Array Information (291)

LISP does not have the following functions:

```
adjustable-array-p
array-element-type
array-row-major-index
```

Functions on Arrays of Bits (293)

LISP does not support arrays of bits. It does not have the following functions:

```
bit
bit-and
sbit
bit-andcl
bit-andc2

bit-eqv
bit-ior
bit-nand
bit-nor
bit-not

bit-orcl
bitorc2
bit-xor
```

Fill Pointers (295)

LISP strings only have active portions. There are no fill pointers. LISP does not have the following functions:

```
array-has-fill-pointer-p
fill-pointer
vector-pop
vector-push
vector-push-extend
```

Strings 18

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

Strings 18

String Access (299)

The LISP char and schar functions execute at the same speed. All strings are simple strings in LISP.

Structures

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

Structures 19

Structure Support (305)

Ignore all of chapter 19 of Common LISP. LISP does not support structures. It does not have the defstruct macro.

The Evaluator

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

The Evaluator 20

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

Run-Time Evaluation of Forms (321)

LISP does not have the following special variables:

```
*applyhook*
*evalhook*
```

LISP does not have the following functions:

```
applyhook
evalhook
```

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, before any diagnostic message or prompting character for the next input line. LISP prints only the primary value returned from a function; the / variable holds a list of all values returned.

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 is fatal, causing the stack to be emptied and the user returned to the top-level loop -- the bottom of the stack.

Streams

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

Streams 21

Standard Streams (327)

LISP stream special variables have the following values:

```
    *standard-input*    #<STREAM TO $INPUT                              >
    *standard-output*   #<STREAM TO $OUTPUT                             >
```

LISP does not support the following special variables:

```
    *debug-io*
    *error-output*
    *terminal-io*
    *trace-output*
    *query-io*
```

Creating New Streams (329)

LISP does not have the following macro:

```
    with-output-to-string
```

Operations on Streams (332)

LISP does not have the stream-element-type function.

Input/Output 22

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

Input/Output

Printed Representation of LISP Objects (333)

The LISP reader operates as described in Common LISP, with the exceptions noted below.

Parsing of Numbers and Symbols (339)

LISP does not recognize the following patterns as valid:

```
ppppp:xxxxx
ppppp::xxxxx
```

LISP does not have the following special variables:

```
*read-base*
*read-suppress*
```

Macro Characters (346)

LISP interprets the ` read macro character so that a backquoted form, when evaluated, produces a result equal to the interpretation shown in Common LISP.

Standard Dispatching Macro Character Syntax (351)

LISP supports only the dispatching macro character uses of #' and #\. Ignore the rest of this subsection of Common LISP.

The Readtable (360)

LISP does not have the *readtable* special variable or the following functions:

```
copy-readtable
get-dispatch-macro-character
readtablep
```

What the Print Function Produces (365)

LISP does not define the output formats of:

```
bit-vectors
complex numbers
pathname objects
random-state objects
ratios

symbols interned in packages
```

LISP does not have the following special variables:

```
*print-array*
*print-base*
*print-circle*
*print-escape*
*print-gensym*

*print-length*
*print-level*
*print-pretty*
*print-radix*
```

Input Functions (374)

LISP supports only character stream input.

Input From Character Streams (374)

LISP does not have the *read-default-float-format* special variable or the following functions:

```
parse-integer
read-char-no-hang
read-delimited-list
read-preserving-whitespace
```

Input From Binary Streams (382)

LISP does not allow input from binary streams. It does not have the read-byte function.

Output Functions (382)

LISP supports only character stream output.

Output to Character Streams (382)

LISP does not have the finish-output function.

Output to Binary Streams (385)

LISP does not allow output to binary streams. It does not have the write-byte function.

Formatted Output to Character Streams (385)

LISP does not support formatted output. It does not have the format function and does not recognize the following format directives:

```
~A
~b
~C
~D
~E

~F
~G
~O
~P
~R

~S
~T
~mincol,colinc,minpad,padchar<str~>
~(str~)
~[str()~;strl~; . . .  ~;strn~]

~{str~}
~;
~]
~{
~}

~%
~&
~(
~*
~<

~<new line>
~?
~~
~^
~>

~|
~$
```

Querying the User (407)

LISP does not have the following functions:

```
y-or-n-p
yes-or-no-p
```

File System Interface

This chapter supplements chapter 23 of Common LISP. The LISP interface with the NOS/VE file system is described.

File System Interface

File Names (409)

LISP does not support the Common LISP pathname concept. Files must be referenced by namestrings containing valid NOS/VE file references.

Pathnames (410)

LISP does not have data objects of type pathname. Pathname components are not recognized. Namestrings cannot be converted to pathnames.

Pathname Functions (413)

LISP does not have the special variable *default-pathname-defaults* or the following functions:

```
directory-namestring
enough-namestring
file-namestring
host-namestring
make-pathname

merge-pathnames
namestring
parse-namestring
pathname
pathname-device

pathname-directory
pathname-host
pathname-name
pathname-type
pathname-version

pathnamep
truename
user-homedir-pathname
```

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 level of the operating system. An action or assignment within LISP which violates 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))
```

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 does not have the with-open-file macro.

open Function Keywords (418)

LISP supports these keywords as described in Common LISP, with the exceptions mentioned in the following subsections.

:element-type (419)

LISP does not recognize the following arguments for the :element-type keyword:

```
    bit
    signed-byte
    (signed-byte n)
    unsigned-byte
    (unsigned-byte n)
```

The :default argument specifies that the unit of transaction is a string-character.

:if-exists (420)

The following :if-exist keyword arguments are accepted but have no effect:

```
    :new-version

    :rename-and-delete
```

LISP does not recognize the :supercede argument.

Renaming, Deleting, and Other File Operations (423)

LISP does not have the following functions:

```
    delete-file
    file-author
    file-length
    file-position
    file-write-date

    probe-file
    rename-file
```

The ve-command function and corresponding NOS/VE SCL commands can be used for these functions.

Loading Files (426)

The filename parameter of the LISP load function must specify a namestring. Object file types do not exist for LISP and therefore cannot be loaded. The :verbose parameter is always NIL (no comments are written to the output file.)

LISP does not have the special variable *load-verbose*.

Accessing Directories (427)

LISP does not have the directory function. The ve-command function and the NOS/VE SCL command display_catalog can be used for this operation.

Errors 24

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

Errors 24

LISP signals an error for the first encountered incorrect argument of a form. There is no interactive debugger.

Appendix C 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. All errors are fatal and return the user to the bottom of the recursion stack (the top level of the interpreter.)

The special variable *break-on-warnings* and the following functions do not exist in LISP:

```
break
cerror
error
```

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.

LISP does not have the assert macro.

Special Forms for Exhaustive Case Analysis (435)

LISP does not have the following macros:

```
ctypecase
ecase
etypecase
```

Miscellaneous Features 25

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

Miscellaneous Features

The Compiler (438)

LISP does not support a separate compiler. It therefore does not have the following functions:

```
compile
compile-file
disassemble
```

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. Do not worry about the speed of code until the compiler is available.

Documentation (445)

LISP does not have the documentation function.

Debugging Tools (440)

LISP does not have the following functions:

```
appropos
appropos-list
describe
dribble
ed

inspect
room
step
time
trace

untrace
```

$save-lisp

This function allows you to save a LISP workspace between terminal sessions. $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.

Please remember that $LOCAL.LISP_BASE_SYSTEM_SPACES might not execute properly under future versions of LISP. To execute this file, you must use the NOS/VE SCL commands

```
SET_PROGRAM_ATTRIBUTES ADD_LIBRARY=$SYSTEM.LISP.BOUND_PRODUCT
LISP
```

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

Substituting for the ed Function (442)

LISP does not have an internal editor. As a substitute for the ed function, you can interrupt the LISP job and use the full screen editor of NOS/VE from within LISP. To do this, enter the function:

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

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

More information about the SCL EDIT_FILE command and the full screen editor can be found in the Full Screen Editor Tutorial/Usage manual.

Files edited with the full screen editor can be subsequently read by LISP if you enter the function:

```
    (load "filename")
```

The argument filename 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.

Environment Inquiries (443)

You can use the LISP ve-command function with NOS/VE SCL commands as arguments to substitute for many of the Common LISP functions listed in the following subsections.

Time Functions (443)

LISP does not have the following functions:

```
    decode-universal-time
    get-internal-real-time
    get-internal-run-time
    encode-universal-time
    get-decoded-time

    get-universal-time
    sleep
```

LISP does not have the following constant:

```
    internal-time-units-per-second
```

Other Environment Inquiries (447)

LISP returns the following values for functions in this section:

Function	Value	Obtained From
lisp-implementation-type	"LISP/VE"	released code
lisp-implementation-version	"Version aaaaa"	aaaaa is a string supplied in the released code
long-site-name	*NOVALUE*	
machine-instance	yyy	yyy is the integer CYBER 180 serial number known to NOS/VE
machine-type	"zz"	zz is the string for the CYBER 180 processor type known to NOS/VE
machine-version	"CDC CYBER 800 series"	released code
short-site-name	*NOVALUE*	

LISP does not have the following functions:

```
software-type
software-version
```

LISP does not have the following special variable:

```
*features*
```

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. Symbols can have static scope (be globally bound) or dynamic scope (be locally bound only within the form currently using them).

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. The object either contains data or contains a pointer to data.

CDR

The portion of a cons cell not included in the CAR. The CDR portion normally contains a pointer to the next element in the list (in effect, CDR points to the rest of the list). See also Dotted Pair.

Cons Cell

The fundamental structure of data storage. A cons cell consists of a CAR portion and a CDR portion.

Constant

A symbol whose binding does not change.

D

Dotted List

A list that is not a true list because its last CDR does not point to NIL.

Dotted Pair

A cons cell construct that is not a list. In a dotted pair cons cell, both the CAR portion and the CDR portion either contain data or a pointer to data; no pointer to another list element exists.

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.

Event

An unexpected or erroneous state. Events are caused by errors or interrupts. Events can be invoked by user code.

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 contains elements separated by blanks and enclosed within parentheses. Lists can be of three types: special forms, macro calls, and function calls. List usually refers to a true list.

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. You should not create functions that cause side effects.

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 that ends with an element whose CDR points to NIL. Contrast with Dotted List.

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.

Character Set

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 B-1. ASCII Character Set Table

Decimal	ASCII Code Hexadecimal	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	
024	18	030	CAN	Cancel	
025	19	031	EM	End of medium	
026	1A	032	SUB	Substitute	
027	1B	033	ESC	Escape	

(Continued)

Table B-1. ASCII Character Set Table (Continued)

Decimal	ASCII Code Hexadecimal	Octal	Graphic or Mnemonic	ASCII Name or Meaning	LISP Definition
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	
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	

(Continued)

Table B-1. ASCII Character Set Table (Continued)

Decimal	ASCII Code Hexadecimal	Octal	Graphic or Mnemonic	ASCII Name or Meaning	LISP Definition
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	
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	

(Continued)

Table B-1. ASCII Character Set Table (Continued)

Decimal	ASCII Code Hexadecimal	Octal	Graphic or Mnemonic	ASCII Name or Meaning	LISP Definition
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 C

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 SCL LISP command. The output file also receives information summarizing such things as variable bindings in effect when the error was detected.

Neither the Common LISP special variable *error-output* nor the NOS/VE $ERRORS file name function are used.

Each fatal error message is prefixed by the characters

```
--LISP ERROR--
```

Nonfatal (informative) error messages are not prefixed.

Fatal Errors

Fatal errors empty the stack but do not abort LISP.

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 valve 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 valve 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 valve 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 valve 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 read table. 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.

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.

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.

Comma used outside of backquote.

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

User Action: Correctly place the comma.

Complex numbers are not yet implemented.

Description: The current version of LISP does not support complex numbers.

User Action: Change your algorithm to use a different representation for the number.

Complex numbers not supported in current implementation.

Description: The current version of LISP does not support complex numbers.

User Action: Change your algorithm to use a different representation for the number.

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.

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 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 cannot be found. xxxxxxxx

Description: The read function attempted to process the file identified by the namestring xxxxxxxx. That file is not accessible to the LISP job.

User Action: If the file is another user's, you might not have permission to read it. If the file does not exist, you must interrupt the LISP job and create the file.

Further Information: See chapter 25 and the NOS/VE SCL System Interface Usage manual.

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 SCL System Interface Usage manual.

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 SCL System Interface 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.

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 corectly specified.

Ill-formed or illegal &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 backquote syntax.

Description: Your backquoted data structure is not specified in a manner that uses the rules allowed by Common LISP.

User Action: Check for `@form or `basic that is a list or a vector, or for a form beginning with a period..

Further Information: See Common LISP page 349.

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, such as a complex number.

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.

Improper substring for comparison.

Description: The substring argument specified is not a valid substring. Comparison is not possible.

User Action: Check the substring content.

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.

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 a 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.

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.

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.

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 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 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.

Unpaired item in keyword portion of macro call.

Description: Each keyword parameter must have a correponding 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.

Use #' for functional args.

Description: The #' macro character syntax can only be used to represent an abbreviation of the function special form; for example, (#' (lambda (y) (+ x y)))) is the same as (function (lambda (y) (+ x y)))). Your current usage does not conform.

User Action: Redesign your program.

Further Information: See Common LISP page 87.

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 — Bad clause in xxxxxxxx.yyyyyyyy

Description: The CDC-written form being evaluated is coded in LISP, using ~S. The argument xxxxxxxx indicates the form involved; the value yyyyyyyy indicates the clause encountered.

User Action: Correct the clause; check for undefined variables.

~S — Illegal type specifier to typep.

Description: The type specifier found is not one defined to LISP.

User Action: Check for a typographical error.

~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 — Ill-formed keyword arg in ~S.

Description: A required symbol is probably missing from the keyword argument found.

User Action: Check for a misplaced parenthesis.

~S — Non-symbol variable name in arglist of ~S.

Description: Variable names must be valid symbols.

User Action: Check the names in the list 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 has an odd number of items in its property list.

Description: Property lists must contain an even number of items.

User Action: Check the property list for a missing item. Check that the correct object is identified as the property list.

~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 illegal or unknown keyword.

Description: The form referenced by the directive contains a keyword LISP does not recognize.

User Action: Check the form for a typographical error in the keyword or for an extra colon before a symbol.

~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 floating point number.

Description: The argument referenced by the directive must be a floating point number when used in its current context.

User Action: Check that the correct argument is referenced. If so, convert the number to floating point and retry the evaluation.

~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 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 not a number.

Description: The argument referenced by the directive must be a number in its current context.

User Action: Check for a typographical error.

~S: index too large.

Description: The argument referenced by the directive is out of bounds for use as an index variable.

User Action: Check for a typographical error.

~S: index too small.

Description: The argument referenced by the directive is out of bounds for use as an index variable.

User Action: Check for a typographical error.

~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.

~S: invalid output type specifier. output-type-spec

Description: The specifier indicated as output-type-spec is not recognizable.

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

Nonfatal Errors

#<array printer not implemented.>

Description: You have used a function that would normally produce a listing of an array as a response. (This occurs when you use aref.) The LISP array printer is not implemented and such a listing cannot be created. The function was evaluated normally.

User Action: None. This is an informative message only.

Index of LISP Symbols D

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.

Symbol	Type	Common LISP Page	Notes
*	function	199	
*	variable	325	
**	variable	325	
***	variable	325	
+	function	199	
+	variable	325	
++	variable	325	
+++	variable	325	
-	function	199	
-	variable	325	
/	function	200	
/	variable	325	
//	variable	325	
///	variable	325	
/=	function	196	
1+	function	200	
1-	function	200	
<	function	192	
<=	function	196	
=	function	196	
>	function	196	
>=	function	196	
"	read macro	347	
(	read macro	346	
•	read macro	355	
`	read macro	349	
)	read macro	347	
,	read macro	351	
'	read macro	347	
;	read macro	347	
acons	function	279	
adjoin	function	276	
adjust-array	function	297	
alpha-char-p	function	235	
alphanumericp	function	236	
and	macro	82	
append	function	268	
apply	function	107	
aref	function	290	
array-dimension	function	292	

Symbol	Type	Common LISP Page	Notes
array-dimensions	function	292	
array-in-bounds-p	function	292	
array-rank	function	292	
array-total-size	function	292	
arrayp	function	76	
assoc	function	280	
assoc-if	function	280	
assoc-if-not	function	280	
atom	function	73	
block	special form	119	
boole-xor	constant	222	
both-case-p	function	235	
boundp	function	90	
butlast	function	271	
c----r	function	263	caaaar thru cddddr
case	macro	117	
ccase	macro	437	
catch	special form	139	
car	function	262	
cdr	function	262	
ceiling	function	217	
char	function	300	
char-bit	function	243	
char-bits	function	243	
char-code	function	239	
char-code-limit	constant	233	
char-control-bit	constant	243	
char-downcase	function	241	
char-equal	function	239	
char-font	function	240	
char-font-limit	constant	234	
char-greaterp	function	239	
char-hyper-bit	constant	243	
char-int	function	242	
char-lessp	function	239	
char-meta-bit	constant	243	
char-name	function	242	
char-not-equal	function	239	
char-not-greaterp	function	239	
char-not-lessp	function	239	
char-super-bit	constant	241	
char-upcase	function	241	
char/=	function	237	
char<	function	237	
char<=	function	237	
char=	function	237	
char>	function	237	
char>=	function	237	
character	function	241	
characterp	function	75	

Symbol	Type	Common LISP Page	Notes
check-type	macro	433	
clear-input	function	380	
clear-output	function	384	
close	function	332	
code-char	function	240	
coerce	function	51	
cond	macro	116	
cons	function	266	
consp	function	74	
constantp	function	324	
copy-alist	function	268	
copy-list	function	268	
copy-seq	function	248	
copy-tree	function	269	
declare	special form	153	only declaration specifier special implemented
define-modify-macro	macro	101	
define-setf-method	macro	105	
defmacro	macro	145	lexical environments not implemented
defparameter	macro	68	
defsetf	macro	102	
defun	macro	57	
defvar	macro	68	
digit-char	function	241	
digit-char-p	function	236	
do	macro	122	
do*	macro	122	
dolist	macro	126	
dotimes	macro	126	
eighth	function	266	
eq	function	77	
eql	function	78	
equal	function	80	
equalp	function	81	
eval	function	321	
evenp	function	196	
exp	function	203	
expt	function	203	
fboundp	function	90	
fceiling	function	217	
ffloor	function	217	
fifth	function	266	
first	function	266	
flet	special form	113	
float	function	214	
float-sign	function	218	
floatp	function	75	
floor	function	215	
fmakunbound	function	92	
force-output	function	384	
fourth	function	266	

Symbol	Type	Common LISP Page	Notes
fresh-line	function	384	
fround	function	217	
ftruncate	function	217	
funcall	function	108	
function	special form	87	
functionp	function	76	
gcd	function	202	
gensym	function	169	
get	function	164	
get-macro-character	function	362	
get-output-stream-string	function	336	
get-properties	function	167	not available for the following place forms: apply, bit, char, char-bit, documentation, elt, fill-pointer, gethash, ldb, mask-field, sbit, schar, string-char, subseq, svref
get-setf-method	function	106	
get-setf-method-multiple-value	function	107	
getf	function	166	not available for the following place forms: apply, bit, char, char-bit, documentation, elt, fill-pointer, gethash, ldb, mask-field, sbit, schar, string-char, subseq, svref
go	special form	133	
identity	function	448	
if	special form	115	
input-stream-p	function	332	
int-char	function	242	
integer-length	function	224	
integerp	function	74	
intern	function	184	
intersection	function	277	:key not implemented
isqrt	function	205	
keywordp	function	170	
labels	special form	113	
last	function	267	
lcm	function	202	
ldiff	function	272	
length	function	248	
let	special form	110	
let*	special form	111	
lisp-implementation-type	function	447	
lisp-implementation-version	function	447	

Symbol	Type	Common LISP Page	Notes
nconc	function	269	
nintersection	function	277	:key not implemented
nil	constant	72	
ninth	function	266	
not	function	82	
nreconc	function	269	
nreverse	function	248	
nstring-capitalize	function	304	
nstring-downcase	function	304	
nstring-upcase	function	304	
nsublis	function	275	
nsubst	function	274	
nsubst-if	function	274	
nsubst-if-not	function	274	
nsubstitute	function	256	
nsubstitute-if	function	256	
nsubstitute-if-not	function	256	
nth	function	265	
nthcdr	function	267	
null	function	73	
numberp	function	74	
nunion	function	276	
oddp	function	196	
open	function	418	
or	macro	83	
output-stream-p	function	332	
pairlis	function	280	
peek-char	function	379	
plusp	function	196	
pop	macro	271	not available for the following place forms: apply, aref, bit, char, char-bit, documentation, elt, fill-pointer, gethash, ldb, mask-field, sbit, schar, string-char, subseq, svref
position	function	257	
pprint	function	383	
prinl	function	383	
prinl-to-string	function	383	
princ	function	383	
princ-to-string	function	383	
print	function	383	
proclaim	function	156	
prog	macro	131	
prog*	macro	131	
progl	macro	109	
prog2	macro	109	
progn	special form	109	
psetf	macro	97	
psetq	macro	92	

Symbol	Type	Common LISP Page	Notes
push	macro	269	not available for the following place forms: apply, aref, bit, char, char-bit, documentation, elt, fill-pointer, gethash, ldb, mask-field, sbit, schar, string-char, subseq, svref
quote	character	346	
quote	special form	86	
rassoc	function	281	
rassoc-if	function	281	
rassoc-if-not	function	281	
read	function	375	
read-char	function	379	
read-from-string	function	380	
read-line	function	378	
rem	function	217	
remf	macro	167	not available for the following place forms: apply, bit, char, char-bit, documentation, elt, fill-pointer, gethash, ldb, mask-field, sbit, schar, string-char, subseq, svref
remprop	function	166	
replace	function	252	
rest	function	266	
return	macro	120	
return-from	special form	120	
revappend	function	269	
reverse	function	248	
rotatef	macro	99	
round	function	215	
rplaca	function	272	
rplacd	function	272	
$save-lisp	function	N/A	unique to LISP
schar	function	300	
second	function	266	
set	function	92	
set-char-bit	function	244	
set-dispatch-macro-character	function	364	
set-macro-character	function	362	
set-syntax-from-char	function	361	
setf	macro	94	not available for the following place forms: apply, bit, char, char-bit, documentation, elt, fill-pointer, gethash, ldb, mask-field, sbit, schar, string-char, subseq, svref

Symbol	Type	Common LISP Page	Notes
setq	special form	91	
seventh	function	266	
shiftf	macro	97	
short-site-name	function	448	
signum	function	206	
sin	function	207	
sixth	function	266	
sqrt	function	205	
standard-input	variable	327	
standard-output	variable	327	
streamp	function	332	
string	function	304	
string-capitalize	function	303	
string-char-p	function	235	
string-downcase	function	303	
string-equal	function	301	
string-greaterp	function	302	
string-left-trim	function	302	
string-lessp	function	302	
string-not-equal	function	302	
string-not-greater-p	function	302	
string-not-lessp	function	302	
string-right-trim	function	302	
string-trim	function	302	
string-upcase	function	303	
string/=	function	301	
string<	function	301	
string<=	function	301	
string=	function	300	
string>	function	301	
string>=	function	301	
stringp	function	75	
sublis	function	274	
subsetp	function	279	
subst	function	273	
subst-if	function	273	
subst-if-not	function	273	
svref	function	291	
symbol-function	function	90	
symbol-name	function	168	
symbol-package	function	170	
symbol-value	function	90	
symbolp	function	73	
symbol-plist	function	166	
t	constant	72	
tailp	function	275	
tagbody	special form	130	
tenth	function	266	
terpri	function	384	
the	special form	162	

Symbol	Type	Common LISP Page	Notes
third	function	266	
throw	special form	142	
tree-equal	function	264	
truncate	function	215	
type-of	function	52	
typecase	macro	118	
typep	function	72	
union	function	276	:key not implemented
unless	macro	115	
unread-char	function	379	
unwind-protect	special form	140	
upper-case-p	function	135	
values	function	134	
values-list	function	135	
vector-p	function	75	
ve-command	function	N/A	unique to LISP
warn	function	432	
when	macro	115	
with-input-from-string	macro	330	
with-open-steam	macro	330	
write	function	382	
write-char	function	384	
write-line	function	384	
write-string	function	384	
write-to-string	function	383	
zerop	function	195	

Tautology Proving Example E

This appendix contains an example of LISP use. There is no corresponding chapter in Common LISP.

The sample LISP statement file shown in figure E-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?") (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")))
    )
```

(Continued)

Figure E-1. Theorem-Prover Code

(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 simplfication.
; 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)))
```

(Continued)

Figure E-1. Theorem-Prover Code

(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))))))
)
```

(Continued)

Figure E-1. Theorem-Prover Code

(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)))))
```

(Continued)

Figure E-1. Theorem-Prover Code

(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))))))


; The following decodes the result of REDUCE and prints knowledgable
; 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 E-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

LISP for NOS/VE Language Definition Usage Supplement 60486213 01

We would like your comments on this manual. While writing it, we made some assumptions about who would use it and how it would be used. Your comments will help us improve this manual. Please take a few minutes to reply.

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/System
___ For Comprehensive Reference
___ For Quick Look-up

What programming languages do you use? ____________________

How Do You Like This Manual? Check those that apply.

Yes	Somewhat	No	
___	___	___	Is the manual easy to read (print size, page layout, and so on)?
___	___	___	Is it easy to understand?
___	___	___	Is the order of topics logical?
___	___	___	Are there enough examples?
___	___	___	Are the examples helpful? (___ Too simple ___ Too complex)
___	___	___	Is the technical information accurate?
___	___	___	Can you easily find what you want?
___	___	___	Do the illustrations help you?
___	___	___	Does the manual tell you what you need to know about the topic?

Comments? If applicable, note page number and paragraph.

Would you like a reply? ___ Yes ___ No

Continue on other side

From:

Name ____________________ Company ____________________

Address ____________________ Date ____________________

____________________ Phone No. ____________________

____

Please send program listing and output if applicable to your comment.

TAPE TAPE

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division
P.O. BOX 3492
Sunnyvale, California 94088-3492

CUT ALONG LINE

FOLD FOLD

CONTROL DATA