Soft Warehouse

Handcrafted Software for the Mind

Reference Manual

muLISP™ 90

LISP Language Programming Environment

Table of Contents

Title Page i
Copyright Notice ii
Preface iii
Table of Contents v

Chapter 1. Introduction

1.1 Major Features 2
1.1.1 Machine Language Kernel 2
1.1.2 Program Development System 3
1.1.3 Source File Library 4
1.1.4 Demonstration Game Programs 5
1.1.5 Tutorial System and Documentation 5
1.2 **muLISP** Diskette Files 6
1.3 System Requirements 8
1.3.1 IBM PC Compatible Computers 8
1.3.2 NEC PC-9801 Compatible Computers 8
1.3.3 Other MS-DOS Computers 8
1.4 Making A Working Copy 9
1.4.1 Hard Disk Procedure 9
1.4.2 Diskette Procedure 10
1.5 Keyboard Notation 11

Chapter 2. Fundamentals

2.1 Starting **muLISP** 13
2.2 Read-eval-print Loop 14
2.3 Line Editing 15
2.4 Programming 17
2.5 Source Files 19
2.6 Memory Image Files 20

Chapter 3. Learning muLISP

3.1 Demonstration Programs 23
3.2 Tutorial System 25

Chapter 4. Interpreter

4.1 Data Objects 27
4.1.1 Symbols 28
4.1.2 Numbers 29
4.1.3 Conses 30
4.2 Memory Management 31
4.2.1 Data Spaces 31
4.2.2 Garbage Collection 32
4.2.3 Data Space Reallocation 32
4.3 Break Option Menu 33
4.4 Console Interrupts 34
4.5 Error Messages 34

Chapter 5. Primitives

5.1 Predicates 40
5.1.1 Recognizers 40
5.1.2 Comparators 45
5.1.3 Logical Operators 50
5.2 Lists 53
5.2.1 Selectors 53
5.2.2 Constructors 60
5.2.3 Modifiers 70
5.2.4 Deletion 77
5.2.5 Search 80
5.2.6 Substitution 85
5.2.7 Sets 89
5.2.8 Association Lists 94
5.2.9 Sorting 98
5.3 Symbols 101
5.3.1 Value 101
5.3.2 Property List 108
5.3.3 Definition 112
5.4 Numbers 121
5.4.1 Precision 122
5.4.2 Numerical Predicates 125
5.4.3 Arithmetic Operators 131
5.4.4 Component Extraction 138
5.4.5 Bit-wise Logical Operators 144
5.4.6 Random Numbers 148
5.5 Print Names 151
5.5.1 Character Functions 151

5.5.2 String Functions 158
5.5.3 Symbol Creation 168
5.6 Control Constructs 169
5.6.1 Sequencing 169
5.6.2 Conditionals 173
5.6.3 Iteration 176
5.6.4 Non-local Exits 179
5.6.5 Subprocess Invocation 183
5.7 Interpreter 184
5.7.1 Form Evaluation 184
5.7.2 Function Application 189
5.7.3 Mapping Functions 191
5.7.4 Mapping Predicates 194
5.7.5 Macro Expansion 197
5.7.6 Memory Management 200
5.8 Communication 206
5.8.1 Input Functions 206
5.8.2 Read macro Characters 214
5.8.3 Input Control Variables 218
5.8.4 Output Functions 220
5.8.5 Output Control Variables 225
5.8.6 Console Screen 229
5.8.7 Hardware Interface 238
5.9 File System 249
5.9.1 Input Files 250
5.9.2 Output Files 255
5.9.3 File and Directory 262
5.9.4 Memory Image Files 266
5.10 User Interface 269
5.10.1 Top-level Loop 269
5.10.2 Read-eval-print Loop 271
5.10.3 Interrupt and Error Handling 273
5.10.4 User Querying 278
5.10.5 Line Editor 279

Chapter 6. Source File Library

6.1 Arrays 281
6.2 Irrational Functions 286
6.3 Structures 294
6.4 Flavors 297
6.5 Closures 304
6.6 Turtle Graphics 309
6.7 Mouse Interface 313

Chapter 7. Program Development System

7.1 Building and Saving the PDS 318
7.2 The Editor 319
7.2.1 The Editor Screen 320
7.2.2 Selecting Menu Options 321
7.2.3 Command Mode Menu Options 322
7.2.4 Edit Mode Commands 326
7.2.5 Theory of Operation 332
7.3 The Debugger 335
7.3.1 Command Mode Menu Options 335
7.3.2 Debugger Display 337
7.4 The Window Manager 339
7.4.1 Window Manager Commands 340
7.4.2 Window Manager Utilities 341

Chapter 8. Compiler

8.1 User Interface 346
8.2 Compilation Procedure 349
8.3 Performance Considerations 351
8.3.1 Interpreter Overhead 351
8.3.2 In-line Machine Code 352
8.3.3 Speed Improvements 352
8.4 Compiler Errors 354
8.4.1 Error Messages 354
8.4.2 Internal Errors 355
8.5 Potential Problems 356
8.5.1 Compiled Function Behavior 356
8.5.2 Name Conflicts 357
8.5.3 Redefining Primitives 358

Chapter 9. Machine-coded Routines

9.1 Data Representation 360
9.2 Argument Passing 362
9.3 Calling Machine-coded Routines 363
9.4 Loading and Linking 364

Chapter 10. LISP References

10.1 Professional Societies 367
10.2 Periodicals 368
10.3 Books 369

Appendixes

Appendix A. PC Video Modes 371
Appendix B. PC Screen Colors 372
Appendix C. PC Screen Characters 374
Appendix D. PC Extended Function Keys 376
Appendix E. Line Editor Commands 377
Appendix F. WordStar-like Editor Commands 378
Appendix G. EMACS-like Editor Commands 380
Appendix H. Base Page 382
Appendix I. Global Storage Area 389
Appendix J. Assembly Language Routine 390

Indexes

Function Index 391
Concept Index 409

Problem Report Form

License Agreement and Limited Warranty

Chapter 1. Introduction

Congratulations on your purchase of the **muLISP™ LISP Language Programming Environment.** It is a powerful system for implementing and delivering sophisticated software products on widely available MS-DOS based personal computers. The information in this manual is arranged as follows:

- **Chapter 1. Introduction** describes the major features of **muLISP**, the files on the master diskettes, the computer system requirements, how to make a working copy, and how the manual denotes keys on the keyboard.
- **Chapter 2. Fundamentals** describes how to start **muLISP**, how to enter commands, how to define functions, how to load source files, and how to load and save memory image files.
- **Chapter 3. Learning muLISP** describes how to load and play the demonstration game programs, and how to start the **muLISP** Tutorial System.
- **Chapter 4. Interpreter** describes the primitive data objects, the memory management system, the break options, how to interrupt a program, and the error messages.
- **Chapter 5. Primitives** describes the functions, special forms, macros, and control variables defined in the machine language kernel and in the COMMON.LSP source file.
- **Chapter 6. Source File Library** describes the facilities provided by the other source files included with **muLISP**. These facilities include arrays, irrational functions, structures, flavors, closures, turtle graphics, and mouse support.
- **Chapter 7. Program Development System** describes how to use the **muLISP** editor, debugger, and window manager.
- **Chapter 8. Compiler** describes how to compile functions using the **muLISP** native-code compiler.
- **Chapter 9. Machine Language Routines** describes how to write, load, and link machine language routines.
- **Chapter 10. LISP References** lists the societies, periodicals, and books relevant to LISP programming.

1.1 Major Features

muLISP consists of a machine language kernel, a program development system, a source file library, several demonstration game programs, a tutorial system, and this reference manual.

1.1.1 Machine Language Kernel

The machine language kernel is distributed as the executable COM file MULISP.COM. It includes the **muLISP** interpreter, garbage collector, and over 400 functions, special forms, and control variables. They are defined in machine language for maximum efficiency. Since the kernel consumes less than 50K (kilobytes) of memory, it leaves a large amount of computer memory free for application programs.

High performance has been achieved through the use of shallow variable binding, address typed data structures, and a closed pointer universe. The following summarizes the capabilities provided by the machine language kernel:

List Processing The kernel provides a large number of functions for selecting, constructing, recognizing, comparing, counting, searching, substituting, and modifying the elements of lists.

Merging and Sorting The kernel provides functions for efficiently merging and sorting lists using user-supplied predicates.

Arithmetic Operators The kernel provides operators (functions) for exact integer and rational arithmetic *and* adjustable precision approximate rational arithmetic. The radix used to express numbers is also adjustable.

Function Mapping The kernel provides functions and predicates for iteratively testing and/or manipulating the elements of any number of lists.

Character and String Processing The kernel provides functions for substring extraction, string searches, case conversion, and string comparisons (both case sensitive and insensitive). Hashing algorithm is used to rapidly intern new print names.

Communication Facilities The kernel provides functions for reading and writing character or binary data from or to the console, the system printer, or DOS files. Both sequential and random file access are supported.

Macro Characters The kernel's LISP reader can be modified by user-defined macro characters (terminating, non-terminating, and comment). For instance, the backquote character greatly simplifies macro definitions.

File System The kernel provides functions for opening, closing, renaming, and deleting files. Also provided are functions for changing the default DOS drive and path, determining a file date, and listing the files in a directory.

Memory Image Files The kernel provides functions for quickly loading and saving compact memory image files.

Control Constructs The kernel provides the following control constructs: "implicit PROGN", "implicit COND", COND, IF, PROGN, PROG1, LOOP, CATCH, THROW, UNWIND-PROTECT, and RETURN. The COMMON.LSP library file provides the following macro-defined control constructs: PROG2, WHEN, UNLESS, CASE, LET, LET*, DO, DO*, DOLIST, and DOTIMES.

Macros The **muLISP** interpreter allows macros to be expanded at run time or pseudo-compile time. Macros can be used to create user-defined control constructs. Functions can be defined to accept any number of arguments and/or to *not* evaluate their arguments.

Pseudo-code Compiler When a function is defined, the definition is automatically compiled into a compact pseudo-code. When a function definition is accessed, the definition is automatically decompiled. The use of pseudo-code to store function definitions reduces memory requirements by a factor of 3 *and* improves the speed of the interpreter.

Machine Language Routines If necessary for some special application or performance need, user-defined machine language routines can loaded and linked to the **muLISP** kernel.

Memory Management The kernel's two-pass, compacting garbage collector automatically reclaims unreferenced data structures. When necessary, data space boundaries are also adjusted to provide more room for heavily used spaces.

Subprocess Invocation The kernel provides functions for temporarily suspending **muLISP** in order to load and run an MS-DOS program or the MS-DOS command processor.

Generic MS-DOS The kernel will run on any of the wide variety of personal computers that use an MS-DOS compatible operating system. If running on a PC compatible computer, the kernel also supports color graphics in a variety of resolutions.

1.1.2 Program Development System

The **muLISP** program development system provides an interactive, window-based environment for software development.

Window Manager The window manager provides a uniform, menu-driven interface for application programs. It includes facilities for opening, splitting, closing, and changing windows.

Editor The screen-oriented text editor has been enhanced to facilitate the editing of LISP source files by incorporating such features as expression evaluation, blinking

parentheses, and undelete. With a single key stroke, editing can be suspended while you switch to a debug window or another edit window.

Debugger The debugger displays function trace output continuously or in single-step mode, making it easy to examine and/or modify intermediate results. Trace output is sent to a window separate from a program's output window.

Compiler The native code compiler *incrementally* compiles function definitions into native machine code, thereby improving the performance of speed-critical functions.

Runtime System After loading or compiling a **muLISP** application program, the program together with the **muLISP** Runtime System can be saved as a directly executable MS-DOS COM file or EXE file. This file can be made commercially available by agreeing to the terms of the **muLISP** Runtime License Agreement and paying the required fee. For details, write **Soft Warehouse, Inc.**, 3660 Waialae Avenue, Suite 304, Honolulu, Hawaii, 96816-3236, U.S.A.

1.1.3 Source File Library

The facilities provided by the source file library complement those provided by the machine language kernel.

Common LISP COMMON.LSP, along with the other source files, provides over 450 Common LISP functions, special forms, macros, and control variables.

Arrays ARRAY.LSP provides a set of functions and macros for creating, accessing, and recognizing multi-dimensional arrays.

Irrational Functions IRRATNAL.LSP provides functions for approximating the exponential, logarithmic, trigonometric, and inverse trigonometric functions to any desired degree of accuracy.

Structures STRUCTUR.LSP provides the DEFSTRUCT macro for creating named and unnamed record structures. Defining a structure in effect creates a new data type complete with accessors, constructors, recognizers, and copiers.

Flavors FLAVORS.LSP provides a set of functions and macros that facilitate object-oriented programming. It supports multiple inherited methods and instance variables, before and after demons, and whopper methods.

Closures CLOSURE.LSP provides functions for creating and manipulating dynamic closures of functions over a list of variables whose values are local to the closures.

Graphics GRAPHICS.LSP provides functions for doing point, line, and turtle graphics. Included as examples are functions for recursively drawing the popular "C" and "Dragon" curves.

Mouse Interface MOUSE.LSP provides functions for reading the position and button status of the Microsoft Mouse™.

1.1.4 Demonstration Game Programs

muLISP includes several interesting game and puzzle programs that demonstrate good **muLISP** and AI programming techniques.

Animal By asking a series of yes-or-no questions, this word game builds and uses a discrimination net to logically determine what animal the player has chosen.

Doctor This game, sometimes called **Eliza**, turns your computer into a psychologist of the Rogerian school of psychology, in which the doctor lets the patients resolve their own problems by asking leading questions.

Eights This program uses heuristic search techniques to "slide" eight "tiles" that are randomly arranged in a 3 by 3 "tray" into a standard configuration.

Tower of Hanoi This program uses character graphics and sound effects to display the recursive solution to this famous puzzle.

Metamind This program can play either side of this code breaking game. The object is to determine the colors and arrangement of four objects based on clues provided by the code maker.

1.1.5 Tutorial System and Documentation

Tutorial System The interactive tutorial system provides an on-line introduction to LISP programming. Topics include the basic LISP data structures and primitives, recursion and iteration, numerical programming techniques, and turtle graphics.

Help Facility The on-line help facility provides ready information on the arguments and/or initial value of the **muLISP** primitives, and the page number in the manual containing more detailed information.

muLISP Reference Manual This detailed manual describes all aspects of **muLISP** programming.

1.2 muLISP Diskette Files

The following files are on the master **muLISP** diskette(s):

- README — Information not yet included in this manual (display contents using the MS-DOS TYPE command)

Executable Files:

- MULISP.COM — **muLISP** interpreter (see Section 2.1)
- COMTOEXE.COM — COM file to EXE file conversion utility (see Section 5.9.4)
- HERCULES.COM — Resident program for Hercules-compatible graphics cards (see Section 5.8.6)

Program Development System:

- EDIT.LSP — Source file editor (see Section 7.1)
- DEBUG.LSP — Program debugger (see Section 7.2)
- WINDOWS.LSP — Window manager (see Section 7.3)
- ADDRESS.LSP — Sample data entry program (see Section 7.3)
- HELP.LSP — Help facility (see Section 3.1)

Source File Library:

- COMMON.LSP — Common LISP library (see Chapter 5)
- ARRAY.LSP — Array library (see Section 6.1)
- IRRATNAL.LSP — Irrational function library (see Section 6.2)
- STRUCTUR.LSP — Structure library (see Section 6.3)
- FLAVORS.LSP — Flavors library (see Section 6.4)
- CLOSURE.LSP — Dynamic closure library (see Section 6.5)
- GRAPHICS.LSP — Turtle graphics library (see Section 6.6)
- MOUSE.LSP — Microsoft Mouse library (see Section 6.7)
- NECGRAPH.LSP — NEC PC-9801 graphics library (see file)
- STATS.LSP — Memory usage statistics library (see file)

Demonstration Game Programs (see Section 3.1):

- ANIMAL.LSP — Animal game
- ANIMAL.MEM — Animal game's prebuilt data base
- DOCTOR.LSP — Doctor or Eliza
- EIGHTS.LSP — Eights puzzle
- HANOI.LSP — Tower of Hanoi puzzle
- METAMIND.LSP — Metamind game

Tutorial System (see Section 3.2):

- LESSONS.LSP — Tutorial System lesson driver
- MULISP1.LES — Data objects and primitive functions
- MULISP2.LES — Defining functions using recursion
- MULISP3.LES — Symbols, numbers, and conses
- MULISP4.LES — List processing & iterative functions
- MULISP5.LES — Numerical programming techniques
- MULISP6.LES — Implementing turtle graphics functions

Native code Compiler (see Chapter 8):

- COMPILE.LSP — Native code compiler loader
- COMPILE0.LSP — User-interface and utility routines
- COMPILE1.LSP — Function body code generator
- COMPILE2.LSP — In-line code generator for special functions and forms
- COMPILE3.LSP — Assembler and native code loader

1.3 System Requirements

muLISP runs on various Intel 8086, 8088, 80286, and 80386 based personal computers running the MS-DOS or compatible disk operating system. This section lists the minimum computer hardware and software required to run **muLISP**.

1.3.1 IBM PC Compatible Computers

- An IBM Personal Computer or Personal System/2 Computer running PC-DOS or the PS/2 operating system; *or* a PC compatible computer running Microsoft Corporation's MS-DOS operating system software, version 2.1 or later.
- A monochrome display adapter (MDA), a color graphics adapter (CGA), an enhanced graphics adapter (EGA), a video graphics adapter (VGA), a Hercules graphics adapter, *or* a compatible adapter installed in the computer.
- A monochrome monitor, a color monitor, an enhanced color monitor, *or* a multisync color monitor.
- At least 256K (kilobytes) of computer memory. **muLISP** can take advantage of up to 640K of memory.
- At least one double-sided (360K) 5 1/4 inch diskette drive *or* one double-sided (720K) 3 1/2 inch diskette drive.

1.3.2 NEC PC-9801 Compatible Computers

- An NEC PC-9801 Personal Computer *or* a PC-9801 compatible computer running Microsoft Corporation's MS-DOS operating system software, version 2.1 or later.
- A monochrome monitor *or* a color monitor.
- At least 256K of computer memory. **muLISP** can take advantage of up to 640K of memory.
- At least one double-sided (1.2M) 5 1/4 inch diskette drive *or* one double-sided (1.2M) 3 1/2 inch diskette drive.

1.3.3 Other MS-DOS Computers

- An Intel 8086, 8088, 80186, 80286, or 80386 based personal computer running Microsoft's MS-DOS operating system software, version 2.1 or later.
- At least 256K of computer memory. **muLISP** can take advantage of up to 640K of memory.
- At least one double-sided (360K) 5 1/4 inch diskette drive *or* one double-sided (720K) 3 1/2 inch diskette drive.

1.4 Making A Working Copy

As a convenience to authorized users, the **muLISP** diskette is normally *not* copy protected. It can be copied using standard DOS copying commands. In return, we ask that you abide by our copyright and *not* distribute copies of the software to unauthorized users.

As a safeguard against the loss, damage, or erasure of the **muLISP** diskette, you should make a working copy of the diskette *before* using **muLISP**. If the working copy is made on a diskette, the diskette must have a label bearing the following copyright notice:

muLISP™
Copyright © 1983, 1987, 1990 by Soft Warehouse, Inc.
All rights reserved worldwide

In addition to the copyright notice, it is a good idea to put the version number (i.e. 6.*nn*), the version date (i.e. *mm/dd/yy*), and the serial number on the diskette label. The version number and serial number are on the master **muLISP** diskette. The version number and version date are displayed when **muLISP** starts. When reporting problems, be sure to include the version number, version date, and serial number on the **muLISP** program **Problem Report Form** printed at the end of this manual.

Before making a working copy, we recommend write protecting the **muLISP** diskette. A 5 1/4 inch diskette is write protected by covering the notch on the right side of the diskette with a write-protect tab. A 3 1/2 inch diskette is write protected by moving the write-protect slide so the window is open. See your computer manuals for details.

1.4.1 Hard Disk Procedure

If your computer has a 5 1/4 inch or a 3 1/2 inch diskette drive for drive A and a hard disk drive for drive C, you can make a working copy of **muLISP** on the hard disk as follows:

- Turn on and boot the computer in the ordinary way from either the hard disk or a DOS system diskette in drive A.
- Insert the **muLISP** diskette in drive A.
- Switch to drive C by entering the DOS command

```
C:
```

- Make a new subdirectory named MULISP by entering the DOS command

```
MD MULISP
```

- Change to the MULISP subdirectory by entering the DOS command

```
CD MULISP
```

- Copy the files from the **muLISP** diskette to the MULISP subdirectory by entering the DOS command

```
COPY A:*.* C:
```

- After all the files have been transferred to the hard disk and the DOS prompt is redisplayed, remove the **muLISP** diskette from drive A and store it in a safe, cool, and dry place that is physically separated from the computer.

1.4.2 Diskette Procedure

If your computer has one or two diskette drives but no hard disk drive, you can make a working copy of **muLISP** on a diskette using the DISKCOPY.COM program described in the DOS manual.

Alternatively, if your computer has diskette drives for drives A and B, you can make a working copy of **muLISP** as follows:

- Turn on and boot the computer in the ordinary way from a DOS system diskette in drive A.
- Insert a blank diskette in drive B.
- Format the blank diskette by entering the DOS command

```
FORMAT B:
```

- Insert the **muLISP** diskette in drive A.
- Copy the files from the **muLISP** diskette to the blank working diskette by entering the DOS command

```
COPY A:*.* B:
```

- After all the files have been transferred to the working diskette and the DOS prompt is redisplayed, remove the **muLISP** diskette from drive A and store it in a safe, cool, and dry place that is physically separated from the working copy of **muLISP**.
- Remove the working diskette from drive B and place the previously described copyright notice on it.

1.5 Keyboard Notation

Throughout this manual keys on the computer keyboard are denoted by names printed in **bold** letters. Some keys may be marked with descriptive symbols instead of names that are spelled out. The following is an alphabetical list of the names and the corresponding symbols used on many keyboards:

Manual Name	Keyboard Symbol	Manual Name	Keyboard Symbol
Alt	Alt	**Ins**	Ins
Backspace	←	**Left arrow key**	←
Ctrl	Ctrl	**PgDn**	PgDn
Del	Del	**PgUp**	PgUp
Down arrow	↓	**Right arrow**	→
End	End	**Shift**	⇧
Enter	↵	**Space**	<space bar>
Esc	Esc	**Tab**	→|
Home	Home	**Up arrow**	↑

Sometimes you need to press two keys at the same time to obtain the desired effect. For example, when the manual says **Shift-Tab**, hold down the **Shift** key and press the **Tab** key. Similarly, **Ctrl-H** means hold down the **Ctrl** key and press the letter **H**, whereas **Alt-E** means hold down the **Alt** key and press the letter **E**.

When the manual says "enter a command", it means to type the command and then press the **Enter** key.

Chapter 2. Fundamentals

This chapter describes how to start **muLISP**, how to enter and edit commands, how to define functions, how to load source files, and how to load and save memory image files.

2.1 Starting muLISP

Once a working copy of the **muLISP** diskette has been made (see Section 1.4), it is a simple matter to start **muLISP** as follows:

- Turn on and boot your computer in the ordinary way from either a hard disk or a DOS system diskette in drive A. If you are not sure how to boot your computer, study the manuals that came with the computer and/or get help from someone experienced in using the computer.
- If your working copy of **muLISP** is on a diskette, insert the diskette in drive A and enter the command

  `A:MULISP`

- If your working copy of **muLISP** is on a hard disk, enter the command

  `MULISP`

  If this does not start **muLISP**, you will have to change to the drive and directory containing the file MULISP.COM before executing the above command. For example, the command

  `CD MULISP`

  changes to the MULISP subdirectory if one exists. See your computer's DOS manual for details on how to change the drive and directory. Once you have changed to the correct drive and directory, you must reissue the MULISP command to actually start **muLISP**. If you are unable to start **muLISP** from the hard disk, you can always start it from a diskette as described earlier.

Whichever method you use to start **muLISP**, there is a short delay while the program loads from the diskette or hard disk. When complete, **muLISP** displays the header message

```
muLISP computer MS-DOS Version 7.nn (mm/dd/yy)
Copyright (c) 1983, 1987, 1990 Soft Warehouse, Inc.
All Rights Reserved Worldwide
```

Appropriate numbers will appear for the version number (7.*nn*), month (*mm*), day (*dd*), and year (*yy*). The version number and version date, along with the serial number, should be included in all inquiries concerning **muLISP**.

The *computer* in the header message indicates the type of computer system on which **muLISP** is being run. The computer type affects which console screen functions are operative (see Section 5.8.6).

The **muLISP** prompt is displayed below the header message. The default prompt consists of the **$** character followed by a space and the console cursor. The prompt indicates the system is ready to accept console input. The prompt can be changed by modifying the string in the **muLISP** base page (see Appendix H).

2.2 Read-eval-print Loop

Following the prompt, you can enter LISP expressions written in list and/or dot notation. Multi-line expressions can be entered since an expression is not evaluated until all parentheses are balanced.

After you enter a complete LISP expression and press **Enter**, **muLISP** reads the expression, evaluates it, and prints the result beginning on a new line. This read-eval-print loop is repeated indefinitely until you enter a SYSTEM command, which terminates **muLISP** and returns control to DOS.

The following printout of a sample **muLISP** session demonstrates the read-eval-print loop. Comments are printed at the right margin preceded by semicolons, which is the **muLISP** comment character. Note that the comments need not be entered for the examples to work.

```
$ 'DOG                                                    ;Quote a symbol
DOG

$ (+ 5 -2 4)                                             ;Add three integers
7

$ (EQUAL 'DOG 'CAT)                                      ;Compare two symbols
NIL

$ (member 'dog '(cat cow dog pig))                          ;Search a list
(DOG PIG)

$ (SYSTEM)                                          ;Quit muLISP, return to DOS

A>                                                          ;The DOS prompt
```

By default **muLISP** converts lower case letters to upper case as they are read. However, if the control variable *READ-UPCASE* is set to NIL, lower case letters are not converted to upper case (see Section 5.8.3).

As a programming convenience, the last three expressions read by the read-eval-print loop are assigned to variables: the last expression to +, the second from last to ++, and the third from last to +++. Also the evaluated result of the last three expressions read

are assigned to variables: the last result to *, the second from last to **, and the third from last to ***. For example:

```
$ (APPEND '(TOM SAM JOAN) '(ANN JOE SUE))
(TOM SAM JOAN ANN JOE SUE)

$ *
(APPEND '(TOM SAM JOAN) '(ANN JOE SUE))

$ **
(TOM SAM JOAN ANN JOE SUE)
```

The pre-defined read-eval-print loop may well be inappropriate for your application. You can define your own driver function by setting the control variable DRIVER to the name of your driver function (see Section 5.10.1).

muLISP output to the console can be "frozen" and then restarted using **Ctrl-S** as a toggle. This is useful to prevent the scrolling of text off the screen before it can be read.

2.3 Line Editing

Often while using **muLISP** you will need to enter and/or edit a line of text. A blinking cursor appears while you are typing in and/or editing the text. When you are finished, press **Enter** or **Ctrl-M** to actually enter the line.

Control keys are used for line editing. There is a control key associated with each letter of the alphabet. To enter a control key, press its associated letter while holding down the **Ctrl** key. On PC compatible computers, the arrow keys located at the right end of the keyboard can also be used for line editing (see Section 1.5).

To correct a mistake made while entering a line of text, press **Backspace, Rub**, or **Ctrl-H** to delete the mistake and then type the correction.

To make a change near the beginning of the line without deleting the characters at the end, press **Left arrow** or **Ctrl-S** to move the cursor to the left without erasing characters. Note that **Backspace** deletes characters as it moves the cursor left, whereas **Left arrow** does not. If you move the cursor too far to the left, press **Right arrow** or **Ctrl-D** to move the cursor to the right.

A **token** is a string of letters or numeric characters terminated by a space, a non-alphanumeric character, or the end of the line. Press **Ctrl-Left arrow** or **Ctrl-A** to move the cursor left a token. Press **Ctrl-Right arrow** or **Ctrl-F** to move it right a token.

Press **Home, Ctrl-]**, or **Ctrl-Q S** (i.e. press **Ctrl-Q** and then press **S**) to move to the left end of the line. Press **End, Ctrl-**, or **Ctrl-Q D** to move to the right end of the line.

If the cursor is within a sequence of characters, press **Del** or **Ctrl-G** to delete the character at the cursor. Press **Ctrl-T** to delete the token at and to the right of the cursor.

If you have so muddled the line you are editing that you would like a fresh start, press **Ctrl-Y** to delete the whole line. Press **Ctrl-Q Y** to delete the right end of the line beginning with the character at the cursor. Press **Ctrl-Q Backspace**, **Ctrl-Q Rub**, or **Ctrl-Q H** to delete the left end of the line.

The effect of typing characters when the cursor is positioned within a line depends on the current text entry mode. In **insert mode**, characters under and to the right of the cursor are moved right to make room for a new character. In **overwrite mode**, the character under the cursor is replaced by a new character. Press **Ins** or **Ctrl-V** to toggle between insert mode and overwrite mode.

Press **Tab** or **Ctrl-I** to move right to the next tab stop. Note that in insert mode characters to the right of the cursor are moved right by a tab command.

To insert a control character in the line of text, press **Ctrl-P** and then press the control key associated with the character. This prevents the control character from being interpreted as an editing command.

Press the **F3** function key or **Ctrl-U** to insert the text from the previous edit line into the current edit line, so it can be edited and then entered again.

These line editing commands are summarized in Appendix E. Note that they are deliberately similar to the editing commands provided by the full-screen, source file editor (see Section 7.1).

The line editing commands can be reassigned to different control keys by modifying the **muLISP** base page (see Appendix H).

2.4 Programming

Although it does not take full advantage of the capabilities of **muLISP**, the following is a valid recursive definition of a function for computing the factorial of a number:

```
(DEFUN FACTORIAL (N)
  (COND
    ((ZEROP N)  1)
    (T (* N (FACTORIAL (SUB1 N)))) ) )
```

The **muLISP** interpreter incorporates numerous extensions to LISP that significantly increase program readability and execution speed, while substantially decreasing the storage requirements for function definitions. The following are the significant extensions:

- When it creates a new symbol for a print name, **muLISP** sets the value of the new symbol to be itself. This auto-quoting of symbols eliminates the need to explicitly quote symbols that are not otherwise assigned a value.
- As with many LISPs, **muLISP** generalizes the COND special form so all the forms following a nonNIL predicate are evaluated in turn. The value returned by this extended COND is the value of the last form evaluated.
- As with many LISPs, **muLISP** uses an implicit PROGN when evaluating function definitions. This eliminates the need for an explicit PROGN to evaluate multiple forms. In addition, **muLISP** uses an implicit COND when evaluating definitions. This often eliminates the need for an explicit COND to implement a branch in a definition.

Comparing the following recursive definition of FACTORIAL with the previous one illustrates some of the extended capabilities of **muLISP**:

```
(DEFUN FACTORIAL (N)
  ((ZEROP N)  1)
  (* N (FACTORIAL (SUB1 N))) )
```

With the exception of the PROG and GO control constructs, LISP is an exemplary structured language. The following features of **muLISP** eliminates the need for these constructs:

- The multiple-exit LOOP control construct provides a convenient way to program structured nonrecursive loops without the need for a GO construct.
- In **muLISP**, excess arguments in a function's formal argument list can be used as local variables within the function. Also, in-line LAMBDA bodies and the LET and LET* control constructs can be used to introduce local variables in functions (see Section 5.6.1) without the need for a PROG construct.

Therefore the PROG and GO control constructs are not primitively defined in **muLISP**. However if necessary for compatibility, the source file COMMON.LSP does provide an interpreter for them.

Instead of requiring numerous explicit control constructs, **muLISP** programs can be guided primarily by the tree structure of function definitions. Program control using implicit PROGNs and implicit CONDs results in function definitions whose structures and meanings are more transparent, require less memory to store, and are much more efficient (see Section 5.6).

As an example of how to replace explicit control constructs with implicit PROGNs and CONDs, consider the following iterative definitions of the FACTORIAL function:

```
(DEFUN FACTORIAL (N)                                    ;Conventional definition
  (PROG (M)
    (SETQ M 1)
  A (COND
      ((ZEROP N)
        (RETURN M) ) )
    (SETQ M (* M N))
    (SETQ N (SUB1 N))
    (GO A) ) )

(DEFUN FACTORIAL (N M)                                  ;muLISP definition
  (SETQ M 1)
  (LOOP
    ((ZEROP N)  M)
    (SETQ M (* M N))
    (SETQ N (SUB1 N)) ) )
```

The first is a conventional nonrecursive definition (It is not a workable **muLISP** function definition unless COMMON.LSP is loaded). The second is an equivalent **muLISP** definition that is also nonrecursive, but is clearly more structured and elegant. Note that the **muLISP** definition requires only 28 conses; the conventional one requires 37 conses. This is a significant space saving for a LISP system running in a limited address space.

After defining FACTORIAL, you can generate some really large factorials in a hurry. For example,

```
$ (FACTORIAL 100)
93326215443944152681699238856266700490715968264381621468592963895217599993229915608941
46397615651828625369792082722375825118521091686400000000000000000000000000
```

2.5 Source Files

Source files are ASCII text files that contain function definitions, variable assignments, and other commands. Numerous source files are included on the **muLISP** diskette (see Section 1.2). If *filename* specifies the drive, directory, and name of a source file, the command

(LOAD '*filename*)

reads and evaluates the expressions in *filename*. If *filename* is not found, LOAD returns NIL. If *filename* does not specify a file name extension, the extension LSP is used. If *filename* does not specify a drive and/or directory, the file is read from the default DOS drive and/or directory. For example, the command

```
(LOAD 'C:\\GAMES\\ANIMAL)
```

loads the source file ANIMAL.LSP from the C:\GAMES drive and directory. Note that two \ characters are required in *filename* where only one appears in DOS pathnames because \ is the single escape character in LISP (see Section 5.8.1).

An alternative way to load a source file is to include its name in the DOS command line used to start **muLISP**. **muLISP** will automatically load the source file immediately after MULISP.COM is loaded. For example, the DOS command

```
A:MULISP C:\GAMES\ANIMAL.LSP
```

loads the source file ANIMAL.LSP from the C:\GAMES drive and directory after MULISP.COM is loaded from drive A.

When an attempt is made to load a source file using the DOS command line and the file is not found, the warning message

File not found

is displayed after the **muLISP** header message.

When loading a source file using either of these methods, the file name extension LSP can be omitted. However, if there is also a memory image file with the same name, it is loaded instead of the source file (see Section 2.6).

LOAD commands in one source file can be used for chain-loading other source files.

2.6 Memory Image Files

muLISP memory image files are compact binary files that contain the environment present at the time they are saved. The environment consists of all the data structures, variable values, property values, and function definitions in **muLISP**. The command

(SAVE '*filename*)

saves the current **muLISP** environment as a memory image file on the drive and directory specified by *filename*. If *filename* does not specify a file name extension, the extension SYS is used. If *filename* does not specify a drive and/or directory, the file is saved on the default DOS drive and/or directory. For example, the command

```
(SAVE '\\PACIFIC\\WHALES)
```

saves the current environment as the memory image file WHALES.SYS in the \PACIFIC directory on the default drive. Note that two \ characters are required where only one appears in DOS pathnames because \ is the single escape character in LISP (see Section 5.8.1).

If *filename* specifies the drive, directory, and name of a memory image file, the command

(LOAD '*filename*)

restores the environment present at the time the memory image file was saved. Since control is immediately transferred to the just loaded environment, LOAD does not return a value. However, if *filename* is not found, LOAD returns NIL. If *filename* does not specify a file name extension, the extension SYS is used. If *filename* does not specify a drive and/or directory, the file is read from the default DOS drive and/or directory. For example, the command

```
(LOAD '\\PACIFIC\\WHALES)
```

loads the memory image file WHALES.SYS from the \PACIFIC directory on the default drive.

The environment present at the time the LOAD command is issued is replaced with that stored in the memory image file. Thus unlike source files, loading memory image files is not cumulative. Instead the new file's environment destroys the existing environment, including all function definitions.

An alternative way to load a memory image file is to include the name of the file in the DOS command line used to start **muLISP**. The memory image file will automatically be loaded immediately after MULISP.COM is loaded. For example, the DOS command

```
MULISP B:\PACIFIC\WHALES
```

loads the memory image file WHALES.SYS from the \PACIFIC directory on the default drive after MULISP.COM is loaded from the default drive.

When an attempt is made to load a memory image file using the DOS command line and the file is not found, the warning message

File not found

is displayed after the **muLISP** header message.

When a memory image file is loaded using either of the above methods, the environment is restored exactly as it was at the time of the save. After loading the file, the amount of memory allocated to each of the various data spaces is adjusted according to the computer's current memory size. This means that the current memory size does not necessarily have to be the same as when the file was created. However if there is insufficient memory to store the new environment, an

Insufficient Memory, Aborting

error occurs and **muLISP** is terminated.

Memory image files are useful for saving intermediate results of an interactive session, so the session can be continued at a future time or after a computer failure. They can also be used to save function definitions entered directly using the read-eval-print loop (see Section 2.2). However, this method of program development provides no way of easily editing existing definitions. Thus it is practical for developing only the most trivial of programs.

The preferred method of program development is to write and debug source files using the Program Development System (see Chapter 7). When the program is perfected, the source files can be saved as a memory image file that can be quickly loaded.

Chapter 3. Learning muLISP

This chapter describes how to load and play the demonstration game programs and how to start the **muLISP** Tutorial System.

3.1 Demonstration Programs

Several demonstration game programs are included with the **muLISP** system. These programs demonstrate numerous programming techniques such as how to manipulate data structures and how to provide a friendly user interface. Each program can be loaded by issuing a LOAD command following the **muLISP** prompt. For example, to load the Animal Game from the disk in drive B, issue the command

```
(LOAD 'B:ANIMAL)
```

- ANIMAL.LSP Think of an animal. The program will try to guess the animal by asking simple yes-or-no questions. The Animal Game is a good example of the use of a database set up as a binary **discrimination net**. If the program fails to guess the animal, it asks you how to distinguish the animal it guessed from the animal you were thinking of. This allows the program to extend its database to include your animal. The more you play, the "smarter" the program becomes!

  The Animal Game database can be saved as a text file for use in a later session. ANIMAL.MEM is a prebuilt database containing over 100 animals that you can load.

- DOCTOR.LSP This is the **muLISP** version of the Eliza (sometimes called Doctor) program written by Joseph Weizenbaum of MIT. It turns your computer into an amateurish, but persistent, psychologist. It mimics the Rogerian school of psychology where the doctor asks leading questions, letting patients resolve their own problems.

  The "P:" (patient) prompt indicates that the "doctor" is waiting for you to comment on your problem. Don't forget to terminate your sentence with a period, question mark, or exclamation point. The "D:" (doctor) herald precedes the doctor's responses to your statements. To exit the program, just type "Goodbye."

  Most of the program consists of a script that contains the keywords, patterns, and responses used by the doctor. The script can be elaborated upon or replaced with a script on an entirely different subject.

- EIGHTS.LSP The Eights Puzzle consists of eight square "tiles" numbered 1 through 8 in a square "tray" that can accommodate nine tiles. The program asks the user to place the tiles on the tray in a scrambled order. Using a heuristic search, the program finds and displays a sequence of moves that rearranges the tiles into a standard order. A move consists of sliding a tile vertically or horizontally into the adjacent empty space.

The standard order can only be reached from half the initial positions. Before beginning the search, the program uses a quick, direct test to determine if the standard order is reachable. The test is based on the number of permutations between the initial and final positions.

- HANOI.LSP This program graphically demonstrates the recursive solution to the Tower of Hanoi Puzzle. There are three pegs, one of which has a stack of rings of successively smaller diameter from bottom to top (i.e. the smallest on the top). The objective is to transfer all the rings to one of the other pegs so they will be in the same order. Only one ring may be moved at a time, and at no time may a larger ring lie on a smaller ring.

  The program provides options that allow you to adjust the number of rings, the speed of operation, the use of sound effects, and the use of color. In normal speed, the program simulates ring movement by showing intermediate positions of the rings and by using sound effects. When HANOI.LSP is loaded, WINDOWS.LSP must be on the default drive and directory.

- METAMIND.LSP Metamind is similar to the popular Master Mind™ board game by Invicta. One player, the "code maker", chooses a secret code consisting of a sequence of four (not necessarily different) colors. The other player, the "code breaker", tries to break the code in a minimal number of guesses. In response to the code breaker's guess as to the code, the code maker states the number of exact matches (i.e. of the correct color in the correct position), and the number of correct colors that are in the wrong position. The computer can be either the code maker or breaker.

- HELP.LSP This file provides a help facility for the functions, special forms, macros, and control variables defined in the machine language kernel or the source file library. After HELP.LSP is loaded, commands of the form

  (HELP *name*)

  or

  (? *name*)

  display *name*'s arguments, primitive type, and the page number containing the detailed description of name.

3.2 Tutorial System

The **muLISP** Tutorial System is an interactive, on-line system designed to teach the fundamentals of LISP programming. The source file LESSONS.LSP contains the **muLISP** program that presents the lessons.

Each of the lesson files, named MULISP*n*.LES, contain the actual text and examples for each lesson. The following is a summary of the information presented in each lesson:

- MULISP1.LES Data objects and primitive functions
- MULISP2.LES Defining functions using recursion
- MULISP3.LES Symbols, numbers, and conses
- MULISP4.LES List processing & iterative functions
- MULISP5.LES Numerical programming techniques
- MULISP6.LES Implementing turtle graphics functions

The first time through, the lessons should be taken in order.

To run the Tutorial System, switch to the drive and directory containing LESSONS.LSP and the lesson files. Then start **muLISP** as described in Section 2.1, and issue the command

```
(LOAD 'LESSONS)
```

following the **muLISP** prompt.

After the lesson program is loaded, it will display a menu listing the available lessons with a brief indication of their contents. To start with the first lesson, simply press the **Space** bar. The screen will clear and the lesson will begin. From this point on the lessons should be self explanatory.

Chapter 4. Interpreter

This chapter describes the primitive data objects, the memory management system, the break options, how to interrupt a program, and the error messages.

4.1 Data Objects

In **muLISP**, the primitively defined **data objects** consist of symbols, numbers, and conses. You can combine data objects to make **data structures** of arbitrary complexity. The type of a data object can be determined using the recognizer functions (see Section 5.1.1).

Each type of data object consists of a fixed number of **data cells**. Symbols and numbers consist of four data cells; conses consist of two data cells. Each data object has at least a **car cell** and a **cdr cell** (pronounced "could-er"). These two cells can only point to (i.e. contain the memory address of) data objects. Only the other cells of a data object are allowed to point to non-data objects such as character strings, number vectors, machine code, and pseudo-code.

The car and cdr cells form an interconnected network of data objects that is closed in the sense that you cannot leave the network by following the pointers. This closure eliminates the need for time consuming run-time type checking by the many functions that follow or manipulate these two data cells.

4.1.1 Symbols

A symbol is a data object consisting of four cells:

Value	Property	Function	Pname

Symbols are uniquely stored in the sense that no two symbols can have identical print names. The four cells of a symbol are used as follows:

- **Value cell** This, the car cell, contains a pointer to the value of the symbol. The value cell can be changed by the value functions (see Section 5.3.1) and the modifier functions (see Section 5.2.3). The value cells of a function's formal arguments are also changed when the function is applied (see Section 5.7.2). When a symbol is created, its value cell is set to the new symbol itself. This automatic self-referencing of symbols is called **auto-quoting**.
- **Property list cell** This, the cdr cell, contains a pointer to the **property list** of the symbol. This list is used and modified by the property list functions (see Section 5.3.2). An atom (i.e. a symbol or a number) on the list is considered to be a flag; a cons is considered to be a property. A property consists of an indicator key consed to a property value. When a symbol is created, its property list cell is set to NIL (i.e. the empty list).
- **Function cell** This cell contains a pointer to the function definition of the symbol. Functions can be defined in either machine code or pseudo-code. When a function is applied, the interpreter uses the function cell to locate its definition. The definition primitives are used to change function definitions (see Section 5.3.3). When a symbol is created, its function cell is set to a routine that generates an Undefined Function error break when the symbol is applied as a function (see Section 5.10.3).
- **Pname cell** This cell contains a pointer to the symbol's **print name string**. The string consists of two bytes that store the length of the string followed by the ASCII characters used to print the symbol. Print name strings can be of arbitrary length. When a string is read by an input function or generated by a string function, a hashing algorithm is used to determine if a symbol already exists with that print name. If so, the existing symbol is used. If not, another symbol is created using the new print name. Once a symbol is created, its print name string cannot be modified.

4.1.2 Numbers

A number is a data object consisting of four cells:

Identity	Sign	Length	Vector

Numbers can be further subdivided into **integers** and **ratios**. Internally, integers are further subdivided into **small integers** (those less than 65536 in magnitude) and **big integers**. Small integers are stored uniquely. Ratios and big integers are *not* stored uniquely, so numbers with the same numerical value can coexist in the system. Except for the function EQ, the distinction between small and big integers does not affect the **muLISP** programmer. The four cells of a number are used as follows:

- **Identity cell** This, the car cell, contains a pointer to the number itself. Thus numbers do not have to be quoted, since they evaluate to themselves. The content of this cell cannot be changed.

- **Sign cell** This, the cdr cell, determines the sign of the number and whether the number is a small integer, a big integer, or a ratio. The cell points to one of six symbols in accordance with the following table:

	Positive	Negative
Small integer	NIL	T
Big integer	LAMBDA	NLAMBDA
Ratio	MACRO	SPECIAL

  The content of the sign cell of a number can be determined by taking the cdr of the number, but the sign cell cannot be changed.

- **Length cell** If the number is a small integer, this cell contains the magnitude of the integer. If the number is a big integer, this cell contains the word length of the number's vector. If the number is a ratio, this cell contains a pointer to the numerator of the ratio, which must be an integer.

- **Vector cell** If the number is a small integer, this cell contains a hash link (i.e. a pointer to another small integer) or a 1. If the number is a big integer, this cell contains a pointer to the least significant two byte word of the integer's binary number vector. If the number is a ratio, this cell contains a pointer to the denominator of the ratio, which must be a positive integer.

  Number vectors are contiguous blocks of memory used to store the unsigned magnitude of big integers. The words of number vectors are kept in reverse order (i.e. more significant words are stored in successively lower memory addresses).

4.1.3 Conses

A cons is a data object consisting of two cells:

Binary trees are the most basic type of data structure in **muLISP**. Internally they are implemented as a network of cell pairs called **conses**. Each cons consists of a car cell and a cdr cell. As mentioned earlier, the cells of a cons can only point to *bona fide* data objects: either symbols, numbers, or conses.

Conses are sometimes called **dotted-pairs**, because of the **dot notation** sometimes used to represent conses. If obj_1 and obj_2 are data objects, the expression

$$(obj_1 \ . \ obj_2)$$

denotes a cons whose car cell points to obj_1 and whose cdr cell points to obj_2. Although cumbersome, dot notation is capable of showing the structure of any binary tree.

In most applications it is more convenient to think in terms of linear **lists** rather than deeply nested binary trees. Therefore cons structures are usually expressed in **list notation**. The list

$$(obj_1 \ obj_2 \ \ldots \ obj_n)$$

represents a chain of conses linked through their cdr cells and whose car cells point to the elements of the list. To be a valid list, the cdr cell of the last cons of the list must point to the atom NIL. The object above expressed in dot notation is

$$(obj_1 \ . \ (obj_2 \ . \ (\ldots \ . \ (obj_n \ . \ \text{NIL})\ldots)))$$

The function READ accepts both dot and list notation (see Section 5.8.1). In list notation, the **null list** or **empty list** is denoted by NIL. Therefore, when "( )" is read by READ, it returns NIL.

The output functions WRITE, PRIN1, PRINC, and PRINT use list notation to the maximum extent possible when printing data objects (see Section 5.8.4). For example,

```
(DOG . (CAT . COW))
```

is displayed by these functions as

```
(DOG CAT . COW)
```

4.2 Memory Management

Dynamic, automatic memory management gives LISP much of its inherent power. This frees the programmer from having to allocate, in advance, sufficient memory for a problem. Such allocation before run-time is difficult, if not impossible, for most AI applications since the problems are generally not of predetermined size. During the execution of a program, all memory not actually required by the program should be available for the creation of new data objects. The following subsections describe how **muLISP** achieves this.

4.2.1 Data Spaces

When **muLISP** starts, it partitions available memory into several distinct data spaces. Unfortunately the address space of computers based on microprocessors in the Intel 8086 family is divided into 64K (kilobytes) segments. This relatively small segment size arbitrarily limits the maximum size of each **muLISP** data space. The following summarizes the contents of each data space and gives maximum size in kilobytes:

- **Atom Space** This space stores symbols and numbers. The symbols and numbers are based at opposite ends of the space. Each atom requires four data cells (8 bytes). The maximum size of the space is 128K.
- **Cons Space** This space stores conses. Each cons requires two data cells (4 bytes). The maximum size of the space is 128K.
- **String Space** This space stores the print names of symbols as character strings. Each string requires two bytes for its length plus one byte for each character in the string. The maximum size of the space is 64K.
- **Vector Space** This space stores the magnitude of big integers as binary number vectors. Each vector requires as many two byte words as is necessary to store the magnitude of the number. The maximum size of the space is 64K.
- **Code Space** This stores user-defined function definitions as pseudo-code. There can be up to six code spaces (see NEW-CODE-SPACE in Section 5.7.6). The maximum size of each space is about 55K.
- **Stack Space** This space stores the control stack and the variable stack. The stacks are based at opposite ends of the space. The maximum size of the space is about 32K.

muLISP can take advantage of up to 640K of computer memory. However, it may happen that one of the data spaces becomes exhausted even though other spaces still have some free space. If this situation should occur during the execution of a program, a Memory Full error break occurs (see Section 4.5).

4.2.2 Garbage Collection

The nature of stack operations makes management of the stack space automatic and continuous. However, management of the remaining spaces requires explicit recycling of data objects that are no longer needed. New data objects are constantly being *created* during the execution of a program, while others are implicitly *discarded* when they are no longer referenced. When the creation process uses up the available resources in any one of the data spaces, a **garbage collection** is performed on *all* data spaces. Storage space vacated by discarded data objects is reclaimed so the program can continue.

A two-pass, mark-sweep garbage collection algorithm is used. The *first* pass consists of marking all active data objects. These are the objects accessible by means of chaining through data cells beginning from the value and property list cells of all system symbols, or from the variable stack, or from pseudo-code. Unreferenced, auto-quoted symbols that have no properties and have no current function definition are not marked. Unmarked symbols are removed during the second pass.

During the *second* pass of the garbage collection, all the marked data compaction objects are collected and compacted into one end of their respective data spaces. This leaves the remainder of each data space available for newly created objects.

4.2.3 Data Space Reallocation

After a garbage collection, one or more of the data spaces may have an insufficient amount of free storage to permit a program to continue execution, despite the fact that the other data spaces may have abundant free space. If this situation occurs, a data space reallocation routine is called to move the partitions dividing the spaces, giving more memory to the exhausted space or spaces. (However, the limits explained in Section 4.2.1 on the size of each data space must be observed by the reallocator.) Thus **muLISP** automatically responds to changing demands placed on the various data spaces by programs.

Although garbage collection and data space reallocation are automatic, their occurrence is not entirely invisible to the user since they cause a short pause in the execution of a program. On PC compatible computers, the blinking cursor temporarily changes to a full block during collections and reallocations.

The amount of time required for a collection or reallocation depends on the amount of data currently in the system and the clock speed of the computer. Garbage collections and data space reallocations typically require a second or two each. Normally this is of no concern to the user, but it should be considered in the design of real-time applications.

A phenomenon known as **thrashing** occurs when the system is forced to spend an inordinate amount of time garbage collecting for a very small return of data space. The symptom is greatly increased execution time for a given task. It can sometimes be resolved by increasing the computer's memory size (up to 640K) and/or modifying the program to reduce storage requirements.

4.3 Break Option Menu

When a console interrupt occurs (see Section 4.4) or a program error is detected (see Section 4.5), execution of the program is suspended and a message is displayed indicating the nature of the break. This is followed on the next line by the break option menu:

Continue, Break, Abort, Top-level, Restart, System

You select one of the options by entering the first letter of its name (C, B, A, T, R, or S). Note that the options are listed in order of increasingly drastic action.

- Continue This option returns control to the program that invoked the break routine. If the break was caused by a console interrupt, execution continues as if the break had not occurred. If the break was caused by an error trap, the form passed to the break by the error handler is returned as the value of the errant function.
- Break This option temporarily suspends program execution and calls the current driver function (see Section 5.10.2). The default driver function implements the read-eval-print loop (see Section 2.2). This allows you to examine and/or change variable bindings, function definitions, etc. To exit the break and resume program execution, press the **Esc** key following the dollar-sign prompt.
- Abort This option aborts program execution, unwinds the variable stack (i.e. restores formal arguments to their original value), and returns control to the current driver function (see Section 5.10.2). Function definitions, property values, and global variables are left unchanged.
- Top-level This option aborts program execution, unwinds the variable stack, makes the console the standard input source and output sink, and returns control to the top-level loop (see Section 5.10.1). Function definitions, property values, and global variable values are left unchanged.
- Restart This option closes all open files, abandons the environment, and initiates a fresh **muLISP** system. All variable bindings, user-defined functions, and property values in the current environment are destroyed.
- System This option closes all open files, terminates execution of **muLISP**, and returns control to the host operating system.

4.4 Console Interrupts

While debugging, you can initiate a console interrupt to stop a "runaway" or nonterminating program, and return control to the console. Press the **Esc** key during the execution of a program to initiate the interrupt. When the console interrupt character is detected, the message

Console Interrupt Break

is displayed on the console followed by the break option menu (see Section 4.3). For console interrupt breaks, the Abort option is normally the appropriate option to choose.

The character used to initiate console interrupts can be changed by modifying the Default Read Table in the **muLISP** base page (see Appendix H).

Programs can disable console interrupts by reassigning the control variable *INTERRUPT-HOOK* to NIL (see Section 5.10.3). Note that while debugging programs that disable console interrupts, the only way to interrupt them is to restart the computer.

4.5 Error Messages

When a program is running and an error condition is detected by **muLISP**, an appropriate error message is displayed on the console, followed by the form that provoked the error and the break option menu (see Section 4.3). The following is an alphabetical list of all the error messages and their significance (**muLISP** compiler errors are described in Chapter 8):

- Dimension Error occurs when MAKE-ARRAY is called with other than a positive integer or a list of positive integers for the dimensions of the array, or when it is called with initial contents incompatible in length with the dimensions of the array (see Section 6.1).

- Disk Full occurs when there is insufficient disk space to store the data being written to a file. When this occurs, a throw is made to the label Disk Full with a value of NIL (see Section 5.8.4). If not caught by the program, the top-level loop catches the throw and generates an error break (see Section 5.10.1).

  Since the file remains open for output, programs can allow the user to delete other files from a full diskette, and then continue writing to the file.

- End-Of-File occurs when an attempt is made to read from beyond the end of a file or from "holes" within a file (see FILE-WRITE-POSITION in Section 5.9.2). The full pathname of the file being read is displayed after the error message.

- Exact Mode occurs when exact mode arithmetic is in effect (i.e. the control variable *EXACT-MODE* is neither NIL nor the symbol RATIONAL) and an irrational numerical function (see Section 6.2) is called upon to generate an approximate result.

- File not found occurs when the source or memory image file given on the DOS command line used to start **muLISP** is not found *or* is a memory image file created under a different version of **muLISP** (see Section 2.6). Memory image files can only be loaded under the same version of **muLISP** used to create them.

  Note that when LOAD fails to find a source or memory image file, it returns NIL instead of displaying a warning message (see Section 5.9.4).

- Include Error occurs when DEFSTRUCT is called upon to include a named structure in an unnamed structure, or to include an unnamed structure in a named structure (see Section 6.3).

- Incompatible Definition occurs when DEFSTRUCT is called upon to create a new definition for a structure function that is incompatible with the existing definition (see Section 6.3).

- Insufficient Arguments occurs when a function is called without enough arguments.

- Insufficient Memory, Aborting occurs when there is not enough memory to load a memory image file. Since there is no other recourse, **muLISP** is immediately terminated and control returns to DOS (see Section 2.6).

  Note that a memory image *can* be loaded on a computer with less memory than the computer on which the memory image was created. This error occurs only when the computer on which the memory image is being loaded does not have enough memory to store the memory image. The only way to load the memory image file is to obtain more memory for your computer.

- Invalid Argument occurs when an irrational numerical function is called with an argument for which the function does not have a real value (see Section 6.2).

- Invalid Place Form occurs when SETF, INCF, or DECF is called with an invalid place form (see Section 5.3.1).

- Keyword Not Handled occurs when MAKE-INSTANCE is called with keywords that were not handled by the inherited and included flavors of the flavor (see Section 6.4).

- Memory Full occurs when there is insufficient memory to continue execution of a program. When this occurs, a throw is made to the label Memory Full with a value of NIL. If not caught by the program, the top-level loop catches the throw and generates an error break (see Section 5.10.1).

Normally, the memory management system provides sufficient storage in each data space to continuously satisfy the demands of programs (see Section 4.2). This error occurs if the memory requirements for storing data objects exhaust all available resources. Statistics in the following form are displayed before the error break message:

GC: *counter* *atom* *cons* *string* *vector* *code* *stack*

The hexadecimal numbers show the available space and the total space in each of the data spaces (see Section 5.7.6). You can use the statistics to help determine the space responsible for the memory exhaustion. However, memory exhaustion can occur even if the statistics show some memory available in all spaces (see Section 4.2.1).

- Missing Required Flavor occurs when MAKE-INSTANCE is called upon to create an instance of a flavor that did not include all the required flavors (see Section 6.4).

- Noninteger Argument occurs when LOGAND, LOGIOR, LOGXOR, LOGNOT, SHIFT, or INTEGER-LENGTH is called with a noninteger argument (see Section 5.4.5).

- Nonnumeric Argument occurs when a numeric function is called with a nonnumeric argument (see Section 5.4).

- Nonsymbolic Argument occurs when SET, SETQ, PSETQ, POP, PUSH, INCQ, DECQ, or DEFSTRUCT is called with a nonsymbolic argument.

- Syntax Error occurs when READ reads either an unmatched right parenthesis (i.e. too many right parentheses) or an error in dot notation, for example

```
(A . )
```

or

```
(A B . C D)
```

Since this error break is generated by the right parenthesis and comma character macros, redefining these character macros enables the error to be handled by a program (see Section 5.8.1).

- Unclaimed Message occurs when SEND is called upon to send a message to an instance of a flavor that does not have a method to handle the message (see Section 6.4).

- Undefined Function occurs when an attempt is made to apply a symbol having no function definition. Following the error message is the form that made the call on the undefined function. A common course of action when confronted with this error is to

select the Break option, appropriately define the undefined symbol, and then continue with the original program by issuing the command

```
(RETURN (EVAL BREAK))
```

- Undefined Structure occurs when DEFSTRUCT is called upon to include an undefined structure in a structure (see Section 6.3).
- Unnamed Structure occurs when DEFSTRUCT is called upon to define a predicate for an unnamed structure (see Section 6.3).
- Unsupported Option occurs when DEFSTRUCT is called with an option not supported by the structure facility (see Section 6.3).
- Zero Divide occurs when /, FLOOR, CEILING, TRUNCATE, ROUND, MOD, REM, or DIVIDE is called with a divisor of zero (see Section 5.4).

Chapter 5. Primitives

This chapter describes the functions, special forms, macros, and control variables defined in the machine language kernel and in the COMMON.LSP source file. All user-defined functions and macros must ultimately be defined in terms of these primitives.

The title line of each primitive indicates its name, its arguments or initial value, and its type. The primitive name is printed in Helvetica type at the left margin.

If the primitive is a function, special form, or macro, its name is followed by a list of names, enclosed in square brackets and separated by commas, that represent each of its arguments. The type of the primitive is printed at the right margin.

If the primitive is a control variable, its name is followed by a colon and its initial value. The phrase "Control variable" is printed at the right margin.

If the primitive is defined in COMMON.LSP, the primitive type printed at the right margin is preceded by the word "Library". These library primitives are only available after issuing a command of the form

(LOAD '*path*\\COMMON.LSP)

where *path* specifies the drive and directory containing the file COMMON.LSP.

Following the title line is a description of the primitive. If the primitive is a function, special form, or macro, its arguments, value, and side-effects, if any, are described. If the primitive is a control variable, its effect on the system is described. There may also be a discussion of subtle points and common uses of the primitive.

Following the plain language description, there is usually a formal definition of the primitive, written in **muLISP**. Functions are defined using DEFUN. Special forms and macros are defined using DEFMACRO.

Occasionally auxiliary functions are introduced to aid in formal definitions. Lower case letters are used for the names of auxiliary functions to indicate that they are not directly accessible to the user. When irreducible, primitive concepts are introduced in formal definitions, they are described in English using lower case letters.

Finally, if appropriate, an interaction is shown that illustrates the use of the primitive. If the expression following the dollar-sign ($) is entered by the user, **muLISP** will evaluate the expression and return the result exactly as shown on the next line of the example. Comments concerning an example are shown to the right of the example following a semicolon, which is the comment delimiter.

5.1 Predicates

A **predicate** is a function used to test if its arguments satisfy some condition. If the test is false, the predicate returns NIL. If the test is true, it returns a nonNIL value, usually T. Predicates are used in conjunction with conditional control constructs (see Section 5.6) to guide the interpreter through function definitions.

5.1.1 Recognizers

A **recognizer** is a predicate used to determine the type of a data object. If its argument is of the specified type, the recognizer returns T, otherwise it returns NIL.

SYMBOLP [*object*] — Function
STRINGP [*object*] — Library function

If *object* is a symbol, (SYMBOLP *object*) returns T, otherwise it returns NIL.

In **muLISP**, no distinction is made between strings and symbols. Thus the definition of STRINGP is *identical* to that of SYMBOLP and is included in COMMON.LSP solely for compatibility.

```
(DEFUN SYMBOLP (OBJ)
  ((if OBJ is a symbol)  T)
  NIL )

$ (SYMBOLP 'DOG)                                        ;DOG is a symbol
T

$ (SYMBOLP ())                                  ;The empty list (= NIL) is a symbol
T

$ (SYMBOLP 100)                                       ;Integers are not symbols
NIL

$ (SYMBOLP 2/3)                                        ;Numbers are not symbols
NIL

$ (SYMBOLP 4.235)                                      ;Numbers are not symbols
NIL

$ (SYMBOLP '(A B C))                                ;Nonempty lists are not symbols
NIL
```

INTEGERP [*object*] Function
NUMBERP [*object*] Function
RATIONALP [*object*] Library function

If *object* is an integer, (INTEGERP *object*) returns T, otherwise it returns NIL.

If *object* is a number, either an integer or a rational, (NUMBERP *object*) returns T, otherwise it returns NIL.

In **muLISP**, rational numbers also serve as floating point numbers. Thus the definition of RATIONALP is *identical* to that of NUMBERP and is included in COMMON.LSP solely for compatibility.

```
(DEFUN INTEGERP (OBJ)
  ((if OBJ is an integer) T)
  NIL )

(DEFUN NUMBERP (OBJ)
  ((INTEGERP OBJ))
  ((if OBJ is a rational number) T)
  NIL )

$ (INTEGERP 100)                                    ;100 is an integer
T

$ (INTEGERP 2/3)                                    ;Ratios are not integers
NIL

$ (INTEGERP 4.235)                                  ;Decimal numbers are not integers
NIL

$ (NUMBERP 100)                                     ;Integers are numbers
T

$ (NUMBERP 2/3)                                     ;Ratios are numbers
T

$ (NUMBERP 4.235)                                   ;Decimal numbers are numbers
T
```

ATOM [*object*] — Function

If *object* is an atom, either a symbol or a number, (ATOM *object*) returns T, otherwise it returns NIL.

```
(DEFUN ATOM (OBJ)
  (OR (SYMBOLP OBJ) (NUMBERP OBJ)) )

$ (ATOM 'DOG)                                          ;Symbols are atoms
T

$ (ATOM 100)                                           ;Integers are atoms
T

$ (ATOM 2/3)                                           ;Numbers are atoms
T

$ (ATOM 4.235)                                         ;Numbers are atoms
T

$ (ATOM '(A B C))                                      ;Nonempty lists are not atoms
NIL

$ (ATOM '(A . B))                                      ;Conses are not atoms
NIL

$ (ATOM ())                                            ;The empty list (= NIL) is an atom
T
```

CONSP [*object*] Function
LISTP [*object*] Function

If *object* is a cons, (CONSP *object*) returns T, otherwise it returns NIL.

If *object* is a list, either a cons or the atom NIL, (LISTP *object*) returns T, otherwise it returns NIL.

```
(DEFUN CONSP (OBJ)
  (NOT (ATOM OBJ)) )

(DEFUN LISTP (OBJ)
  (OR (CONSP OBJ) (NULL OBJ)) )

$ (CONSP 'DOG)                                    ;Symbols are not conses
NIL

$ (CONSP 4.235)                                   ;Numbers are not conses
NIL

$ (CONSP '(A B C))                                      ;Lists are conses
T

$ (CONSP ())                              ;The empty list (= NIL) is not a cons
NIL

$ (LISTP 'DOG)                                     ;Symbols are not lists
NIL

$ (LISTP 4.235)                                    ;Numbers are not lists
NIL

$ (LISTP '(A B C))                                         ;(A B C) is a list
T

$ (LISTP ())                                ;The empty list (= NIL) is a list
T
```

NULL [*object*] Function
ENDP [*object*] Function

If *object* is the empty list, (NULL *object*) returns T, otherwise it returns NIL. The empty or null list is represented by the symbol NIL.

If *object* is the empty list, (ENDP *object*) returns T. If *object* is a cons, ENDP returns NIL. Otherwise (i.e. if *object* is an atom), ENDP generates a Non-list Object error break.

Since the atom NIL represents both the empty list and the truth value false, the functions NULL and NOT always return identical values. However for program readability, it is better to use NULL when testing for the empty list and to use NOT when testing for the truth value false. When debugging it is even better to use ENDP instead of NULL when testing for the empty list, since an error break occurs if ENDP is given a non-list.

```
(DEFUN NULL (OBJ)
  (EQ OBJ NIL) )

(DEFUN ENDP (OBJ)
  ((NULL OBJ))
  ((ATOM OBJ)
    (BREAK (LIST 'ENDP OBJ) "Non-list Object") ) )

$ (NULL NIL)                                          ;NIL is the empty list
T

$ (NULL ())                                            ;() is the empty list
T

$ (NULL 'DOG)                                ;A nonNIL symbol is not the empty list
NIL

$ (NULL '(A B C))                            ;A nonempty list is not the empty list
NIL

$ (NULL '(NIL))                              ;A nonempty list is not the empty list
NIL

$ (ENDP NIL)                                             ;NIL is the end of a list
T

$ (ENDP '(A B C))                                  ;A nonempty list is not the end
NIL

$ (ENDP 'DOG)                                       ;A nonNIL symbol is not a list
Non-list Object Break: (ENDP DOG)
```

5.1.2 Comparators

A **comparator** is a predicate used to compare two data objects. If its arguments satisfy the comparison, the comparator returns T, otherwise it returns NIL.

EQ [*object*$_1$, *object*$_2$] Function
NEQ [*object*$_1$, object$_2$] Function

If *object*$_1$ is identical to *object*$_2$, (EQ *object*$_1$ *object*$_2$) returns T, otherwise it returns NIL.

(NEQ *object*$_1$ *object*$_2$) is equivalent to (NOT (EQ *object*$_1$ *object*$_2$)).

EQ tests for the *identity* of its two arguments (i.e. they point to the same data object in memory). Since symbols and small integers are stored uniquely (see Section 4.1), EQ is an efficient way to determine if two symbols or two small integers are equal.

```
(DEFUN EQ (OBJ1 OBJ2)
  (= (LOCATION OBJ1) (LOCATION OBJ2)) )

$ (EQ 'Alice 'Alice)                          ;Identical symbols are EQ
T

$ (EQ 7 (+ 3 4))                             ;Equal small integers are EQ
T

$ (EQ 100000 100000)                       ;Equal big integers may not be EQ
NIL

$ (EQ 4.75 4.75)                         ;Equal decimal numbers may not be EQ
NIL

$ (EQ '(A B C) '(A B C))                      ;Equal lists may not be EQ
NIL
```

EQL [*object*$_1$, *object*$_2$] Function
NEQL [*object*$_1$, *object*$_2$] Function

If *object*$_1$ is identical to *object*$_2$ or if both are numbers having the same numeric value, (EQL *object*$_1$ *object*$_2$) returns T, otherwise it returns NIL.

(NEQL *object*$_1$ *object*$_2$) is equivalent to (NOT (EQL *object*$_1$ *object*$_2$)).

Since big integers and ratios are *not* stored uniquely (see Section 4.1), EQL rather than EQ should be used to determine if two numbers are equal. Nonatomic arguments are EQL if and only if they point to the same cons in memory.

```
(DEFUN EQL (OBJ1 OBJ2)
  ((AND (NUMBERP OBJ1) (NUMBERP OBJ2))
    (= OBJ1 OBJ2) )
  (= (LOCATION OBJ1) (LOCATION OBJ2)) )

$ (EQL 'Alice 'Alice)                          ;Identical symbols are EQL
T

$ (EQL 7 (* 3 4))                                  ;Equal numbers are EQL
T

$ (EQL 100000 100000)                              ;Equal numbers are EQL
T

$ (EQL 4.75 4.75)                                  ;Equal numbers are EQL
T

$ (EQL '(A B C) '(A B C))                     ;Equal lists may not be EQL
NIL
```

EQUAL [*object*$_1$, *object*$_2$, *test*] Function
TREE-EQUAL [*object*$_1$, *object*$_2$, *test*] Library function

If *object*$_1$ and *object*$_2$ are isomorphic tree structures with leaves such that (*test leaf*$_1$ *leaf*$_2$) is nonNIL, (EQUAL *object*$_1$ *object*$_2$ *test*) returns T, otherwise it returns NIL. If the *test* argument is NIL or not given, EQUAL uses EQL for *test*. The definition of TREE-EQUAL is *identical* to that of EQUAL and is included in COMMON.LSP solely for compatibility.

Thus, EQUAL tests for the *equality* of its two arguments. Two nonatomic arguments are EQUAL if the car branches are EQUAL and the cdr branches are EQUAL. Two EQUAL data structures look the same when printed. Since EQL is faster, it should be used in place of EQUAL in cases where at least one of the arguments is known to be atomic.

```
(DEFUN EQUAL (OBJ1 OBJ2 TEST)
  ((ATOM OBJ1)
    (FUNCALL (OR TEST 'EQL) OBJ1 OBJ2) )
  ((ATOM OBJ2)  NIL)
  ((EQUAL (CAR OBJ1) (CAR OBJ2) TEST)
    (EQUAL (CDR OBJ1) (CDR OBJ2) TEST) ) )
```

```
$ (EQUAL 'Alice 'Alice)                          ;Identical symbols are EQUAL
T

$ (EQUAL 7 (+ 3 4))                                  ;Equal numbers are EQUAL
T

$ (EQUAL 4.75 4.75)                                  ;Equal numbers are EQUAL
T

$ (EQUAL '(A B C) '(A B C))                            ;Equal lists are EQUAL
T

$ (EQUAL '(A B C) '(C B A))                         ;Nonequal lists are not EQUAL
NIL
```

TAILP [*list*$_1$, *list*$_2$] Function

If *list*$_1$ is a sublist (i.e. one of the top-level conses) of *list*$_2$, (TAILP *list*$_1$ *list*$_2$) returns T, otherwise it returns NIL.

```
(DEFUN TAILP (LST1 LST2)
  ((EQ LST1 LST2))
  ((ATOM LST2)  NIL)
  (TAILP LST1 (CDR LST2)) )

$ (SETQ FOO '(A B C D))                                  ;Initialize FOO
(A B C D)

$ (TAILP '(C D) FOO)                                    ;Not a tail of FOO
NIL

$ (SETQ BAZ (CDDR FOO))                                  ;Initialize BAZ
(C D)

$ (TAILP BAZ FOO)                                      ;BAZ is a tail of FOO
T
```

ORDERP [$object_1$, $object_2$] Function

If $object_1$ is ordered before $object_2$, (ORDERP $object_1$ $object_2$) returns T, otherwise it returns NIL.

ORDERP provides an efficient **chronological ordering** for atoms that is useful for writing sorting and searching routines. Numbers are ordered before symbols, which are ordered before conses. Two numbers are ordered in increasing numerical order. Two symbols are ordered by comparing their physical memory address (i.e. according to their order of introduction). Two conses are ordered by comparing their physical memory address.

```
(DEFUN ORDERP (OBJ1 OBJ2)
  ((NUMBERP OBJ1)
    ((NUMBERP OBJ2)
      (< OBJ1 OBJ2) )
    T )
  ((NUMBERP OBJ2)  NIL)
  (< (LOCATION OBJ1) (LOCATION OBJ2)) )

$ (ORDERP 5 9)                                  ;Numbers are ordered numerically
T

$ (ORDERP 'DOG 5)                               ;Symbols are ordered after numbers
NIL

$ (ORDERP 'DOG 'CAT)                            ;DOG was introduced before CAT
T

$ (ORDERP 'CAT 'DOG)                            ;CAT was not introduced before DOG
NIL
```

5.1.3 Logical Operators

A **logical operator** is a predicate used to perform Boolean operations on the truth values of its arguments. NIL has the truth value false. All other data objects have the truth value true.

NOT [*object*] Function

If *object* is the atom NIL, (NOT *object*) returns T, otherwise it returns NIL.

Since the atom NIL represents both the empty list and the truth value false, the functions NULL and NOT always return identical values. However for program readability, it is better to use NULL when testing for the empty list and to use NOT when testing for the truth value false.

```
(DEFUN NOT (OBJ)
  (EQ OBJ NIL) )

$ (NOT NIL)                                        ;NIL has truth value false
T

$ (NOT '())                                         ;() has truth value false
T

$ (NOT 'FOO)                                ;NonNIL symbols have truth value true
NIL

$ (NOT '(A . B))                                   ;Conses have truth value true
NIL

$ (NOT (EQ 'DOG 'CAT))                                  ;DOG is not EQ to CAT
T
```

AND [$form_1$, $form_2$, ..., $form_n$] Special form

(AND $form_1$ $form_2$ ... $form_n$) evaluates each form in turn until one of them evaluates to NIL or all the forms are evaluated. If a form evaluates to NIL, AND returns NIL, otherwise it returns the value of $form_n$. The call (AND) returns T, an identity for the AND operator. (AND $form_1$ $form_2$ ... $form_n$) is equivalent to the call

(IF $form_1$ (IF $form_2$ (IF ... (IF $form_{n-1}$ $form_n$))))

Note that AND evaluates a form if and only if all the preceding forms have evaluated to nonNIL values.

```
(DEFMACRO AND LST
  ((NULL LST))
  ((NULL (CDR LST))
    (CAR LST) )
  (LIST 'IF (CAR LST) (MACROEXPAND (CONS 'AND (CDR LST)))) )

$ (AND (EQ 'DOG 'CAT) (< 2 3))                          ;One comparison is false
NIL

$ (AND (EQ 'DOG 'DOG) (< 2 3))                         ;Both comparisons are true
T

$ (AND (ATOM 'DOG) (MEMBER 'DOG '(CAT DOG COW)))                ;Returns last value
(DOG COW)

$ (AND (SYMBOLP 5) (PRINT 'HELLO))                          ;Since test is false,
NIL                                                   ;(PRINT HELLO) is not evaluated
```

OR [*form*$_1$, *form*$_2$, ..., *form*$_n$] Special form

(OR *form*$_1$ *form*$_2$ ... *form*$_n$) evaluates each form in turn until one of them evaluates to a nonNIL value or all the forms are evaluated. If a form is nonNIL, OR returns that nonNIL value, otherwise it returns NIL. The call (OR) returns NIL, the identity for the OR operator. (OR *form*$_1$ *form*$_2$ ... *form*$_n$) is equivalent to the call

(COND (*form*$_1$) (*form*$_2$) ... (*form*$_n$))

Note that OR evaluates a form if and only if all the preceding forms have evaluated to NIL.

```
(DEFMACRO OR LST
  ((NULL LST) NIL)
  (CONS 'COND
      (MAPCAR '(LAMBDA (FORM) (LIST FORM)) LST)) )

$ (OR (EQ 'DOG 'CAT) (< 2 3))                          ;One comparison is true
T

$ (OR (EQ 'DOG 'CAT) (< 3 2))                        ;Neither comparison is true
NIL

$ (SETQ FOO 7)                                                  ;Initialize FOO
7

$ (OR (ZEROP FOO) (/ 35 FOO))                          ;OR returns last value
5

$ (SETQ FOO 0)                                                  ;Initialize FOO
0

$ (OR (ZEROP FOO) (/ 35 FOO))                        ;No "Zero Divide" error break
T
```

5.2 Lists

The functions in this section perform operations on lists including list construction and element selection, substitution, deletion, searching, and sorting.

5.2.1 Selectors

The functions in this subsection are for accessing the elements of lists and the components of trees.

CAR [*object*] Function
CDR [*object*] Function

(CAR *object*) returns the car of *object*, that is, the data object pointed to by the **car cell** of *object*. (CDR *object*) returns the cdr of *object*, that is, the data object pointed to by the **cdr cell** of *object*.

The proper interpretation for the car and the cdr of a data object depends on the data type of the object (see Section 4.1). If the object is a symbol, the car cell points to its current value and the cdr cell points to its property list. If the object is a number, the car cell points to the number itself and the cdr cell points to a symbol indicating its sign and number type. If the object is a list, the car cell points to the first element of the list and the cdr cell points to its tail (i.e. everything but the first element of the list). If the object is a binary tree, the car cell points to its left or **car branch** and the cdr cell points to its right or **cdr branch**.

```
(DEFUN CAR (OBJ)
  (the object pointed to by the car cell of OBJ) )

(DEFUN CDR (OBJ)
  (the object pointed to by the cdr cell of OBJ) )

$ (CAR '(A B C D E))                                    ;Car of a list
A

$ (CDR '(A B C D E))                                    ;Cdr of a list
(B C D E)

$ (CAR '((A . B) . C))                            ;Car of a binary tree
(A . B)

$ (CDR '((A . B) . C))                            ;Cdr of a binary tree
C
```

CAAR [*object*]	Function
CDAR [*object*]	Function
CADR [*object*]	Function
CDDR [*object*]	Function
CAAAR [*object*]	Function
CDAAR [*object*]	Function
CADAR [*object*]	Function
CDDAR [*object*]	Function
CAADR [*object*]	Function
CDADR [*object*]	Function
CADDR [*object*]	Function
CDDDR [*object*]	Function
CAAAAR [*object*]	Function
CDAAAR [*object*]	Function
CADAAR [*object*]	Function
CDDAAR [*object*]	Function
CAADAR [*object*]	Function
CDADAR [*object*]	Function
CADDAR [*object*]	Function
CDDDAR [*object*]	Function
CAAADR [*object*]	Function
CDAADR [*object*]	Function
CADADR [*object*]	Function
CDDADR [*object*]	Function
CAADDR [*object*]	Function
CDADDR [*object*]	Function
CADDDR [*object*]	Function
CDDDDR [*object*]	Function

Although successive applications of CAR and CDR are sufficient to access any part of a binary tree, all combinations of CAR and CDR up to four compositions are defined in machine language for efficiency and convenience. The name of these functions begins with a C and ends with an R. In between is a sequence of As and Ds indicating the composition of cars and cdrs that the function performs.

```
(DEFUN CDAR (OBJ)
  (CDR (CAR OBJ)) )
```

```
(DEFUN CADDR (OBJ)
  (CAR (CDR (CDR OBJ))) )

$ (SETQ TREE '(                                                    ;Initialize TREE
    (((A . B) . (C . D)) . ((E . F) . (G . H))) .
    (((I . J) . (K . L)) . ((M . N) . (O . P)))) )
((((A . B) C . D) (E . F) G . H) ((I . J) K . L) (M . N) O . P)

$ (CAR TREE)                                                           ;Car of TREE
(((A . B) C . D) (E . F) G . H)

$ (CDR TREE)                                                           ;Cdr of TREE
(((I . J) K . L) (M . N) O . P)

$ (CAAR TREE)                                                   ;Car of car of TREE
((A . B) C . D)

$ (CDAR TREE)                                                   ;Cdr of car of TREE
((E . F) G . H)

$ (CADR TREE)                                                   ;Car of cdr of TREE
((I . J) K . L)

$ (CDDR TREE)                                                   ;Cdr of cdr of TREE
((M . N) O . P)

$ (CAAAR TREE)                                          ;Car of car of car of TREE
(A . B)

$ (CADDDR TREE)                                   ;Car of cdr of cdr of cdr of TREE
O

$ (CDADAR TREE)                                   ;Cdr of car of cdr of car of TREE
F
```

FIRST [*list*] Function
SECOND [*list*] Function
THIRD [*list*] Function
FOURTH [*list*] Function
FIFTH [*list*] Function
SIXTH [*list*] Function
SEVENTH [*list*] Function
EIGHTH [*list*] Function
NINTH [*list*] Function
TENTH [*list*] Function
REST [*list*] Function

These selectors return any one of the first ten elements of a list. REST is included to complement FIRST.

These are the only primitives that refer to the car of a list as the *first* element. All the other primitives (e.g. NTH, SUBLIST, REPLACE-NTH, etc.) refer to the car of a list as the *zeroth* element.

```
(DEFUN SIXTH (LST)
  (NTH 5 LST) )

(DEFUN REST (LST)
  (CDR LST) )

$ (FIRST '(A B C D E))                          ;The first element
A

$ (SECOND '(A B C D E))                       ;The second element of a list
B

$ (THIRD '(A B C D E))                         ;The third element of a list
C

$ (FOURTH '(A B C D E))                       ;The fourth element of a list
D

$ (FIFTH '(A B C D E))                         ;The fifth element of a list
E

$ (REST '(A B C D E))                           ;The rest of a list
(B C D E)
```

LAST [*list*] Function

(LAST *list*) returns the last top-level cons of *list*. Note that LAST returns the last cons, *not* the last element, of *list*. If *list* is an atom, LAST returns NIL. As the final example below shows, the last element can be extracted by taking the CAR of (LAST *list*).

```
(DEFUN LAST (LST)
  ((ATOM LST)  NIL)
  ((ATOM (CDR LST))  LST)
  (LAST (CDR LST)) )

$ (LAST '(A B C D))                                  ;The last cons of a list
(D)

$ (LAST '(A B C . D))                              ;The last of cons a binary tree
(C . D)

$ (LAST 'FOO)                                          ;The last of a symbol
NIL

$ (LAST 5)                                             ;The last of a number
NIL

$ (CAR (LAST '(A B C)))                            ;The last element of a list
C
```

NTHCDR [*n*, *list*] Function
NTH [*n*, *list*] Function

If *n* is zero or a positive integer, (NTHCDR *n list*) returns the *n*th cdr of *list*. (NTH *n list*) returns the *n*th element of *list*, where the car of *list* is the zeroth element. NTHCDR and NTH return NIL if *n* is neither zero nor a positive integer, or if *list* has *n* or fewer elements.

```
(DEFUN NTHCDR (N LST)
  ((ZEROP N)  LST)
  ((AND (INTEGERP N) (PLUSP N))
    ((ATOM LST)  NIL)
    (NTHCDR (SUB1 N) (CDR LST)) ) )

(DEFUN NTH (N LST)
  ((ATOM (NTHCDR N LST))  NIL)
  (CAR (NTHCDR N LST)) )

$ (NTHCDR 0 '(A B C D))                          ;No cdrs of a list
(A B C D)

$ (NTH 0 '(A B C D))                             ;Zeroth element of a list
A

$ (NTHCDR 2 '(A B C D))                          ;Two cdrs of a list
(C D)

$ (NTH 2 '(A B C D))                             ;Second element of a list
C

$ (NTHCDR 4 '(A B C D))                          ;Four cdrs of a list
NIL

$ (NTH 4 '(A B C D))                             ;No fourth element of a list
NIL

$ (NTHCDR 2 '(A B . C))                          ;Two cdrs of a tree
C
```

LENGTH [*object*] Function
LIST-LENGTH [*object*] Library function

(LENGTH *object*) returns the length of *object* appropriate for its data type. If *object* is NIL or a list, LENGTH returns the number of top-level conses (i.e. elements) in *object*. If *object* is a circular list, LENGTH returns NIL. If *object* is a symbol, LENGTH returns the number of characters in *object*'s print name string. If *object* is a number, LENGTH returns the number of words (a word equals two bytes) required to store *object*'s number vector.

The definition of LIST-LENGTH is *identical* to that of LENGTH and is included in COMMON.LSP solely for compatibility.

```
(DEFUN LENGTH (OBJ)
  ((NULL OBJ)  0)
  ((SYMBOLP OBJ)
    (the number of characters in OBJ's print name string) )
  ((NUMBERP OBJ)
    (CEILING (ABS OBJ) 65536) )
  (ADD1 (LENGTH (CDR OBJ))) )

$ (LENGTH '(A B C D))                                  ;Length of a list
4

$ (LENGTH NIL)                                         ;Length of the empty list
0

$ (LENGTH 'MULISP)                                     ;Number of characters in a print name
6

$ (LENGTH 100000)                                      ;Number of words to store a number
2

$ (LENGTH -13)                                         ;Number of words to store a number
1
```

5.2.2 Constructors

The functions in this subsection are for making new data structures (binary trees or linked lists) from simpler structures using conses.

CONS [*object*$_1$, *object*$_2$] Function

(CONS *object*$_1$ *object*$_2$) returns a cons whose car cell points to *object*$_1$ and whose cdr cell points to *object*$_2$. If the value of *FREE-LIST* is a cons, CONS modifies and returns that cons and sets *FREE-LIST* to the cdr of *FREE-LIST* (see Section 5.7.6).

The proper interpretation of (CONS *object*$_1$ *object*$_2$) depends on the data type of *object*$_2$. If *object*$_2$ is a list, CONS returns a list whose first element is *object*$_1$ and whose tail is *object*$_2$. If *object*$_2$ is an atom or a binary tree, CONS returns a tree whose left or car branch is *object*$_1$, and whose right or cdr branch is *object*$_2$. Note that CONS creates a new cons; it does *not* alter either *object*$_1$ or *object*$_2$.

```
(DEFUN CONS (OBJ1 OBJ2)
  ((ATOM *FREE-LIST*)
    (a new cons whose car cell points to OBJ1 and whose cdr cell points to OBJ2) )
  (RPLACA *FREE-LIST* OBJ1)
  (RPLACD (PROG1 *FREE-LIST* (POP *FREE-LIST*)) OBJ2) )

$ (CONS 'A '(B C D))                                  ;Cons a symbol onto a list
(A B C D)

$ (CONS '(A B) '(C D))                                ;Cons a list onto a list
((A B) C D)

$ (CONS 'A 'B)                                        ;Cons a symbol onto a symbol
(A . B)

$ (CONS '(A B C) 'D)                                  ;Cons a list onto a symbol
((A B C) . D)

$ (CONS 'A NIL)                                       ;Cons a symbol onto NIL
(A)
```

ADJOIN [*object*, *list*, *test*] Function

If *object* is a member of *list* (i.e. there is an *element* of *list* for which (*test object element*) is nonNIL), (ADJOIN *object list test*) returns *list*. Otherwise, ADJOIN returns (CONS *object list*). If the *test* argument is NIL or not given, ADJOIN uses an EQL test.

```
(DEFUN ADJOIN (OBJ LST TEST)
  ((MEMBER OBJ LST TEST)  LST)
  (CONS OBJ LST) )

$ (ADJOIN 'DOG '(CAT COW DOG PIG))                    ;DOG already an element of list
(CAT COW DOG PIG)

$ (ADJOIN 'DOG '(CAT COW RAT PIG))                       ;DOG not an element of list
(DOG CAT COW RAT PIG)

$ (ADJOIN '(A B C) '((R S T) (A B C) (X Y Z)))         ;Not EQL to an element of list
((A B C) (R S T) (A B C) (X Y Z))

$ (ADJOIN '(A B C) '((R S T) (A B C) (X Y Z)) 'EQUAL)     ;EQUAL to an element of list
((R S T) (A B C) (X Y Z))
```

LIST [$object_1$, $object_2$, ..., $object_n$] Function
LIST* [$object_1$, $object_2$, ..., $object_n$] Function

(LIST $object_1$ $object_2$ ... $object_n$) constructs and returns the list consisting of the elements $object_1$ through $object_n$. If called without arguments, LIST returns NIL. LIST can take any number of arguments.

(LIST* $object_1$ $object_2$ ... $object_n$) conses $object_1$ through $object_{n-1}$ onto $object_n$ and returns the resulting list. LIST* is useful for consing several items onto the front of a list. If called with a single argument, LIST* returns that argument.

```
(DEFUN LIST LST
  ((NULL LST)  NIL)
  (CONS (CAR LST) (APPLY 'LIST (CDR LST))) )

(DEFUN LIST* LST
  ((NULL LST)  NIL)
  ((NULL (CDR LST))
    (CAR LST) )
  (CONS (CAR LST) (APPLY 'LIST* (CDR LST))) )

$ (LIST 'A 'B 'C 'D)                                  ;Returns a 4 element list
(A B C D)

$ (LIST* 'A 'B 'C 'D)                                   ;Returns a binary tree
(A B C . D)

$ (LIST 'A 'B '(C D))                                 ;Returns a 3 element list
(A B (C D))

$ (LIST* 'A 'B '(C D))                                ;Returns a 4 element list
(A B C D)

$ (LIST 'A)                                           ;Returns a 1 element list
(A)

$ (LIST* 'A)                                                   ;Returns a leaf
A

$ (LIST)                                                ;Returns the empty list
NIL
```

APPEND [$list_1$, $list_2$, ..., $list_n$] Function

(APPEND $list_1$ $list_2$ ... $list_n$) constructs and returns a list consisting of the elements of the lists $list_1$, $list_2$, through $list_n$. APPEND copies the top-level conses of each of its arguments except the last. If called with a single argument, APPEND simply returns the argument without copying it. Therefore COPY-LIST rather than APPEND must be used to copy a list.

APPEND uses consing to construct and return a new list rather than modifying its arguments. See NCONC in the next subsection for a destructive version of APPEND.

```
(DEFUN APPEND (LST1 LST2)
   ((ATOM LST1)  LST2)
   (CONS (CAR LST1) (APPEND (CDR LST1) LST2)) )

$ (SETQ FOO '(D E F))                                    ;Set FOO to list
(D E F)

$ (APPEND '(A B C) FOO '(G H I))                          ;Append lists
(A B C D E F G H I)

$ FOO                                                     ;FOO unchanged
(D E F)

$ (APPEND '(A B C D) 'E 'F 'G)                  ;Atoms cannot be appended
(A B C D . G)
```

MAKE-LIST [*n*, *object*, *list*] Function
COPY-LIST [*list*] Function
COPY-TREE [*object*] Function

(MAKE-LIST *n object list*) constructs and returns a list of *n* elements, each initialized to *object* concatenated onto *list* as if using NCONC. *n* defaults to 0, *object* and *list* default to NIL.

(COPY-LIST *list*) copies the *top-level* conses of *list*, returning a list that is EQUAL to *list*. Sublists are *not* copied.

(COPY-TREE *object*) copies *all* conses of *object*, returning an object that is EQUAL to *object*.

See COPY-ALIST for copying association lists (see Section 5.2.8).

```
(DEFUN MAKE-LIST (N OBJ LST)
  ((AND (INTEGERP N) (PLUSP N))
    (CONS OBJ (MAKE-LIST (SUB1 N) OBJ LST)) )
  LST )

(DEFUN COPY-LIST (LST)
  ((ATOM LST)  LST)
  (CONS (CAR LST) (COPY-LIST (CDR LST))) )

(DEFUN COPY-TREE (OBJ)
  ((ATOM OBJ)  OBJ)
  (CONS (COPY-TREE (CAR OBJ)) (COPY-TREE (CDR OBJ))) )

$ (MAKE-LIST 4)                                    ;Make 4 element list
(NIL NIL NIL NIL)

$ (MAKE-LIST 3 '(A B C))                           ;Make 3 element list
((A B C) (A B C) (A B C))

$ (MAKE-LIST 3 5 '(2 3))                           ;Add three 5's to list
(5 5 5 2 3)

$ (COPY-LIST '(A B (C . D) E))                     ;Does not copy (C . D)
(A B (C . D) E)

$ (COPY-TREE '(A B (C . D) E))                     ;Does copy (C . D)
(A B (C . D) E)
```

SUBLIST [*list*, *n*, *m*] Function

If *n* and *m* are integers and $0 \leq n \leq m$, (SUBLIST *list n m*) copies and returns the *n*th through *m*th elements of *list*, where the car of *list* is the zeroth element. If *m* is not an integer or is greater than or equal to the length of *list*, *m* is equivalent to being one less than the length of *list*. If *n* is not an integer greater than or equal to 0 or if $n > m$, SUBLIST returns NIL.

```
(DEFUN SUBLIST (LST N M)
  ((INTEGERP N)
    ((INTEGERP M)
      (FIRSTN (ADD1 (- M N)) (NTHCDR N LST)) )
    (NTHCDR N LST) ) )

$ (SUBLIST '(A B C D E F) 2 4)                    ;2nd thru 4th element
(C D E)

$ (SUBLIST '(A B C D E F) 2 2)                          ;2nd element
(C)

$ (SUBLIST '(A B C D E F) 2 1)                          ;No elements
NIL

$ (SUBLIST '(A B C D E F) 2)                     ;2nd thru last element
(C D E F)

$ (SUBLIST '(A B C D E F) 0 3)                    ;0th thru 3rd element
(A B C D)
```

FIRSTN [*n*, *list*] Function
BUTLAST [*list*, *n*] Function

If *n* is a positive integer, (FIRSTN *n list*) copies and returns the first *n* elements of *list*. FIRSTN returns NIL if *n* is not a positive integer. If *list* has *n* or fewer elements, FIRSTN copies and returns the elements of *list*.

If *n* is zero or a positive integer, (BUTLAST *list n*) copies and returns all but the last *n* elements of *list*. If *n* is omitted or is neither zero nor a positive integer, BUTLAST copies and returns all but the last element of *list*. See NBUTLAST in the next subsection for a destructive version of BUTLAST.

```
(DEFUN FIRSTN (N LST)
  ((AND (INTEGERP N) (PLUSP N))
    ((ATOM LST) NIL)
    (CONS (CAR LST) (FIRSTN (SUB1 N) (CDR LST))) ) )

(DEFUN BUTLAST (LST N)
  ((AND (INTEGERP N) (>= N 0))
    (FIRSTN (- (LENGTH LST) N) LST) )
  (BUTLAST LST 1) )

$ (FIRSTN 0 '(A B C D))                    ;First 0 elements of a 4 element list
NIL

$ (FIRSTN 2 '(A B C D))                    ;First 2 elements of a 4 element list
(A B)

$ (FIRSTN 5 '(A B C D))                    ;First 5 elements of a 4 element list
(A B C D)

$ (FIRSTN 3 '(A B . C))                      ;First 3 elements of a binary tree
(A B)

$ (BUTLAST '(A B C D E))                           ;But last element of a list
(A B C D)

$ (BUTLAST '(A B C D E) 2)                       ;But last 2 elements of a list
(A B C)

$ (SETQ FOO '(A B C))                                          ;Initialize FOO
(A B C)

$ (BUTLAST FOO)                                          ;But last element of FOO
(A B)

$ FOO                                                     ;FOO remains unchanged
(A B C)
```

LDIFF [*list*, *tail*] Function

If *tail* is a tail (one of the top-level conses) of *list*, (LDIFF *list tail*) copies and returns the elements of *list* up to, but not including, *tail*. If *tail* is not a tail of *list*, LDIFF copies and returns *list*. In neither case is *list* modified or destroyed.

LDIFF is short for list difference.

```
(DEFUN LDIFF (LST TAIL)
  ((ATOM LST)  LST)
  ((EQ LST TAIL)  NIL)
  (CONS (CAR LST) (LDIFF (CDR LST) TAIL)) )

$ (SETQ NUMS '(1 2 3 4 5))                          ;Initialize NUMS
(1 2 3 4 5)

$ (SETQ TAIL-NUMS (CDDDR NUMS))                     ;Tail of NUMS
(4 5)

$ (LDIFF NUMS TAIL-NUMS)                            ;But TAIL-NUMS of NUMS
(1 2 3)

$ (LDIFF NUMS '(4 5))                               ;(4 5) is not a tail of NUMS
(1 2 3 4 5)
```

REVERSE [*list*, *object*] — Function
REVAPPEND [*list*, *object*] — Library function

(REVERSE *list*) constructs and returns a list consisting of the elements in *list*, but in reverse order. (REVERSE *list object*) returns the reverse of *list* appended to *object*. The result is the same as (APPEND (REVERSE *list*) *object*), but the two argument call on REVERSE is more efficient.

The definition of REVAPPEND is *identical* to that of REVERSE and is included in COMMON.LSP solely for compatibility.

REVERSE uses consing to construct and return a new list rather than modifying *list*. See the next subsection for a destructive version of REVERSE.

```
(DEFUN REVERSE (LST OBJ)
   ((ATOM LST)  OBJ)
   (REVERSE (CDR LST) (CONS (CAR LST) OBJ)) )

$ (REVERSE '(A B C D E))                                  ;Reverse list
(E D C B A)

$ (REVERSE '(A B C) '(D E F))                            ;Reverse and append
(C B A D E F)

$ (REVERSE '(A B C) 'D)                                  ;Reverse and append
(C B A . D)

$ (SETQ FOO '(A B C))                                     ;Initialize FOO
(A B C)

$ (REVERSE FOO)                                           ;Reverse of FOO
(C B A)

$ FOO                                                     ;FOO unchanged
(A B C)
```

OBLIST [ ] Function
STACK-LIST [ ] Function

(OBLIST) constructs and returns the **object list**, a list of the currently active symbols in the system. (It would be more appropriate to call it the **symbol list** but we bow to tradition.) The symbols are listed in the reverse order from which they were read in and/or generated by the string functions: newer symbols to the left of older symbols. Note that unneeded symbols are automatically collected during a **garbage collection** (see Section 4.2).

(STACK-LIST) constructs and returns a list of the objects currently on the **variable stack** (see Section 4.2). The objects are listed in order: more recently pushed objects to the left of less recently pushed objects. STACK-LIST is primarily useful during an error break to help determine what caused the error.

```
(DEFUN OBLIST ()
  (construct and return a list of the symbols currently in the system) )

(DEFUN STACK-LIST ()
  (construct and return a list of the objects on the variable stack) )

$ (LENGTH (OBLIST))                          ;Current number of symbols in system
421

$ (COUNT-IF 'GETD (OBLIST))                 ;Current number of functions in system
365

$ (STACK-LIST)                                ;List of objects on variable stack
(EVAL T DRIVER)
```

5.2.3 Modifiers

The functions in this subsection are for modifying the car or cdr cell of data objects. Modifier functions are used primarily for their *effect* rather than their *value*. By modifying an existing data structure, they can eliminate the need for consing together a whole new structure when only a small part needs to be changed. Since these functions can produce unwanted **side-effects**, they should be used with care.

RPLACA [$object_1$, $object_2$] — Function
RPLACD [$object_1$, $object_2$] — Function

If $object_1$ is neither the symbol NIL nor a number, (RPLACA $object_1$ $object_2$) replaces the car cell of $object_1$ with a pointer to $object_2$, and (RPLACD $object_1$ $object_2$) replaces the cdr cell of $object_1$ with a pointer to $object_2$. Both functions return the newly modified $object_1$.

The effect of replacing the car or cdr of an object depends on its data type (see Section 4.1). Replacing the car of a list replaces its first element with $object_2$. Replacing the cdr of a list replaces its tail with $object_2$. Replacing the car of a binary tree replaces its left branch with $object_2$. Replacing the cdr of a binary tree replaces its right branch with $object_2$. Replacing the car of a symbol sets its value cell to $object_2$. Replacing the cdr of a symbol sets its property list to $object_2$.

```
(DEFUN RPLACA (OBJ1 OBJ2)
  ((OR (NUMBERP OBJ1) (NULL OBJ1))  OBJ1)
  (replace car cell of OBJ1 with pointer to OBJ2)
  OBJ1 )

(DEFUN RPLACD (OBJ1 OBJ2)
  ((OR (NUMBERP OBJ1) (NULL OBJ1))  OBJ1)
  (replace cdr cell of OBJ1 with pointer to OBJ2)
  OBJ1 )
```

```
$ (SETQ FOO '(C A T))                                  ;Initialize FOO
(C A T)

$ (RPLACA FOO 'R)                                      ;Replace car of FOO
(R A T)

$ (RPLACD FOO '(E A D))                                ;Replace cdr of FOO
(R E A D)

$ FOO                                                  ;Newly modified FOO
(R E A D)
```

FILL [*list*, *object*] Function
REPLACE [$list_1$, $list_2$] Function

By modifying the car of the top-level conses of *list*, (FILL *list object*) replaces the elements of *list* with *object* and returns the newly modified *list*.

By modifying the car of the top-level conses of *list*1, (REPLACE $list_1$ $list_2$) replaces successive elements of $list_1$ with successive elements of $list_2$. If the lists are of unequal lengths, replacement ceases when the end of the shorter list is reached.

```
(DEFUN FILL (LST OBJ)
  ((ATOM LST)  LST)
  (FILL (CDR LST) OBJ)
  (RPLACA LST OBJ) )

(DEFUN REPLACE (LST1 LST2)
  ((OR (ATOM LST1) (ATOM LST2))  LST1)
  (REPLACE (CDR LST1) (CDR LST2))
  (RPLACA LST1 (CAR LST2)) )

$ (SETQ FOO '(X Y Z W))                          ;Initialize FOO
(X Y Z W)

$ (REPLACE FOO '(1 2 3))                        ;Replace 3 elements of FOO
(1 2 3 W)

$ FOO                                            ;Newly modified FOO
(1 2 3 W)

$ (FILL FOO 4)                                   ;Fill FOO with 4's
(4 4 4 4)

$ FOO                                            ;Newly modified FOO
(4 4 4 4)
```

NCONC [$list_1$, $list_2$, ..., $list_n$] Function

(NCONC $list_1$ $list_2$ ... $list_n$) returns a list consisting of the elements of the lists $list_1$, $list_2$, through $list_n$ in this order, by modifying the cdr of the last top-level cons of $list_1$ through $list_{n-1}$.

Note that if NCONC and APPEND, defined in the previous subsection, are given the same arguments, they will return EQUAL lists. However, APPEND uses conses to copy all but its last argument, whereas NCONC actually modifies all but its last argument by redirecting their final cdr cells.

(NCONC *list list*) will turn *list* into a **circular list**. If an attempt is then made to display *list*, the display will continue indefinitely or until halted by a console interrupt (see Section 4.4).

Compare the following examples with those given for APPEND in the previous subsection.

```
(DEFUN NCONC (LST1 LST2)
  ((ATOM LST1)  LST2)
  (RPLACD (LAST LST1) LST2)
  LST1 )

$ (SETQ FOO '(D E F))                                    ;Initialize FOO
(D E F)

$ (NCONC '(A B C) FOO '(G H I))                          ;Concatenate lists
(A B C D E F G H I)

$ FOO                                                    ;Newly modified FOO
(D E F G H I)

$ (NCONC '(A B C D) 'E 'F 'G)                            ;Nconcing atoms
(A B C D . G)
```

NREVERSE [*list*, *object*] Function
NRECONC [*list*, *object*] Library function

By modifying the top-level conses of *list*, (NREVERSE *list*) reverses the elements in *list*. (NREVERSE *list object*) returns the nreverse of *list* concatenated to *object*. The result is the same as (NCONC (NREVERSE *list*) *object*), but the two argument call on NREVERSE is more efficient.

The definition of NRECONC is *identical* to that of NREVERSE and is included in COMMON.LSP solely for compatibility.

By destructively modifying *list*, NREVERSE requires no additional conses to reverse *list*. REVERSE is a non-destructive version of NREVERSE (see Section 5.2.2).

```
(DEFUN NREVERSE (LST OBJ)
  ((ATOM LST)  OBJ)
  (NREVERSE (CDR LST) (RPLACD LST OBJ)) )

$ (NREVERSE '(A B C D E))                                  ;Reverse list
(E D C B A)

$ (NREVERSE '(A B C) '(D E F))                            ;Reverse and nconc
(C B A D E F)

$ (NREVERSE '(A B C) 'D)                                  ;Reverse and nconc
(C B A . D)

$ (SETQ FOO '(A B C))                                       ;Initialize FOO
(A B C)

$ (NREVERSE FOO)                                            ;Reverse FOO
(C B A)

$ FOO                                        ;FOO has been destructively modified!
(A)

$ (SETQ FOO '(A B C))                                       ;Initialize FOO
(A B C)

$ (SETQ FOO (NREVERSE FOO))                                 ;Reverse FOO
(C B A)

$ FOO                                                  ;FOO is correctly reversed
(C B A)                                      ;Moral: Always use SETQ with NREVERSE
```

NBUTLAST [*list*, *n*] Function

If *n* is zero or a positive integer, (NBUTLAST *list n*) returns a list consisting of all but the last *n* elements of *list*, by replacing the cdr of the *n*th cons from the end of *list* with NIL. *n* defaults to 1.

By destructively modifying *list*, NBUTLAST requires no additional conses to return all but the last *n* elements of *list*. BUTLAST is a non-destructive version of NBUTLAST (see Section 5.2.2).

```
(DEFUN NBUTLAST (LST N)
  ((AND (INTEGERP N) (>= N 0))
    (NREVERSE (NTHCDR N (NREVERSE LST))) )
  (NBUTLAST LST 1) )

$ (NBUTLAST '(A B C D))                          ;Remove last element
(A B C)

$ (NBUTLAST '(A B C D) 2)                        ;Remove last 2 elements
(A B)

$ (SETQ FOO '(A B C))                            ;Initialize FOO
(A B C)

$ (NBUTLAST FOO)                                 ;Remove last element
(A B)

$ FOO                                            ;Newly modified FOO
(A B)

$ (NBUTLAST FOO 2)                               ;Remove last 2 elements
NIL

$ FOO                                            ;FOO is not modified!
(A B)

$ (SETQ FOO (NBUTLAST FOO 2))                    ;Remove last 2 elements
NIL

$ FOO                                            ;FOO is now modified!
NIL                                              ;Moral: Always use SETQ with NBUTLAST
```

REPLACE-NTH [*item*, *list*, *n*] Function
INSERT-NTH [*item*, *list*, *n*] Function
DELETE-NTH [*list*, *n*] Function

By modifying *list*, (REPLACE-NTH *item list n*) replaces the *n*th element of *list* with *item*, where the car of *list* is the *0*th element of *list*. If *list* has *n* or fewer elements, REPLACE-NTH returns *list*.

By modifying *list*, (INSERT-NTH *item list n*) inserts *item* before the *n*th element of *list*. If *list* has fewer than *n* elements, INSERT-NTH returns *list*.

By modifying *list*, (DELETE-NTH *list n*) deletes the *n*th element of *list*. If *list* has *n* or fewer elements, DELETE-NTH returns *list*.

```
(DEFUN INSERT-NTH (ITEM LST N)
  ((AND (INTEGERP N) (>= N 0))
    ((ZEROP N)
      (CONS ITEM LST) )
    ((ATOM LST)  LST)
    (RPLACD LST (INSERT-NTH ITEM (CDR LST) (SUB1 N))) ) )

(DEFUN DELETE-NTH (LST N)
  ((AND (INTEGERP N) (>= N 0))
    ((ATOM LST)  LST)
    ((ZEROP N)
      (CDR LST) )
    (RPLACD LST (DELETE-NTH (CDR LST) (SUB1 N))) ) )

$ (REPLACE-NTH 'A '(B C D E) 2)                    ;Replace 2nd element
(B C A E)

$ (INSERT-NTH 'A '(B C D E) 2)                      ;Insert before 2nd
(B C A D E)

$ (DELETE-NTH '(A B C D) 1)                        ;Delete 1st element
(A C D)

$ (DELETE-NTH '(A B C D) 4)                        ;Delete 4th element
(A B C D)
```

TCONC [*dotted-pair, object*] Function
LCONC [*dotted-pair, list*] Function

By modifying the list, (TCONC *dotted-pair object*) adds the new element *object* to the end of the list pointed to by the car of *dotted-pair*. TCONC returns *dotted-pair* with the cdr cell modified to point to the new end of the list.

By modifying the list, (LCONC *dotted-pair list*) appends *list* to the end of the list pointed to by the car of *dotted-pair*. LCONC returns *dotted-pair* with the cdr cell modified to point to the new end of the list.

TCONC and LCONC respectively add and append elements to the tail end of a list, efficiently using RPLACD to modify the list. Their first argument is a cons, the car of which points to the beginning of a list and the cdr of which points to the end (i.e. the last cons) of the list. Their second argument is the item to be added to the end of the list. As shown in the examples, the first time they are called, their first argument should be the atom NIL or the list (NIL).

```
(DEFUN TCONC (PAIR OBJ)
  (LCONC PAIR (LIST OBJ)) )

(DEFUN LCONC (PAIR LST)
  ((ATOM LST)  PAIR)
  ((ATOM PAIR)
    (CONS LST (LAST LST)) )
  ((ATOM (CDR PAIR))
    (RPLACA PAIR LST)
    (RPLACD PAIR (LAST LST)) )
  (RPLACD (CDR PAIR) LST)
  (RPLACD PAIR (LAST LST)) )

$ (SETQ FOO (TCONC NIL 'A))                    ;Add A to end of empty list
((A) A)                                        ;Note that SETQ is required

$ (TCONC FOO 'B)                               ;Add B to end of list
((A B) B)

$ (CAR FOO)                                    ;Extract list
(A B)

$ (SETQ FOO (LIST NIL))                        ;Initialize FOO to (NIL)
(NIL)

$ (LCONC FOO '(A B))                           ;Add (A B) to end of empty list
((A B) B)                                      ;Note: SETQ is not required

$ (LCONC FOO '(C D))                           ;Add (C D) to end of list
((A B C D) D)
```

5.2.4 Deletion

The functions in this subsection are for deleting elements from lists. Both nondestructive and destructive versions are provided.

REMOVE [*item*, *list*, *test*] Function
DELETE [*item*, *list*, *test*] Function

If *test* is a comparator, (REMOVE *item list test*) returns a top-level copy of *list* except for those elements for which (*test item element*) is nonNIL. *list* is *not* altered.

By modifying the cdr of the top-level conses of *list*, (DELETE *item list test*) deletes all elements of *list* for which (*test item element*) is nonNIL. Although *list* is altered, do *not* assume that *list* is EQ to the value returned by DELETE.

These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN REMOVE (ITEM LST TEST)
  ((ATOM LST)  LST)
  ((FUNCALL (OR TEST 'EQL) ITEM (CAR LST))
    (REMOVE ITEM (CDR LST) TEST) )
  (CONS (CAR LST) (REMOVE ITEM (CDR LST) TEST)) )

(DEFUN DELETE (ITEM LST TEST)
  ((ATOM LST)  LST)
  ((FUNCALL (OR TEST 'EQL) ITEM (CAR LST))
    (DELETE ITEM (CDR LST) TEST) )
  (delete-aux ITEM LST LST TEST) )

(DEFUN delete-aux (ITEM LST TAIL TEST)
  ((ATOM (CDR TAIL))  LST)
  ((FUNCALL (OR TEST 'EQL) ITEM (CADR TAIL))
    (RPLACD TAIL (CDDR TAIL))
    (delete-aux ITEM LST TAIL TEST) )
  (delete-aux ITEM LST (CDR TAIL) TEST) )

$ (REMOVE '(2 5) '((5 2) (2 5) (2 3)))                    ;Lists are not EQL
((5 2) (2 5) (2 3))

$ (REMOVE '(2 5) '((5 2) (2 5) (2 3)) 'EQUAL)              ;Lists are EQUAL
((5 2) (2 3))

$ (SETQ ANIMALS '(DOG RAT CAT PIG RAT COW))               ;Initialize ANIMALS
(DOG RAT CAT PIG RAT COW)

$ (DELETE 'RAT ANIMALS)                                     ;Deletes all RATs
(DOG CAT PIG COW)                                        ;and modifies ANIMALS
```

REMOVE-IF [*test, list*] Function
DELETE-IF [*test, list*] Function

If *test* is a recognizer, (REMOVE-IF *test list*) returns a top-level copy of *list* except for those elements for which (*test element*) is nonNIL. *list* is *not* altered.

By modifying the cdr of the top-level conses of *list*, (DELETE-IF *test list*) deletes all elements of *list* for which (*test element*) is nonNIL. Although *list* is altered, do *not* assume that *list* is EQ to the value returned by DELETE-IF.

```
(DEFUN REMOVE-IF (TEST LST)
  ((ATOM LST)  LST)
  ((FUNCALL TEST (CAR LST))
    (REMOVE-IF TEST (CDR LST)) )
  (CONS (CAR LST) (REMOVE-IF TEST (CDR LST))) )

(DEFUN DELETE-IF (TEST LST)
  ((ATOM LST)  LST)
  ((FUNCALL TEST (CAR LST))
    (DELETE-IF TEST (CDR LST)) )
  (delete-if-aux TEST LST LST) )

(DEFUN delete-if-aux (TEST LST TAIL)
  ((ATOM (CDR TAIL))  LST)
  ((FUNCALL TEST (CADR TAIL))
    (RPLACD TAIL (CDDR TAIL))
    (delete-if-aux TEST LST TAIL) )
  (delete-if-aux TEST LST (CDR TAIL)) )

$ (SETQ NUMS '(-2 0 7 -0.1 3))                    ;Initialize NUMS
(-2 0 7 -0.1 3)

$ (REMOVE-IF 'MINUSP NUMS)                        ;Remove all negative numbers
(0 7 3)

$ NUMS                                            ;NUMS left unchanged
(-2 0 7 -0.1 3)

$ (DELETE-IF 'MINUSP NUMS)                        ;Delete all negative numbers
(0 7 3)

$ NUMS                                            ;NUMS now modified
(-2 0 7 3)                                        ;but not EQ to answer
```

REMOVE-DUPLICATES [*list*, *test*] Function
DELETE-DUPLICATES [*list*, *test*] Function

(REMOVE-DUPLICATES *list test*) returns a copy of *list* with all duplicate elements removed. Duplicate elements are those for which (*test* $element_1$ $element_2$) is nonNIL.

By modifying the cdr of the top-level conses of *list*, (DELETE-DUPLICATES *list test*) removes duplicate elements of *list*. Duplicate elements are those for which (*test* $element_1$ $element_2$) is nonNIL. Although *list* is altered, do *not* assume that *list* is EQ to the value returned by DELETE-DUPLICATES.

These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN REMOVE-DUPLICATES (LST TEST)
  ((ATOM LST)  LST)
  ((MEMBER (CAR LST) (CDR LST) TEST)
    (REMOVE-DUPLICATES (CDR LST) TEST) )
  (CONS (CAR LST) (REMOVE-DUPLICATES (CDR LST) TEST)) )

(DEFUN DELETE-DUPLICATES (LST TEST)
  ((ATOM LST)  LST)
  ((MEMBER (CAR LST) (CDR LST) TEST)
    (DELETE-DUPLICATES (CDR LST) TEST) )
  (delete-duplicates-aux LST LST TEST) )

(DEFUN delete-duplicates-aux (LST TAIL TEST)
  ((ATOM (CDR TAIL))  LST)
  ((MEMBER (CADR TAIL) (CDDR TAIL) TEST)
    (RPLACD TAIL (CDDR TAIL))
    (delete-duplicates-aux LST TAIL TEST) )
  (delete-duplicates-aux LST (CDR TAIL) TEST) )

$ (SETQ NUMS '(4 2 3 4 5 3 0))                          ;Initialize NUMS
(4 2 3 4 5 3 0)

$ (REMOVE-DUPLICATES NUMS)                              ;Remove duplicates
(2 4 5 3 0)

$ NUMS                                                  ;NUMS left unchanged
(4 2 3 4 5 3 0)

$ (DELETE-DUPLICATES NUMS)                              ;Delete duplicates
(2 4 5 3 0)

$ NUMS                                                  ;NUMS destructively modified
(4 2 4 5 3 0)                                           ;Moral: Use SETQ with DELETE-DUPLICATES
```

5.2.5 Search

The functions in this subsection are for searching through the elements of lists to test for certain conditions. If no test is specified, these functions all use an EQL test.

FIND [*object, list, test*] Function
FIND-IF [*test, list*] Function

(FIND *object list test*) performs a linear search of *list* looking for an element for which (*test object element*) is nonNIL. If the test argument is NIL or not given, FIND uses an EQL test.

(FIND-IF *test list*) searches *list* for an element for which (*test element*) is nonNIL.

For both functions, if an element satisfying the test is found, the element is returned; otherwise NIL is returned.

```
(DEFUN FIND (OBJ LST TEST)
  ((ATOM LST) NIL)
  ((FUNCALL (OR TEST 'EQL) OBJ (CAR LST))
    (CAR LST) )
  (FIND OBJ (CDR LST) TEST) )

$ (FIND 'EAT '(CORN WHEAT OATS RICE) 'FINDSTRING)              ;Element containing EAT
WHEAT

$ (FIND 7 '(5 7 3 8 12 9 3) '<)                          ;First element that 7 is less than
8

$ (SETQ WEIGHT '((SUE 120) (TOM 182) (AL 174)))
((SUE 120) (TOM 182) (AL 174))

$ (FIND 'TOM WEIGHT '(LAMBDA (X Y)(EQ X (CAR Y))))                          ;Tom's weight
(TOM 182)

$ (FIND-IF '(LAMBDA (Y) (EQ 'TOM (CAR Y))) WEIGHT)                          ;Tom's weight
(TOM 182)

$ (FIND-IF 'MINUSP '(4 0 -6 2 -3 7))                     ;First negative number in list
-6

$ (FIND-IF '(LAMBDA (X) (MINUSP (CADR X))) '((X 3.7) (Y 0.5) (Z -2.4)))
(Z -2.4)                                                 ;First negative coordinate in list
```

POSITION [*object, list, test*] Function
POSITION-IF [*test, list*] Function

(POSITION *object list test*) performs a linear search of *list* looking for an element for which (*test object element*) is nonNIL. If the test argument is NIL or not given, POSITION uses an EQL test.

(POSITION-IF *test list*) searches *list* for an element for which (*test element*) is nonNIL.

For both functions, if an element satisfying the test is found, the zero-based index of the element is returned; otherwise NIL is returned.

```
(DEFUN POSITION (OBJ LST TEST)
  (position-aux OBJ LST TEST 0) )

(DEFUN position-aux (OBJ LST TEST INDEX)
  ((ATOM LST)  NIL)
  ((FUNCALL (OR TEST 'EQL) OBJ (CAR LST))  INDEX)
  (position-aux OBJ (CDR LST) TEST (ADD1 INDEX)) )

$ (POSITION 23 '(2 3 5 7 11 13 17 19 23 29 31 37))          ;Number of primes before 23
8

$ (POSITION '(A B C) '((R S T) (C A B) (A B C)) 'EQUAL)    ;Position of sublist in list
2

$ (POSITION '(A B C) '((R S T) (C A B) (A B C)))    ;Sublist not EQL to element of list
NIL

$ (POSITION-IF 'PLUSP '(-2.5 0 X 3.7 -5.3))         ;Position of first positive element
3

$ (POSITION-IF 'EVENP '(2 3 5 7 11 13))                  ;Position of first even element
0
```

COUNT [*object, list, test*] Function
COUNT-IF [*test, list*] Function

(COUNT *object list test*) returns the number of elements in *list* for which (*test object element*) is nonNIL. If the test argument is NIL or not given, COUNT uses an EQL test.

(COUNT-IF *test list*) returns the number of elements in *list* for which (*test element*) is nonNIL.

```
(DEFUN COUNT (OBJ LST TEST)
  (count-aux OBJ LST TEST 0) )

(DEFUN count-aux (OBJ LST TEST COUNTER)
  ((ATOM LST)  COUNTER)
  ((FUNCALL (OR TEST 'EQL) OBJ (CAR LST))
    (count-aux OBJ (CDR LST) TEST (ADD1 COUNTER)) )
  (count-aux OBJ (CDR LST) TEST COUNTER) )

$ (COUNT 'DOG '(CAT DOG COW PIG DOG ANT))                    ;Number of DOGs in list
2

$ (COUNT 7 '(5 8 12 7 9 3) '<)                  ;Number of numbers that 7 is less than
3

$ (COUNT 'T (UNPACK 'SUBSTITUTE))                          ;Number of Ts in SUBSTITUTE
3

$ (COUNT-IF 'EVENP '(3 -6 8 7 0))                       ;Number of even integers in list
3

$ (COUNT-IF '(LAMBDA (CHAR) (MEMBER CHAR '(A E I O U))) (UNPACK 'AUTOMOBILE))
6                                                        ;Number of vowels in AUTOMOBILE
```

MISMATCH [$list_1$, $list_2$, *test*] Function

(MISMATCH $list_1$ $list_2$ *test*) compares the elements of $list_1$ and $list_2$ using the two argument predicate *test* until a mismatch is found (i.e. until *test* returns NIL), or the end of one or both of the lists is reached. If the *test* argument is NIL or not given, MISMATCH uses an EQL test.

If the lists are of equal length and no mismatch is found, MISMATCH returns NIL. If the lists are of unequal length and no mismatch is found, MISMATCH returns the length of the shorter list. Otherwise, MISMATCH returns the zero-based index in the lists of the first pair of elements that fail to match.

```
(DEFUN MISMATCH (LST1 LST2 TEST)
  (mismatch-aux LST1 LST2 TEST 0) )

(DEFUN mismatch-aux (LST1 LST2 TEST INDEX)
  ((AND (ATOM LST1) (ATOM LST2))  NIL)
  ((OR (ATOM LST1) (ATOM LST2))  INDEX)
  ((NOT (FUNCALL (OR TEST 'EQL) (CAR LST1) (CAR LST2)))  INDEX)
  (mismatch-aux (CDR LST1) (CDR LST2) TEST (ADD1 INDEX)) )

$ (MISMATCH '(A B C D E) '(A B D C E))                    ;Index of first mismatch
2

$ (MISMATCH '(A B C D E) '(A B C D))                      ;Length of shorter list
4

$ (MISMATCH '(5 -4 6 9) '(7 -3 9 8) '<)          ;Index of first element greater than
3

$ (MISMATCH '(5 -4 6 2) '(7 -3 9 8) '<)                   ;All elements less than
NIL
```

MEMBER [*object, list, test*] Function
MEMBER-IF [*test, list*] Function

(MEMBER *object list test*) performs a linear search of *list* looking for an element for which (*test object element*) is nonNIL. If the *test* argument is NIL or not given, MEMBER uses an EQL test.

(MEMBER-IF *test list*) searches *list* for an element for which (*test element*) is nonNIL.

For both functions, if an element satisfying the test is found, the tail of *list* beginning with that element is returned; otherwise NIL is returned.

Note that in some versions of LISP, MEMBER uses an EQUAL test. Of course, this effect can easily be achieved by the call (MEMBER *object list* 'EQUAL).

```
(DEFUN MEMBER (OBJ LST TEST)
  ((ATOM LST)  NIL)
  ((FUNCALL (OR TEST 'EQL) OBJ (CAR LST))  LST)
  (MEMBER OBJ (CDR LST) TEST) )

$ (MEMBER 'A '(B C D))                          ;A is not EQL to a member of list
NIL

$ (MEMBER 'A '(B A D))                              ;A is EQL to a member of list
(A D)

$ (MEMBER '(3 4) '((7 8) (3 4) (5 2)))          ;(3 4) is not EQL to a member of list
NIL

$ (MEMBER '(3 4) '((7 8) (3 4) (5 2)) 'EQUAL)     ;(3 4) is EQUAL to a member of list
((3 4) (5 2))

$ (MEMBER-IF 'NUMBERP '(A B 3 C (-7)))                  ;There is a number in list
(3 C (-7))

$ (MEMBER-IF 'MINUSP '(A B 3 C (-7)))           ;There is no negative number in list
NIL
```

5.2.6 Substitution

The functions in this subsection are for substituting new objects for old objects in lists and binary trees. The substitution can be accomplished either by copying or by modifying the original list or binary tree.

SUBSTITUTE [*new, old, list, test*] Function
NSUBSTITUTE [*new, old, list, test*] Function

(SUBSTITUTE *new old list test*) returns a top-level copy of *list*, replacing with *new* the elements of *list* for which (*test old element*) is nonNIL. If the *test* argument is NIL or not given, SUBSTITUTE uses an EQL test. *list* is *not* altered.

By modifying the car of the top-level conses of *list*, (NSUBSTITUTE *new old list test*) replaces with *new* the elements of *list* for which (*test old element*) is nonNIL. *list* is altered.

These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN SUBSTITUTE (NEW OLD LST TEST)
  ((ATOM LST)  LST)
  (CONS (IF (FUNCALL (OR TEST 'EQL) OLD (CAR LST)) NEW (CAR LST))
        (SUBSTITUTE NEW OLD (CDR LST) TEST)) )

(DEFUN NSUBSTITUTE (NEW OLD LST TEST)
  ((ATOM LST)  LST)
  (NSUBSTITUTE NEW OLD (CDR LST) TEST)
  ((FUNCALL (OR TEST 'EQL) OLD (CAR LST))
    (RPLACA LST NEW) )
  LST )

$ (SUBSTITUTE 'CANNIBALS 'NOUN '(NOUN LIKE TO EAT NOUN))                ;Replace NOUNs
(CANNIBALS LIKE TO EAT CANNIBALS)

$ (SETQ EXPN '(+ 3 X (* 2 X) X))                                       ;Initialize EXPN
(+ 3 X (* 2 X) X)

$ (SUBSTITUTE 5 'X EXPN)                                     ;Replace top-level Xs with 5
(+ 3 5 (* 2 X) 5)

$ EXPN                                                              ;EXPN left unchanged
(+ 3 X (* 2 X) X)

$ (NSUBSTITUTE 5 'X EXPN)                                    ;Replace top-level Xs with 5
(+ 3 5 (* 2 X) 5)

$ EXPN                                                              ;EXPN now modified
(+ 3 5 (* 2 X) 5)
```

SUBSTITUTE-IF [*new, test, list*] Function
NSUBSTITUTE-IF [*new, test, list*] Function

(SUBSTITUTE-IF *new test list*) returns a top-level copy of *list*, replacing with *new* all the elements of *list* for which (*test element*) is nonNIL. *list* is *not* altered.

By modifying the car of the top-level conses of *list*, (NSUBSTITUTE-IF *new test list*) replaces with *new* the elements of *list* for which (*test element*) is nonNIL. *list* is altered.

```
(DEFUN SUBSTITUTE-IF (NEW TEST LST)
  ((ATOM LST)  LST)
  (CONS (IF (FUNCALL TEST (CAR LST)) NEW (CAR LST))
        (SUBSTITUTE-IF NEW TEST (CDR LST))) )

(DEFUN NSUBSTITUTE-IF (NEW TEST LST)
  ((ATOM LST)  LST)
  (NSUBSTITUTE-IF NEW TEST (CDR LST))
  ((FUNCALL TEST (CAR LST))
    (RPLACA LST NEW) )
  LST )

$ (SUBSTITUTE-IF 0                    ;Replace top-level small magnitude numbers with 0
                 '(LAMBDA (N) (AND (NUMBERP N) (< -0.1 N 0.1)))
                 '(0.5 -0.09 (1.1 0.03) 0.2))
(0.5 0 (1.1 0.03) 0.2)

$ (SETQ EXPN '(3 -2 (-7 5) -4.7))                                  ;Initialize EXPN
(3 -2 (-7 5) -4.7)

$ (SUBSTITUTE-IF 0 'MINUSP EXPN)                  ;Replace top-level negatives with 0
(3 0 (-7 5) 0)

$ EXPN                                                            ;EXPN left unchanged
(3 -2 (-7 5) -4.7)

$ (NSUBSTITUTE-IF 0 'MINUSP EXPN)                 ;Replace top-level negatives with 0
(3 0 (-7 5) 0)

$ EXPN                                                             ;EXPN now modified
(3 0 (-7 5) 0)
```

SUBST [*new, old, object, test*] Function
NSUBST [*new, old, object, test*] Function

(SUBST *new old object test*) returns a copy of *object*, replacing with *new* all subexpressions of *object* for which (*test old subexpression*) is nonNIL. To prevent unnecessary consing, SUBST uses as much of *object* as possible in the result; however in no case is *object* altered.

By modifying the conses of *object*, (NSUBST *new old object test*) replaces with *new* all subexpressions of *object* for which (*test old subexpression*) is nonNIL. *object* is altered.

These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN SUBST (NEW OLD OBJ TEST)
  ((FUNCALL (OR TEST 'EQL) OLD OBJ)  NEW)
  ((ATOM OBJ)  OBJ)
  (LET ( (LEFT (SUBST NEW OLD (CAR OBJ) TEST))
         (RIGHT (SUBST NEW OLD (CDR OBJ) TEST)) )
    ((AND (EQ LEFT (CAR OBJ)) (EQ RIGHT (CDR OBJ)))  OBJ)
    (CONS LEFT RIGHT) ) )

(DEFUN NSUBST (NEW OLD OBJ TEST)
  ((FUNCALL (OR TEST 'EQL) OLD OBJ)  NEW)
  ((ATOM OBJ)  OBJ)
  (RPLACA OBJ (NSUBST NEW OLD (CAR OBJ) TEST))
  (RPLACD OBJ (NSUBST NEW OLD (CDR OBJ) TEST)) )

$ (SETQ EXPN '(+ 3 X (* 2 X) X))                    ;initialize EXPN
(+ 3 X (* 2 X) X)

$ (SUBST 5 'X EXPN)                                 ;Replace all Xs with 5
(+ 3 5 (* 2 5) 5)

$ EXPN                                              ;EXPN left unchanged
(+ 3 X (* 2 X) X)

$ (NSUBST 5 'X EXPN)                                ;Replace all Xs with 5
(+ 3 5 (* 2 5) 5)

$ EXPN                                              ;EXPN now modified
(+ 3 5 (* 2 5) 5)
```

SUBST-IF [*new, test, object*] Function
NSUBST-IF [*new, test, object*] Function

(SUBST-IF *new test object*) returns a copy of *object*, replacing with *new* all subexpressions of *object* for which (*test subexpression*) is nonNIL. To prevent unnecessary consing, SUBST-IF uses as much of *object* as possible in the result; however in no case is *object* altered.

By modifying the conses of *object*, (NSUBST-IF *new test object*) replaces with *new* all subexpressions of *object* for which (*test subexpression*) is nonNIL.

```
(DEFUN SUBST-IF (NEW TEST OBJ)
  ((FUNCALL TEST OBJ)  NEW)
  ((ATOM OBJ)  OBJ)
  (LET ( (LEFT (SUBST-IF NEW TEST (CAR OBJ)))
         (RIGHT (SUBST-IF NEW TEST (CDR OBJ))) )
    ((AND (EQ LEFT (CAR OBJ)) (EQ RIGHT (CDR OBJ)))  OBJ)
    (CONS LEFT RIGHT) ) )

(DEFUN NSUBST-IF (NEW TEST OBJ)
  ((FUNCALL TEST OBJ)  NEW)
  ((ATOM OBJ)  OBJ)
  (RPLACA OBJ (NSUBST-IF NEW TEST (CAR OBJ)))
  (RPLACD OBJ (NSUBST-IF NEW TEST (CDR OBJ))) )

$ (SUBST-IF 0                          ;Replace all small magnitude numbers with 0
           '(LAMBDA (N) (AND (NUMBERP N) (< -0.1 N 0.1)))
           '(0.5 -0.09 (1.1 0.03) 0.2))
(0.5 0 (1.1 0) 0.2)

$ (SETQ EXPN '(3 -2 (-7 5) -4.7))                                ;Initialize EXPN
(3 -2 (-7 5) -4.7)

$ (SUBST-IF 0 'MINUSP EXPN)                              ;Replace all negatives with 0
(3 0 (0 5) 0)

$ EXPN                                                          ;EXPN left unchanged
(3 -2 (-7 5) -4.7)

$ (NSUBST-IF 0 'MINUSP EXPN)                             ;Replace all negatives with 0
(3 0 (0 5) 0)

$ EXPN                                                           ;EXPN now modified
(3 0 (0 5) 0)
```

5.2.7 Sets

The functions in this subsection treat lists as if they were sets (i.e. the functions do not consider the order of the elements in lists to be important).

UNION [*list*$_1$, *list*$_2$, *test*] Function
NUNION [*list*$_1$, *list*$_2$, *test*] Library function

(UNION *list*$_1$ *list*$_2$ *test*) constructs and returns a list of those elements that are in either *list*$_1$ *or list*$_2$. If an element is in both lists, it will not be repeated in the result.

NUNION is a destructive version of UNION that modifies the top-level conses of *list*$_1$. NUNION is defined in COMMON.LSP.

For both functions, *test* is a two argument predicate used to determine if elements in *list*$_1$ match elements in *list*$_2$. The first argument of *test* is always from *list*$_1$, the second is always from *list*$_2$. These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN UNION (LST1 LST2 TEST)
  ((ATOM LST1)  LST2)
  ((MEMBER (CAR LST1) LST2 TEST)
    (UNION (CDR LST1) LST2 TEST) )
  (CONS (CAR LST1) (UNION (CDR LST1) LST2 TEST)) )

(DEFUN NUNION (LST1 LST2 TEST)
  ((ATOM LST1)  LST2)
  ((MEMBER (CAR LST1) LST2 TEST)
    (NUNION (CDR LST1) LST2 TEST) )
  (RPLACD LST1 (NUNION (CDR LST1) LST2 TEST)) )

$ (UNION '(LISP FORTH APL) '(APL PASCAL LISP C))
(FORTH APL PASCAL LISP C)
```

INTERSECTION [*list*$_1$, *list*$_2$, *test*] — Function
NINTERSECTION [*list*$_1$, *list*$_2$, *test*] — Library function

(INTERSECTION *list*$_1$ *list*$_2$ *test*) constructs and returns a list of the elements that are in both *list*$_1$ *and list*$_2$. If an element is not in both lists, it will not be included in the result.

NINTERSECTION is a destructive version of INTERSECTION that modifies the top-level conses of *list*$_1$. NINTERSECTION is defined in COMMON.LSP.

For both functions, *test* is a two argument predicate used to determine if elements in *list*$_1$ match elements in *list*$_2$. The first argument of *test* is always from *list*$_1$, the second is always from *list*$_2$. These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN INTERSECTION (LST1 LST2 TEST)
  ((ATOM LST1)  NIL)
  ((MEMBER (CAR LST1) LST2 TEST)
    (CONS (CAR LST1) (INTERSECTION (CDR LST1) LST2 TEST)) )
  (INTERSECTION (CDR LST1) LST2 TEST) )

(DEFUN NINTERSECTION (LST1 LST2 TEST)
  ((ATOM LST1)  NIL)
  ((MEMBER (CAR LST1) LST2 TEST)
    (RPLACD LST1 (NINTERSECTION (CDR LST1) LST2 TEST)) )
  (NINTERSECTION (CDR LST1) LST2 TEST) )

$ (INTERSECTION '(3 8 2 0) '(5 2 7 3 4))
(3 2)
```

SET-DIFFERENCE [$list_1$, $list_2$, *test*] Function
NSET-DIFFERENCE [$list_1$, $list_2$, *test*] Library function

(SET-DIFFERENCE $list_1$ $list_2$ *test*) constructs and returns a list of those elements of $list_1$ that do not appear in $list_2$. If an element is in both lists, it will not be included in the result.

NSET-DIFFERENCE is a destructive version of SET-DIFFERENCE that modifies the top-level conses of $list_1$. NSET-DIFFERENCE is defined in COMMON.LSP.

For both functions, *test* is a two argument predicate used to determine if elements in $list_1$ match elements in $list_2$. The first argument of *test* is always from $list_1$, the second is always from $list_2$. These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN SET-DIFFERENCE (LST1 LST2 TEST)
  ((ATOM LST1)  NIL)
  ((MEMBER (CAR LST1) LST2 TEST)
    (SET-DIFFERENCE (CDR LST1) LST2 TEST) )
  (CONS (CAR LST1) (SET-DIFFERENCE (CDR LST1) LST2 TEST)) )

(DEFUN NSET-DIFFERENCE (LST1 LST2 TEST)
  ((ATOM LST1)  NIL)
  ((MEMBER (CAR LST1) LST2 TEST)
    (NSET-DIFFERENCE (CDR LST1) LST2 TEST) )
  (RPLACD LST1 (SET-DIFFERENCE (CDR LST1) LST2 TEST)) )

$ (SET-DIFFERENCE '(APL PASCAL LISP C) '(LISP FORTH APL))
(PASCAL C)

$ (SET-DIFFERENCE '(4 2 8 5 6) '(3 6 7 2))
(4 8 5)

$ (SET-DIFFERENCE '(3 6 7 2) '(4 2 8 5 6))
(3 7)
```

SET-EXCLUSIVE-OR [$list_1$, $list_2$, *test*] — Function
NSET-EXCLUSIVE-OR [$list_1$, $list_2$, *test*] — Library function

(SET-EXCLUSIVE-OR $list_1$ $list_2$ *test*) constructs and returns a list of those elements of $list_1$ and $list_2$ that do not appear in both lists (i.e. those that appear in exactly one list).

NSET-EXCLUSIVE-OR is a destructive version of SET-EXCLUSIVE-OR that modifies the top-level conses of $list_1$ and $list_2$. NSET-EXCLUSIVE-OR is defined in COMMON.LSP.

For both functions, *test* is a two argument predicate used to determine if elements in $list_1$ match elements in $list_2$. The first argument of *test* is always from $list_1$, the second is always from $list_2$. These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN SET-EXCLUSIVE-OR (LST1 LST2 TEST)
  (NCONC (SET-DIFFERENCE LST1 LST2 TEST) (SET-DIFFERENCE LST2 LST1 TEST)) )

(DEFUN NSET-EXCLUSIVE-OR (LST1 LST2 TEST
    TMP)
  ((ATOM LST1)  LST2)
  ((MEMBER (CAR LST1) LST2 TEST)
    (SETQ TMP (CAR LST1))
    (NSET-EXCLUSIVE-OR (DELETE TMP LST1 TEST) (DELETE TMP LST2 TEST) TEST) )
  (RPLACD LST1 (SET-DIFFERENCE (CDR LST1) LST2 TEST)) )

$ (SET-EXCLUSIVE-OR '(4 2 8 5 6) '(3 6 7 2))
(4 8 5 3 7)
```

SUBSETP [*list*$_1$, *list*$_2$, *test*] Function

If every element of *list*$_1$ is in *list*$_2$, (SUBSETP *list*$_1$ *list*$_2$ *test*) returns T, otherwise it returns NIL.

test is a two argument predicate used to determine if elements in *list*$_1$ match elements in *list*$_2$. The first argument of *test* is always from *list*$_1$, the second is always from *list*$_2$. SUBSETP use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN SUBSETP (LST1 LST2 TEST)
  ((ATOM LST1))
  ((MEMBER (CAR LST1) LST2 TEST)
    (SUBSETP (CDR LST1) LST2 TEST) ) )

$ (SUBSETP '(A B C) '(D A E C B))                    ;(A B C) is a subset
T

$ (SUBSETP '(A B C) '(D A E F B))                    ;(A B C) is not a subset
NIL
```

5.2.8 Association Lists

An **association list** (a-list) is a convenient way to associate indicator keys with data objects. Each element of an a-list is a cons whose car cell points to an **indicator key** and whose cdr cell points to an object. The functions in this subsection are for constructing and using a-lists.

ACONS [*key, object, a-list*] Function
PAIRLIS [*keys, objects, a-list*] Function
COPY-ALIST [*a-list*] Function

(ACONS *key object a-list*) constructs and returns the result of consing the pair (*key* . *object*) onto the head of *a-list*. ACONS is useful for constructing a-lists.

(PAIRLIS *keys objects a-list*) constructs pairs from the elements of *keys* and *objects* and returns the resulting association list appended onto *a-list*. If the lists are of unequal length, the construction stops when the end of the shorter list is reached. *a-list* defaults to NIL.

(COPY-ALIST *a-list*) copies the top-level conses of *a-list* and the top-level cons of each element of *a-list* that is a cons, returning a list that is EQUAL to *a-list*. To prevent unwanted side-effects, use COPY-ALIST instead of COPY-LIST to copy a-lists.

```
(DEFUN ACONS (KEY OBJ ALST)
  (CONS (CONS KEY OBJ) ALST) )

(DEFUN PAIRLIS (KEYS OBJS ALST)
  ((OR (ATOM KEYS) (ATOM OBJS))  ALST)
  (ACONS (CAR KEYS) (CAR OBJS) (PAIRLIS (CDR KEYS) (CDR OBJS) ALST)) )

(DEFUN COPY-ALIST (ALST)
  ((ATOM ALST)  ALST)
  ((ATOM (CAR ALST))
    (CONS (CAR ALST) (COPY-ALIST (CDR ALST))) )
  (ACONS (CAAR ALST) (CDAR ALST) (COPY-ALIST (CDR ALST))) )

$ (ACONS 'EYES 'BLUE '((WEIGHT . 170) (AGE . 39)))                ;Add pair to a-list
((EYES . BLUE) (WEIGHT . 170) (AGE . 39))

$ (PAIRLIS '(X Y Z) '(5 2 8))                                      ;Make an a-list
((X . 5) (Y . 2) (Z . 8))

$ (COPY-ALIST '((A . B) (C . (D E))))                             ;Does not copy (D E)
((A . B) (C . (D E)))
```

ASSOC [*key, a-list, test*] Function
ASSOC-IF [*test, a-list*] Function

(ASSOC *key a-list test*) performs a linear search of *a-list* looking for a pair for which (*test key* (CAR *pair*)) is nonNIL. If the test argument is NIL or not given, ASSOC uses an EQL test. If a pair satisfying the test is found, the pair is returned; otherwise NIL is returned.

(ASSOC-IF *test a-list*) searches *a-list* for a pair for which (*test* (CAR *pair*)) is nonNIL. If a pair satisfying the test is found, the pair is returned; otherwise NIL is returned.

```
(DEFUN ASSOC (KEY ALST TEST)
  ((ATOM ALST)  NIL)
  ((ATOM (CAR ALST))
    (ASSOC KEY (CDR ALST) TEST) )                          ;Skip atoms on a-list
  ((FUNCALL (OR TEST 'EQL) KEY (CAAR ALST))
    (CAR ALST) )                                         ;Return pair if match found
  (ASSOC KEY (CDR ALST) TEST) )                           ;Otherwise, keep searching

(DEFUN ASSOC-IF (TEST ALST)
  ((ATOM ALST)  NIL)
  ((ATOM (CAR ALST))
    (ASSOC-IF TEST (CDR ALST)) )                           ;Skip atoms on a-list
  ((FUNCALL TEST (CAAR ALST))
    (CAR ALST) )                                         ;Return pair if match found
  (ASSOC-IF TEST (CDR ALST)) )                            ;Otherwise, keep searching

$ (SETQ CAPITALS '((USA . WASHINGTON) (FRANCE . PARIS) (JAPAN . TOKYO)))
((USA . WASHINGTON) (FRANCE . PARIS) (JAPAN . TOKYO))              ;Initialize a-list

$ (ASSOC 'FRANCE CAPITALS)                          ;The capital of France is Paris
(FRANCE . PARIS)

$ (ASSOC 'AUSTRALIA CAPITALS)            ;The capital of Australia is not in the a-list
NIL
```

RASSOC [*key, a-list, test*] Function
RASSOC-IF [*test, a-list*] Function

(RASSOC *key a-list test*) performs a linear search of *a-list* looking for a pair for which (*test key* (CDR *pair*)) is nonNIL. If the test argument is NIL or not given, RASSOC uses an EQL test. If a pair satisfying the test is found, the pair is returned; otherwise NIL is returned.

(RASSOC-IF *test a-list*) searches *a-list* for a pair for which (*test* (CDR *pair*)) is nonNIL. If a pair satisfying the test is found, the pair is returned; otherwise NIL is returned. RASSOC is short for reverse associate.

```
(DEFUN RASSOC (KEY ALST TEST)
  ((ATOM ALST)  NIL)
  ((ATOM (CAR ALST))
    (RASSOC KEY (CDR ALST) TEST) )                          ;Skip atoms on a-list
  ((FUNCALL (OR TEST 'EQL) KEY (CDAR ALST))
    (CAR ALST) )                                          ;Return pair if match found
  (RASSOC KEY (CDR ALST) TEST) )                           ;Otherwise, keep searching

(DEFUN RASSOC-IF (TEST ALST)
  ((ATOM ALST)  NIL)
  ((ATOM (CAR ALST))
    (RASSOC-IF TEST (CDR ALST)) )                           ;Skip atoms on a-list
  ((FUNCALL TEST (CDAR ALST))
    (CAR ALST) )                                          ;Return pair if match found
  (RASSOC-IF TEST (CDR ALST)) )                            ;Otherwise, keep searching

$ (SETQ CAPITALS '((USA . WASHINGTON) (FRANCE . PARIS) (JAPAN . TOKYO)))
((USA . WASHINGTON) (FRANCE . PARIS) (JAPAN . TOKYO))                ;Initialize a-list

$ (RASSOC 'PARIS CAPITALS)                            ;Paris is the capital of France
(FRANCE . PARIS)

$ (RASSOC 'CANBERRA CAPITALS)                  ;Canberra is not a capital in the a-list
```

SUBLIS [*a-list, object, test*] Function
NSUBLIS [*a-list, object, test*] Function

(SUBLIS *a-list object test*) returns a copy of *object*, replacing all subexpressions of *object* for which there is a key in *a-list* such that (*test key subexpression*) is nonNIL with the value associated with the key. To prevent unnecessary consing, SUBLIS uses as much of the original object as possible in the result; however in no case is *object* altered.

By modifying the conses of *object*, (NSUBLIS *a-list object test*) replaces all subexpressions of *object* for which there is a key in *a-list* such that (*test key subexpression*) is nonNIL with the value associated with the key. *object* is altered.

These functions use an EQL test if the *test* argument is NIL or not given.

```
(DEFUN SUBLIS (ALST OBJ TEST)
  ((ASSOC OBJ ALST TEST)
    (CDR (ASSOC OBJ ALST TEST)) )
  ((ATOM OBJ)  OBJ)
  (LET ( (LEFT (SUBLIS ALST (CAR OBJ) TEST))
         (RIGHT (SUBLIS ALST (CDR OBJ) TEST)) )
    ((AND (EQ LEFT (CAR OBJ)) (EQ RIGHT (CDR OBJ)))  OBJ)
    (CONS LEFT RIGHT) ) )

(DEFUN NSUBLIS (ALST OBJ TEST)
  ((ASSOC OBJ ALST TEST)
    (CDR (ASSOC OBJ ALST TEST)) )
  ((ATOM OBJ)  OBJ)
  (RPLACA OBJ (NSUBLIS ALST (CAR OBJ) TEST))
  (RPLACD OBJ (NSUBLIS ALST (CDR OBJ) TEST)) )

$ (SETQ EXPN '(* (+ X Y) (- X 2) Y))                    ;Initialize EXPN
(* (+ X Y) (- X 2) Y)

$ (SETQ VALS '((X . 5) (Y . 3)))                        ;Initialize VALS
((X . 5) (Y . 3))

$ (SUBLIS VALS EXPN)                                  ;Substitute for X and Y
(* (+ 5 3) (- 5 2) 3)

$ EXPN                                                  ;EXPN unchanged
(* (+ X Y) (- X 2) Y)

$ (NSUBLIS VALS EXPN)                                 ;Substitute for X and Y
(* (+ 5 3) (- 5 2) 3)

$ EXPN                                                  ;Newly modified EXPN
(* (+ 5 3) (- 5 2) 3)
```

5.2.9 Sorting

The functions in this subsection are for sorting and merging lists.

SPLIT [*list*] Function

(SPLIT *list*) destructively separates *list* into two lists by replacing with NIL the cdr of the top-level cons halfway down *list*. SPLIT returns the tail of *list* beginning where the split was made.

If *list* has an even number of elements, the head and tail of the split will have the same number of elements. If *list* has an odd number of elements, the head will have one more element than the tail.

```
(DEFUN SPLIT (LST)
  ((ATOM LST)  LST)
  (split-aux LST (CDR LST)) )

(DEFUN split-aux (HEAD TAIL)
  ((ATOM TAIL)
    (PROG1 (CDR HEAD) (RPLACD HEAD NIL)) )
  (POP TAIL)
  ((ATOM TAIL)
    (PROG1 (CDR HEAD) (RPLACD HEAD NIL)) )
  (split-aux (CDR HEAD) (CDR TAIL)) )

$ (SETQ NAMES '(TOM SUE JOE PAT SAM))                    ;Set NAMES
(TOM SUE JOE PAT SAM)

$ (SPLIT NAMES)                                  ;SPLIT returns last half
(PAT SAM)

$ NAMES                                          ;NAMES is now first half
(TOM SUE JOE)
```

MERGE [$list_1$, $list_2$, *test*] Function

If *test* is a two argument ordering predicate, (MERGE $list_1$ $list_2$ *test*) destructively merges the elements of $list_1$ and $list_2$ based on *test* and returns the result. (*test* $object_1$ $object_2$) should return NIL if and only if $object_1$ is ordered after $object_2$ or is equal to $object_2$.

MERGE uses a **stable list merge** in the sense that if an element of $list_1$ is equal to an element of $list_2$, the element of $list_1$ will precede that of $list_2$ in the result.

Since MERGE modifies the top-level conses of the lists, it does not require any additional conses. If the lists should not be destroyed, call MERGE with copies of the lists made by the function COPY-LIST.

```
(DEFUN MERGE (LST1 LST2 TEST)
  ((ATOM LST1)  LST2)
  ((ATOM LST2)  LST1)
  ((FUNCALL TEST (CAR LST2) (CAR LST1))
    (RPLACD LST2 (MERGE LST1 (CDR LST2) TEST)) )
  (RPLACD LST1 (MERGE (CDR LST1) LST2 TEST)) )

$ (MERGE '(2 4 6 8 10) '(2 3 5 7 11) '<)                    ;Merge lists of integers
(2 2 3 4 5 6 7 8 10 11)

$ (MERGE '(APPLE ORANGE PEAR) '(BANANA GRAPE PLUM) 'STRING<)   ;Merge lists of symbols
(APPLE BANANA GRAPE ORANGE PEAR PLUM)
```

SORT [*list*, *test*] — Function
STABLE-SORT [*list*, *test*] — Library function

If *test* is a two argument ordering predicate, (SORT *list test*) destructively sorts the elements of *list* based on *test* and returns the result. (*test* $object_1$ $object_2$) should return NIL if and only if $object_1$ is ordered after $object_2$ or is equal to $object_2$.

SORT uses a **list merge sort** requiring time proportional to $n \log(n)$ where n is the length of *list* (see Knuth, The Art of Computer Programming, Volume 3, Addison-Wesley, 1973). The sort is a **stable sort** in the sense that elements of *list* which *test* finds equal to each other will stay in the same order in the result.

Since SORT modifies the top-level conses of *list*, it does not require any additional conses. If *list* should not be destroyed, call SORT with a copy of *list* made by COPY-LIST.

The definition of STABLE-SORT is *identical* to that of SORT and is included in COMMON.LSP solely for compatibility.

```
(DEFUN SORT (LST TEST)
  ((NULL LST) NIL)
  ((NULL (CDR LST)) LST)
  (SETQ LST (CONS (SORT (SPLIT LST) TEST) (SORT LST TEST)))
  (MERGE (CDR LST) (CAR LST) TEST) )
```

```
$ (SORT '(5 2 3/4 7 0 3 -4) '<)                          ;Sort list of numbers
(-4 0 0.75 2 3 5 7)

$ (SORT '(SMITH JONES MILLER COX SCHMIDT) 'STRING<)       ;Sort list of names
(COX JONES MILLER SCHMIDT SMITH)
```

5.3 Symbols

A **symbol** is a primitive data object having a value, a property list, a print name, and a function definition (see Section 4.1.1).

5.3.1 Value

The primitives in this subsection are for accessing and modifying the value of symbols by modifying their value cell. They can be used to assign values to program variables. If a variable is not a **local variable** (i.e. not a formal parameter of the currently executing function), assignments to the variable will remain in effect even after the function is exited. As with the modifier functions (see Section 5.2.3), this phenomenon is called a **side-effect** of the function.

SYMBOL-VALUE [*symbol*] — Library function
BOUNDP [*symbol*] — Library function

If *symbol* is a symbol, (SYMBOL-VALUE *symbol*) returns *symbol*'s value (i.e. the data object pointed to by *symbol*'s value cell). Otherwise, SYMBOL-VALUE returns NIL.

If *symbol* is NIL or a symbol having a value other than itself, (BOUNDP *symbol*) returns T, otherwise it returns NIL.

Since the **muLISP** interpreter automatically evaluates symbols, explicit use of SYMBOL-VALUE and BOUNDP are rarely needed. They are defined in COMMON.LSP.

```
(DEFUN SYMBOL-VALUE (SYM)
  ((SYMBOLP SYM)
    (CAR SYM) ) )

(DEFUN BOUNDP (SYM)
  ((SYMBOLP SYM)
    ((NULL SYM))
    (NEQ SYM (CAR SYM)) ) )

$ (BOUNDP 'NUM)                                    ;Test if NUM is bound
NIL

$ (SETQ NUM 5)                                      ;Assign NUM a value
5

$ (BOUNDP 'NUM)                                    ;Test if NUM is bound
T

$ (SYMBOL-VALUE 'NUM)                               ;The value of NUM
5
```

SET [*symbol*, *object*] Function

If *symbol* is a nonNIL symbol, (SET *symbol object*) replaces the value cell of *symbol* with a pointer to *object* and returns *object*. If *symbol* is NIL, SET returns object without changing the value of NIL. (Note that the value of the symbol T is *not* protected.) If *symbol* is not a symbol, a Nonsymbolic Argument error break occurs.

RPLACA and SET are identical as far as their effect on their first argument. However, RPLACA returns its first argument, whereas SET returns its second argument. As a general rule, SET should be used if the first argument is a symbol; RPLACA should be used if the first argument is a cons.

```
(DEFUN SET (SYM OBJ)
  ((SYMBOLP SYM)
    ((NULL SYM)  OBJ)
    (RPLACA SYM OBJ)
    OBJ )
  (BREAK (LIST 'SET SYM OBJ) "Nonsymbolic Argument") )

$ (SET 'FOO '(A B C))                                  ;Assign FOO a value
(A B C)

$ FOO                                                  ;New value of FOO
(A B C)

$ (SET 'PET 'DOG)                                      ;Assign PET a value
DOG

$ (SET PET 'ROVER)                          ;Assign the value of PET (i.e. DOG)
ROVER                                                                  ;a value

$ DOG                                                  ;The value of DOG
ROVER

$ PET                                                  ;The value of PET
DOG
```

SETQ [$symbol_1$, $form_1$, ..., $symbol_n$, $form_n$] Special form

(SETQ *symbol form*) evaluates *form*, sets the value of *symbol* to result, and returns the result. If SETQ is given more than two arguments, the form evaluation and assignments are made sequentially. If there are an odd number of arguments, the last symbol is set to NIL. SETQ returns the value assigned to the last symbol.

Note that SETQ is a special form and that its first argument is *not* evaluated. If the first argument is not a symbol, a Nonsymbolic Argument error break occurs.

SETQ is short for SET quote. Although SET is more primitive, SETQ is used far more often than SET because assignments are usually made to a variable rather than to the value of a variable.

```
(DEFMACRO SETQ (SYM OBJ)
  (LIST 'SET (LIST 'QUOTE SYM) OBJ) )

$ (SETQ FOO '(A B C))                                   ;Assign FOO to list
(A B C)

$ FOO                                                   ;Current value of FOO
(A B C)

$ (SETQ FOO (CDR FOO))                                  ;Assign FOO to cdr of list
(B C)

$ FOO                                                   ;Current value of FOO
(B C)

$ (SETQ NUM 5)                                          ;Assign NUM to 5
5

$ (SETQ NUM (+ 3 4) SQR (* NUM NUM))                    ;Assigns NUM to 7 and SQR to 7*7
49

$ NUM                                                   ;The value of NUM is now 7
7

$ SQR                                                   ;The value of SQR is 49 (not 25)
49
```

PSETQ [$symbol_1$, $form_1$, ..., $symbol_n$, $form_n$] Special form

(PSETQ $symbol_1$ $form_1$ ... $symbol_n$ $form_n$) evaluates $form_1$ through $form_n$, simultaneously sets the value of $symbol_1$ through $symbol_n$ to the results, and returns NIL. Note that PSETQ is a special form and that the symbols are *not* evaluated. If any $symbol_i$ is not a symbol, PSETQ generates a Nonsymbolic Argument error break.

The effect of PSETQ on its arguments is identical to that of SETQ except that all the forms are evaluated before making any assignments to symbols. PSETQ is short for parallel SETQ.

As the last example shows, PSETQ can be used to efficiently interchange the value of two variables without having to introduce an intermediary.

```
$ (SETQ NUM 5)                                          ;Assign NUM to 5
5

$ (PSETQ NUM (+ 3 4) SQR (* NUM NUM))                ;Assigns NUM to 7 and SQR to 5*5
NIL

$ NUM                                                   ;The value of NUM is now 7
7

$ SQR                                                ;The value of SQR is 25 (not 49)
25

$ (PSETQ NUM SQR SQR NUM)                               ;Swap value of NUM & SQR
NIL

$ NUM                                                   ;Value of NUM is now 25
25

$ SQR                                                   ;Value of SQR is now 7
7
```

SETF [*place*$_1$, *form*$_1$, ..., *place*$_n$, *form*$_n$] Library macro

If *place* is a valid place form, (SETF *place form*) evaluates *form*, stores the result in *place*, and returns the result. If SETF is given more than two arguments, the form evaluation and assignments are made sequentially. If there are an odd number of arguments, the last place form is set to NIL. SETF returns the value assigned to the last place form.

If the first argument of SETF is not a valid place form an Invalid Place Form error break occurs. The following are the valid **place forms**:

- symbols,
- lists beginning with one of these selector functions: CAR, CDR, CAAR, CADR, ..., CDDDDR, FIRST, SECOND, THIRD, ..., TENTH, REST, NTH, GET, MACRO-FUNCTION, SYMBOL-VALUE, SYMBOL-PLIST, and SYMBOL-FUNCTION, and
- macro calls that expand into valid place forms.

SETF is defined in COMMON.LSP.

```
(DEFMACRO SETF LST
  ((NULL LST)  NIL)
  ((NULL (CDDR LST))
    ((ATOM (CAR LST))
      (CONS 'SETQ LST) )
    (MAKE-SET-FORM (CAR LST) (CADR LST)) )
  (CONS 'PROGN (setf-aux LST)) )

(DEFUN setf-aux (LST)
  ((NULL LST)  NIL)
  ((ATOM (CAR LST))
    (CONS (LIST 'SETQ (CAR LST) (CADR LST)) (setf-aux (CDDR LST))) )
  (CONS (MAKE-SET-FORM (CAR LST) (CADR LST)) (setf-aux (CDDR LST))) )

$ (SETQ FOO '(A B C))                                    ;Initialize FOO
(A B C)

$ (SETF (SECOND FOO) 5)                            ;2nd element of FOO set to 5
5

$ FOO                                                    ;Newly modified FOO
(A 5 C)
```

POP [*symbol*] Special form

(POP *symbol*) returns the "top" (i.e car) of the **stack** (i.e. list) named *symbol* and updates the value of *symbol* to the rest (i.e. cdr) of the stack. If *symbol* is not a symbol, a Nonsymbolic Argument error break occurs. If the value of *symbol* is not a cons, POP returns NIL.

This special form provides the LISP analog of the machine language pop instruction used to fetch information from the top of a stack.

```
(DEFMACRO POP (SYM)
  (LIST 'PROG1
        (LIST 'CAR SYM)
        (LIST 'SETQ SYM (LIST 'CDR SYM))) )

$ (SETQ STACK-LIST '(A B C D E F))                    ;Initialize stack
(A B C D E F)

$ (POP STACK-LIST)                                    ;Pop A off stack
A

$ (POP STACK-LIST)                                    ;Pop B off stack
B

$ (POP STACK-LIST)                                    ;Pop C off stack
C

$ STACK-LIST                                          ;Shortened stack
(D E F)
```

PUSH [*form, symbol*] Special form
PUSHNEW [*form, symbol, test*] Library macro

(PUSH *form symbol*) evaluates *form* and pushes (i.e. conses) the result onto the "stack" named *symbol* and updates the value of *symbol* to the larger stack. PUSH returns the enlarged stack (i.e. list). If *symbol* is not a symbol, a Nonsymbolic Argument error break occurs. This special form provides the LISP analog of the machine language push instruction used to save information on top of a stack.

If the value of *form* is *not* a member of the "stack" named *symbol*, (PUSHNEW *form symbol test*) pushes (i.e. conses) the value of *form* onto the stack, updates the value of *symbol* to the larger stack, and returns the enlarged stack. Otherwise, PUSHNEW simply returns the stack. If *symbol* is not a symbol, a Nonsymbolic Argument error break occurs. PUSHNEW is defined in COMMON.LSP.

```
(DEFMACRO PUSH (OBJ SYM)
  (LIST 'SETQ SYM (LIST 'CONS OBJ SYM)) )

(DEFMACRO PUSHNEW (OBJ SYM TEST)
  (LIST 'SETQ SYM (LIST 'ADJOIN OBJ SYM TEST)) )

$ (SETQ STACK-LIST NIL)                                  ;Empty stack
NIL

$ (PUSH 'A STACK-LIST)                                   ;Push A on stack
(A)

$ (PUSH 'B STACK-LIST)                                   ;Push B on stack
(B A)

$ (PUSHNEW 'C STACK-LIST)                                ;Push C on stack
(C B A)

$ (PUSHNEW 'B STACK-LIST)                                ;B already on stack
(C B A)

$ STACK-LIST                                             ;Current stack
(C B A)
```

5.3.2 Property List

The functions in this subsection are for managing the properties and flags on the property lists of symbols. They aid the construction of dynamically extensible data bases from which information can be rapidly extracted or modified.

A symbol's **property list** is an a-list (see Section 5.2.8) used to store the properties and flags associated with the symbol. Since flags are atoms and properties are conses, they are easily distinguishable from one another on property lists.

PUT [*symbol, key, object*] Function

(PUT *symbol key object*) places *object* on the property list of *symbol* under the indicator *key*.

Since PUT uses an EQUAL test when searching the property list for a match to *key*, property keys need not be atomic. Note that any previous value under the indicator *key* is automatically discarded.

Since PUT uses a RPLACD operation to add a property value to a property list, care should used when sharing property lists.

```
(DEFUN PUT (SYM KEY OBJ)
  ((OR (NULL SYM) (NOT (SYMBOLP SYM))) NIL)
  ((ASSOC KEY (CDR SYM))
    (RPLACD (ASSOC KEY (CDR SYM)) OBJ)
    OBJ )
  (RPLACD SYM (ACONS KEY OBJ (CDR SYM)))
  OBJ )

$ (PUT 'USA 'POPULATION 250000000)                    ;Store population of USA
250000000

$ (PUT 'FRANCE 'POPULATION 57000000)                 ;Store population of FRANCE
57000000

$ (PUT 'JAPAN 'POPULATION 125000000)                  ;Store population of JAPAN
125000000

$ (PUT 'USA 'AREA 3536855)                  ;Store area of USA (in square miles)
3536855

$ (PUT 'FRANCE 'AREA 211208)                                ;Store area of FRANCE
211208

$ (PUT 'JAPAN 'AREA 142726)                                  ;Store area of JAPAN
142726
```

GET [*symbol, key, default*] — Function
GETF [*plist, key, default*] — Library function
GET-PROPERTIES [*plist, key-list*] — Library function
SYMBOL-PLIST [*symbol*] — Library function

(GET *symbol key default*) returns the property value associated with *symbol* under the indicator *key*. If no such value exists, GET returns *default* or NIL if no default is given.

(GETF *plist key default*) returns the property value on the property list *plist* under the indicator *key*. If no such value exists, GETF returns *default* or NIL if no default is given. It is defined in COMMON.LSP.

(GET-PROPERTIES *plist key-list*) returns the first property (*key* . *value*) associated with any indicator in *key-list* on the property list *plist*. It is defined in COMMON.LSP.

If *symbol* is a symbol, (SYMBOL-PLIST *symbol*) returns *symbol*'s property list (i.e. the data object pointed to by *symbol*'s property list cell). Otherwise, SYMBOL-PLIST returns NIL. It is defined in COMMON.LSP.

To run the following examples, first run the examples for PUT on the previous page.

```
(DEFUN GET (SYM KEY)
  (CDR (ASSOC KEY (CDR SYM))) )

(DEFUN GET-PROPERTIES (LST1 LST2)
  (ASSOC-IF '(LAMBDA (PAIR) (MEMBER (CAR PAIR) LST2)) LST1) )

(DEFUN SYMBOL-PLIST (SYM)
  ((SYMBOLP SYM)
    (CDR SYM) ) )

$ (SYMBOL-PLIST 'USA)                                    ;Property list of USA
((AREA . 3536855) (POPULATION . 250000000))

$ (GET 'USA 'POPULATION)                                  ;Population of USA
250000000

$ (GET 'AUSTRALIA 'POPULATION)                ;Population of Australia is not stored
NIL

$ (DEFUN POPULATION-DENSITY (COUNTRY)          ;Define a population density function
    (/ (GET COUNTRY 'POPULATION)
       (GET COUNTRY 'AREA)) )
POPULATION-DENSITY

$ (POPULATION-DENSITY 'FRANCE)                       ;French persons per square mile
269.8761408

$ (POPULATION-DENSITY 'JAPAN)                       ;Japanese persons per square mile
875.8039867
```

REMPROP [*symbol*, *key*] — Function
REMF [*place*, *key*] — Library macro

(REMPROP *symbol key*) removes from *symbol*'s property list the property associated with *key*. REMPROP returns the old property value if one was found, otherwise it returns NIL.

REMF is the generalized variable form of REMPROP (see SETF in Section 5.3.1).

Since REMPROP and REMF use a destructive RPLACD operation to remove an element from a property list, care should used when sharing property lists.

To run the following examples, first run the examples for PUT given on a previous page.

```
(DEFUN REMPROP (SYM KEY)
  ((ATOM (CDR SYM)) NIL)
  ((EQUAL (CAADR SYM) KEY)
    (SETQ KEY (CDADR SYM))
    (RPLACD SYM (CDDR SYM))
    KEY )
  (REMPROP (CDR SYM) KEY) )

$ (REMPROP 'USA 'AREA)                                  ;Remove property
3536855

$ (GET 'FRANCE 'AREA)                                   ;France's area
211208

$ (GET 'USA 'AREA)                                      ;USA's area is unknown
NIL
```

FLAG [*symbol, flag*] Function
FLAGP [*symbol, flag*] Function
REMFLAG [*symbol, flag*] Function

If *symbol* is a nonNIL symbol, (FLAG *symbol flag*) adjoins *flag* to *symbol*'s property list, and returns *flag*. Otherwise, it returns NIL.

If *flag* is a member of *symbol*'s property list, (FLAGP *symbol flag*) returns a nonNIL value (specifically the property list beginning with *flag*). Otherwise, it returns NIL.

If *flag* is a member of *symbol*'s property list, (REMFLAG *symbol flag*) removes *flag* from the property list, and returns T. Otherwise, it returns NIL.

Flags must be atomic data objects (normally symbols) so the property list functions can distinguish them from properties on property lists.

```
(DEFUN FLAG (SYM FLAG)
  ((OR (NULL SYM) (NOT (SYMBOLP SYM))) NIL)
  (RPLACD SYM (ADJOIN FLAG (CDR SYM) 'EQUAL))
  FLAG )

(DEFUN FLAGP (SYM FLAG)
  (MEMBER FLAG (CDR SYM) 'EQUAL) )

(DEFUN REMFLAG (SYM FLAG)
  ((ATOM (CDR SYM)) NIL)
  ((EQUAL FLAG (CADR SYM))
    (RPLACD SYM (CDDR SYM))
    T )
  (REMFLAG (CDR SYM) FLAG) )

$ (FLAG 'JAN 'MALE)                                  ;Flag Jan to be male
MALE

$ (FLAGP 'JAN 'MALE)                                 ;Jan is flagged as male
(MALE)

$ (FLAGP 'SAM 'MALE)                                 ;Sam is not flagged as male
NIL

$ (FLAG 'JAN 'TALL)                                  ;Flag Jan to be tall
TALL

$ (REMFLAG 'JAN 'MALE)                               ;Remove Jan's male flag
T

$ (FLAGP 'JAN 'MALE)                                 ;Jan is no longer flagged as male
NIL
```

5.3.3 Definition

The primitives in this subsection are for managing the function definition cell associated with each symbol. When a function is defined, the definition is automatically transformed into an extremely dense form called **pseudo-code**. The inverse process of transforming a definition into a linked list is also automatic.

GETD [*symbol*, *flag*]	Function
SPECIAL	Keyword
LAMBDA	Keyword
NLAMBDA	Keyword
MACRO	Keyword

If *symbol* is a user-defined function, (GETD *symbol*) decompiles and returns the definition of *symbol* in linked list form. If *symbol* is a pre-defined function or special form, (GETD *symbol*) returns the physical memory address (in the 8086 code segment) of the **machine language** routine. GETD returns the offset of the routine in the code segment. If *symbol* is not a function or special form, (GETD *symbol*) returns NIL.

If *flag* is nonNIL, (GETD *symbol flag*) returns SPECIAL if *symbol* is a special form, LAMBDA if *symbol* is an eval function, NLAMBDA if *symbol* is a no-eval function, MACRO if *symbol* is a macro, or NIL if *symbol* is undefined. See Section 5.7.1 for an explanation of the different types of functions.

If a program only needs to know if *symbol* is a defined function, it is faster to call GETD with *flag* nonNIL since no consing is required.

```
(DEFUN GETD (SYM FLAG)
  ((NOT (SYMBOLP SYM)) NIL)
  ((undefined SYM) NIL)
  ((NULL FLAG)
    ((machine language routine SYM)
      (the memory address of the routine) )
    (the S-expression equivalent of the pseudo-code definition of SYM) )
  ((if SYM is a special form) 'SPECIAL)
  ((if SYM is an eval function) 'LAMBDA)
  ((if SYM is a no-eval function) 'NLAMBDA)
  ((if SYM is a macro) 'MACRO) )

$ (GETD 'CONS)                                    ;Address of CONS routine
2673                                     ;result depends on version of muLISP

$ (GETD 'CONS T)                                  ;CONS is an eval function
LAMBDA
```

SYMBOL-FUNCTION [*symbol*] Library function

If *symbol* is a defined function, (SYMBOL-FUNCTION *symbol*) returns *symbol*'s definition (i.e. the decompiled pseudo-code or the address of the machine code pointed to by *symbol*'s function definition cell). If *symbol* is undefined, an Undefined Function error break occurs.

SYMBOL-FUNCTION is defined in COMMON.LSP.

```
(DEFUN SYMBOL-FUNCTION (SYM)
  ((SYMBOLP SYM)
    ((GETD SYM T)
      (GETD SYM) )
    (BREAK (LIST 'SYMBOL-FUNCTION SYM) "Undefined function") ) )

$ (SYMBOL-FUNCTION 'FBOUNDP)                                  ;FBOUNDP's definition
(LAMBDA (SYM) (GETD SYM T))
```

EVAL-FUNCTION-P [*symbol*] — Library function
NO-EVAL-FUNCTION-P [*symbol*] — Library function
MACRO-P [*symbol*] — Library function
SPECIAL-FORM-P [*symbol*] — Library function
FBOUNDP [*symbol*] — Library function

These predicates return T if *symbol* is a function of the given type, otherwise they return NIL. EVAL-FUNCTION-P tests for eval functions. NO-EVAL-FUNCTION-P tests for no-eval functions. MACRO-P tests for macros. SPECIAL-FORM-P tests for special forms. FBOUNDP tests for functions of any of these types.

These functions are defined in COMMON.LSP.

```
(DEFUN EVAL-FUNCTION-P (SYM)
  (EQ (GETD SYM T) 'LAMBDA) )

(DEFUN NO-EVAL-FUNCTION-P (SYM)
  (EQ (GETD SYM T) 'NLAMBDA) )

(DEFUN MACRO-P (SYM)
  (EQ (GETD SYM T) 'MACRO) )

(DEFUN SPECIAL-FORM-P (SYM)
  (EQ (GETD SYM T) 'SPECIAL) )

(DEFUN FBOUNDP (SYM)
  (GETD SYM T) )

$ (EVAL-FUNCTION-P 'APPEND)                    ;APPEND is a function
T

$ (MACRO-P 'DO)                                          ;DO is a macro
T

$ (SPECIAL-FORM-P 'IF)                         ;IF is a special form
T

$ (FBOUNDP NIL)                          ;NIL is not a defined function
NIL
```

PUTD [*symbol, definition*] Function

If *definition* is a lambda-list, (PUTD *symbol definition*) compiles *definition* into pseudo-code, modifies *symbol*'s function definition cell to point to the pseudo-code, and returns *symbol*. Quoted nonatomic objects (i.e. **constants**) within a function definition are not compiled but remain stored as a linked list. Therefore, the interpreter does not have to decompile such constants each time they are encountered, and constants can contain conses with nonNIL atomic cdr cells.

If *definition* is zero or a positive integer less than 65536, (PUTD *symbol definition*) modifies *symbol*'s function cell to *definition*, and returns *symbol*. This makes it possible to link to user-defined machine language routines (see Section 9.4).

The variable *CONDENSE* controls the condensing of pseudo-code while compiling. The variable MACROEXPAND controls the expansion of macros while compiling pseudo-code (see Section 5.7.5).

```
(DEFUN PUTD (SYM DEFN)
  ((OR (NOT (SYMBOLP SYM)) (SYMBOLP DEFN)) NIL)
  ((NUMBERP DEFN)
    ((AND (INTEGERP DEFN) (PLUSP DEFN))
      (store DEFN in SYM's function cell)
      SYM ) )
  (compile DEFN into pseudo-code)
  (store a pointer to the pseudo-code in SYM's
      function cell)
  SYM )

$ (PUTD 'SQUARE                                          ;Define a function
        '(LAMBDA (NUM) (* NUM NUM)) )
SQUARE

$ (SQUARE 5)                                             ;Call the function
25
```

MOVD [$symbol_1$, $symbol_2$] Function

(MOVD $symbol_1$ $symbol_2$) copies $symbol_1$'s function definition cell to $symbol_2$'s and returns $symbol_2$. In cases where MOVD is sufficient, it should be used instead of a combination of GETD and PUTD since no pseudo-code compilation is required.

The examples show how MOVD can be used to make more mnemonic symbols for some of the primitive LISP selector functions without sacrificing execution efficiency.

```
(DEFUN MOVD (SYM1 SYM2)
  ((AND (SYMBOLP SYM1) (SYMBOLP SYM2))
    (replace the contents of SYM2's function
        cell with the contents of SYM1's function cell)
    SYM2 ) )

$ (MOVD 'CAR 'FIRST)                                    ;Define FIRST
FIRST

$ (MOVD 'CDR 'REST)                                     ;Define REST
REST

$ (MOVD 'CADR 'SECOND)                                  ;Define SECOND
SECOND

$ (MOVD 'CADDR 'THIRD)                                  ;Define THIRD
THIRD

$ (MOVD 'CADDDR 'FOURTH)                                ;Define FOURTH
FOURTH

$ (FIRST '(A B C D))                                    ;Call FIRST
A
```

REMD [*symbol*] Function
FMAKUNBOUND [*symbol*] Library function

(REMD *symbol*) sets *symbol*'s function definition cell to point to a machine language routine that generates an Undefined Function error break when *symbol* is called as a function (see Section 5.10.3). REMD returns *symbol*.

The definition of FMAKUNBOUND is *identical* to that of REMD and is included in COMMON.LSP solely for compatibility.

The primary use of REMD is to make symbols be undefined so they can be collected by the garbage collector (see Section 4.2.2).

```
(DEFUN REMD (SYM)
  ((SYMBOLP SYM)
    (MOVD 'UNDEFINED SYM) ) )

$ (DEFUN RCONS (SYM1 SYM2) (CONS SYM2 SYM1))                    ;Define RCONS function
RCONS

$ (RCONS 'DOG 'CAT)                                                        ;Call RCONS
(CAT . DOG)

$ (REMD 'RCONS)                                              ;Remove RCONS's definition
RCONS

$ (GETD 'RCONS)                                                  ;RCONS now undefined
NIL

$ (RCONS 'DOG 'CAT)                                                        ;Call RCONS
Undefined Function Break: (RCONS 'DOG 'CAT)
```

DEFUN [*symbol*, *arglist*, $form_1$, ..., $form_n$] Special form
DEFUN [*symbol*, (*ftype arglist* $form_1$ ... $form_n$)] Special form
DEFUN [*symbol*, *address*] Special form

The function defining special form DEFUN can be called in either of three ways: The first way is for defining eval functions. The second way is for defining no-eval functions and macros. The third way is for linking to user-defined machine language routines.

(DEFUN *symbol arglist* $form_1$ ... $form_n$) compiles *arglist* and the forms into pseudo-code assuming an eval or LAMBDA function type. DEFUN then modifies *symbol*'s function definition cell to point to the resulting pseudo-code, and returns *symbol*.

If *ftype* is the symbol LAMBDA, NLAMBDA, or MACRO, the call (DEFUN *symbol* (*ftype arglist* $form_1$ ... $form_n$)) compiles *arglist* and the forms into pseudo-code and makes *symbol* an eval function, a no-eval function, or a macro depending on *ftype*. DEFUN then modifies *symbol*'s function cell to point to the resulting pseudo-code, and returns *symbol*.

If *address* is zero or a positive integer less than 65536, (DEFUN *symbol address*) modifies *symbol*'s function cell to *address*, and returns *symbol*. See Section 9.4 for details on linking to machine language routines.

Although it is defined in terms of the more primitive function PUTD, DEFUN is usually a more convenient means of defining functions. Examples of DEFUN appear throughout this chapter.

```
(DEFMACRO DEFUN (SYM . BODY)
  ((ATOM BODY) NIL)
  ((NUMBERP (CAR BODY))
    (LIST 'PUTD (LIST 'QUOTE SYM) (CAR BODY)) )
  ((MEMBER (CAAR BODY) '(LAMBDA NLAMBDA MACRO))
    (LIST 'PUTD (LIST 'QUOTE SYM) (CONS 'QUOTE BODY)) )
  (LIST 'PUTD (LIST 'QUOTE SYM) (LIST 'QUOTE (CONS 'LAMBDA BODY))) )
```

DEFMACRO [*symbol*, *arglist*, $form_1$, ..., $form_n$] Special form

(DEFMACRO *symbol arglist* $form_1$ $form_2$ ... $form_n$) destructures and compiles *arglist* and all the forms into pseudo-code, modifies *symbol*'s function definition cell to point to the resulting pseudo-code, thereby making *symbol* a macro. DEFMACRO returns *symbol*.

Unlike DEFUN, DEFMACRO provides for the **destructuring** of *arglist*. This facility means that arguments are assigned to the symbols in *arglist* based on the linked list structure of *arglist*. If *arglist* is a symbol, the actual arguments in the macro call are assigned to the symbol. Otherwise the car of the arguments are recursively assigned to the car of *arglist* and the cdr of the arguments are assigned to the cdr of *arglist*.

The formal definition of DEFUN on the previous page is an example of the use of DEFMACRO's destructuring capability.

```
(PUTD DEFMACRO '(MACRO BODY
  (POP BODY)
  (LIST 'PUTD
        (LIST 'QUOTE (CAR BODY))
        (LIST 'QUOTE
              (LIST 'MACRO
                    'BODY
                    (LIST (LIST* 'LAMBDA
                                 (Leaves (CADR BODY)) (CDDR BODY))
                                 (Destructure (CADR BODY) '(CDR BODY))))))) )

(DEFUN Leaves (TREE)
  ((ATOM TREE)
    (LIST TREE) )
  ((NULL (CDR TREE))
    (Leaves (CAR TREE)) )
  (NCONC (Leaves (CAR TREE)) (Leaves (CDR TREE))) )

(DEFUN Destructure (TREE FORM)
  ((ATOM TREE)
    (LIST FORM) )
  ((NULL (CDR TREE))
    (Destructure (CAR TREE) (LIST 'CAR FORM)) )
  (NCONC (Destructure (CAR TREE) (LIST 'CAR FORM))
         (Destructure (CDR TREE) (LIST 'CDR FORM))) )
```

CONDENSE: NIL Control variable

As a function definition is being compiled, a search is made to find existing pseudo-code equivalent to each subexpression in the definition. If such equivalent pseudo-code is found, it is used instead of creating more pseudo-code, thereby saving memory. This process is called **condensing** pseudo-code.

If *CONDENSE* is NIL, the condensing search is limited to the definition's own pseudo-code. Although it increases the pseudo-code space memory usage, limiting the condensing search space greatly reduces the time required to load large source files.

If *CONDENSE* is nonNIL, the condensing search spans the entire existing pseudo-code. This **full condensing** reduces the amount of memory required to store a given set of function definitions at a cost of increased time required to load large source files. Using a small set of generic names (e.g. NUM1, NUM2, LST, OBJ, ATM, etc.) in the formal argument lists of user defined functions can significantly increase the code density benefits derived from full condensing.

When a user-defined function is redefined using PUTD, MOVD, or REMD, the pseudo-code used to store the definition is automatically reclaimed if full condensing has *not* been performed (i.e. if *CONDENSE* has been NIL) since the function was defined. Thus, it is wise to have *CONDENSE* NIL during interactive program development when functions are frequently redefined.

5.4 Numbers

The functions in this section implement the basic mathematical operations for both integer and rational numbers. The user can select either exact or approximate **rational arithmetic**. For exact rational arithmetic the size of integers, numerators, and denominators is "limited" to thousands of digits. For approximate rational (also called **floating slash**) arithmetic the precision used can be adjusted to any desired degree of accuracy within this limitation.

If a function requiring integer arguments is called with a noninteger argument, a Noninteger Argument error break occurs. If a function requiring numeric (i.e. integer or rational) arguments is called with a nonnumeric argument, a Nonnumeric Argument error break occurs. If a function is called with too few arguments, an Insufficient Arguments error break occurs. If an attempt is made to divide by zero, a Zero Divide error break occurs.

When an **error break** occurs, a diagnostic message, the errant function call, and the break option prompt are displayed on the console. The break allows the user to determine the cause of the error and how to proceed (see Section 4.3).

The control variables *READ-BASE*, *PRINT-BASE*, and *PRINT-POINT* affect how numbers are expressed for input and output (see Section 5.8). Numerical results in the examples in this section are shown as they would appear with these variables set to their default value.

Numbers can be entered using either **decimal notation** (e.g. -0.75) or **rational notation** (e.g. -3/4). Note that there should be no spaces around a dot, slash, or minus sign. Also at least one decimal digit should precede a dot.

Regardless of how a number is entered, internally it is stored as the reduced ratio of two integers. This eliminates the many problems that can result from having a nonunique means of storing numbers. For example, in **muLISP** the form

```
(= -0.75 -3/4 -6/8)
```

returns T, as it should.

5.4.1 Precision

The control variable in this subsection selects either exact or approximate mode rational arithmetic. The functions determine the precision and underflow threshold used for numerical operations.

The rounding algorithm used for the approximate rational arithmetic was inspired by the work of Matula and Kornerup (see D. W. Matula and P. Kornerup, *Finite Precision Rational Arithmetic: Slash Number Systems*, **IEEE Transactions of Computers**, C-34, No. 1, January, 1985, pp. 3-18).

PRECISION [*n*] — Function
EXACT-MODE: NIL — Control variable

If *n* is a positive integer, (PRECISION *n*) sets the precision to *n* and returns the previous precision. If *n* is omitted or not a positive integer, (PRECISION *n*) returns the current precision without changing it. The initial precision setting is 1.

If *EXACT-MODE* is nonNIL, the rational numerical functions (+, -, *, /, GCD, and LCM) use exact rational arithmetic and do *not* round results.

If *EXACT-MODE* is NIL and the current precision is *n*, the numerical functions use approximate rational arithmetic to round results as follows: a) if the exact result is a ratio having a magnitude greater than 65536^n, the result is rounded to the truncated integer quotient of its numerator over its denominator; b) if the exact result is a ratio having a magnitude less than 65536^{-n}, the result is rounded to the reciprocal of the truncated integer quotient of its denominator over its numerator; otherwise c) if the exact result is a ratio requiring a total of more than $n+1$ words of memory to store its numerator and denominator, the result is rounded to one of the two closest rational approximations that require a total of no more than $n+1$ words of memory to store its numerator and denominator. (A word is equivalent to 2 bytes or 16 bits of memory).

In short, a precision of *n* generally provides about $2.5 + 4.5n$ decimal digits of accuracy. Thus, a precision of 1 roughly corresponds to **single precision** floating point; a precision of 3 roughly corresponds to **double precision** floating point; and so on.

```
(DEFUN PRECISION (N)
  ((AND (INTEGERP N) (PLUSP N))
    (set precision to N)
    (return previous precision) )
  (return current precision) )
```

```
$ 1.00001                                          ;Does round to 1
1

$ (PRECISION 3)                                    ;Set double precision
1

$ 1.00001                                          ;Does not round to 1
1.00001

$ (SETQ *EXACT-MODE* T)                            ;Set for exact mode
T

$ (SETQ *PRINT-POINT* 30)                          ;Print 30 digits
30

$ 1.000000000000000000000000000001                 ;Does not round to 1
1.000000000000000000000000000001

$ (SETQ *EXACT-MODE* NIL)                          ;Set for approximate mode
NIL

$ (PRECISION 1)                                    ;Restore single precision
3
```

UNDERFLOW [*n*] Function

If *n* is a positive integer, (UNDERFLOW *n*) sets the value at which underflow occurs to 65536^{-n} and returns the previous underflow setting. If *n* is zero, (UNDERFLOW *n*) prevents underflow from occurring and returns the previous underflow setting. If *n* is omitted or is neither zero nor a positive integer, (UNDERFLOW *n*) returns the current setting of underflow without changing it. The initial underflow setting is 0.

Sometimes when doing calculations with approximate arithmetic, it is desirable to have numbers smaller in absolute value than a specified number "underflow" to 0. For example, when computing a series approximation of a function, the series can be terminated when the terms of the series underflow to zero.

The following table gives the approximate numerical value at which underflow occurs for various settings of *n*:

n	Underflow	n	Underflow	n	Underflow	n	Underflow
1	$1.53\ 10^{-5}$	4	$5.42\ 10^{-20}$	7	$1.93\ 10^{-34}$	10	$6.84\ 10^{-49}$
2	$2.33\ 10^{-10}$	5	$8.27\ 10^{-25}$	8	$2.94\ 10^{-39}$	11	$1.04\ 10^{-53}$
3	$3.55\ 10^{-15}$	6	$1.26\ 10^{-29}$	9	$4.48\ 10^{-44}$	12	$1.59\ 10^{-58}$

```
(DEFUN UNDERFLOW (N)
  ((ZEROP N)
    (disable underflow feature)
    (return previous underflow setting) )
  ((AND (INTEGERP N) (PLUSP N))
    (set underflow point to 65536^-N)
    (return previous underflow setting) )
  (return current underflow setting) )

$ 0.000001                                          ;No underflow
0.000001

$ (UNDERFLOW 1)                                     ;Set underflow to 1
0                                                 ;Returns previous setting

$ 0.000001                                          ;Underflows to 0
0

$ (UNDERFLOW 0)                                     ;Disable underflow
7

$ 0.00000000000000000000000000001                   ;No underflow
0.00000000000000000000000000001
```

5.4.2 Numerical Predicates

The numerical predicates are divided into recognizers and comparators. Since the numerical recognizers are used to identify objects, they do *not* generate a Nonnumeric Argument error break, even if given a symbol or a cons as an argument; they simply return NIL. However the numerical comparators, used to compare two or more numbers, do generate a Nonnumeric Argument error break if any argument is *not* a number.

ZEROP [*object*] Function
PLUSP [*object*] Function
MINUSP [*object*] Function

If *object* is the number zero, (ZEROP *object*) returns T, otherwise it returns NIL.

If *object* is a positive number, (PLUSP *object*) returns T, otherwise it returns NIL.

If *object* is a negative number, (MINUSP *object*) returns T, otherwise it returns NIL.

```
(DEFUN ZEROP (OBJ)
  (AND (NUMBERP OBJ) (EQL OBJ 0)) )

(DEFUN PLUSP (OBJ)
  (AND (NUMBERP OBJ) (> OBJ 0)) )

(DEFUN MINUSP (OBJ)
  (AND (NUMBERP OBJ) (< OBJ 0)) )

$ (ZEROP 0.0)
T

$ (ZEROP 3.1416)
NIL

$ (ZEROP 'DOG)
NIL

$ (PLUSP 3.1416)
T

$ (PLUSP 0)
NIL

$ (MINUSP 3.1416)
NIL

$ (MINUSP -7/3)
T
```

ODDP [*object*] Function
EVENP [*object*] Function

If *object* is an odd integer, (ODDP *object*) returns T, otherwise it returns NIL.

If *object* is an even integer, (EVENP *object*) returns T, otherwise it returns NIL.

```
(DEFUN ODDP (OBJ)
  (AND (INTEGERP OBJ) (NOT (ZEROP (REM OBJ 2)))) )

(DEFUN EVENP (OBJ)
  (AND (INTEGERP OBJ) (ZEROP (REM OBJ 2))) )

$ (ODDP 12)
NIL

$ (ODDP -41)
T

$ (ODDP 3.1416)
NIL

$ (ODDP 'DOG)
NIL

$ (EVENP 12)
T

$ (EVENP -41)
NIL

$ (EVENP 3.1416)
NIL

$ (EVENP 'DOG)
NIL
```

= $[n_1, n_2, ..., n_m]$ Function
/= $[n_1, n_2, ..., n_m]$ Function

If n_1 is equal to n_2, and n_2 is equal to n_3, and ..., and n_{m-1} is equal to n_m, then (= n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (=) and (= n) return T.

If n_1 is *not* equal to n_2, and n_2 is *not* equal to n_3, and ..., and n_{m-1} is *not* equal to n_m, then (/= n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (/=) and (/= n) return T.

Note that if /= returns T, no *adjacent* pair of arguments are equal. However, it can still be the case that two *nonadjacent* arguments are equal.

```
(DEFUN = LST
  ((OR (NULL LST) (NULL (CDR LST))))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((EQL (CAR LST) (CADR LST))
      (APPLY '= (CDR LST)) ) )
  ((BREAK (LIST '= (CAR LST) (CADR LST)) "Nonnumeric Argument")
    (APPLY '= (CDR LST)) ) )

(DEFUN /= LST
  ((OR (NULL LST) (NULL (CDR LST))))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((EQL (CAR LST) (CADR LST))  NIL)
    (APPLY '/= (CDR LST)) )
  ((BREAK (LIST '/= (CAR LST) (CADR LST)) "Nonnumeric Argument")
    (APPLY '/= (CDR LST)) ) )

$ (= 4 4 7)
NIL

$ (= 3 3.0)
T

$ (= 0.75 6/8)
T

$ (/= 4 4 7)
NIL

$ (/= 3 3.0)
NIL

$ (/= 0.75 6/8)
NIL

$ (/= 6 -2 6)                                  ;No adjacent pairs are equal
T
```

< [n_1, n_2, ..., n_m] Function
> [n_1, n_2, ..., n_m] Function

If $n_1 < n_2$, and $n_2 < n_3$, and ..., and $n_{m-1} < n_m$, then (< n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (<) and (< n) return T.

If $n_1 > n_2$, and $n_2 > n_3$, and ..., and $n_{m-1} > n_m$, then (> n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (>) and (> n) return T.

The comparator < is commonly used to determine if a number lies within an **open interval** by calling it with three arguments: the number and the interval's end-points. For example, the test

```
(< 10 NUM 100)
```

determines if NUM is between 10 and 100.

```
(DEFUN < LST
  ((OR (NULL LST) (NULL (CDR LST))))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((less than (CAR LST) (CADR LST))
      (APPLY '< (CDR LST)) ) )
  ((BREAK (LIST '< (CAR LST) (CADR LST)) "Nonnumeric Argument")
    (APPLY '< (CDR LST)) ) )

(DEFUN > LST
  ((OR (NULL LST) (NULL (CDR LST))))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((greater than (CAR LST) (CADR LST))
      (APPLY '> (CDR LST)) ) )
  ((BREAK (LIST '> (CAR LST) (CADR LST)) "Nonnumeric Argument")
    (APPLY '> (CDR LST)) ) )

$ (< 5 9)
T

$ (< 4 -7)
NIL

$ (< 3/5 2/3)
T

$ (> 5 9)
NIL

$ (> 7 3 4)
NIL

$ (> 7/2 3.2)
T
```

<= [n_1, n_2, ..., n_m] Function
>= [n_1, n_2, ..., n_m] Function

If $n_1 \le n_2$, and $n_2 \le n_3$, and ..., and $n_{m-1} \le n_m$, then (<= n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (<=) and (<= n) return T.

If $n_1 \ge n_2$, and $n_2 \ge n_3$, and ..., and $n_{m-1} \ge n_m$, then (>= n_1 n_2 ... n_m) returns T, otherwise it returns NIL. (>=) and (>= n) return T.

The comparator <= is commonly used to determine if a number lies in a **closed interval** by calling it with three arguments: the number and the interval's end-points. For example, the test

```
(<= 10 NUM 100)
```

determines if NUM is between 10 and 100 inclusive.

```
(DEFUN <= LST
  ((OR (NULL LST) (NULL (CDR LST))))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((> (CAR LST) (CADR LST)) NIL)
    (APPLY '<= (CDR LST)) )
  ((BREAK (LIST '<= (CAR LST) (CADR LST)) "Nonnumeric Argument")
    (APPLY '<= (CDR LST)) ) )

(DEFUN >= LST
  ((OR (NULL LST) (NULL (CDR LST))))
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((< (CAR LST) (CADR LST)) NIL)
    (APPLY '>= (CDR LST)) )
  ((BREAK (LIST '>= (CAR LST) (CADR LST)) "Nonnumeric Argument")
    (APPLY '>= (CDR LST)) ) )

$ (<= 5 9 9)
T

$ (<= 4 -7)
NIL

$ (<= 3 3.0)
T

$ (>= 4 -7)
T

$ (>= 3 3.0)
T

$ (>= 3/5 2/3)
NIL
```

MAX [$n_1, n_2, ..., n_m$] Function
MIN [$n_1, n_2, ..., n_m$] Function

(MAX n_1 n_2 ... n_m) returns its greatest argument (i.e. the argument furthest to the right on the real number line).

(MIN n_1 n_2 ... n_m) returns its least argument (i.e. the argument furthest to the left on the real number line).

If these functions are called with no arguments, an Insufficient Arguments error break occurs.

```
(DEFUN MAX LST
  ((NULL LST)
    (BREAK (LIST 'MAX) "Insufficient Arguments") )
  ((NULL (CDR LST))
    (CAR LST) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((< (CAR LST) (CADR LST))
      (APPLY 'MAX (CADR LST) (CDDR LST)) )
    (APPLY 'MAX (CAR LST) (CDDR LST)) )
  (APPLY 'MAX (BREAK (LIST 'MAX (CAR LST) (CADR LST)) "Nonnumeric Argument")
              (CDDR LST)) )

(DEFUN MIN LST
  ((NULL LST)
    (BREAK (LIST 'MIN) "Insufficient Arguments") )
  ((NULL (CDR LST))
    (CAR LST) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((< (CAR LST) (CADR LST))
      (APPLY 'MIN (CAR LST) (CDDR LST)) )
    (APPLY 'MIN (CADR LST) (CDDR LST)) )
  (APPLY 'MIN (BREAK (LIST 'MIN (CAR LST) (CADR LST)) "Nonnumeric Argument")
              (CDDR LST)) )

$ (MAX 5 -7 4)
5

$ (MAX 2/3 0.6)
0.6666666

$ (MAX -4)
-4

$ (MIN 2/3 0.6)
0.6
```

5.4.3 Arithmetic Operators

The functions in this subsection perform the basic mathematical operations on both integer and rational numbers. They generate a Nonnumeric Argument error break if any argument is *not* a number.

+ $[n_1, n_2, ..., n_m]$ Function
− $[n_1, n_2, ..., n_m]$ Function

(+ n_1 n_2 ... n_m) returns the sum of the numbers n_1 through n_m. (+) returns 0, the identity for addition.

(− n_1 n_2 ... n_m) returns the difference between n_1 and the sum of n_2 through n_m. (− n) returns minus n. (−) generates an Insufficient Arguments error break.

```
(DEFUN + LST
  ((NULL LST)  0)
  ((NULL (CDR LST))
    (CAR LST) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    (APPLY '+ ((CAR LST) plus (CADR LST)) (CDDR LST)) )
  (APPLY '+ (+TRAP (CAR LST) (CADR LST)) (CDDR LST)) )

(DEFUN - LST
  ((NULL LST)
    (BREAK (LIST '-) "Insufficient Arguments") )
  ((NULL (CDR LST))
    ((NUMBERP (CAR LST))
      (minus (CAR LST)) )
    (-TRAP (CAR LST)) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((NULL (CDDR LST))
      ((CAR LST) minus (CADR LST)) )
    (APPLY '- ((CAR LST) minus (CADR LST)) (CDDR LST)) )
  (APPLY '- (-TRAP (CAR LST) (CADR LST)) (CDDR LST)) )

$ (+ 1/6 0.7)
0.8666666

$ (- 12 5)
7

$ (- 8)
-8
```

* [n_1, n_2, ..., n_m] Function
/ [n_1, n_2, ..., n_m] Function

(* n_1 n_2 ... n_m) returns the product of the numbers n_1 through n_m. (*) returns 1, the identity for multiplication.

(/ n_1 n_2 ... n_m) returns the result of n_1 divided by the product of n_2 through n_m. (/ n) returns the reciprocal of n. (/) generates an Insufficient Arguments error break. If any argument other than the first is zero, a Zero Divide error break occurs.

```
(DEFUN * LST
  ((NULL LST)  1)
  ((NULL (CDR LST))
    (CAR LST) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    (APPLY '* ((CAR LST) times (CADR LST)) (CDDR LST)) )
  (APPLY '* (*TRAP (CAR LST) (CADR LST)) (CDDR LST)) )

(DEFUN / LST
  ((NULL LST)
    (BREAK (LIST '/) "Insufficient Arguments") )
  ((NULL (CDR LST))
    ((NUMBERP (CAR LST))
      ((ZEROP (CAR LST))
        (/TRAP (CAR LST)) )
      (reciprocal of (CAR LST)) )
    (/TRAP (CAR LST)) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    ((NULL (CDDR LST))
      ((ZEROP (CADR LST))
        (/TRAP (CAR LST) (CADR LST)) )
      (quotient (CAR LST) (CADR LST)) )
    ((ZEROP (CADR LST))
      (APPLY '/ (/TRAP (CAR LST) (CADR LST)) (CDDR LST)) )
    (APPLY '/ (quotient (CAR LST) (CADR LST)) (CDDR LST)) )
  (APPLY '/ (/TRAP (CAR LST) (CADR LST)) (CDDR LST)) )

$ (* 3 4 5)
60

$ (* 5/6 -0.7)
-0.5833333

$ (/ 12 8)
1.5

$ (/ 5)
0.2
```

ADD1 [*n*] Function
SUB1 [*n*] Function

(ADD1 *n*) returns (+ *n* 1). (ADD1) generates an Insufficient Arguments error break.

(SUB1 *n*) returns (– *n* 1). (SUB1) generates an Insufficient Arguments error break.

```
(DEFUN ADD1 (N)
  ((NUMBERP N)
    (+ N 1) )
  (BREAK (LIST 'ADD1 N) "Nonnumeric Argument") )

(DEFUN SUB1 (N)
  ((NUMBERP N)
    (- N 1) )
  (BREAK (LIST 'SUB1 N) "Nonnumeric Argument") )

$ (ADD1 3)
4

$ (ADD1 2/3)
1.6666666

$ (ADD1 -3)
-2

$ (SUB1 3)
2

$ (SUB1 0.25)
-0.75

$ (SUB1 -3)
-4
```

INCQ [*symbol*, *n*] Special form
DECQ [*symbol*, *n*] Special form

If *symbol* is a symbol and *n* is a number, (INCQ *symbol n*) adds *n* to the value of *symbol*, sets *symbol* to the result, and returns the result. If *n* is omitted, *symbol* is incremented by 1.

If *symbol* is a symbol and *n* is a number, (DECQ *symbol n*) subtracts *n* from the value of *symbol*, sets *symbol* to the result, and returns the result. If *n* is omitted, *symbol* is decremented by 1.

If the first argument of INCQ or DECQ is not a symbol, a Nonsymbolic Argument error break occurs.

```
(DEFMACRO INCQ (SYM N)
  ((NULL N)
    (LIST 'SETQ SYM (LIST 'ADD1 SYM)) )
  (LIST 'SETQ SYM (LIST '+ SYM N)) )

(DEFMACRO DECQ (SYM N)
  ((NULL N)
    (LIST 'SETQ SYM (LIST 'SUB1 SYM)) )
  (LIST 'SETQ SYM (LIST '- SYM N)) )

$ (SETQ CTR 5)                                          ;Set counter to 5
5

$ (INCQ CTR)                                           ;Increment counter
6

$ (INCQ CTR 2)                                          ;Add 2 to counter
8

$ (DECQ CTR 5)                                       ;Subtract 5 from counter
3

$ CTR                                                   ;Value of counter
3
```

INCF [*place*, *n*] Library macro
DECF [*place*, *n*] Library macro

If *place* is a valid place form and *n* is a number, (INCF *place n*) adds *n* to the value in *place*, stores the result in *place*, and returns the result. If *n* is omitted, *place* is incremented by 1.

If *place* is a valid place form and *n* is a number, (DECF *place n*) subtracts *n* from the value in *place*, stores the result in *place*, and returns the result. If *n* is omitted, *place* is decremented by 1.

If the first argument of INCF or DECF is not a valid place form (see SETF in Section 5.3.1), an Invalid Place Form error break occurs. INCF and DECF are defined in COMMON.LSP.

```
(DEFMACRO INCF (PLACE N)
  ((NULL N)
    (MAKE-SET-FORM PLACE (LIST 'ADD1 PLACE)) )
  (MAKE-SET-FORM PLACE (LIST '+ PLACE N)) )

(DEFMACRO DECF (PLACE N)
  ((NULL N)
    (MAKE-SET-FORM PLACE (LIST 'SUB1 PLACE)) )
  (MAKE-SET-FORM PLACE (LIST '- PLACE N)) )

$ (SETQ LST '(5 2 3))                                  ;Initialize LST
(5 2 3)

$ (INCF (CAR LST))                                     ;Increment car of LST
6

$ (INCF (NTH 1 LST) 2)                                 ;Add 2 to second of LST
4

$ (DECF (CADR LST) 5)                                  ;Subtract 5 from second of LST
-1

$ LST                                                  ;Value of LST
(6 -1 3)
```

GCD [n_1, n_2, ..., n_m] Function

(GCD n_1 n_2 ... n_m) returns the gcd (**greatest common divisor**) of the numbers n_1 through n_m. (GCD n) returns the absolute value of n. (GCD) returns 0, the identity for this operation.

The gcd of two integers is the largest non-negative integer that evenly divides both integers. The gcd of two rational numbers is the gcd of the numerators divided by the lcm of the denominators. In either case if the gcd is divided into the two numbers, the result is the smallest two integers having the same ratio as the original two numbers.

```
(DEFUN GCD LST
  ((NULL LST)  0)
  ((NULL (CDR LST))
    ((NUMBERP (CAR LST))
      (ABS (CAR LST)) )
    (GCDTRAP (CAR LST)) )
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    (APPLY 'GCD (/ (Integer-GCD (NUMERATOR (CAR LST)) (NUMERATOR (CADR LST)))
                   (Integer-LCM (DENOMINATOR (CAR LST)) (DENOMINATOR (CADR LST))))
                (CDDR LST)) )
  (APPLY 'GCD (GCDTRAP (CAR LST) (CADR LST)) (CDDR LST)) )

(DEFUN Integer-GCD (N M)
  ((ZEROP M)
    (ABS N) )
  (Integer-GCD M (REM N M)) )

(DEFUN Integer-LCM (N M)
  (/ (* N M) (Integer-GCD N M)) )

$ (GCD 4 6)
2

$ (GCD 12 -16 20)
4

$ (GCD 14/15 8/9)
0.0444444

$ (GCD 0 5)
5

$ (GCD 0 0)
0

$ (GCD -7)
7
```

LCM $[n_1, n_2, \ldots, n_m]$ Function

(LCM n_1 n_2 ... n_m) returns the lcm (**least common multiple**) of the numbers n_1 through n_m. (LCM n) returns the absolute value of n. (LCM) generates an Insufficient Arguments error break, since this operation has no finite identity.

The lcm of two integers is the least non-negative integer that is a multiple of both integers. In general, the lcm of the two numbers n and m, either integer or rational, can be defined as

$$\frac{|n\ m|}{\gcd(n, m)}$$

```
(DEFUN LCM LST
  ((NULL LST)
    (BREAK (LIST 'LCM) "Insufficient Arguments") )
  ((NULL (CDR LST))
    ((NUMBERP (CAR LST))
      (ABS (CAR LST)) )
    (LCMTRAP (CAR LST)) )
  ((OR (ZEROP (CAR LST)) (ZEROP (CADR LST)))  0)
  ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
    (APPLY 'LCM (/ (ABS (* (CAR LST) (CADR LST))) (GCD (CAR LST) (CADR LST)))
                (CDDR LST)) )
  (APPLY 'LCM (LCMTRAP (CAR LST) (CADR LST)) (CDDR LST)) )

$ (LCM 4 6)
12

$ (LCM 12 -16 20)
240

$ (LCM 14/15 8/9)
18.6666666                                          ;This result is = to 56/3

$ (LCM 0 5)
0
```

5.4.4 Component Extraction

ABS [*n*] Function
SIGNUM [*n*] Function

If *n* is a number, (ABS *n*) returns the **absolute value** of *n*. If *n* is not a number, ABSTRAP is called to generate a Nonnumeric Argument error break (see Section 5.10.3).

If *n* is a number, (SIGNUM *n*) returns 1, -1, or 0 depending on whether *n* is positive, negative, or zero. If *n* is not a number, a Nonnumeric Argument error break occurs.

```
(DEFUN ABS (N)
  ((NUMBERP N)
    ((MINUSP N)
      (- N) )
    N )
  (ABSTRAP N) )

(DEFUN SIGNUM (N)
  ((NUMBERP N)
    ((ZEROP N) 0)
    ((PLUSP N) 1)
    -1 )
  (BREAK (LIST 'SIGNUM N) "Nonnumeric Argument") )

$ (ABS 4/6)
0.6666666

$ (ABS -3)
3

$ (ABS 0)
0

$ (SIGNUM 7/2)
1

$ (SIGNUM 0)
0

$ (SIGNUM -0.2)
-1
```

NUMERATOR [*n*] Function
DENOMINATOR [*n*] Function

If *n* is a ratio, (NUMERATOR *n*) returns the numerator of *n*. If *n* is an integer, (NUMERATOR *n*) returns *n*. The numerator is always an integer. If *n* is not a number, NUMTRAP is called to generate a Nonnumeric Argument error break (see Section 5.10.3).

If *n* is a ratio, (DENOMINATOR *n*) returns the denominator of *n*. If *n* is an integer, (DENOMINATOR *n*) returns 1. The denominator is always a positive integer. If *n* is not a number, DENTRAP is called to generate a Nonnumeric Argument error break (see Section 5.10.3).

```
(DEFUN NUMERATOR (N)
  ((NUMBERP N)
    ((INTEGERP N)  N)
    (the numerator of N) )
  (NUMTRAP N) )

(DEFUN DENOMINATOR (N)
  ((NUMBERP N)
    ((INTEGERP N)  1)
    (the denominator of N) )
  (DENTRAP N) )

$ (NUMERATOR 10/8)
5

$ (DENOMINATOR 10/8)
4

$ (NUMERATOR 0.75)
3

$ (DENOMINATOR 0.75)
4

$ (NUMERATOR -8)
-8

$ (DENOMINATOR -8)
1
```

FLOOR [*n*, *m*] Function
CEILING [*n*, *m*] Function

If *n* is an integer, (FLOOR *n*) returns *n*. If *n* is a ratio, FLOOR returns the largest integer less than *n*. In other words, FLOOR truncates *n* toward minus infinity. If *n* and *m* are numbers, (FLOOR *n m*) truncates toward minus infinity the quotient of *n* divided by *m*. (FLOOR *n m*) equals (FLOOR (/ *n m*)), but is more efficient.

If *n* is an integer, (CEILING *n*) returns *n*. If *n* is a ratio, CEILING returns the smallest integer greater than *n*. In other words, CEILING truncates *n* toward plus infinity. If *n* and *m* are numbers, (CEILING *n m*) truncates toward plus infinity the quotient of *n* divided by *m*. (CEILING *n m*) equals (CEILING (/ *n m*)), but is more efficient.

If the second argument of FLOOR or CEILING (i.e. the divisor) is zero, a Zero Divide error break occurs.

```
(DEFUN FLOOR (N M)
  ((NUMBERP M)
    ((ZEROP M)
      (BREAK (LIST 'FLOOR N M) "Zero Divide") )
    (FLOOR (/ N M)) )
  ((NUMBERP N)
    ((INTEGERP N)  N)
    (the largest integer less than N) )
  (BREAK (LIST 'FLOOR N M) "Nonnumeric Argument") )

(DEFUN CEILING (N M)
  ((NUMBERP M)
    ((ZEROP M)
      (BREAK (LIST 'CEILING N M) "Zero Divide") )
    (CEILING (/ N M)) )
  ((NUMBERP N)
    ((INTEGERP N)  N)
    (the smallest integer greater than N) )
  (BREAK (LIST 'CEILING N M) "Nonnumeric Argument") )
```

```
$ (FLOOR 7.3)
7

$ (FLOOR -3.5)
-4

$ (FLOOR 8 3)
2

$ (FLOOR -15/4 2)
-2
```

```
$ (CEILING 7.3)
8

$ (CEILING -3.5)
-3

$ (CEILING 8 3)
3

$ (CEILING -15/4 2)
-1
```

TRUNCATE [*n*, *m*] Function
ROUND [*n*, *m*] Function

If *n* is an integer, (TRUNCATE *n*) returns *n*. If *n* is a positive ratio, TRUNCATE returns the largest integer less than *n*. If *n* is a negative ratio, TRUNCATE returns the smallest integer greater than *n*. In other words, TRUNCATE truncates *n* toward zero. If *n* and *m* are numbers, (TRUNCATE *n m*) truncates toward zero the quotient of *n* divided by *m*. (TRUNCATE *n m*) equals (TRUNCATE (/ *n m*)), but is more efficient.

If *n* is an integer, (ROUND *n*) returns *n*. If *n* is a ratio, ROUND returns the integer closest in value to *n*. If *n* is equidistant between two integers (i.e. 2 *n* is an integer), ROUND returns the one that is divisible by two. If *n* and *m* are numbers, (ROUND *n m*) rounds the quotient of *n* divided by *m*. (ROUND *n m*) equals (ROUND (/ *n m*)), but is more efficient.

If the second argument of TRUNCATE or ROUND (i.e. the divisor) is zero, a Zero Divide error break occurs.

```
(DEFUN TRUNCATE (N M)
  ((NUMBERP M)
    ((ZEROP M)
      (BREAK (LIST 'TRUNCATE N M) "Zero Divide") )
    (TRUNCATE (/ N M)) )
  ((NUMBERP N)
    ((INTEGERP N)  N)
    ((PLUSP N)  (the largest integer less than N) )
    (the smallest integer greater than N) )
  (BREAK (LIST 'TRUNCATE N M) "Nonnumeric Argument") )

(DEFUN ROUND (N M)
  ((NUMBERP M)
    ((ZEROP M)
      (BREAK (LIST 'ROUND N M) "Zero Divide") )
    (ROUND (/ N M)) )
  ((NUMBERP N)
    ((INTEGERP N)  N)
    ((INTEGERP (* 2 N))
      ((EVENP (TRUNCATE N))  (TRUNCATE N) )
      (ADD1 (TRUNCATE N)) )
    (the integer closest to N) )
  (BREAK (LIST 'ROUND N M) "Nonnumeric Argument") )
```

```
$ (TRUNCATE -2.4)
-3

$ (TRUNCATE 7 2)
3
```

```
$ (ROUND -2.4)
-2

$ (ROUND 7 2)
4
```

MOD [*n*, *m*] Function
REM [*n*, *m*] Function

If *n* and *m* are numbers, (MOD *n m*) returns the **modulus** of *n* divided by *m*. *m* defaults to 1. MOD is defined so that FLOOR and MOD preserve the fundamental relation between quotients and remainders:

$$n = m \text{ floor } (n, m) + \text{mod } (n, m)$$

If *n* and *m* are numbers, (REM *n m*) returns the **remainder** of *n* divided by *m*. *m* defaults to 1. REM is defined so that TRUNCATE and REM preserve the fundamental relation between quotients and remainders:

$$n = m \text{ truncate } (n, m) + \text{rem } (n, m)$$

If the second argument of MOD or REM (i.e. the divisor) is zero, a Zero Divide error break occurs.

```
(DEFUN MOD (N M)
  ((NULL M)
    (MOD N 1) )
  ((AND (NUMBERP N) (NUMBERP M))
    ((ZEROP M)
      (BREAK (LIST 'MOD N M) "Zero Divide") )
    (- N (* M (FLOOR N M))) )
  (BREAK (LIST 'MOD N M) "Nonnumeric Argument") )

(DEFUN REM (N M)
  ((NULL M)
    (REM N 1) )
  ((AND (NUMBERP N) (NUMBERP M))
    ((ZEROP M)
      (BREAK (LIST 'REM N M) "Zero Divide") )
    (- N (* M (TRUNCATE N M))) )
  (BREAK (LIST 'REM N M) "Nonnumeric Argument") )
```

```
$ (MOD 7 3)
1

$ (MOD -7 3)
2

$ (MOD 7 -3)
-2

$ (MOD -7 -3)
-1
```

```
$ (REM 7 3)
1

$ (REM -7 3)
-1

$ (REM 7 -3)
1

$ (REM -7 -3)
-1
```

DIVIDE [*n*, *m*] Function

If *n* is a number, (DIVIDE *n*) returns a cons whose car is (TRUNCATE *n*) and whose cdr is (REM *n*). If *n* and *m* are numbers, (DIVIDE *n m*) returns a cons whose car is (TRUNCATE *n m*) and whose cdr is (REM *n m*).

If the second argument of DIVIDE (i.e. the divisor) is zero, a Zero Divide error break occurs.

When both a truncated quotient and remainder are required, using the DIVIDE function is more efficient than computing them separately.

```
(DEFUN DIVIDE (N M)
  ((AND (NUMBERP N) (NUMBERP M))
    ((ZEROP M)
      (BREAK (LIST 'DIVIDE N M) "Zero Divide") )
    (CONS (TRUNCATE N M) (REM N M)) )
  (BREAK (LIST 'DIVIDE N M) "Nonnumeric Argument") )

$ (DIVIDE 7 3)
(2 . 1)

$ (DIVIDE -7 3)
(-2 . -1)

$ (DIVIDE 7 -3)
(-2 . 1)

$ (DIVIDE -7 -3)
(2 . -1)
```

5.4.5 Bit-wise Logical Operators

The functions in this subsection are for performing bit-wise logical operations over the integers. If called with noninteger arguments, a Noninteger Argument error break occurs.

LOGNOT [*n*] — Function
LOGTEST [*n*, *m*] — Library function
LOGBITP [*index*, *n*] — Library function

(LOGNOT *n*) returns the bit-wise **logical not** of *n* (i.e. the integer whose bits are the complement of the bits in *n*). Since LOGNOT operates as if negative integers are stored in "half-infinite" two's-complement notation, the result is always of the opposite sign as *n*.

If any 1 bits in *n* are also 1 bits in *m*, (LOGTEST *n m*) returns T, otherwise it returns NIL. LOGTEST is defined in COMMON.LSP.

If the bit in *n* whose weight is 2^{index} is 1, (LOGBITP *index n*) returns T, otherwise it returns NIL. LOGBITP is defined in COMMON.LSP.

```
$ (LOGNOT 9)                                    ;Complement 9
-10

$ (LOGNOT -1)                                   ;Complement -1
0

$ (LOGTEST 4 13)                                ;Test the 2nd bit in 13
T

$ (LOGBITP 2 13)                                ;Test the 2nd bit in 13
T
```

LOGIOR [$n_1, n_2, ..., n_m$] Function
LOGXOR [$n_1, n_2, ..., n_m$] Function
LOGAND [$n_1, n_2, ..., n_m$] Function
LOGEQV [$n_1, n_2, ..., n_m$] Library function

These functions perform logical operations on the bits of their integer arguments. They operate as if integers are stored in "half-infinite" two's-complement notation.

(LOGIOR n_1 n_2 ... n_m) returns the bit-wise **logical inclusive or** of its arguments. (LOGIOR) returns 0, the identity for this operation.

(LOGXOR n_1 n_2 ... n_m) returns the bit-wise **logical exclusive or** of its arguments. (LOGXOR) returns 0, the identity for this operation.

(LOGAND n_1 n_2 ... n_m) returns the bit-wise **logical and** of its arguments. (LOGAND) returns -1, the identity for this operation.

(LOGEQV n_1 n_2 ... n_m) returns the bit-wise **logical equivalence** of its arguments. (LOGEQV) returns -1, the identity for this operation. LOGEQV is defined in COMMON.LSP.

```
(DEFUN LOGIOR LST
  (REDUCE '(LAMBDA (INT1 INT2)
      ((AND (INTEGERP INT1) (INTEGERP INT2))
        (the logical OR of INT1 and INT2) )
      (BREAK (LIST 'LOGIOR INT1 INT2) "Noninteger Argument") )
    LST 0) )

$ (LOGIOR 11 6)
15

$ (LOGXOR -6 6)
-4

$ (LOGAND 11 6)
2

$ (LOGEQV 11 6)
-14
```

SHIFT [*n*, *count*] — Function
ASH [*n*, *count*] — Library function

If *n* is an integer and if *count* is a positive integer, (SHIFT *n count*) returns the result of shifting *n* left by *count* bits. If *count* is a negative integer, *n* is shifted right -*count* bits. The result is always of the same sign as *n*. Shifting left (right) is a fast way to multiply (divide) a number by a power of 2.

ASH is similar to SHIFT except that it operates as if negative integers are stored in "half-infinite" two's-complement notation. ASH is defined in COMMON.LSP.

```
(DEFUN SHIFT (N COUNT)
  ((AND (INTEGERP N) (INTEGERP COUNT))
    ((ZEROP COUNT)  N)
    ((PLUSP COUNT)
      (SHIFT (* N 2) (SUB1 COUNT)) )
    (SHIFT (TRUNCATE N 2) (SUB1 COUNT)) )
  (BREAK (LIST 'SHIFT N COUNT) "Noninteger Argument") )

(DEFUN ASH (N COUNT)
  ((OR (>= N 0) (>= COUNT 0))
    (SHIFT N COUNT) )
  (ASH (SHIFT (SUB1 N) -1) (ADD1 COUNT)) )

$ (SHIFT 5 -1)                                          ;halve 5
2

$ (SHIFT 5 1)                                           ;double 5
10

$ (SHIFT 5 2)                                           ;quadruple 5
20

$ (SHIFT -7 -2)                                         ;logical right shift
-1

$ (ASH -7 -2)                                           ;arithmetic right shift
-2
```

INTEGER-LENGTH [*n*] Function

If *n* is an integer, (INTEGER-LENGTH *n*) returns the number of bits required to represent the magnitude *n*.

```
(DEFUN INTEGER-LENGTH (N)
  ((INTEGERP N)
    ((ZEROP N)  0)
    (ADD1 (INTEGER-LENGTH (SHIFT N -1))) )
  (BREAK (LIST 'INTEGER-LENGTH N) "Noninteger Argument") )

$ (INTEGER-LENGTH 0)
0

$ (INTEGER-LENGTH 1)
1

$ (INTEGER-LENGTH 4)
3

$ (INTEGER-LENGTH 7)
3

$ (INTEGER-LENGTH 8)
4

$ (INTEGER-LENGTH -8)
4
```

5.4.6 Random Numbers

The functions in this subsection are for generating pseudo-random numbers. All are defined in COMMON.LSP.

RANDOM [*n*, *state*] — Library function

If *n* is a positive integer, (RANDOM *n*) returns an approximately uniformly distributed random integer x such that $0 \le x < n$. If *n* is a number but not a positive integer, (RANDOM *n*) returns an approximately uniformly distributed random rational number x such that $0 \le x < |n|$, subject to the current precision as set by PRECISION (see Section 5.4.1).

The non-negative integer value of the random state variable *RANDOM-STATE* is used and maintained by RANDOM as the present state of the random-number generator. If *state* is zero or a positive integer, (RANDOM *n state*) sets *RANDOM-STATE* to *state* before returning (RANDOM *n*).

```
(DEFUN RANDOM (N STATE)
  (IF (RANDOM-STATE-P STATE)
      (SETQ *RANDOM-STATE* STATE) )
  ((AND (INTEGERP N) (PLUSP N))
    (LET ((TEMP 0)
          (COUNT 0))
      (LOOP
        (SETQ *RANDOM-STATE* (LOGAND 4294967295
                                     (ADD1 (* 3141592653 *RANDOM-STATE*))))
        (SETQ TEMP (+ (* TEMP 32) *RANDOM-STATE*))
        ((> (INCQ COUNT 32) (+ (INTEGER-LENGTH N) 10))
          (SHIFT (* N TEMP) (- COUNT)) ) ) ) )
  ((NUMBERP N)
    (LET ((TEMP (EXPT 2 (- (* 16 (ADD1 (PRECISION))) 10))))
        (/ (* (ABS N) (RANDOM TEMP)) TEMP) ) ) )

$ (RANDOM 100)                                  ;Random integer less than 100
84                                                          ;Result will vary

$ (RANDOM -100)                                 ;Random number less than 100
61.2102272                                                  ;Result will vary
```

RANDOM-STATE Library control variable

MAKE-RANDOM-STATE [*state*] Library function

RANDOM-STATE is used to store the current state of the random number generator. The function RANDOM automatically updates the value of *RANDOM-STATE* when generating random numbers. *RANDOM-STATE* should only be set to values returned by the function MAKE-RANDOM-STATE. Initially *RANDOM-STATE* is set to (MAKE-RANDOM-STATE T).

If *state* is T, (MAKE-RANDOM-STATE *state*) returns a random-number state based on the system clock. If *state* is NIL or omitted, MAKE-RANDOM-STATE returns *RANDOM-STATE*. If *state* is an acceptable random-number state, MAKE-RANDOM-STATE returns *state*. Otherwise, MAKE-RANDOM-STATE returns NIL.

```
(DEFUN MAKE-RANDOM-STATE (STATE)
  ((NULL STATE)
    *RANDOM-STATE* )
  ((EQ STATE 'T)
    (TIME) )
  ((RANDOM-STATE-P STATE)
    STATE ) )

(SETQ *RANDOM-STATE* (MAKE-RANDOM-STATE 'T))

$ (SETQ *RANDOM-STATE* (MAKE-RANDOM-STATE 1000))
1000

$ (RANDOM 100)
45

$ (SETQ *RANDOM-STATE* (MAKE-RANDOM-STATE 1000))
1000

$ (RANDOM 100)                                  ;Resetting *RANDOM-STATE* makes
45                                                  ;RANDOM results repeatable
```

RANDOM-STATE-P [*object*] Library function

If *object* is an acceptable random-number state object, (RANDOM-STATE-P *object*) returns T, otherwise it returns NIL. The acceptable random-number states are integers greater than or equal to zero.

```
(DEFUN RANDOM-STATE-P (OBJ)
  (AND (INTEGERP OBJ) (>= OBJ 0)) )

$ (RANDOM-STATE-P 546)                    ;Positives are valid random states
T

$ (RANDOM-STATE-P (TIME))                 ;TIME gives valid random states
T

$ (RANDOM-STATE-P -73)                    ;Negatives are not valid random states
NIL

$ (RANDOM-STATE-P 12.7)                   ;Ratios are not valid random states
NIL
```