Table of contents

0.1 History 0-1
0.1.1 Design goals 0-1
0.1.2 Outcome 0-2
0.1.3 Current state of the system 0-2
0.1.4 Le-Lisp Version 15.26 0-2
0.2 Reader's guide 0-3

1 Use and installation **1-1**
1.1 Le-Lisp on various systems 1-1
1.2 Starting with Le-Lisp 1-2
1.3 Starting the Le-Lisp system under Unix 1-3
1.4 Installation of the Le-Lisp system under Unix 1-4
1.4.1 Installing the system 1-4
1.4.2 Modification of the system configuration 1-7
1.4.3 Modification of data zone sizes 1-9
1.4.4 Linking the Le-Lisp system with C modules 1-11
1.4.5 Calling the shell 1-11
1.5 Starting the Le-Lisp system under VMS 1-11
1.6 Practical advice 1-11

2 Evaluation **2-1**
2.1 Basic objects 2-1
2.1.1 Atomic objects 2-1
2.1.2 Compound objects 2-4
2.2 Basic evaluation actions 2-5
2.2.1 Evaluation of atomic objects 2-5
2.2.2 Evaluation of composite objects 2-6
2.3 Evaluation of functions 2-6
2.3.1 Functions of the `subr` kind 2-7
2.3.2 Functions of the `fsubr` kind 2-8
2.3.3 Functions of the `msubr` kind 2-8
2.3.4 Functions of the `dmsubr` kind 2-9
2.3.5 Functions of the `expr` kind 2-9

2.3.6 Functions of the `expr` kind with a `&nobind` argument 2-11
2.3.7 Functions of the `fexpr` kind 2-11
2.3.8 Functions of the `macro` kind 2-12
2.3.9 Functions of the `dmacro` kind 2-12
2.4 Defining functions 2-13
2.5 Packages 2-13
2.6 Extended types 2-13
2.7 Meta-circular definition of the evaluator 2-15

3 Predefined functions 3-1
3.1 Evaluation functions 3-2
3.2 Application functions 3-6
3.2.1 Simple application functions 3-7
3.2.2 Application functions of the `map` type 3-7
3.2.3 Other application functions 3-10
3.3 Environment manipulation functions 3-13
3.4 Function-definition functions 3-17
3.4.1 Static function definitions 3-17
3.4.2 Advanced use of `macro` functions 3-20
3.4.3 Definition of closures 3-22
3.4.4 Definition of dynamic functions 3-23
3.4.5 Generalized assignment 3-23
3.5 Variable functions 3-24
3.6 Basic control functions 3-26
3.7 Lexical control functions 3-33
3.7.1 Primitive lexical control forms 3-33
3.7.2 Iteration functions of the `prog` kind 3-34
3.7.3 Iteration functions of the `do` kind 3-35
3.8 Dynamic, non-local control functions 3-37
3.9 Basic predicates 3-42
3.10 Functions on lists 3-49
3.10.1 Search functions on lists 3-49
3.10.2 List creation functions 3-52
3.10.3 Functions on cells of labelled lists 3-58

3.10.4 Physical modification functions 3-59
3.10.5 Functions on A-lists 3-65
3.10.6 Sorting functions 3-68
3.11 Functions on symbols 3-69
3.11.1 Functions that access symbol values 3-69
3.11.2 Functions that modify symbol values 3-70
3.11.3 Functions on P-lists 3-75
3.11.4 Access to function definitions 3-78
3.11.5 Access to symbol special fields 3-81
3.11.6 Symbol creation functions 3-83
3.11.7 Symbol management functions 3-84
3.12 Functions on character strings 3-86
3.12.1 Basic manipulation functions 3-87
3.12.2 Character string conversion 3-89
3.12.3 Comparison of character strings 3-91
3.12.4 Character-string creation functions 3-91
3.12.5 Character-string access functions 3-93
3.12.6 Functions that physically modify strings 3-94
3.12.7 Search functions on character strings 3-95
3.13 Functions on characters 3-97
3.14 Functions on vectors 3-98
3.15 Functions on arrays 3-103
3.16 Hash tables 3-104
3.16.1 Hash table creation functions 3-105
3.16.2 Hash table access functions 3-105
3.17 Mathematical sets 3-106
3.17.1 Operations on sets 3-107
3.17.2 Comparisons on sets 3-109
3.17.3 Transitive closure 3-109

4 Arithmetic functions **4-1**

4.1 Generic arithmetic 4-1
4.1.1 Generic arithmetic interrupt 4-2
4.1.2 Type tests 4-2

4.1.3 Numeric conversions 4-3
4.1.4 Generic arithmetic functions 4-4
4.1.5 Predicates of generic arithmetic 4-7
4.1.6 Circular and mathematical functions 4-10
4.1.7 Extensions to generic arithmetic 4-12
4.2 Integer arithmetic 4-12
4.2.1 Integer arithmetic functions 4-13
4.2.2 Fixed-precision integer comparison functions 4-14
4.2.3 Boolean functions 4-16
4.2.4 Functions on bit fields 4-17
4.2.5 Pseudo-random functions 4-19
4.3 Extended integer arithmetic 4-19
4.4 Floating-point arithmetic 4-22
4.4.1 Floating-point arithmetic functions 4-22
4.4.2 Floating-point arithmetic comparisons 4-23
4.5 Mixed arithmetic 4-25

5 Object-oriented programming 5-1
5.1 Structures 5-1
5.1.1 Structure definition 5-2
5.1.2 Instance creation 5-2
5.1.3 Access to fields 5-3
5.1.4 Type tests 5-3
5.1.5 Implementation of structures 5-4
5.2 Le-Lisp typology 5-4
5.3 Object-oriented programming 5-7
5.3.1 Searching for methods 5-7
5.3.2 Invoking methods 5-9
5.3.3 Predefined methods 5-12
5.4 Abbreviation functions 5-13
5.5 MicroCeyx 5-15
5.5.1 Specific errors 5-15
5.5.2 Definition of structures 5-16
5.5.3 Tclass and record instances 5-17

5.5.4 Methods and message sending 5-18

6 Input and output **6-1**

6.1 Introduction 6-1
6.1.1 Characters 6-1
6.1.2 Character strings 6-1
6.1.3 Lines 6-2
6.1.4 Channels 6-2
6.1.5 Programmable I/O interrupts 6-2
6.2 Basic input functions 6-3
6.2.1 Inside `read` 6-7
6.3 Use of the terminal for input 6-8
6.4 Standard reader 6-9
6.4.1 Reading symbols 6-9
6.4.2 Reading character strings 6-10
6.4.3 Reading integer and rational numbers 6-11
6.4.4 Reading floating-point numbers 6-12
6.4.5 Reading lists 6-12
6.4.6 Reading vectors 6-13
6.4.7 Reading comments 6-13
6.4.8 Types of characters 6-14
6.4.9 Macro characters 6-18
6.5 Basic output functions 6-33
6.6 Controlling the output functions 6-35
6.6.1 Limitations on printing 6-35
6.6.2 Standard printing environment 6-36
6.6.3 Extending the printer 6-38
6.7 Input/output for lists 6-39
6.8 Input/output on character strings 6-40
6.9 Input/Output buffer management 6-41
6.9.1 Input buffer 6-41
6.9.2 Output buffer 6-44
6.10 Functions on I/O streams 6-47
6.10.1 Default directories and extensions 6-48

6.10.2 Selecting I/O streams 6-48
6.10.3 End-of-file programmable interrupt 6-52
6.10.4 Functions on files 6-53
6.10.5 `load` function and `autoload` mode 6-54
6.10.6 File access paths 6-55
6.10.7 Access to libraries 6-56
6.11 Event loop 6-57
6.11.1 Functions 6-57
6.11.2 Technical notes 6-60
6.11.3 Precautions 6-61
6.12 Virtual file system 6-63
6.12.1 Pathnames 6-63
6.12.2 Relative access paths 6-66
6.12.3 Pathname manipulation functions 6-67
6.12.4 Portability 6-74
6.13 Features 6-74

7 System functions **7-1**
7.1 Programmable interrupts 7-1
7.2 Machine interrupts 7-3
7.2.1 User interrupt 7-3
7.2.2 Real-time clock 7-4
7.3 Multiple tasks 7-5
7.3.1 Basic sequencers 7-5
7.4 Errors provoked by the Le-Lisp system 7-7
7.4.1 Standard error processing 7-8
7.4.2 Explicit call and error test 7-8
7.4.3 Examples of exception handling 7-9
7.5 Access to the evaluator 7-10
7.6 Core-image files 7-11
7.7 Installation functions 7-13
7.8 Calling and leaving the system 7-16
7.9 Top level 7-18
7.10 Garbage collector 7-19

7.11 Date processing 7-23
7.12 Other access to the system 7-23

8 S-expression pretty-printer **8-1**
8.0.1 Pretty-print functions 8-2
8.0.2 Control of pretty-printer functions 8-2
8.0.3 Standard pretty-printer format 8-3
8.0.4 Extending the pretty-printer 8-4

9 Specialized output **9-1**
9.1 Formatted output 9-1
9.1.1 Print-formatting function 9-1
9.1.2 Format directives 9-3
9.2 Handling circular or shared objects 9-12
9.3 Multi-language messages 9-14
9.3.1 Languages 9-15
9.3.2 Messages 9-16
9.3.3 Advice 9-17

10 Rational and complex arithmetic **10-1**
10.1 Rationals (file Q) 10-2
10.1.1 Rational number I/O 10-2
10.1.2 Tests for type 10-3
10.1.3 Generic rational arithmetic 10-3
10.1.4 Functions limited to Z 10-4
10.1.5 Functions limited to rational arguments (limited to Q) 10-4
10.1.6 Two examples 10-5
10.2 Complex numbers (the field C) 10-6
10.2.1 Complex number I/O 10-6
10.2.2 Tests for type 10-6
10.2.3 Complex generic arithmetic 10-7
10.2.4 Functions limited to the complex numbers (limited to C) 10-7
10.2.5 Polar coordinates 10-8
10.2.6 Hyperbolic functions 10-9
10.3 A complex mini-extension of generic arithmetic 10-10

10.3.1 Representation 10-10
10.3.2 I/O for C 10-10
10.3.3 Complex arithmetic 10-11
10.3.4 exp, log and sqrt functions 10-12

11 Debugging tools **11-1**
11.1 Tracing 11-1
11.1.1 Stepping functions 11-1
11.1.2 Trace parameters 11-2
11.1.3 Trace global variables 11-3
11.1.4 Example of trace use 11-5
11.2 Break and debug mode 11-7
11.2.1 Inspection (or debug) loop 11-8
11.2.2 Debug mode functions 11-11
11.3 Stepwise execution 11-12

12 Loader/assembler LLM3 **12-1**
12.1 Access to memory and the CPU 12-1
12.2 LLM3 memory loader 12-4
12.3 LLM3 instruction format 12-5
12.4 Modules and labels 12-5
12.5 LLM3 instruction operands 12-6
12.6 Pseudo-instructions 12-8
12.7 Basic instructions 12-9
12.7.1 Moving pointers 12-9
12.7.2 Pointer comparisons 12-9
12.7.3 Control 12-9
12.8 Stack 12-10
12.8.1 Management of the stack pointer 12-10
12.8.2 As a control stack 12-10
12.8.3 As a data stack 12-11
12.9 List cell `cons` operations 12-11
12.9.1 Test for list cell type 12-11
12.9.2 Access to list cell fields 12-12
12.9.3 Creation of list cells 12-12

12.10 `nil` 12-12
12.11 Symbols 12-12
12.11.1 Test for symbol type 12-12
12.11.2 Access to the fields of a symbol 12-13
12.11.3 Variables 12-13
12.12 Numbers 12-13
12.12.1 16-bit integer numbers 12-14
12.12.2 Floating-point numbers 12-16
12.13 Vectors of Lisp pointers 12-17
12.13.1 Test for vector of pointers 12-17
12.13.2 Access to internal fields of a vector of pointers 12-18
12.13.3 Access to elements of a vector of pointers 12-18
12.13.4 Creation 12-18
12.14 Character strings 12-18
12.14.1 Test for character string 12-18
12.14.2 Access to the internal fields of a character string 12-18
12.14.3 Access to characters 12-19
12.14.4 Creation 12-19
12.15 Heap zone 12-19
12.16 Extending the Loader/Assembler 12-19
12.17 Functions 12-20
12.17.1 Types of functions 12-20
12.17.2 Function calling rules 12-20
12.18 Examples 12-20

13 Compilation **13-1**

13.0.1 Calling the compilers 13-1
13.0.2 Compiler macros 13-3
13.0.3 Closed macros 13-3
13.0.4 Open macros 13-4
13.0.5 Modules 13-6
13.0.6 Source files 13-11
13.0.7 Object files 13-12
13.0.8 Controlling evaluation 13-12

13.1 Use and manipulation of modules 13-15
13.1.1 Functions on modules 13-16
13.1.2 Loading and compiling modules 13-16
13.2 Complice 13-18
13.2.1 Compatibility messages 13-18
13.2.2 Errors and warnings 13-19
13.2.3 General remarks and examples 13-25

14 External interfaces **14-1**
14.1 Interface functions 14-1
14.2 Links with external procedures under Unix 14-5
14.2.1 Principles 14-5
14.2.2 Calling functions written in C 14-5
14.2.3 Calling Lisp from C 14-16

15 Virtual terminal **15-1**
15.1 Virtual terminal functions 15-2
15.1.1 Standard functions 15-3
15.1.2 Required functions 15-4
15.1.3 Optional functions 15-7
15.2 Screen functions 15-9
15.3 Using the virtual terminal 15-10
15.4 Defining a virtual terminal 15-11

16 Full-page editor **16-1**
16.1 Functions to call the full-page editor 16-1
16.2 Full-page editor commands 16-2
16.2.1 Extensions to the full-page editor 16-3

17 Terminal-based line editor **17-1**
17.1 Loading the terminal-based line editor 17-1
17.2 Terminal-based line-editor commands 17-2

18 Virtual bitmap display **18-1**
18.1 Loading the description file 18-1
18.2 Screen preparation 18-2

18.2.1 Managing a single screen 18-2
18.2.2 Managing several screens 18-3
18.3 Functions on screens 18-5
18.4 Functions on windows 18-9
18.4.1 Global coordinates 18-9
18.4.2 Local coordinates 18-9
18.5 Windows 18-10
18.5.1 Creating windows 18-11
18.5.2 Drawing in windows 18-14
18.5.3 Attaching properties to windows 18-15
18.5.4 Functions on windows 18-16
18.5.5 Primitive functions on windows 18-20
18.5.6 Minimal graphics primitives 18-21
18.5.7 Character strings 18-22

19 Virtual mouse **19-1**

19.1 Events 19-1
19.2 Structure and types of events 19-2
19.3 Mouse modes 19-5
19.4 Events queue 19-6
19.5 Programmable interrupts 19-8
19.6 Synchronous mouse tracking 19-9
19.7 Virtual menus 19-9
19.8 Cut and Paste 19-11

20 Graphics primitives **20-1**

20.1 Graphics environments 20-1
20.1.1 Current font 20-1
20.1.2 Foreground and background colors 20-2
20.1.3 Cursor 20-5
20.1.4 Line style 20-6
20.1.5 Fill patterns (or textures) 20-7
20.1.6 Drawing (combination) mode 20-7
20.1.7 Clipping zone 20-8
20.2 Graphics primitives 20-9

20.3 Extended graphics functions 20-10
20.3.1 Line-drawing functions 20-10
20.3.2 Fill functions 20-12
20.3.3 Displaying text 20-13
20.3.4 Bitmaps 20-13
20.3.5 Compatibility between types of bitmaps 20-18

FOREWORD

LE-LISP [Chailloux et al.] is a LISP system developed at INRIA (*Institut National de Recherche en Informatique et en Automatique*) in France. This dialect of the LISP language [McCarthy 62] is the 'spiritual child' of Vlisp [Greussay 77], [Chailloux 80], from which LE-LISP gets its conciseness and interpreter speed, and the 'natural child' of MacLisp or, more precisely, of the *Post-MacLisp* LISPs such as MIT Lisp Machine Lisp [Weinreb and Moon 81], NIL [White 79] [Burke and Carette 82] and Franz Lisp [Foderaro and Sklower 81], to which LE-LISP owes its strength as a language for the development of sound and powerful compiled applications.

0.1 History

In 1981, researchers at INRIA began to develop an ambitious VLSI design system, using a unique structure to represent all aspects of the project: graphics, simulation, etc. For the implementation language, LISP was an obvious choice. The INRIA designers were thinking, more precisely, of one of the powerful *Post-MacLisp* languages. Fortunately (or unfortunately, as the case may be), hardware was quickly evolving, and the VLSI project called upon many heterogeneous and incompatible machines. Various existing LISP systems—otherwise perfectly satisfactory—were simply not available on the required range of machines. For instance, at that time, Franz Lisp only existed on Vaxen. The implementation of the LISP system on a new computer had to be faster, of course, than the translation of the VLSI design system from one LISP dialect to the other. Out of this context grew the LE-LISP system, to fulfil well-identified goals. And, to this day, portability of the LISP system and its applications has remained a hallmark of LE-LISP.

0.1.1 Design goals

Efficiency and flexibility

LE-LISP is designed for maximum efficiency both in terms of execution resources (main memory and execution time) and development resources (application development time and workstation configurations).

Extensions and integration

The LE-LISP system allows you to make full use of existing system facilities by means of the concept of *virtual devices*. In this way, access to file-management procedures, graphic libraries, routines written in other languages and operating-system calls is provided with no sacrifice of portability. The user can create new instances of such facilities while retaining a standardized interface.

Portability of applications

Since all system facilities are accessed via their virtual counterpart, user applications are never required to contain implementation-specific details—even though they might call upon the full richness of the system.

Compatibility among implementations

Since the implementation of LE-LISP is based upon a virtual machine (called LLM3), only this virtual machine is ported from one real machine to another. The rest of the system is guaranteed to be compatible, because all implementations share the same code. A corollary is the rapid availability of top-quality implementations on new machines that appear on the market.

0.1.2 Outcome

All the above-mentioned goals were attained in the resulting system. The first LE-LISP system began to run—during the autumn of 1981—on an Exormacs (Motorola's machine based upon the 68000). That first implementation formed the basis of a VLSI design workstation that incorporated a colored bit-mapped display and a mouse [Chailloux et al.]. The system was then implemented on the VAX, under UNIX, during the autumn of 1982. After its successful beginnings, LE-LISP simply "went forth and multiplied".

0.1.3 Current state of the system

Today, LE-LISP is a rich LISP system that is implemented on more than thirty kinds of machines, based upon more than twelve CPUs running under a variety of operating systems. The industrial availability of all these versions is a reflection of the countless enhancements made to LE-LISP over the years. Indeed, since its inception, LE-LISP has been enlarged to include user-extensible generic arithmetic, an object-oriented type system, virtual graphic libraries, a modular compiler, full access to foreign routines and many other marvels.

More importantly, environments constructed using LE-LISP provide a huge array of functionalities. The application environments include numerous expert-system generators, simulation tools, database access tools, graphic development tools, music and acoustic research tools, symbolic mathematics systems, CASE environments and, of course (since it was the initial motivation behind the creation of the language), VLSI design environments.

0.1.4 Le-Lisp Version 15.26

LE-LISP Version 15.26 (December 1, 1993) is totally compatible (as well as backward compatible) with Lisp code - both interpreted and compiled.

The LLM3 virtual machine has been ported onto the HP/PA and DEC ALPHA processors.

Some new functions have been introduced : `copyfile`, `create-directory`, `delete-directory`, `map-expand-pathname`, `at-end` and `printf`.

The information issued from the function `gcinfo` has been fine tuned.

A new programmable interruption `at-end` is now available.

The `defextern` function systematically creates a test of type dynamic.

The new functions C `C_LL_FIX`, `LL_C_FIX`, `C_LL_FLOAT` and `LL_C_FLOAT` are available in the external interface.

0.2 Reader's guide

The present document is a *reference manual*. As such, it provides precise and complete information—rather than tutorial material—on the subject of LE-LISP. For newcomers to the language, excellent introductory books on LE-LISP exist already in French, English and other European languages.

Bibliography

[Abelson and Sussman] Harold Abelson and Gerald Jay Sussman. *The Structure and Interpretation of Computer Programs*. The MIT Press, McGraw Hill Book Company, Cambridge, 1985.

[Allen 78] John Allen. *The Anatomy of Lisp*. McGraw-Hill, 1978.

[Audoire 85] Louis Audoire. "Un Processeur Spécialisé mLLM3 sur SPS7", Actes des journées SM90. Versailles, Décembre 1985.

[Boston 85] First Common Lisp Standardization Meeting. Boston, December 1985.

[Brook and Gabriel 84] Rodney A. Brook and Richard P. Gabriel. "A Critique of Common Lisp", *1984 ACM Symposium on Lisp and Functional Programming*. Austin, Texas, July 1984.

[Burke and Carette 82] Burke and Carette. *NIL Notes for Release 0*. Massachusetts Institute of Technology, Cambridge, December 1982.

[Cayrol 83] Michel Cayrol. *Le langage LISP*. Cepadues Editions, Toulouse, 1983.

[Chailloux 80] Jérôme Chailloux. "Le modèle Vlisp : description, évaluation et interprétation", Thèse de 3ème cycle, Université de Paris VI. Paris, Avril 1980.

[Chailloux 83] Jérôme Chailloux. *Le-Lisp 80 version 12, le manuel de référence*, Rapport technique INRIA no 27. Rocquencourt, Juillet 1983.

[Chailloux et al.] Jérôme Chailloux, Matthieu Devin et Jean-Marie Hullot. "Le-Lisp : a Portable an Efficient Lisp System", *1984 ACM Symposium on Lisp and Functional Programming*. Austin, July, 1984.

[Chailloux 85a] Jérôme Chailloux. *Le-Lisp version 15, le manuel de référence*. Documentation INRIA, Rocquencourt, Février 1985.

[Chailloux 85b] Jérôme Chailloux. "La machine virtuelle LLM3", Rapport technique no 55, INRIA. Rocquencourt, Juin 1985.

[Cointe 82] Pierre Cointe. "Fermetures dans les lambda-interprètes. Application aux langages LISP, PLASMA et SMALLTALK", Thèse de 3ème cycle, Université de Paris VI. Paris, 1982.

[Dana 86] Michel Dana. "Le-Lisp v15.2 sous système VAX/VMS", Rapport ENST, Janvier 1986.

[Devin 85a] Matthieu Devin. "Le portage du système Le-Lisp : mode d'emploi", Rapport Technique no 50. INRIA, Rocquencourt, Mai 1985.

[Devin 85b] Matthieu Devin, "La Microprogrammation du système Le-Lisp : une première approche", Rapport de Recherche no. 441. INRIA, Rocquencourt, Septembre 85.

[Farreny 84] Henry Farreny. *LISP*. Masson, Paris, 1984.

[Foderaro and Sklower 81] Franz Lisp Manual. Univ. of California, Berkeley, September 1981.

[Gabriel 86] Richard Gabriel. *The Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, 1986.

[Girardot 85] Jean-Jacques Girardot. "Les langages et les systèmes LISP", édi tests, Paris, 1985.

[Greussay 77] Patrick Greussay. "Contribution à la définition interprétative et à l'implémentation des lambda-langages", Thèse, Université de Paris VI Paris, Novembre 1977.

[Hullot 83] Jean-Marie Hullot. "Ceyx, a Multiformalism Programming Environment" IFIP83, R.E.A. Masson (ed), North Holland, Paris 1983.

[Hullot 85a] Jean-Marie Hullot. "Programmer en Ceyx", Rapports techniques no 44-45-46. INRIA, Rocquencourt, Février 1985.

[Hullot 85b] Jean Marie Hullot. "Alcyone, La boîte à outils Objets", Rapport Technique no 60. INRIA, Rocquencourt, Novembre 1985.

[Kiremitdjian and Roy 85] Georges Kiremitdjian et Jean-Paul Roy. *Lire Lisp, le langage de l'Intelligence Artificielle*. Cedic-Nathan, Paris, 1985.

[Lang and Dupont 87] Bernard Lang et Francis Dupont. "Incremental Incrementally Compacting Garbage Collection", *ACM/SIGPLAN Symposium on Interpreters and Interpretive Techniques*. St. Paul, June 1987.

[LMDL 86] Jérôme Chailloux, éditeur. "Les Comptes Rendus des Mardis du Lisp", Rapports internes du projet VLSI. INRIA, Rocquencourt, 1986.

[McCarthy 62] John McCarthy. *LISP 1.5 Programmer's Manual*. M.I.T. Press, Cambridge, 1962.

[Padget et. al. 86] Julian Padget, Jérôme Chailloux, Thomas Christaller, Matthieu Devin, John Fitch, Tim Krumnack, Ramon Lopez, Eugen Neidl, Stephen Pope, Christian Queinnec, Luc Steels, Herbert Stoyan. "Desiderata for a standardization of LISP", *1986 ACM Conference on Lisp and Functional Programming*. Boston, August 1986.

[Queinnec 82] Christian Queinnec. *Lisp : langage d'un autre type*. Eyrolles, Paris, 1982.

[Queinnec 84] Christian Queinnec. *Lisp : mode d'emploi*. Eyrolles, Paris, 1984.

[Serpette et al 89] Bernard Serpette, Jean Vuillemin, and Jean-Claude Hervé "BigNum: A Portable and Efficient Package for Arbitrary Precision Arithmetic", DEC PRL, Paris, 1989.

[Spir 87] Eric Spir. "Implémentation d'un Glanneur de Cellules pour Le-Lisp Version 16", Rapport de DEA d'Informatique fondamentale de l'Université Paris VII. Paris, Septembre 1987.

[Steele 84] Guy L. Steele, Jr. *Common Lisp, The Language*. Digital Press, Bedford, 1984.

[Stoyan and Görz 84] Herbert Stoyan and Günter Görz. *Lisp, Eine Einführung in die Programmierung*. Springer-Verlag, Berlin, 1984.

[Stoyan et. al. 86] Herbert Stoyan, Julian Padget, Jérôme Chailloux, Thomas Christaller, Matthieu Devin, John Fitch, Tim Krumnack, Ramon Lopez, Eugen Neidl, Stephen Pope, Christian Queinnec, Luc Steels. "Towards a LISP standard", Proceedings of the 7th ECAI. Brighton, July 1986.

[Teitelman 83] Warren Teitelman. *Interlisp Reference Manual.* XEROX PARC. Palo Alto, October 1983.

[Vuillemin 87] Jean Vuillemin. "Exact real computer arithmetic with continued fractions", Rapport de recherche INRIA, no. 760. Rocquencourt, Novembre 1987.

[Weinreb and Moon 81] Daniel Wienreb and David Moon. *Lisp Machine Manual, Fourth Edition.* Artificial Intelligence Laboratory, M.I.T., Cambridge, July 1981.

[Wertz 85] Harald Wertz. *Lisp, une introduction à la programmation.* Masson, Paris, 1985.

[Winston and Horn 84] Henry Winston et Berthold P.K. Horn. *Lisp, 2nd Edition.* Addison Wesley, New York, 1984.

[White 79] Jon L. White. "NIL - a perspective", Proc. of the Macsyma User's Conference. Washington D.C., June 1979.

Table of contents

0.1 History 0-1
0.1.1 Design goals 0-1
0.1.2 Outcome 0-2
0.1.3 Current state of the system 0-2
0.1.4 Le-Lisp Version 15.26 0-2
0.2 Reader's guide 0-3

Function Index

Chapter 1

Use and installation

1.1 Le-Lisp on various systems

LE-LISP has been ported to the following processors:

- Motorola 88x00
- DEC VAX 11
- Intel 80386/80486/Pentium
- Ridge 32/SPS9
- IBM/RS 6000
- SPARC
- MIPS R4x00
- HP-PA
- DEC ALPHA

In the following list of machines on which LE-LISP works at present, the name returned by the `system` function is indicated between square brackets:

- APOLLO (MC680x0 base, under system Domain/OS, SysV et BSD `[apollo]`).
- DecStation 3100 and 5000 (MIPS base, under ULTRIX `[decstation]`).
- HP9000 300 et 400 series (MC680x0 base, under HP/UX `[hp9300]` `[hp9400]`).
- HP9700 (HP-PA base, under system HP/UX `[hp9700]`).
- Silicon Graphics IRIS (MIPS base, under system IRIX4 `[iris4d ]`).
- Silicon Graphics IRIS (MIPS base, under system IRIX5 `[irix5] `).
- PC Compatibles (under DOS system `[msdos]`).
- PC Compatibles (under Windows `[windows]`).
- PC Compatibles (under Windows NT `[nt386]`).

- PC Compatibles (under UNIX SCO `[sco386]`.
- PC Compatibles (under Solaris x86 `[solaris386]`).
- IBM/RS 6000 (RS6000 base or PowerPC under AIX `[rs6 000]`).
- SM90/SPS7 (MC68000 base, under sytems SMX et SPIX `[sm90]` —[spix]—).
- Ridge 32/SPS9 (under ROS `[sps9]`).
- SUN 3 (MC680x0 base, under SUN OS `[sun]`).
- SUN 4 (SPARC base, under SUN OS `[sun4]`).
- SUN 4 (SPARC base, under Solaris `[solaris]`).
- VAX 11 (under systems Ultrix and VMS `[vaxunix]` `[vaxvms]`).
- ALPHA (under systems OSF et VMS `[alphaosf]` `[alphavms]`).

1.2 Starting with Le-Lisp

To start up LE-LISP on any of the systems that support it, just type the command `lelisp` on the terminal. The system responds with

```
; Le-Lisp (by INRIA) version 15.26 (dd/mm/yy) [system]
; Standard modular system: << date of the core-image >>
=  << list of the pre-loaded system features >>
```

where `(dd/mm/yy)` is the last system-modification date and `[system]` is the type of the LE-LISP system being used. At this point, LE-LISP enters the main interactive loop, which reads an expression from the terminal, evaluates it, and prints its value, indefinitely. LE-LISP indicates that it is waiting to read an expression by printing the question-mark character `?` on the terminal at the beginning of each line. The value of an evaluation is printed preceded by an equal sign `=`.

Here is an example of a LE-LISP session run on a `Sun 4/75`:

```
% # we use the c-shell (csh)
% lelisp

; Le-Lisp (by INRIA) version 15.26 (17/Nov/91) [sun4]
; Standard modular system: Sat 28 Dec 91 19:34:39
= (31bitfloats edlin microceyx abbrev date debug setf pepe
       virbitmap virtty compiler pretty loader pathname
       defstruct callext module messages)
? ()                                ; the null list!
= ()
? (length (oblist))                 ; number of active symbols
= 3190
? (version)                         ; number of the current version
```

```
= 15.26
? (system)                              ; the system type
= sun4

? (+ 1 2 3 4)
= 10
? (defun fib (n)
?     (cond ((= n 1) 1)
?           ((= n 2) 1)
?           (t (+ (fib (1- n)) (fib (- n 2))))))
= fib
? (fib 20)
= 6765
? (time '(fib 20))
= 0.42
?
? (gcinfo t)                            ; the initial working space
= (gc 0 0 0 0 0 0 0 0 cons (32) symbol 5120 string 5120
vector 4096 float 0 fix 0 heap (256) code (1500))
? (gc t)                                ; remaining work space
= (gc 24 0 0 0 0 0 0 2 cons 25646 symbol 2027 string 2093
vector 4041 float 0 fix 0 heap (168) code (908))
? (time '(gc))                          ; the time a gc takes
= .14
? ^Lhanoi                               ; loading a library
= /usr/local/lelispv15.26/llib/hanoi.ll
? (hanoi 4)                             ; try this.. it's nice!
= hanoi
? (end)                                 ; to leave Le-Lisp.
Que Le-Lisp soit avec vous.
```

1.3 Starting the Le-Lisp system under Unix

The command used to start up LE-LISP under the UNIX operating system has the following form:

```
lelisp [n] [[-r] file]
```

The strings in square brackets are optional.

- The `n` argument is the size of the list zone in multiples of 8K list cells. By default, `n` has the value `4`. In other words, `32k` list cells are allocated. The theoretical limit on this value is `128`, for a maximum of `1024k` list cells. The practical limit is the physical memory size of the host machine, or the maximum process size of the host operating system.
- The `file` argument provides a way to specify the name of a file containing LISP programs to be loaded before the system enters its main `read-eval-print` loop.

- The `-r file` argument string lets you specify a core-image file to be loaded before the system enters its main interactive `read-eval-print` loop.

After loading the standard executable core-image file, LE-LISP automatically loads the file named `$HOME/.lelisp`.

1.4 Installation of the Le-Lisp system under Unix

This section covers LE-LISP systems running under UNIX: Vax-11 (Ultrix), DecStation (Ultrix), Sun 3 & 4 (SunOS), HP9300 (HP/UX), etc.

1.4.1 Installing the system

The LE-LISP system, version 15.26, is distributed on magnetic medium (1600-bpi `tar`-format tape, DMA or Streamer cartridge) that should be copied to disk in the system's *installation directory*. Usually this is the `/usr/local/lelisp` directory, or maybe the `/usr/local/lelispv15.26` directory when you want to keep several different versions of LE-LISP. When installed, the system occupies about twelve megabytes of disk storage.

Since a typical LE-LISP implementation is locked when you receive it, make sure that you have installed the access key that came along with the product.

The installation directory contains several sub-directories, one of which, called the *system directory*, is named according to the host machine (VAXUNIX, SUN4, etc.).

This system directory must contain a sub-directory called `llcore`, containing core-image files appropriate to the system involved.

```
$ ls -F /usr/local/lelispv15.26
README        ceyx/         llobj/        vaxunix/
LLUSERFILES   common/       lltest/       virbitmap/
TARUSER*      llib/         llub/         virtty/
benchmarks/   llmod/        manl/
```

It is also necessary for all users to have write permission in the `virtty` sub-directory. (Use `chmod a+w virtty`.)

Installation requires the initialization of some absolute pathnames and the construction of LE-LISP core-image files.

Initializing some absolute pathnames

Change directories (`cd`) to the system directory and execute the `newdir` command without any arguments. This command is to be done only once, after the system has been copied from the magnetic medium onto disk. If the LE-LISP system is ever moved to a new location in the file system, this procedure must be carried out again.

```
$ cd /usr/local/lelispv15.26/vaxunix
$ newdir
2000
DIR=/usr/local/lelispv15.26
SYSDIR=/usr/local/lelispv15.26/vaxunix
2000
2000
DIR=/usr/local/lelispv15.26
2000
20828
      (defvar #:system:directory "/usr/local/lelispv15.26/"))
20826
```

Building Le-Lisp core-image files

Change directories to the system directory and execute the `make` command with the name of the core-image file to construct as argument.

There are several entry points (targets) in the system makefile, allowing the construction of different core-image files (`lelisp-`, `lelisp`, `cmplc`, `lelispX11`) with different memory configurations (normal, +, ++). This makefile can be extended to accomodate the construction of new systems.

The makefile uses the `config` command, which builds a shell script that launches LE-LISP using the constructed core-image file. This shell script should be copied into a command directory (the directory `/usr/local/bin`, for instance) in the host file system.

Core-image files are stored in the `system/llcore` directory. Since they occupy a lot of disk storage space, it might be advantageous to mount the `llcore` directory onto a special disk partition.

```
$ cd /usr/ilog/lelispv15.26/sun4
$ make lelisp
./config lelisp lelispbin Lelispconf.ll -stack 6 -code 1500 -heap 256\\
-number 0 -vector 4 -string 5 -symbol 5 -cons 4 -float 0
; Le-Lisp (by INRIA) version 15.26  (17/nov/91) [sun4]
= (Version:  15.26)
= (Subversion:  1)
= unix system
 (load-std sav min pepe env ld llcp)  to load standard environment,
 (load-stm sav min pepe env ld llcp)  to load modular  environment,
 (load-cpl sav min meme env ld cmpl)  to load complice environment.
= /usr/ilog/lelispv15.26/llib/startup.ll
? (setq #:system:name (quote |lelisp|))
= lelisp
? (progn
    (load-stm #:system:name t t t t t)
    (add-feature (if (eq 0.0 0.0)
                     '31BITFLOATS
```

```
                    '64BITFLOATS))
    )
Loading loader.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/llpatch.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/messages.lm ... already loaded.
Loading /usr/ilog/lelispv15.26/llmod/path.lm ... already loaded.
Loading /usr/ilog/lelispv15.26/llmod/files.lm ... already loaded.
Loading /usr/ilog/lelispv15.26/llmod/module.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/defs.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/genarith.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/toplevel.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/cpmac.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/llcp.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/peephole.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/virtty.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/virbitmap.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/pepe.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/setf.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/defstruct.lm ... already loaded.
Loading /usr/ilog/lelispv15.26/llmod/sort.lm ... already loaded.
Loading /usr/ilog/lelispv15.26/llmod/array.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/callext.lm ... already loaded.
Loading /usr/ilog/lelispv15.26/llmod/trace.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/pretty.lm ... already loaded.
Loading /usr/ilog/lelispv15.26/llmod/debug.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/ttywindow.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/abbrev.lm ... done.
Loading /usr/ilog/lelispv15.26/llmod/microceyx.lm ... done.
Wait, saving: Standard modular system
; Le-Lisp (by INRIA) version 15.26 (17/Nov/91)   [sun4]
; Standard modular system : Mon 17 Dec  91 14:36:02
= (31bitfloats microceyx abbrev date debug setf pepe virbitmap virtty
compiler pretty loader pathname defstruct callext module messages)
? (end)
May Le-Lisp be with you.
$ cp lelisp /usr/local/bin
$
The installation is finished!
$
$ lelisp
; Le-Lisp (by INRIA) version 15.26 ( 2/Jan/91) [sun4]
; Standard modular system: Sat 29 Dec 90 19:34:39
= (31bitfloats edlin display date microceyx debug setf pepe virbitmap
virtty compiler pretty abbrev loader callext defstruct pathname messages)
?
```

Generally, the installation just described suffices. For certain applications, it is sometimes necessary to modify the configuration of the system, that is, the set of LISP files loaded in as the core-image, or to fine-tune the organization of the LISP memory space.

1.4.2 Modification of the system configuration

The standard distribution allows the system to be built in three different configurations, which are not exclusive, and which correspond to different `makefile` entry points:

- `lelisp`
  Complete environment with debugging tools, an editor and the standard compiler.
- `cmplc++`
  Complete environment with debugging tools, an editor and the modular COMPLICE compiler used by the `complice` command.

The exact composition of each of these system configurations is described in a file in the `conf` sub-directory: `lelispconf.ll` and `cmplcconf.ll`. You can change a configuration by editing one of these files. It is also possible to create a new entry point in the `makefile` describing a new system configuration.

Example:

Construction of a core-image file named `mylisp` containing the standard environment without an editor, but with the `edlin` line editor and with the `scheduler` scheduler functions. This new system can be started up by the command `mylisp`.

A configuration file (for example, `conf/mylispconf.ll`) must be created.

```
$ cat conf/mylispconf.ll
(load-std ()       ;    load the environment without backup
          t        ;    the minimum environment,
          ()       ;    no editor,
          t        ;    the complete environment,
          t        ;    the loader,
          t)       ;    and the compiler.
(libload edlin)    ;    load the edlin line editor
(libload schedule) ;    and the scheduler.
(progn
   (llcp-std #:system:name)    ; compilation and construction of the image
   (edlin)                     ; execution of edlin after initialization
   "Welcome to my-lisp"))      ; welcome message.
```

Then add the entry point (target) `mylisp` to the makefile in the system directory. The `mylisp` system is built using the `lelispbin` standard system (no supplementary C modules), and the standard `SIZE` memory zone sizes.

```
mylisp:        mylispconf.ll
        ./config mylisp lelispbin mylispconf.ll $(SIZE)
```

After this is done, the system can be built by executing the `make mylisp` command.

```
$ make mylisp USERLELISP=monlelisp USERLELISPBIN=mylispbin
USER0=monc.o cc -o monlelispbin \
        o/llnumb.o o/llmain.o o/llstdio.o o/llfloat.o \
        o/lelisp.o o/getgloba.o lelisp31bin.o \
        monc.o \
          -z -x  -A systype,bsd4.3 -A runtype,bsd4.3
./config monlelisp monlelispbin monlelispconf.ll  -stack 6 -code 600 \\
-heap 256  -number 0 -vector 4 -string 5  -symbol 5 -cons 4 -float 0
; Le-Lisp (by INRIA) version 15.26  (17/nov/91)  [apollo]
= (Version:  15.26)
= subversion
= herald
= defvar
= syste‘me unix
 (load-std sav min pepe env ld llcp)  to load standard environment,
 (load-stm sav min pepe env ld llcp)  to load modular environment,
 (load-cpl sav min meme env ld cmpl)  to load complice environment.
= /usr/ilog/lelispv15.26/llib/startup.ll
? (setq #:system:name (quote |monlelisp|))
= monlelisp
? (load-stm ()       ;    load environment without backup,
?           t        ;    minimum environment,
?           ()       ;    no editor,
?           t        ;    complete environment,
?           t        ;    loader,
?           ())      ;    no compiler
Loading loader.lm
Loading /usr/ilog/lelispv15.26/llmod/llpatch.lm
Loading /usr/ilog/lelispv15.26/llmod/module.lm
Loading /usr/ilog/lelispv15.26/llmod/defs.lm
Loading /usr/ilog/lelispv15.26/llmod/genarith.lm
Loading /usr/ilog/lelispv15.26/llmod/toplevel.lm
Loading /usr/ilog/lelispv15.26/llmod/virtty.lm
Loading /usr/ilog/lelispv15.26/llmod/virbitmap.lm
Loading /usr/ilog/lelispv15.26/llmod/setf.lm
Loading /usr/ilog/lelispv15.26/llmod/defstruct.lm
Loading /usr/ilog/lelispv15.26/llmod/sort.lm
Loading /usr/ilog/lelispv15.26/llmod/array.lm
Loading /usr/ilog/lelispv15.26/llmod/callext.lm
Loading /usr/ilog/lelispv15.26/llmod/trace.lm
Loading /usr/ilog/lelispv15.26/llmod/pretty.lm
Loading /usr/ilog/lelispv15.26/llmod/debug.lm
Loading /usr/ilog/lelispv15.26/llmod/ttywindow.lm
Loading /usr/ilog/lelispv15.26/llmod/abbrev.lm
```

```
Loading /usr/ilog/lelispv15.26/llmod/microceyx.lm
= ()
? (libload edlin)
= /usr/local/lelispv15.26/llib/edlin.ll
? (libload schedule)
= /usr/local/lelispv15.26/llib/schedule.ll
? (progn (save-std #:system:name)
?        (edlin)
?        "Welcome to my lisp")))
Wait, saving: Standard modular system
; Le-Lisp (by INRIA) version 15.26 ( 2/Jan/91) [vaxunix]
; Standard modular system: Sat 29 Dec 90 19:34:39
= (31bitfloats edlin display date microceyx debug setf pepe virbitmap
virtty compiler pretty abbrev loader callext defstruct pathname messages)
? (end)
Que Le-Lisp soit avec vous.
$
$ cp mylisp /usr/local/bin
$ mylisp
; Le-Lisp (by INRIA) version 15.26 ( 2/Jan/91) [vaxunix]
; Standard modular system: Sat 29 Dec 90 19:34:39
= Welcome to my lisp
?
```

1.4.3 Modification of data zone sizes

The sizes of the different system data zones are fixed during the construction of each core-image file, and they cannot be modified dynamically. It is quite possible to saturate one of these zones, which causes one of the fatal errors described in section 7.10:

```
*** fatal error : no room for XXXXs.
```

Notice the following points:

- The size of the list storage zone can be modified at each invocation of the system. So, it is not necessary to construct a new core-image file in order to rectify a situation that causes the `list-memory-zone-full` error.
- The saturation of a zone could well be due to an error in a user program.
- The LISP function `gcinfo` gives information on how full memory zones are.

The `makefile` has entry points that permit the construction of each system configuration (`lelisp`, `cmplc`) with different zone sizes. For instance the system `lelisp++` is a LE-LISP system with very big memory zones.

The makefile's `SIZE`x variables contain the values indicating the size of the different zones.

For example, the size of the LE-LISP system is described by the `SIZE` variable:

```
SIZE= -stack 6 -code 256 -heap 100 -vector 3 -number 0 -float 0
      -string 4 -symbol 3 -cons 4
```

Each `-zone` option sets the size of `zone`. Here are the units that describe these sizes:

LE-LISP memory zones

Name	LISP object	Unit	Real size
`stack`	Stack	K-word	1 = 4 K-byte
`code`	Compiled code	K-byte	1 = 1 K-byte
`heap`	Heap	K-byte	1 = 1 K-byte
`vector`	Vectors	K-vector	1 = 1 K-vector (8Kb)
`float`	Floating-point numbers	K-float	1 = 1 K-float (8Kb) (*)
`string`	Character strings	K-string	1 = 1 K-string (8Kb)
`symbol`	Symbols	K-symbol	1 = 1 K-symbol (64 Kb)
`cons`	Pairs	8K-pair	1 = 1 K-pair (64 Kb)

(*) set to 0 for 31-bit float system.

Strings and vectors take up space in the `heap`, as indicated in the following table:

Heap use

n-object vector	8+4n bytes
Character string	9+n bytes

To build a core-image file with new zone sizes, the definitions of the `SIZE`*x* variables must be changed, or new entry points (targets) using other `SIZE` parameters must be added to the makefile, and then the `make` command must be executed again.

Example:

To add an entry point that constructs a system named `lelispv` with 10,240 (`10K`) vectors.

```
SIZEV= -stack 6 -code 256 -heap 100 -vector 10 -number 0 -float 1
       -string 4 -symbol 3 -cons 4

lelispv:       lelispconf.ll
        ./config lelispv lelispbin lelispconf.ll $(SIZEV)
```

Next, the execution of the `make` command builds the core-image file for the `lelispv` system.

```
$ make lelispv
./config lelispv ...
..
$ cp lelispv /usr/local/bin
$ lelispv
; Le-Lisp by INRIA ...
...
```

1.4.4 Linking the Le-Lisp system with C modules

Chapter 14 of this manual describes how to link the `lelispbin` system with C programs to make a new executable file.

To perform this operation, the `lelispbin.o` binary file in the system directory must be used. This file is the result of a link (using `ld -r`) of all of the LE-LISP system's constituant modules, except the C modules. These last modules are obtained by compiling the C files in the `common` directory.

The generic entries of the Makefile allow you to do that in a very homogeneous manner. See the on-line UNIX information called `lelisp-manl`.

1.4.5 Calling the shell

The *shell* can be called by using the `comline` function (and the `!` macro character). This will execute `/bin/sh` which a local interpretation of the `cd` command.

```
% lelisp
; Le-Lisp (by INRIA) version 15.26 ( 2/Jan/91) [vaxunix]
; Standard modular system: Sat 29 Dec 90 19:34:39
= (compiler debug defstruct loader pepe pretty virbitmap virtty)
?
? !pwd
/usr/local/lelispv15.26/vaxunix
= t
? !cd ../
= t
? !pwd
/usr/local/lelispv15.26
.....
```

1.5 Starting the Le-Lisp system under VMS

Complete documentation covering the installation and execution of LE-LISP under VMS is available in [Dana86]. The command used to start the LE-LISP system under the VMS operating system has the following form:

```
$ lelisp
```

1.6 Practical advice

Finally, here is some practical information that will be developed in detail in the chapters to come. In particular, we give hints about how to use the system comfortably right from the very start.

- To erase a character, use the BACKSPACE key or the LEFT arrow. The character to the left of the cursor will be erased from the screen.
- To kill a line of input, type CONTROL-`X`, obtained by holding down the CONTROL key and pressing the `X` key. We write this key combination as `^X`. The line is erased from the screen. You can obtain the same result by typing `^U`.
- To return to the main interactive loop when you are in the midst of typing an expression, provoke an error by typing two dots (periods), one right after the other. If nothing (visible) happens, you are probably within a comment or a character string.
- To return to the main interactive loop when you have lost control of things, or LISP is no longer responding, type BREAK, DELETE or `^C` on UNIX hosts, or `^C` under VMS.
- To return to the host OS in really desperate circumstances, enter `^\` on UNIX systems, or `^Y` under VMS. In this case you loose your LISP environment.
- To send a command line to the host system (if the latter permits this kind of request), type the exclamation point character `!` followed by the command, as shown here:

  ```
  ? !dir
  ```

- To load a previously-created file, simply type

  ```
  ? ^Lfile
  ```

  To obtain CONTROL-`L`, hold down the CONTROL key and type `L` simultaneously. Follow this immediately by the name of the file, without its extension, which is `.ll` by default.

- To edit or create a file, call one of the full-screen editors by simply typing

  ```
  ? ^Efile
  ```

  To obtain CONTROL-`E`, hold down the CONTROL key and type `E` simultaneously. Follow this by the name of the file, without its extension, which is `.ll` by default.

- To see a replay of the commands recently executed, use the following command from within the editor:

  ```
  ESC ?
  ```

- To edit a particular function that is in a file, call one of the screen editors by simply typing

  ```
  ? ^Ffunction
  ```

  To obtain CONTROL-`F`, hold down the CONTROL key and type `F` simultaneously. Follow this by the name of the function. The system will try to locate the file that contains it, and will call one of the available full-screen editors on this file, as with the `^E` command.

- To insert a comment into a LISP expression, type a semi-colon. The rest of the line, up to the end-of-line character, will be ignored.

Finally, you have access to the source code of a large number of utilities that are written in LISP. The editor, the pretty-printer, debugging tools, visualization tools, etc. are included in the distribution. Go ahead and read them, try to understand them, and do not be afraid to extend and improve them.

Table of contents

1 Use and installation **1-1**

1.1 Le-Lisp on various systems 1-1

1.2 Starting with Le-Lisp 1-2

1.3 Starting the Le-Lisp system under Unix 1-3

1.4 Installation of the Le-Lisp system under Unix 1-4

1.4.1 Installing the system 1-4

1.4.2 Modification of the system configuration 1-7

1.4.3 Modification of data zone sizes 1-9

1.4.4 Linking the Le-Lisp system with C modules 1-11

1.4.5 Calling the shell 1-11

1.5 Starting the Le-Lisp system under VMS 1-11

1.6 Practical advice 1-11

Function Index

Chapter 2

Evaluation

2.1 Basic objects

The LE-LISP language operates on objects called **symbolic expressions**, generally referred to as *S-expressions*.

An S-expression can have any one of the following types:

- **Atomic objects**
  - Symbol.
  - Number: an integer, a floating-point number—often referred to simply as a *float*—or an arbitrary-precision number.
  - Character string.
- **Composite objects**
  - List.
  - Vector.

LE-LISP also allows you to define new types, called *extended types*.

Inside the machine, every S-expression is represented by a pointer to its value. You can think of this pointer as the address of the value. The value of an object is always accessed by means of an *indirection*. Consequently, LE-LISP is optimized for pointer manipulation.

LE-LISP runs principally on processors with 32-bit pointers. This allows you to manipulate addresses up to 2^{32}: that is, in the four-gigabyte region. Hardware limitations sometimes restrict the address space to only 24 or even 20 bits. However, in most cases, the address space of this kind of processor is adequate.

2.1.1 Atomic objects

Symbols

Symbols play the rôle of identifiers that are used to name variables, functions and labels. They are created automatically when read from the input stream. They can also be created explicitly by the

`symbol`, `implode`, `concat` and `gensym` functions. So, they do not need to be declared.

The external name of a symbol—referred to as its *print name*, abbreviated to `p-name`—can be any string of no more than 128 characters containing at least one non-numeric character. You can include special characters or delimiters in a `p-name` by surrounding the entire `p-name` by the character referred to as *absolute value bar*: represented by a vertical bar. (See the section on the standard reader.)

A symbol is represented in the system by a pointer to a descriptor stored in a special memory zone. This descriptor has the following nine *intrinsic properties*:

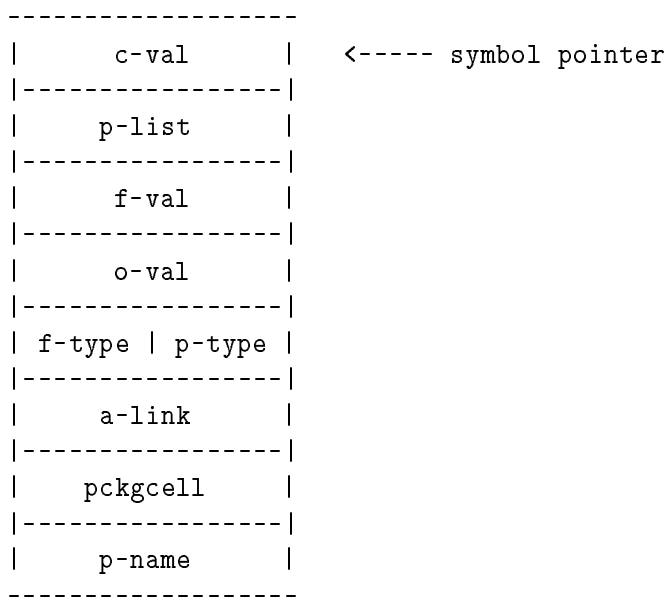
- `c-val` (an abbreviation for *cell value*) always contains the value associated with a symbol that is considered as a variable. Access to this value is extremely rapid. When a symbol is created, its `c-val` is *undefined*. Any attempt to reference a symbol that has not had a value assigned to it raises the `errudv` error.
- `p-list` (an abbreviation for *property list*) always contains the property list of the symbol. These properties are managed by the user by means of special functions that operate on P-lists: `addprop`, `putprop`, `getprop`, `remprop` and `defprop`. By default, the list of properties has the value `()`.
- `f-val` (an abbreviation for *function value*) always contains the value associated with a symbol that is considered as a function. This value can be of two kinds:
  - In the case of `subr` functions, it is a machine address.
  - In the case of `expr`, `fexpr`, `macro` and `dmacro` functions, it is a list.

  You can fully access the `f-val` of a symbol by means of the `valfn`, `setfn`, `remfn` and `getdef` functions. If the symbol has no function definition this property has the value `0`.
- `f-type` (an abbreviation for *function type*) contains the type of the function stored in the `f-val` field. Together, the `f-val`/`f-type` combination enables the evaluator to execute function calls very rapidly. You can access directly the `f-type` of symbols by means of the `typefn` and `setfn` functions. When a symbol does not have a function definition, this property has the value `()`.
- `p-type` (an abbreviation for *print type*) contains necessary information for the edition of the external representation of the symbol. There are two possibilities:
  - The symbol is a variable. The `p-name` string can be enclosed within a pair of *absolute value* characters.
  - The symbol designates a function. In this case, the pretty-printer uses the appropriate print format.

  The `p-type` cell of a symbol is accessed by means of the special `ptype` function.
- `o-val` (an abbreviation for *object value*) can contain any S-expression, and can be used to hold special values. This field is particularly useful in the implementation of object-oriented extensions.

- `a-link` (an abbreviation for *atom link*) contains the address of the next symbol in the symbol table. Among other things, this link facilitates the hashing of the symbol table. This attribute cannot be accessed directly by the user.
- `pckgcell` (an abbreviation for *package cell*) contains the name of the package to which the symbol belongs. The package is accessed by means of the `symbol` and `packagecell` functions.
- `p-name` (an abbreviation for *print name*) contains the address of the character string that represents the name of the symbol.

These intrinsic properties are stored in memory with the following layout:



Certain symbols are defined at system-initialization time:

- Symbolic constants that contain their own names as values: `||`, `t`, `lambda`, `flambda`, `mlambda`, `subr0`, `subr1`, `subr2`, `subr3`, `nsubr`, `fsubr`, `expr`, `fexpr`, `macro`, `dmacro` and `quote`.
- Predefined functions.
- System variables.

Numbers

LE-LISP uses 16-bit integers, allowing calculations in the range of -2^{15} to $+(2^{15})-1$: that is, from -32768 to $+32767$. Floating-point numbers are composed of either 31, 32, 48 or 64 bits, depending on the implementation. LE-LISP also incorporates libraries for arbitrary-precision arithmetic.

The `p-name` of a number is the representation of its value in the output conversion base. (See the `obase` function.) The *value* of a number is, of course, the number itself.

Character strings

LE-LISP has character strings, which have the sequence of characters enclosed in quote characters as external representations. The quote character can be represented in strings by inserting a sequence of two quote characters in a row. A character string cannot be longer than 32767 characters in the current implementation. Character strings are stored in a special memory segment which is dynamically compacted in by a linear-time garbage-collection algorithm. The value of a character string is the string itself, thus there is no need to *quote* it.

Each character string can also have its own particular type. By default, the type of a string is `string`.

`"foo bar"`	corresponds to the string	`foo bar`
`"""abc"""`	corresponds to the string	`"abc"`
`""""`	corresponds to the string	`"`

All these three examples are of type `string`.

`#:foo:"abc"`	corresponds to the string	`abc`

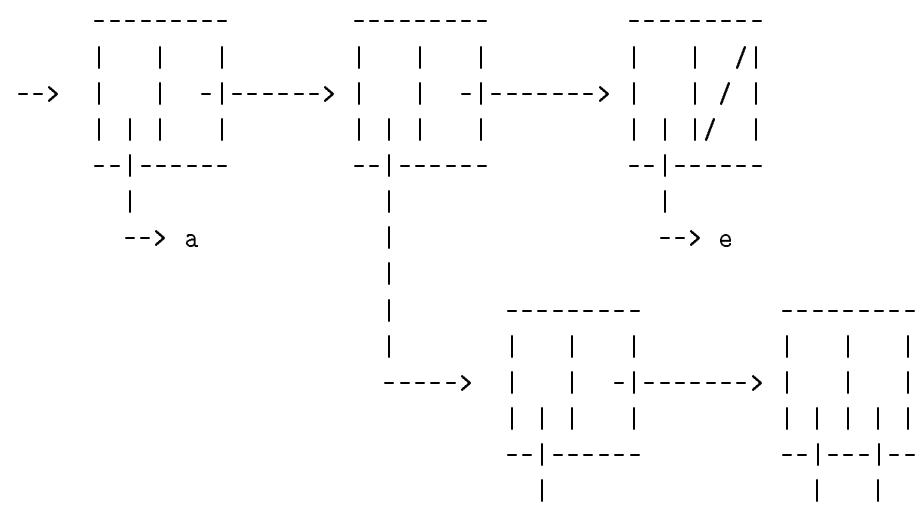
In this fourth example, on the contrary, the type is `foo`.

2.1.2 Compound objects

Objects are said to be *compound* when they are made up of other LE-LISP objects.

Lists

LE-LISP represents lists in a standard fashion. The following diagram shows, for example, how the list `(a (b c . d) e)` is stored in memory:



A list element is stored in a *list cell* made up of a pair of pointers. It is often referred to as a `cons` cell. The first member, or `car`, of this cell contains a pointer to the list element. Its second member, or `cdr`, contains a pointer to the next list element, or to a special end-of-list marker. LE-LISP uses the symbol `||`, whose name has a length of zero, as an end-of-list marker.

LE-LISP allows a list cell to be labelled. A label is a special mark that can be tagged on to any list cell. The label can be removed from the cell to which it is attached. Also, you can search for a particular labelled cell. These operations are carried out by specialized functions such as `tcons`, `tconsp`, `tconsmk` and `tconscl`. The mark is invisible to all other list-manipulation functions, and neither slows nor otherwise modifies normal access to list cell members. Among other things, these labels allow you to define new user types. (See the section on extended types.)

Vectors of S-expressions

LE-LISP has a *vector of S-expressions* type, which allows for indexed access to LISP objects. The external representation of a vector is

`#[`s_1 s_2 ... s_n`]`

The vector elements are shown here as s_1 s_2 ... s_n. Our use of subscripts aims solely at improving the readability of this manual, and should not be taken literally, since there are no real subscripts in LE-LISP. Access to a vector element is very rapid. In the current implementation, a vector can contain no more than 32767 elements. Since vectors are authentic LISP objects, vectors of vectors are allowed. Vectors are stored in a special memory space that is dynamically compacted by a linear-time garbage-collection algorithm. Since the value of a vector is the vector itself, there is no need to *quote* them. Each vector can also have its own particular type. By default, the type of a vector is `vector`.

For example, `#[a b #[x y z] d e]` is a vector with five elements, of which the third is a vector of three elements. And `#:foo:#[1 2]` is a vector of two elements, of type `foo`.

2.2 Basic evaluation actions

2.2.1 Evaluation of atomic objects

The value of a symbol considered as a variable is its `c-val`. The evaluation of a symbol whose `c-val` is undefined—that is, a symbol that does not yet have a value—raises the `errudv` error at evaluation time. Its screen display is

```
** <fn> : undefined variable :  <symb>
```

Here, the name of the symbol at fault is `symb`, and `fn` is the name of the function that caused the error. Usually, it is one of the two evaluation functions: `eval` or `symeval`.

Variables are used in LISP in three circumstances:

- As *global variables*, which are constantly accessible by all functions. It is recommended to initialize them with the `defvar` function. (See the following chapter.)
- As *local variables*, which only retain their values during the execution of a function. Indeed, they are the parameters of the function.
- Finally, they can be used in the style of the `own` variables of ALGOL. In this case, they are local to a function, but they do not loose their value between successive evaluations of this function. These variables must be enclosed within a function by means of the `closure` function.

The value of a number or a character string is the number or character string itself. So, there is no need to *quote* them.

2.2.2 Evaluation of composite objects

The evaluator always considers that a list is a function call. The list is referred to as a *form*. The `car` of the form is the *function*, and its `cdr` is the *argument list* of the function.

The value of a form is the value returned when the function is applied to its arguments.

The value of a vector of S-expressions is the vector itself. As in the case of numbers and character strings, it is not necessary to *quote* vectors, which are always treated like constants.

2.3 Evaluation of functions

A function—the `car` of a form—can be a symbol or a special list. The `cdr` of the form is the argument list of the function. If this list of arguments is not terminated by `()`, the `errbal` error is raised. Its screen display is

```
** <fn> : bad arguments list : <a>
```

Here, `fn` is the name of the function that caused the error, and `a` is the final `cdr` of the list of arguments.

```
(cons 1 . 2)      ⟹      ** cons : bad argument list : 2
(if () 2 3 . 4)   ⟹      ** if : bad argument list : 4
```

If the function is a symbol, LE-LISP uses the function associated with the `f-val` of this symbol.

- This association can be established at system initialization. This is the case for predefined functions, which are also called *standard functions*.
- The user can make this association by means of static or dynamic definition functions.

If, even after searching through the extended types (see that section), no function has been associated with this symbol, the `errudf` error is raised. Its screen display is

```
** <fn> : undefined function : <symb>
```

Here, `symb` is the name of the undefined symbol, and `fn` is the function that caused the error. It is usually one of the functions `eval`, `apply` or `funcall`.

```
(cons 'a 'b)        ⟹   (a . b)
(setq kons 'cons)   ⟹   cons
(kons 'x 'y)        ⟹      ** eval : undefined function : kons
```

When the function is a number, a vector or a character string, the evaluator also raises the `errudf` error, whose screen display is the same as in the preceding case.

```
(3 '(1 2 3))   ⟹      ** eval : undefined function : 3
```

When the function is a list, an anonymous function is explicitly declared.

The first element of the list must be one of the special symbols `lambda`, `flambda` or `mlambda`.

```
((lambda (x) (+ x x 2)) 5)        ⟹   12
((flambda (x) x) (+ x x))         ⟹   (+ x x)
((lambda (x) x) (lambda (x) x))   ⟹   (lambda (x) x)
```

A calculated function call is made by using the `funcall` function. LISP is one of the rare languages in which it is possible to write convenient calls of the form

```
(funcall (if (< n 0) '* '+) val 2)
```

LE-LISP has two main classes of functions: those written in the LLM3 machine language, and those written in LISP. All functions written in LISP can be translated into LLM3 machine language by the compiler. Each of these two classes of functions has four types of functions:

- Functions that evaluate their arguments: called `subr` in LLM3 and `expr` in LISP.
- Functions that do not evaluate their arguments: called `fsubr` in LLM3 and `fexpr` in LISP.
- Simple macro functions: called `msubr` in LLM3 and `macro` in LISP.
- Substitution macro functions: called `dmsubr` in LLM3 and `dmacro` in LISP.

2.3.1 Functions of the `subr` kind

Functions of the `subr` kind are written in LLM3 machine language. The arguments of these functions are always evaluated. Some `subr` functions have a fixed numbers of arguments. Depending on the precise number of arguments, they are referred to as either `subr0`, `subr1`, `subr2` or `subr3`. Other `subr` functions have a variable numbers of arguments, and are sometimes referred to as `nsubr` functions. In the case of `subr` functions with a fixed number of arguments, the evaluator checks whether the correct number of arguments has been passed to the function. If not, the `errwna` error is raised. Its screen display is

```
** <fn> : wrong number of arguments : <n>
```

Here, `fn` is the name of the `subr` that was called, and `n` is the number of arguments required by this function.

In the case of `subr` functions with a variable number of arguments, the evaluator sometimes checks whether a minimum number of arguments required by the `nsubr` is passed in the function call. If this is not the case, the `errwna` error is raised. Its screen display is the same as that for `subr` functions.

It is possible to add `subr` functions to the system by writing them directly in LLM3 machine language, or by compiling appropriate `expr` functions.

There are many `subr` or `nsubr` functions in the system: between 400 and 500, depending on the system being used.

```
(cons)                  ==>    ** cons : wrong number of arguments : 2
(cons 'a)               ==>    ** cons : wrong number of arguments : 2
(cons 'a 'b)            ==>  (a . b)
(cons 'a 'b 'c)         ==>    ** cons : wrong number of arguments : 2
(apply 'cons '())       ==>    ** cons : wrong number of arguments : 2
(apply 'cons '(a))      ==>    ** cons : wrong number of arguments : 2
(apply 'cons '(a b))    ==>  (a . b)
(apply 'cons '(a b c))  ==>    ** cons : wrong number of arguments : 2
(funcall 'cons)         ==>    ** cons : wrong number of arguments : 2
(funcall 'cons 'a)      ==>    ** cons : wrong number of arguments : 2
(funcall 'cons 'a 'b)   ==>  (a . b)
(funcall 'cons 'a 'b 'c) ==>   ** cons : wrong number of arguments : 2
```

2.3.2 Functions of the `fsubr` kind

`fsubr`s are also functions written in machine language. They are resident in the system from initialization time onward, and they are executed very rapidly. These functions take a variable number of arguments, which are never evaluated. In certain cases, though, these arguments are evaluated by the function itself. These very special functions are used primarily as control functions or for name manipulation and are often referred as *special forms*. There are relatively few `fsubr` functions in LE-LISP.

It is possible to add functions of this type by writing them directly in LLM3 machine language, or by compiling appropriate `fexpr` functions.

2.3.3 Functions of the `msubr` kind

Functions of the `msubr` kind are also written in LLM3 machine language. They are created by compiling `macro` functions.

These functions have a variable number of arguments, which are never evaluated.

2.3.4 Functions of the `dmsubr` kind

Functions of the **dmsubr** kind are also written in LLM3 machine language. They are created by compiling **dmacro** functions.

These functions have a variable number of arguments, which are never evaluated.

2.3.5 Functions of the `expr` kind

Functions of the **expr** kind are written in LISP. They are evaluated by the functions: **eval**, **apply** and **funcall**. According to their definition, these functions expect a certain number of arguments, represented by a list or a tree of **lvar** variables. They also expect a function body, made up of a sequence of expressions $s_1 \ldots s_n$, to be evaluated.

A function of this kind is written by using a list, called a **lambda**-expression, of the form

(lambda lvar $s_1 \ldots s_n$)

Here, the **lambda** symbol designates an **expr**-type function, **lvar** is the list or tree of parameters, and $s_1 \ldots s_n$ is the body of the function.

A typical example is `(lambda (x y) (cons (car x) (cdr y)))`.

The definition of an **expr**-type function consists of associating a **lambda**-expression with a symbol. This is done by means of the **defun** function.

The evaluation of an **expr**-type function call takes place in three steps:

1. After current values of function parameter names are saved on the stack, the argument values are bound to the function parameter names. Calls to **expr**-type functions are by value.
2. The expressions in the function body $s_1 \ldots s_n$ are evaluated. The value returned by the function is the value of the last evaluation: that is, the result of the evaluation of s_n.
3. The bindings performed in step 1 are undone. The previous values of the parameter names, saved on the stack, are restored.

The binding of argument values to function parameters is carried out by a recursive procedure that treats the list of **lvar** parameters as a tree. This binding takes place between the leaves of the parameter tree and the list of argument values. This kind of binding allows the value of an argument to be split, when the function is called, between different parameter variables.

If there are too few or too many elements in the argument list after the binding is performed, the **errwna** error is raised. Its screen display is

```
** <fn> : wrong number of arguments : <s>
```

Here, **fn** is the name of the function that was called, or **lambda** in the case of an anonymous function. If too many arguments were supplied, **s** is the remainder of the argument list. If there were too few, **()** is displayed.

If a value cannot be bound to a parameter tree, the **errilb** error is raised. This happens if an attempt is made to bind an atomic value of the list of argument values to a non-atomic parameter. The screen display is

```
** <fn> : illegal binding : (<p> <v>)
```

Here, `fn` is the name of the function that was called, or `lambda` in the case of an anonymous function, and `(p v)` is a list of the parameter tree `p` and the value `v` that could not be bound.

If the parameter tree contains no variables, the `errbpa` error is raised. Its screen display is

```
** <fn> : bad parameter : <s>
```

Here, `fn` is the name of the function that was called, or `lambda` in the case of an anonymous function, and `s` is the faulty parameter.

If an error occurs during the operation of the extended binding mechanism, no parameter binding is performed.

Here is the extended binding mechanism, written in LISP:

```
(defun bindvar (tvar lval)
    ; tvar : the parameter tree
    ; lval : the list of argument values
    ; push (using the push function) the
    ; "value - variable" pairs
    (cond ((null tvar)
              (when lval
                    (error 'eval 'errwna lval)))
          ((variablep tvar)
              (push (symeval tvar))
              (push tvar)
              (set tvar lval))
          ((consp tvar)
               (if (atom lval)
                   (error 'eval 'errilb (list tvar lval))
                   (bindvar (car tvar) (car lval))
                   (bindvar (cdr tvar) (cdr lval))))
          (t (error 'eval 'errbpa tvar))))
```

Here are some examples of `expr` binding:

```
? ((lambda (x y z) (list x y z)) (1+ 1) (1+ 2) (1+ 3))
= (2 3 4)   ? ((lambda (x y z) (list x y z)) (1+ 1))
** lambda : wrong number of arguments : ()
? ((lambda (x y z) (list x y z)) 1 2 3 4 5)
** lambda : wrong number of arguments : (4 5)
? ((lambda x x) (1+ 1) (1+ 2) (1+ 3))
= (2 3 4)   ? ((lambda (x y . z) (list x y z))
?     (1+ 1) (1+ 2) (1+ 3) (1+ 4))
= (2 3 (4 5))   ? ((lambda ((x . y) . z) (list x y z)) (cons 1 2) 'c))
= (1 2 (c))   ? ((lambda (x (y . z)) (list x y z)) 'a 'b)
** lambda : illegal binding : ((y . z) b)
```

```
? ((lambda (t 1) (list x)) 'a 'b)
** lambda : bad parameter : t
? ((lambda (x 1) (list x)) 'a 'b)
** lambda : bad parameter : 1
```

2.3.6 Functions of the `expr` kind with a `&nobind` argument

If the argument list of an `expr` function is made up uniquely of the keyword `&nobind`, then binding is carried out in a different manner. No variables are bound. The actual number of arguments of the call is returned by the `(arg)` function. The arguments are retrieved using the `(arg n)` function, where `n` is the number of the argument, starting with `n = 0` for the first argument. This feature allows for the definition of functions with a variable number of arguments without the necessity of building an argument list.

This construction resembles `lexpr` function of MacLisp.

```
? (defun gog &nobind
?    (list (arg) (arg 0) (arg 1)))
= gog    ? (gog 10 11)
= (2 10 11)    ? (gog 10 11 12 13 14 15)
= (6 10 11)    ? (defun magog &nobind
?     (if (> (arg) 2)
?         (error 'magog "unexpected arguments" (arg))
?         (let ((arg1 (if (< (arg) 1) 'default1 (arg 0)))
?               (arg2 (if (< (arg) 2) 'default2 (arg 1))))
?              (list arg1 arg2))))
= magog    ? (magog)
= (default1 default2)    ? (magog 10)
= (10 default2)    ? (magog 10 11)
= (10 11)    ? (magog 10 11 12 13)
** magog : unexpected arguments : 4
```

2.3.7 Functions of the `fexpr` kind

`fexpr`s are functions written in LISP and evaluated by the standard evaluation functions (`eval`, `apply`, or `funcall`). Just like `exprs`, these functions have a variable number of parameters, described in a parameter list (or tree) `lvar`, and a function body made up of a number of expressions `s1` ... `sN`.

A function of this type is described by using a list, called an `flambda`-expression, of the following form:

(`flambda` `lvar` `s1` ... `sN`)

where the symbol `flambda` is a function indicator of type `fexpr`, `lvar` is the list (or tree) of parameters, and `s1` ... `sN` make up the body of the function.

Example:

```
(flambda (var val) (set var (eval val)))
```

The definition of `fexpr`-type functions (that is, the association of a `flambda`-expression with a symbol) is done by means of the `df` function.

These functions differ from `expr`s only in the way parameters are bound in `fexpr`s. It is the set of arguments that are *not evaluated*: that is, the `cdr` of the form itself, which is bound to the parameter list `lvar` by the same `bindvar` procedure used for `expr`s.

These function calls can raise the same errors as `expr`s: `errwna`, `errilb`, and `errbpa`.

2.3.8 Functions of the `macro` kind

The evaluator permits another type of functions, the `macro` functions. Just like `expr`s and `fexpr`s, these functions, written in LISP, have variable numbers of parameters which are stored in a list `lvar`, and function bodies made up of expressions `s1` ... `sN`.

`macro` functions are described by using a list, called an `mlambda`-expression, of the form

(`mlambda` `lvar` `s1` ... `sN`)

where the symbol `mlambda` is a function indicator of type `macro`, `lvar` is the parameter list, and `s1` ... `sN` make up the body of the function.

Example:

```
(mlambda (n var) (list 'setq var (list 'cdr var)))
```

The definition of `macro`-type functions (that is, the association of a `mlambda`-expression with a symbol) is done by means of the `dm` function.

To evaluate a form which has a `macro` as its function, the evaluator first evaluates the function associated with this `macro` using the whole form (obviously, *not* evaluated) as its argument, using the `bindvar` procedure. Then it re-evaluates the value returned by this initial evaluation. The evaluation of a `macro` thus happens in two steps.

It is the entire `macro` call which is taken as the argument; it is therefore possible to physically modify the form itself during the first evaluation of the `macro`.

The use of `macro`s, truly a sport in itself, allows for easy and powerful extensions of the language, LISP being both the `macro` description language and the target language of `macro` expansions.

`macro` function calls can raise the same errors as `expr` calls: `errwna`, `errlib`, `errbpa`.

2.3.9 Functions of the `dmacro` kind

The evaluator accepts a second kind of `macro` definition, the `dmacro`s. `dmacro`s are defined with the `defmacro` function.

The evaluation of a function of this type differs from the evaluation of `macro` functions in two ways:

- the `cdr` of the form (not evaluated) is used as the argument (the same as with `fexpr`s),
- after the first evaluation has been performed, the entire form is physically replaced by the value returned by the `displace` function. (Whence the name, `dmacro`.)

Though less general than the `macro` functions, the `dmacro`s are more frequently used. In fact most occurrences of macro functions are calls to `dmacro`.

`dmacro` function calls can raise the same errors as `expr` calls: `errwna`, `errlib`, `errbpa`.

2.4 Defining functions

Two kinds of functions are encountered in LE-LISP:

- **Global functions**
  These functions are defined globally and keep their definition as long as they are not explicitly modified. This kind of definition does not allow dynamic recovery of previous definitions. See the `defun`, `df`, `dm` and `defmacro` functions.
- **Local functions**
  The definition of these functions can change during certain evaluations and then assume their previous definition. See the `flet` and `letn` functions.

Recall that function definitions are stored in the `f-val` and `f-type` intrinsic properties of symbols. Since direct access to these properties is not at all convenient, there are a number of predefined functions facilitating static and dynamic function definition at very little cost. (See the Function Definition section of the next chapter.)

2.5 Packages

LE-LISP has a multiple name-space. Each symbol, unique in the system, has an external name: its so-called `pname`. This name is not necessarily unique. Each symbol also has a package name, indicating to which package the symbol belongs. A package name is also a symbol, which can in turn have a package name, and so on. The global package is named `||`. It is possible to manage name hierarchies. In LE-LISP, packages allow you to perform the following operations:

- The name space can be shared among several programs.
- Functions can be invoked within specific names spaces.
- Method execution within a simple hierarchy—of a SmallTalk variety—can be implemented in a very efficient manner.

2.6 Extended types

In LE-LISP, you can define other types of objects. These extended types are always built with a labelled list cell (`tcons`), which contains the symbolic name of the extended type in the `car`.

Its value, stored in the `cdr`, is either a typed vector or a typed string. The extended types are recognized by the LE-LISP evaluator and printer.

When the evaluator (`eval`) evaluates a form which has a labelled list cell as the first element, and if the `car` of this labelled list is a symbol or a typed vector or typed string to be evaluated, it searches to see if it can find a function named `eval` in the package with the same name as the result of evaluating the `type-of` function on its argument. If this function exists, it is called by `eval`; if not, or if there is no such package, an error is raised. The search for the `eval` function in the package uses the `getfn` function with third argument equal to `()`. The evaluator's search up the package hierarchy stops just below the level of the global package, thus avoiding an infinite evaluation loop.

Warning: The search for the evaluation function takes place after the normal evaluation. It is therefore impossible, in the present implementation, to use defined function names as names of extended types.

When the printer (the function `prin`) prints a LISP object, it searches for a function named `prin` in the package with the same name as the result of evaluating the `type-of` function on its argument. If this function exists, it is called by `prin` in place of the standard LE-LISP `prin` function. The search for this function in the package uses the `getfn` function with third argument equal to `()`. The evaluator's search up the package hierarchy stops just below the level of the global package, thus avoiding an infinite print loop.

The functionality of the printer can be expressed in LISP in the following manner:

```
(if (getfn (type-of x) 'prin)
    (funcall (getfn (type-of x) 'prin ()) x)
    (prin x))
```

As an example, let us define the `list-of-numerals` extended type. The input of such a list will be done with the #-macro `<>`.

```
#<cccccccc>
```

where the numerals ..ccc.. are surrounded by the delimiters `<>`.

Here is the LISP definition of the read function:

```
(defsharp |<| ()
   (list (let (x c v)
              (while (neq (setq c (readcn)) #/>)
                     (newl x c))
              (setq v (apply 'vector (nreverse x)))
              (typevector v 'list-of-numerals)
              v)))
```

This function behaves as follows:

```
? '#<01234>
= #:list-of-numerals:#[48 49 50 51 52]    ? (type-of '#<123>)
= list-of-numerals
```

Let us now define the printer for this extended type:

```
(defun #:list-of-numerals:prin (x)
    (prin "#<")
    (mapvector 'princn x)
    (prin ">"))
```

Here is the behavior of the new device:

```
? #<0246>
= #<0246>   ? (cons #<0234> #<067>)
= (#<0234> . #<067>)
```

2.7 Meta-circular definition of the evaluator

This final section proposes a meta-circular description of the LE-LISP evaluator. This description—which provides a global view of the operation of the evaluator—is obviously incomplete, and does not represent the actual implementation. The style in which the evaluator is really implemented is far more efficient than what you see here, especially in the number of `cons` operations used in the evaluator functions, and in the management of the stack. In fact, the LE-LISP evaluator performs no `cons` operations at all for its own use.

```
; Special package

(defvar #:sys-package:colon 'eval)

; Main loop

(defun :toplevel ()
    (tag :error
         (print "<toplevel>")
         (let ((stack) (it))
              (setq it (:eval (with ((prompt ":? ")) (read))))
              (print ":= " it)))
    (:toplevel))

; Handle errors

(defvar #:system:error-flag t)
(defvar #:system:debug t)

(defun :error (f m a)
    (when #:system:error-flag
          (print "** " f " : " m " : " a))
    (exit :error))

; Evaluate a form
```

```
(defun :eval (form)
    (cond
      ((symbolp form)
         (if (boundp form)
             (symeval form)
             (:error 'eval 'errudv form)))
      ((or (fixp form) (floatp form))
          form)
      ((atom form)
          (if (getfn (type-of form) 'eval ())
              (:funcall (getfn (type-of form) 'eval ()) form)
              form))
      (t (let ((funct (car form)) (lval (cdr form)))
           (cond
              ((symbolp funct)
                  (cond ((typefn funct)
                             (:evalinternal form
                                            funct
                                            (typefn funct)
                                            (valfn funct)
                                            lval))
                        ((and (tconsp form)
                              (getfn funct 'eval ()))
                            (:funcall (getfn funct 'eval ()) form))
                        (t (:error 'eval 'errudf funct))))
              ((atom funct)
                  (:error 'eval 'errudf funct))
              ((eq (car funct) 'lambda)
                  (:evalinternal form funct 'expr (cdr funct) lval))
              ((eq (car funct) 'flambda)
                  (:evalinternal form funct 'fexpr (cdr funct) lval))
              ((eq (car funct) 'mlambda)
                  (:evalinternal form funct 'macro (cdr funct) lval))
              (t (:error 'eval 'errudf funct)))))))

; Evaluate a function according to its type

(defun :evalinternal (form funct ftype fval lval)
    (if (or (and (atom lval) lval)
            (and (consp lval) (cdr (last lval))))
        (:error funct 'errbal (if (atom lval) lval (cdr (last lval))))
        (selectq ftype
          (subr0 (if (= (length lval) 0)
                     (call fval () () ())
                     (:error funct 'errwna 0)))
```

```
            (subr1 (if (= (length lval) 1)
                       (call fval (:eval (car lval)) () ())
                       (:error funct 'errwna 1)))
            (subr2 (if (= (length lval) 2)
                       (call fval (:eval (car lval))
                                  (:eval (cadr lval))
                                  ())
                       (:error funct 'errwna 2)))
            (subr3 (if (= (length lval) 3)
                       (call fval (:eval (car lval))
                                  (:eval (cadr lval))
                                  (:eval (caddr lval)))
                       (:error funct 'errwna 3)))
            (nsubr (calln fval (:evlis lval)))
            (msubr (call fval form () ()))
            (dmsubr (displace form (call fval (cdr form) () ())))
            (fsubr
               ; All the FSUBRs are evaluated directly
               ; since they can call EVAL again.
               ; Only the most important ones are defined here.
               (selectq funct
                   (quote    (car lval))
                   (lambda   form)
                   (if       (if (:eval (car lval))
                                 (:eval (cadr lval))
                                 (:eval (caddr lval))))
                   (progn    (:eprogn lval))
                   (defun    (setfn (car lval) 'expr   (cdr lval)))
                   (df       (setfn (car lval) 'fexpr  (cdr lval)))
                   (dm       (setfn (car lval) 'macro  (cdr lval)))
                   (defmacro (setfn (car lval) 'dmacro (cdr lval)))
                   (setq     (while (cddr lval)
                                    (set (nextl lval)
                                         (:eval (nextl lval))))
                             (set (nextl lval) (:eval (car lval))))
                   (t        (:error 'eval 'errudf funct))))
            (expr    (:evalambda fval (:evlis lval)))
            (fexpr   (:evalambda fval lval))
            (macro   (:eval (:evalambda fval form)))
            (dmacro  (:eval (displace form
                                 (:evalambda fval (cdr form)))))
            (t       (:error 'eval 'errudf funct)))))

; Build the list of values of the evaluations of all list elements
```

```
(defun :evlis (l)
    (if (null l)
        ()
        (cons (:eval (car l)) (:evlis (cdr l)))))

; Evaluate the body of l

(defun :eprogn (l)
    (if (null (cdr l))
        (:eval (car l))
        (:eval (car l))
        (:eprogn (cdr l))))

; Apply an F-VAL of type (<lvar> <s1> ... <sn>)
; to the list of arguments lval

(defun :evalambda (fval lval)
    (:push ())
    (:bindvar (car fval) lval)
    (protect (:eprogn (cdr fval))
             (:unbindvar)))

; Carry out a generalized tree binding

(defun :bindvar (lvar lval)
    (cond ((null lvar)
               (when lval
                     (:error 'eval 'errwna lval)))
          ((variablep lvar)
              (:push (if (boundp lvar) (symeval lvar) '<undef>))
              (:push lvar)
              (set lvar lval))
          ((consp lvar)
               (if (atom lval)
                   (:error 'eval 'errilb (list lvar lval))
                   (:bindvar (car lvar) (car lval))
                   (:bindvar (cdr lvar) (cdr lval))))
          (t (:error 'eval 'errbpa lvar))))

(defun :unbindvar ()
    (untilexit :done
               (set (or (:pop) (exit :done)) (:pop))))

; With the stack manipulation functions

(defun :push l (while l (newl stack (nextl l))))
```

```
(defun :pop () (nextl stack))

; Apply funct to the lval arguments

(defun :apply (funct lval)
    (cond
      ((symbolp funct)
          (:applyinternal funct
                          (typefn funct)
                          (valfn funct)
                          lval))
      ((atom funct)
          (:error ':apply 'errudf funct))
      ((eq (car funct) 'lambda)
          (:applyinternal funct 'expr funct lval))
      ((eq (car funct) 'flambda)
          (:applyinternal funct 'fexpr funct lval))
      ((eq (car funct) 'mlambda)
          (:applyinternal funct 'macro funct lval))
      (t (:error ':apply 'errudf funct))))

; Application of a function according to its type

(defun :applyinternal (funct ftype fval lval)
    (selectq ftype
       (subr0 (if (= (length lval) 0)
                  (call fval () () ())
                  (:error funct 'errwna 0)))
       (subr1 (if (= (length lval) 1)
                  (call fval (car lval) () ())
                  (:error funct 'errwna 1)))
       (subr2 (if (= (length lval) 2)
                  (call fval (car lval) (cadr lval) ())
                  (:error funct 'errwna 2)))
       (subr3 (if (= (length lval) 3)
                  (call fval (car lval)
                             (cadr lval)
                             (caddr lval))
                  (:error funct 'errwna 3)))
       (nsubr  (calln fval lval))
       (msubr  (:eval (call fval (cons funct lval) () ())))
       (dmsubr (:eval (call fval lval () ())))
       (fsubr  (call fval lval () () ))
       (expr   (:evalambda fval lval))
```

```
        (fexpr  (:evalambda fval (list lval)))
        (macro  (:eval (:evalambda fval (cons funct lval))))
        (dmacro (:eval (displace form (:evalambda fval lval))))
        (t (:error ':apply 'errudf funct))))

; Application to funcall

(defun :funcall (funct . larg)
    (:apply funct larg))
```

Table of contents

2 Evaluation **2-1**

2.1 Basic objects 2-1

2.1.1 Atomic objects 2-1

2.1.2 Compound objects 2-4

2.2 Basic evaluation actions 2-5

2.2.1 Evaluation of atomic objects 2-5

2.2.2 Evaluation of composite objects 2-6

2.3 Evaluation of functions 2-6

2.3.1 Functions of the `subr` kind 2-7

2.3.2 Functions of the `fsubr` kind 2-8

2.3.3 Functions of the `msubr` kind 2-8

2.3.4 Functions of the `dmsubr` kind 2-8

2.3.5 Functions of the `expr` kind 2-9

2.3.6 Functions of the `expr` kind with a `&nobind` argument 2-11

2.3.7 Functions of the `fexpr` kind 2-11

2.3.8 Functions of the `macro` kind 2-12

2.3.9 Functions of the `dmacro` kind 2-12

2.4 Defining functions 2-13

2.5 Packages 2-13

2.6 Extended types 2-13

2.7 Meta-circular definition of the evaluator 2-15

Function Index

Chapter 3

Predefined functions

The functions described in this lengthy chapter are always resident in all versions of the LE-LISP system. Each function is listed here with its type—`function` or `special-form`—as well as the number of arguments it expects. For each argument, the requested or required argument type is indicated, using the following notation:

- `s` for S-expressions
- `l` for lists
- `a` for atoms: symbols, numbers or character strings
- `symb` for symbols
- `n` for numbers
- `strg` for character strings, or objects that could be converted to character strings by using the `string` function
- `vect` for vectors of S-expressions
- `ch` for characters: that is, for the first character of character strings or objects that could be converted to character strings by using the `string` function
- `cn` for internal character codes
- `fn` for functions: symbols defined as functions, and anonymous or named lambda-expressions.

Whenever possible, functions of the `subr` type are described here in LISP terms, in the form of `defun`, `df`, `dm` or `defmacro` definitions. These descriptions are merely LISP equivalents of the functions. A description of this kind represents no more than the *behavior* of the corresponding function, but not its actual implementation in LE-LISP.

Numerous type tests are carried out by the functions presented in this chapter. The following errors can be raised:

- `errnaa`, with the following default screen display:

```
** <fn> : not an atom : <e>
```

- `errnla`, with the following default screen display:

  ```
  ** <fn> : not a list : <e>
  ```

- `errsym`, with the following default screen display:

  ```
  ** <fn> : not a symbol : <e>
  ```

- `errnva`, with the following default screen display:

  ```
  ** <fn> : not a variable : <e>
  ```

- `errnsa`, with the following default screen display:

  ```
  ** <fn> : not a string : <e>
  ```

In these displays, `e` is name of the the faulty argument, and `fn` is the function that caused the error.

3.1 Evaluation functions

(eval s env) *[function with one or two arguments]*

This is the main function of the interpreter. `eval` returns the value of the evaluation of the argument `s`. (See the complete description of this function in the preceding chapter.) The second argument, `env`, if supplied, is a lexical environment passed by the `stepeval` function. (See section 5 of chapter 7.)

```
(eval '(1+ 55))                        ⟹   56
(eval (list '+ 8 '(1+ 3) 3))           ⟹   15
(eval (list (car '(cdr)) ''(a b c)))   ⟹   (b c)
```

(evlis l) *[function with one argument]*

Returns a list of the values of the evaluations of all the elements of the list `l`.

In LE-LISP, `evlis` could be defined in the following manner:

```
(defun evlis (l)
    (if (null l)
        ()
        (cons (eval (car l)) (evlis (cdr l)))))
```

```
(setq l '((1+ 5) (1+ 7) (1+ 9)))  ⟹   ((1+ 5) (1+ 7) (1+ 9))
(evlis l)                         ⟹   (6 8 10)
```

(eprogn l) [*function with one argument*]

Sequentially evaluates all the elements in the list `l`, and returns the value of the final element of `l`. If `l` is not a well-formed list, the `errbal` error is raised.

In LE-LISP, `eprogn` could be defined in the following manner:

```
(defun eprogn (l)
    (cond ((consp l)
           (if (null (cdr l))
               (eval (car l))
               (eval (car l))
               (eprogn (cdr l))))
          ((null l) ())
          (t (error 'eprogn 'errbal l))))
```

```
? (setq l '((prin 1) (prin 2) (prin 3)))
= ((prin 1) (prin 2) (prin 3))   ? (eprogn l)
123
= 3
```

(prog1 s_1 ... s_n) [*special form*]

Sequentially evaluates the expressions $s_1 \ldots s_n$, and returns the value of the first evaluation: that of s_1.

In LE-LISP, `prog1` could be defined in the following manner:

```
(df prog1 (first . rest)
    (let ((result (eval first)))
         (eprogn rest)
         result))
```

Alternatively, it could be defined in the form of an `expr`:

```
(defun prog1 l (car l))
```

```
? (prog1 (prin 1) (prin 2) (prin 3))
123
= 1
```

`prog1` is always used to generate side-effects.

Here is a `macro`, `(exch var1 var2)`, that exchanges the values of `var1` and `var2` without using other memory than for `var1` and `var2`:

```
(dm exch (exch var1 var2)
    (list 'setq var1
```

```
        (list 'prog1 var2
              (list 'setq var2 var1))))))
```

A form such as `(exch v1 v2)` is expanded to `(setq v1 (prog1 v2 (setq v2 v1)))`.

(prog2 s_1 s_2 ... s_n) [*special form*]

Sequentially evaluates the expressions s_1 s_2 ... s_n and returns the value of the second evaluation: that of s_2.

In LE-LISP, `prog2` could be defined in the following manner:

```
(df prog2 (first second . rest)
    (eval first)
    (let ((result (eval second)))
         (eprogn rest)
         result))
```

Alternatively, it could be defined in the form of an `expr`:

```
(defun prog2 (first second . l) second)
```

```
? (prog2 (prin 1) (prin 2) (prin 3))
123
= 2
```

Like `prog1`, `prog2` is always used to generate side-effects.

(progn s_1 ... s_n) [*special form*]

Evaluates the expressions s_1 ... s_n in sequence, and returns the value of the last evaluation: that is, the evaluation of s_n.

`progn` is the `fsubr` form of the `eprogn` function. So, it can be described in the form of an `fexpr`, in the following manner:

```
(df progn l (eprogn l))
```

Alternatively, it could be defined in the form of an `expr`:

```
(defun progn l (car (last l)))
```

```
? (progn (prin 1) (prin 2) (prin 3))
123
= 3
```

(quote s) [*special form*]

Returns the unevaluated S-expression `s`. This function is used to inhibit argument-evaluation in calls to functions.

A predefined macro character, the apostrophe sign, can be used in place of the `quote` primitive. Written as `'`, it is often referred to as *quote*. Placed before any expression `s`, this apostrophe is read as `(quote s)`.

For readability reasons, output functions print lists of the form `(quote s)` as `'s`.

In LE-LISP, `quote` could be defined in the following manner:

```
(df quote (s) s)
```

```
(quote (1+ 4))      ⟹   (1+ 4)
'(a (b c))          ⟹   (a (b c))
'a                  ⟹   a
''a                 ⟹   'a
'''a                ⟹   ''a
(quote (quote a))   ⟹   'a
'(quote a b)        ⟹   (quote a b)
```

(function fn) [*special form*]

If a lexical environment is present, this function returns the *lexical closure* composed of the function `fn` and this lexical environment. If there is no lexical environment present, `function` is equivalent to `quote`. LE-LISP Version 15.22 only builds a lexical environment for the labels defined by the `tagbody` special-form and the block names defined by the `block` special-form. The `function` function cannot be described in LISP.

(arg n) [*function with an optional argument*]

This function is used inside an `expr &nobind`. Without arguments, `(arg)` returns the number of arguments of the last `expr &nobind` function call. With a numeric argument, `(arg n)` returns the value of the n^{th} argument of the last `expr &nobind` function call. If `n` is greater than the number of arguments, the result is indeterminate, since `arg` does not test the validity of `n`.

(identity s) [*function with one argument*]

As its name indicates, this function is the identity function. It returns its argument.

In LE-LISP, `identity` could be defined in the following manner:

```
(defun identity (s) s)
```

```
(identity 'a)       ⟹   a
(identity (1+ 5))   ⟹   6
```

(comment e_1 ... e_n) [*special form*]

Returns the symbol `comment` itself, and does not evaluate the arguments. This function is very useful for commenting out an S-expression in the middle of a file. It is not recommended for the introduction of program comments, which should be written after the special *semi-colon* character.

Warning: This function can only be inserted into an implicit or explicit `progn`. Otherwise, the evaluation will be disrupted.

In LE-LISP, `comment` could be defined in the following manner:

```
(df comment l 'comment)
```

```
(comment 'foo bar)   ⟹   comment
(comment)            ⟹   comment
(comment but no)     ⟹   comment
```

3.2 Application functions

(lambda l s_1 ... s_n) [*special form*]

(flambda l s_1 ... s_n) [*special form*]

(mlambda l s_1 ... s_n) [*special form*]

The value of a `lambda`, `flambda` or `mlambda` lambda-expression is the expression itself. These new functions were added to eliminate the need to *quote* anonymous explicit lambda-expressions in application functions.

In LE-LISP, `lambda` could be defined in the following manner:

```
(df lambda l (cons 'lambda l))
```

Alternatively, it could be defined in the form of a macro:

```
(dm lambda l (kwote l))
```

The second definition is more exact, since it preserves the physical address of the lambda-expression.

In LE-LISP, `flambda` could be defined in the following manner:

```
(df flambda l (cons 'flambda l))
```

In LE-LISP, `mlambda` could be defined in the following manner:

```
(df mlambda l (cons 'mlambda l))
```

```
? (lambda (x) x)
= (lambda (x) x)    ? (mapc (lambda (x) (prin x)) '(a b c))
abc
= ()
```

3.2.1 Simple application functions

(apply `fn` s_1 ... s_n `l`**)** [*function with a variable number of arguments*]

The arguments s_1 ... s_n are optional. `apply` returns the value of the application of the function `fn` to the list of arguments `l` with the arguments s_1 ... s_n appended in front, if they are present. The argument `l` must be a list, which might be empty. Otherwise, the `errbal` error is raised. The function `fn` is of one of the following types: `subr`, `nsubr`, `fsubr`, `msubr`, `dmsubr`, `expr`, `fexpr`, `macro` or `dmacro`.

```
(apply 'cons (list (1+ 1) (1+ 2)))                    ⟹   (2 . 3)
(apply 'cons (1+ 1) (list (1+ 3)))                    ⟹   (2 . 4)
(apply 'list 1 2 3 '(4 5))                            ⟹   (1 2 3 4 5)
(apply (lambda (x y) (+ x y)) (list (1+ 8) (- 10 3)))  ⟹   16
(apply (flambda (x y) (cons x y)) '((1+ 1) (1- 2)))    ⟹   ((1+ 1) 1- 2)
```

(funcall `fn` s_1 ... s_n**)** [*function with a variable number of arguments*]

This is another form of the `apply` function. The form `(funcall fn` s_1 ... s_n`)` is equivalent to the form `(apply fn (list` s_1 ... s_n`))`. But `funcall` does not really construct any lists. Consequently, it is much more efficient than `apply`.

In LE-LISP, `funcall` could be defined in the following manner:

```
(defun funcall (fnt . larg)
    (apply fnt larg))
```

```
(funcall (lambda (x y) (cons x y)) 'a 'b)  ⟹   (a . b)
(funcall '+ (1+ 1) (1+ 2) (1+ 3))          ⟹   9
(setq kons 'cons)                          ⟹   cons
(funcall kons (1+ 1) (1+ 2))               ⟹   (2 . 3)
```

3.2.2 Application functions of the `map` type

These functions allow for the repeated application of a function to a set of argument lists. The applied function can have any number of arguments, and the applications stop when the end of one of these argument lists is reached. To apply *invariant* arguments, include them in a circular list, by using for example the `cirlist` function. Obviously, at least one of these arguments lists must terminate in the context of the evaluation, so that these application functions do not loop indefinitely.

(allcar l) [*function with one argument*]

In the LISP descriptions accompanying these functions, we shall use the following auxiliary functions:

```
(defun allcar (e)

    ; e is a list of lists.
    ; This function returns the list of all their cars.

    (if (null e) () (cons (caar e) (allcar (cdr e)))))

(defun allcdr (e)

    ; e is a list of lists.
    ; This function returns a list of all their cdrs.

    (if (null e) () (cons (cdar e) (allcdr (cdr e)))))
```

(mapl fn l_1 ... l_n) [*function with a variable number of arguments*]

(map fn l_1 ... l_n) [*function with a variable number of arguments*]

These apply the function fn to the lists l_i, then to all the cdrs of these lists, until the end of one of the lists is reached.

In LE-LISP, mapl and map could be defined in the following manner:

```
(defun mapl (f . l)
    (when (every 'consp l)
          (apply f l)
          (apply 'mapl f (allcdr l))))
```

```
? (mapl 'print '(a (b c) d) '(x y z))
(a (b c) d)(x y z)
((b c) d)(y z)
(d)(z)
= ()
```

(mapc fn l_1 ... l_n) [*function with a variable number of arguments*]

Applies the function fn to the cars of the lists l_i, then to all the cadrs of these lists, then to the caddrs, and so on, until the end of one of these lists is reached.

In LE-LISP, mapc could be defined in the following manner:

```
(defun mapc (f . l)
    (when (every 'consp l)
          (apply f (allcar l))
          (apply 'mapc f (allcdr l))))
```

```
? (mapc 'print '(a (b c) d) '(x y z))
ax
(b c)y
dz
= ()
```

(maplist fn l_1 ... l_n) [*function with a variable number of arguments*]

Applies the function `fn` to the lists l_i, then to the `cdr`s of these lists ... until the end of one of the lists is reached. So, `maplist` is functionally similar to the `map` or `mapl` functions, but it returns as its value the list of the values of the applications.

In LE-LISP, `maplist` could be defined in the following manner:

```
(defun maplist (f . l)
    (when (every 'consp l)
          (cons (apply f l)
                (apply 'maplist f (allcdr l)))))
```

```
(maplist 'length '(a (b c) d))  ⟹  (3 2 1)
```

(mapcar fn l_1 ... l_n) [*function with a variable number of arguments*]

Applies the function `fn` to the `car`s of the lists l_i, then to the `cadr`s of these lists, then to their `caddr`s ... until the end of one of the lists is reached. So, `mapcar` is functionally similar to the `mapc` function, but it returns as its value the list of values of the applications.

In LE-LISP, `mapcar` could be defined in the following manner:

```
(defun mapcar (f . l)
    (when (every 'consp l)
          (cons (apply f (allcar l))
                (apply 'mapcar f (allcdr l)))))
```

```
(mapcar 'cons '(a b c) '(1 2))  ⟹  ((a . 1) (b . 2))
(mapcar 'list '(a b c d e f) (cirlist 1 2))  ⟹  ((a 1) (b 2) (c 1) (d 2) (e 1) (f 2))
```

(mapcon fn l_1 ... l_n) [*function with a variable number of arguments*]

Applies the function `fn` to the lists l_i, then to their `cdr`s, then to their `cddr`s ... until the end of one of the lists is reached. `mapcon` is therefore functionally similar to the `map` function.

However, each application must return a list as its value, and **mapcon** returns the list of values of these applications, which are **nconc**ed together (see the description of this function) to give the return value. If one of the returned values is not a list, it is ignored and does not appear in the list returned by **mapcon**.

In LE-LISP, **mapcon** could be defined in the following manner:

```
(defun mapcon (f . l)
    (when (every 'consp l)
          (nconc (apply f l)
                 (apply 'mapcon f (allcdr l)))))
```

```
? (mapcon 'list '(1 2 3) '(4 5 6))
= ((1 2 3) (4 5 6) (2 3) (5 6) (3) (6))    ? (mapcon (lambda (x) (list (car (last x)))) '(a
b c))
= (c c c)
```

The following code makes the **last** function loop the third time:

```
(mapcon 'last '(a b c))  ⟹   ? ?
```

(mapcan fn l_1 ... l_n**)** *[function with a variable number of arguments]*

Applies the function **fn** to the **car**s of the lists l_i, then to the **cadr**s of these lists, then to their **caddr**s ... until the end of one of these lists is reached. **mapcan** is therefore functionally similar to the **mapc** function. However, each application must return a list as its value and **mapcan** returns the list of values of these applications, which are **nconc**ed together (see the description of this function) to give the return value. If one of these values is not a list, it is ignored and does not appear in the list returned by **mapcan**.

In LE-LISP, **mapcan** could be defined in the following manner:

```
(defun mapcan (f . l)
    (when (every 'consp l)
          (nconc (apply f (allcar l))
                 (apply 'mapcan f (allcdr l)))))
```

```
? (mapcan (lambda (x y) (list (1+ x) (1- y)))
?  '(1 2 3)
?  '(1 2 3))
= (2 0 3 1 4 2)    ? (mapcan 'list '(a b c d)
?  (cirlist 1 2)
?  '(w x y z)
?  (cirlist 0))
= (a 1 w 0 b 2 x 0 c 1 y 0 d 2 z 0)
```

3.2.3 Other application functions

(every fn l_1 ... l_n) [*function with a variable number of arguments*]

Applies the function `fn` to the `car`s of the lists l_i, then to the `cadr`s of these lists, then to their `caddr`s ... until the value returned by one of these applications is `()`, or until the end of one of the lists l_i is reached. In the latter case, `every` returns the value of the last application. Due to the way the boolean functions are represented in LISP, with `()` representing *false*, this function is principally used to apply a predicate to all the elements of a list.

In LE-LISP, `every` could be defined in the following manner:

```
(defun every (f . l)
    (if (caar l)
        (ifn (cdar l)
             (apply f (allcar l))
             (and (apply f (allcar l))
                  (apply 'every f (allcdr l))))
        t))
```

This description assumes that all list arguments have the same length.

```
(every 'consp '((1) (2) (3)))  ⟹  (3)
(every 'eq '(1 2 3) '(1 2 3))  ⟹  t
(every 'eq '(1 2) '(1 2 3))    ⟹  t
(every 'eq '(1 2 3) '(1 2 4))  ⟹  ()
```

(any fn l_1 ... l_n) [*function with a variable number of arguments*]

Applies the function `fn` to the `car`s of the lists l_i, then to the `cadr`s of these lists, then to their `caddr`s, ... until the value returned by one of these applications is not equal to `()`, or until the end of one of the lists l_i is reached. Due to the way the boolean functions are represented in LISP, with `()` representing *false*, this function is principally used to apply a predicate to all the elements of a list.

In LE-LISP, `any` could be defined in the following manner:

```
(defun any (f . l)
    (when (car l)
          (or (apply f (allcar l))
              (apply 'any f (allcdr l)))))
```

This description assumes that all list arguments have the same length.

```
(any 'consp '(1 "foo" (1) 10))  ⟹  (1)
(any '= '(1 2 3) '(10 2 30))    ⟹  2
```

(mapvector fn vect) [*function with two arguments*]

Applies the function `fn`, which must be defined to take a single argument, to each element of the vector `vect`, beginning with the zeroth element. `mapvector` returns `()` as its value.

In LE-LISP, mapvector could be defined in the following manner:

```
(defun mapvector (fn vect)
    (for (i 0 1 (1- (vlength vect)))
         (funcall fn (vref vect i))))
```

```
? (mapvector 'prin #[a b c])
abc
= ()
```

(mapoblist fn) [*function with one argument*]

Applies the function fn, which must be defined to take a single argument, to all the elements—in turn—of the oblist. This function is therefore equivalent to (mapc fn (oblist)), but it is much more efficient, mainly because it does not build the entire list of symbols: typically about two thousand elements.

Here, for example, is a function that prints the names of all the nsubrs in the system:

```
(defun printnsubr ()
    (mapoblist (lambda (symb)
                       (when (eq (typefn symb) 'nsubr)
                             (print symb)))))
```

(mapcoblist fn) [*function with one argument*]

Applies the function fn, which must be defined to take a single argument, to all the elements—in turn—of the oblist. Each evaluation returns a list, and these are nconced together to give the return value. If one of the values is not a list, it is ignored and it does not appear in the result. mapcoblist is therefore equivalent to (mapcan fn (oblist)), but it is much more efficient, mainly because it does not build the entire list of symbols: about two thousand elements.

Here is a function that returns a list of the form (symb objval symb objval ...) for all the symbols that have an objval:

```
(defun findoval ()
    (mapcoblist (lambda (symb)
                        (when (objval symb)
                              (list symb
                                    (objval symb))))))
```

(maploblist fn) [*function with one argument*]

Applies the function fn, which must be defined to take a single argment, to all the elements—in turn—of the oblist. Each evaluation returns a boolean value, and maploblist returns the list of symbols in the oblist of which the function, usually a predicate, is true.

In LE-LISP, maploblist could be defined in the following manner:

```
(defun maploblist (fn)
    (mapcoblist
        (lambda (s)
                (when (funcall fn s) (list s)))))
```

To obtain the number of functions in the system, evaluate

```
(length (maploblist 'typefn))
```

Here is a function that returns the list of all the functions of type `fsubr` in the system:

```
(defun findfsubr ()
    (maploblist (lambda (symb)
                        (eq (typefn symb) 'fsubr))))
```

3.3 Environment manipulation functions

These functions temporarily change the dynamic environment. Here, the term 'environment' means the variable-value bindings. The initial environment, which was current before one of these function calls, is automatically restored upon function termination.

(let lv s_1 ... s_n) [*special form*]

By means of `let`, you can make anonymous `expr`-type function calls. In other words, you can create lambda-expressions.

`lv` is a list with elements of the form

- `var`, or
- `(var val)`.

$s_1 \ldots s_n$ is the body of the function.

The `let` function binds the variables var_i, dynamically and simultaneously, with the values val_i, if the latter are supplied, or with `()` by default. It then executes the body of the function $s_1 \ldots s_n$. After completing this execution, `let` unbinds the variables and binds them to their previous values. `let` simplifies the definition and initialization of local variables.

`let` is an abbreviated form of anonymous `expr`-type function call, and can therefore raise the same errors: `errbpa`, `errlib` or `errbal`.

The form

(let ((var val)) $s_1 \ldots s_n$)

corresponds to the call

((lambda (var) $s_1 \ldots s_n$) val)

and the form

```
(let ((var1 val1) ... (varn valn))
     s1 ... sm)
```

corresponds to the call

```
((lambda (var1 ... varn) s1 ... sm)
     val1 ... valn)
```

In LE-LISP, `let` could be defined in the following manner:

```
(defmacro let (lv . body)
     (cond ((null lv)
               (cons (cons 'lambda (cons () body))))
           (t (cons (cons 'lambda
                          (cons (mapcar
                                   (lambda (l)
                                       (if (consp l)
                                           (car l)
                                           l))
                                   lv)
                                body))
                    (mapcar (lambda (l)
                              (if (consp l)
                                  (cadr l)
                                  ()))
                            lv)))))
```

Here is a more readable definition:

```
(defmacro let (lv . body)
     (if (null lv)
         `((lambda () ,@body))
         `((lambda ,(mapcar (lambda (l) (if (consp l) (car l) l))
                            lv)
                   ,@body)
             ,@(mapcar (lambda (l) (if (consp l) (cadr l) ()))
                       lv))))
```

```
(let ((i 10) (j 20)) (+ i j))                      ⟹   30
(let ((a 'foo) ((b . c) (cons 1 2))) (list a b c))  ⟹   (foo 1 2)
(let ((i 10) j k) (list i j k))                    ⟹   (10 () ())
```

(letv lvar lval s1 ... sn) [*special form*]

This function is identical to the `let` function, but allows for the calculation of the variable names to be bound. `lvar` is an argument that should result, when evaluated, in a tree of symbols. `lval` is an argument that should result, when evaluated, in a list of values. `letv` links the symbols of the `lvar` tree with the values of the `lval` list, and then evaluates the expressions $s_1 \ldots s_n$ in this

new environment. `letv` returns the value of the last evaluation (that of s_n), and then restores the environment to the state it was in before the call. `letv` can raise the same errors (`errbpa`, `errilb`, `errwna` or `errbal`) as the evaluation of an anonymous function.

In LE-LISP, `letv` could be defined in the following manner:

```
(defmacro letv (lvar lval . body)
     `((lambda (,(eval lvar)) ,@body) ,lval))
```

```
(letv '(a (b . c) d) '(1 (2 3) 4) (list a b c d))  ⟹  (1 2 (3) 4)
```

(letvq lvar lval s_1 ... s_n) [*special form*]

This function is identical to the preceding one, except that the first argument, `lvar` (the tree of variables), is not evaluated.

In LE-LISP, `letvq` could be defined in the following manner:

```
(defmacro letvq (lvar lval . body)
     `((lambda (,lvar) ,@body) ,lval))
```

```
(letvq (a . b) '(1 2) (list a b))  ⟹  (1 (2))
```

(lets lv s_1 ... s_n) [*special form*]

(slet lv s_1 ... s_n) [*special form*]

(let* lv s_1 ... s_n) [*special form*]

These functions are identical to the `let` function, but the arguments are bound sequentially, not in parallel. They differ from `let` in the same way that `psetq` differs from `setq`. So, a value already bound in the `lets` can be used to calculate the value of another expression in the list `lv`. There is no difference between `lets`, `slet` and `let*`.

The form

```
(lets ((var1 val1) (var2 val2) ... (varn valn))
      s1 ... sn)
```

corresponds, therefore, to

```
(let ((var1 val1))
     (let ((var2 val2))
        ......
        (let ((varn valn))
           s1 ... sn) ... ))
```

and also to

```
((lambda (var1)
         ((lambda (var2)
                  ......
                  ((lambda (varn)
                       s1 ... sn)
                   valn) ... )
          val2)
 val1)
```

```
(let ((i 10)) (slet ((i 20) (j (+ i i))) (list i j)))  ==>  (20 40)
```

(letn symb lv s_1 ... s_n) [*special form*]

Makes it possible to call named **expr**-type functions. **symb** is the temporary function name, **lv** is a list of variable/value pairs, as in the **let** function, and $s_1 \ldots s_n$ is a function body.

The **letn** function will dynamically, and in parallel, bind the variables with the values in the list **lv**. It will then execute the body of the function $s_1 \ldots s_n$, after having associated (during the evaluation of the body) the same lambda-expression as in the **let** function to the name **symb**. **letn** therefore simplifies the dynamic definition and call of a recursive function.

The form

(letn symb ((var_1 val_1) ... (var_n val_n)) s_1 ... s_n)

is equivalent to

(flet ((symb (var_1 ... var_n) s_1 ... s_n)) (symb val_1 ... val_n))

In LE-LISP, **letn** could be defined in the following manner:

```
(defmacro letn (name lv . body)
     `(flet ((,name ,(mapcar 'car `,lv) ,@body))
            (,name ,@(mapcar 'cadr `,lv)))))))))
```

Let us look at an example that uses **letn**: the **index** function. It returns the index of the sub-string **pname1** of the string **pname2**, starting at index **n**:

```
(defun index (pname1 pname2 n)
    (letn inxaux1 ((p1 (pname pname1))
                   (p2 (nthcdr n (pname pname2)))
                   (n n))
          (cond ((< (length p2) (length p1)) ())
                ((<> (car p1) (car p2))
                   (inxaux1 p1 (cdr p2) (1+ n)))
                (t (letn inxaux2 ((pp1 (cdr p1))
                                  (pp2 (cdr p2)))
                         (cond ((null pp1) n)
                               ((= (car pp1) (car pp2))
                                    (inxaux2 (cdr pp1)
                                             (cdr pp2)))
                               (t (inxaux1 p1 (cdr p2)
                                           (1+ n)))))))))
```

3.4 Function-definition functions

These functions allow for the definition of new functions. All of them test the validity of their arguments:

- Names of functions must be symbols.
- All the variables in the list or in the parameter tree must also be symbols.

These symbols can only appear once in the list or in the parameter tree.

If not, the `errbdf` or `errbpa` errors are raised, with default screen displays:

```
** <fn> : bad definition : <symb>
```

```
** <fn> : bad parameter : <e>
```

There are two types of function definition: static definitions and dynamic definitions. (See the preceding chapter on the operation of the interpreter.)

3.4.1 Static function definitions

All these functions will permanently change the functions associated with the symbols. These changes are controlled by the following variables:

`#:system:previous-def-flag` [*variable*]

`#:system:previous-def` [*property*]

All function redefinitions can be preceded by a save of the previous definition associated with the symbol—if it had one—on its P-list by use of the `#:system:previous-def` property. This save is controlled by the `#:system:previous-def-flag` variable. If it is true—that is, if its value is different from `()`—the save takes place. If it is false, no save is performed. By default the value of this variable is `()`.

`#:system:redef-flag` [*variable*]

The `#:system:redef-flag` flag allows for warning on function redefinition.
If `#:system:redef-flag` is false—that is, equal to `()`, as it is by default—a warning message is printed on any attempt to redefine a function. If this flag is true—different from `()`—function redefinitions cause no warning messages. Such a message will have the form

```
** <fn> : redefined function : <symb>
```

where `fn` is the definition function that was called, and `symb` is the name of the function that was just redefined.

#:system:loaded-from-file [*variable*]

#:system:loaded-from-file [*property*]

Moreover, all the following definition functions add, to the P-list of the newly defined symbols by use of the `#:system:loaded-from-file` property, the name of the file from which the function was loaded. This name will be found in the `#:system:loaded-from-file` variable. If the value of this variable is `()`, nothing is added on to the P-list of the function name symbol.

(defun symb lvar s_1 ... s_n) [*special form*]

(de symb lvar s_1 ... s_n) [*special form*]

`defun` allows for the static definition of new `expr`s. `symb` is the name of the symbol to which a lambda-expression of the type

(lambda lvar s_1 ... s_n)

will be attached.

`defun` returns the name of the function, `symb`, as its value. There is no difference between `de` and `defun`.

Up to validity checks, the form

(defun symb lvar s_1 ... s_n)

is equivalent to

(setfn 'symb 'expr '(lvar s_1 ... s_n)

```
(defun foo (x) (if (null (cdr x)) x (foo (cdr x))))  ⟹  foo
(valfn 'foo)    ⟹  ((x) (if (null (cdr x)) x (foo (cdr x))))
(typefn 'foo)   ⟹  expr
(foo '(a b c))  ⟹  (c)
```

(df symb lvar s_1 ... s_n) [*special form*]

Allows for the static definition of new `fexpr`s. `symb` is the name of the symbol to which a lambda expression of the type

(flambda lvar s_1 ... s_n)

will be attached.

`df` returns the name of the defined function, `symb`, as its value.

Up to validity checks,

(df symb lvar s_1 ... s_n)

is equivalent to

(setfn 'symb 'fexpr '(lvar s_1 ... s_n)

```
? (df increm (var . val)
?  (set var (if (consp val)
?  (+ (eval var) (eval (car val)))
?  (1+ (eval var)))))
= increm   ? (typefn 'increm)
= fexpr   ? (setq nb 7)
= 7   ? (increm nb)
= 8   ? nb
= 8
```

(dm symb lvar s_1 ... s_n) [*special form*]

Allows for the static definition of new macros. symb is the name of the symbol to which a lambda expression of the type

(mlambda lvar s_1 ... s_n)

will be attached.

dm returns the name of the defined function, symb, as its value.

Up to validity checks, the form

(dm symb lvar s_1 ... s_n)

is equivalent to

(setfn 'symb 'macro '(lvar s_1 ... s_n)

```
? (dm decr (decr var . val)
?      (list 'setq var
?            (if val
?                (list '- var (car val))
?                (list '1- var))))
= decr   ? (setq n 10)
= 10   ? (decr n)
= 9
```

(defmacro symb lvar s_1 ... s_n) [*special form*]

(dmd symb lvar s_1 ... s_n) [*special form*]

defmacro allows for the static definition of a new dmacro named symb that accepts a tree of variables lvar. This tree of variables will be bound to the cdr of the call to this macro. defmacro returns symb as its value. There is no difference between defmacro and dmd.

In LE-LISP, defmacro could be defined in the following manner:

```
(df defmacro (name larg . corps)
```

```
(eval `(dm ,name -l-
           (letvq ,larg (cdr -l-)
                  (displace -l-
                       ,(if (and (consp corps)
                                 (consp (car corps))
                                 (null (cdr corps)))
                            (car corps)
                            (cons 'progn corps)))
                  -l-))))
```

Up to validity checks, the form

`(defmacro symb lvar` s_1 ... s_n`)`

is equivalent to

`(setfn 'symb 'dmacro '(lvar` s_1 ... s_n`)`

Here is an example:

```
(defmacro if (test then else1 . elsen)
    `(cond ((,test ,then) (t ,else1 ,@elsen))))
```

(ds symb type adr) [*special form*]

Allows for the static definition of functions whose type is `subr` or `fsubr`. `symb` is the name of a symbol to which a machine-language function, beginning at address `adr`, will be attached. Its type is given by the argument `type`. This function, which is little used but included for reasons of homogeneity, is an `fsubr` variant of `setfn`.

Up to validity tests, the form

`(ds symb type adr)`

is equivalent to

`(setfn 'symb 'type 'adr)`

3.4.2 Advanced use of `macro` functions

Writing `macro` functions is a challenging activity. LE-LISP encourages their use over `fexpr`s, which cause variable capture. In order to facilitate the writing of `macro`s, the backquote macro-character is provided (described in the chapter on input/output), along with the next two functions, which allow for control of the results of `macro`-expansions.

(macroexpand1 s) [*function with one argument*]

If the expression `s` is a `macro`, `dmacro`, `msubr` or `dmsubr` call, `macroexpand1` returns the expanded value of this `macro`. This is obviously very useful for testing your own `macro`s.

In LE-LISP, `macroexpand1` could be defined in the following manner:

```
(defun macroexpand1 (x)
    ; expands a macro one level
    (cond ((eq (typefn (car x)) 'macro)
               (apply (cons 'lambda (valfn (car x))) x))
          ((eq (typefn (car x) 'dmacro)
               (apply (cons 'lambda (valfn (car x))) (cdr x)))
          ((eq (typefn (car x)) 'msubr)
               (call (valfn (car x)) x () ()))
          ((eq (typefn (car x)) 'dmsubr)
               (call (valfn (car x)) (cdr x) () ()))
          (t x))))
```

```
(defmacro foo (x1 x2) `(mcons ,x1 foo ,x2))  ⟹   foo
(macroexpand1 '(foo 10 20))                  ⟹   (mcons 10 foo 20)
```

(macroexpand s) *[function with one argument]*

Allows for the multi-level expansion of `macro`, `dmacro`, `msubr` or `dmsubr` calls in the expression `s`.

In LE-LISP, `macroexpand` could be defined in the following manner:

```
(defun macroexpand (x)
    ; expand a call until no macros remain
    (cond ((atom x) x)
          ((symbolp (car x))
             (cond ((eq (typefn (car x)) 'macro)
                    (macroexpand
                         (apply (cons 'lambda (valfn (car x)))
                                x)))
                   ((eq (typefn (car x)) 'dmacro)
                    (macroexpand
                       (apply (cons 'lambda (valfn (car x)))
                              (cdr x))))
                   ((eq (typefn (car x)) 'msubr)
                    (macroexpand
                       (call (valfn (car x)) x () ())))
                   ((eq (typefn (car x)) 'dmsubr)
                    (macroexpand
                       (call (valfn (car x)) (cdr x) () ())))
                   ((eq (car x) 'quote) x)
                   (t (let ((s ()))
                         (while (consp x)
                             (newl s (macroexpand (nextl x))))
                         (prog1 (nreverse s) (rplacd s x))))))
          (t (let ((s ()))
                 (while (consp x)
                        (newl s (macroexpand (nextl x))))
```

```
                    (prog1 (nreverse s) (rplacd s x))))))

(defmacro foo (x1 x2) `(mcons ,x1 foo ,x2))  ==>  foo
(dm bar (bar x) `(foo ,x 30))                ==>  bar
(macroexpand '(bar (foo 10 20)))             ==>  (mcons (mcons 10 foo 20) foo 30)
```

3.4.3 Definition of closures

(closure lvar fn) *[function with two arguments]*

Allows for the definition of a closure made up of a list of variables `lvar` and a function `fn`. Each variable in the list `lvar` must have a non-indefinite value that will be used to construct the closure.

In LE-LISP, `closure` could be defined in the following manner:

```
(defun closure (lvarclot fnt)
    (let ((listval (mapcar '(lambda (val)
                                    (kwote (eval val)))
                           lvarclot))
          (lvar (cadr fnt))
          (corps (cddr fnt)))
        `(lambda ,lvar
            ((lambda ,lvarclot
                 (protect (progn ,@corps)
                      ,@(mapcar
                           '(lambda (slot var)
                               `(rplaca
                                  (cdr ,`(quote ,slot))
                                  ,var))
                            listval  lvarclot)))
             ,@listval))))
```

For example, here is a natural number generator:

```
(setfn 'nextint  'expr
       (let ((n 1))
            (cdr (closure '(n)
                     '(lambda () (setq n (1+ n)))))))
```

Here is a Fibonacci number generator:

```
(setfn 'fib 'expr
      (let ((x 1) (y 1))
           (cdr (closure '(x y)
                  '(lambda ()
                       (setq y (prog1 (+ x y)
                                 (setq x y))))))))
```

```
(nextint)  ⟹   2
(nextint)  ⟹   3
n          ⟹      ** eval : undefined variable : n
(fib)      ⟹   2
(fib)      ⟹   3
(fib)      ⟹   5
```

3.4.4 Definition of dynamic functions

(flet `l` s_1 ... s_n**)** [*special form*]

Allows for the definition of dynamic functions—analogous to the `let` function for variables.

The argument `l` is a list of lists. Each sub-list has the form `(symb l` e_1 ... e_n`)`.

The arguments s_1 ... s_n represent the function body.

The `flet` function binds a new function to each symbol `symb` during the evaluation of s_1 ... s_n. Each of these functions—which are themselves `expr`-type expressions—has an argument list `l` and a body e_1 ... e_n. The return value of `flet` is the evaluation of the last function body in the `flet`: that of s_n. After the `flet`, the symbols `symb` retrieve their previous definitions ... if they had one.

Due to the introduction of programmable interrupts, `flet` is not used very often. Moreover, its use leads to poor performance in compiled code because of the totally dynamic nature of the redefinition. Consequently, it should be avoided.

```
? (flet ((car (x) (cdr x))) (car '(a b c)))
= (b c)    ? (car '(a b c))
= a    ? (flet ((car (x) (cdr x))
?  (cdr (x) (cddr x)))
?  (cdr (car '(a b c d e))))
= (e)
```

3.4.5 Generalized assignment

LISP allows assignment of values to variables by means of functions such as `setq`, `set` and `psetq`. Often, however, values are stored in different kinds of data structures: lists, vectors and structures. For each type of storage, two functions need to be defined: an access function and an assignment function. They are akin to the `vref` and `vset` functions for reading from and writing to vector storage.

Generalized assignment allows you to use the access function to carry out an assignment by means of the `setf` macro.

The advantage of generalized assignment is the economy gained in the initial assignment of values to symbols. It also provides a uniform means of writing such assignment functions.

(setf loc_1 val_1 ... loc_n val_n**)** [*macro*]

Updates the locations `loc`$_1$ ... `loc`$_n$ with the values of the expressions `val`$_1$ ... `val`$_n$. As in the case of the `setq` function, the assignments are carried out sequentially. `setf` returns the value of the last assignment that was performed.

`setf` is a macro that is expanded into assignment functions for the locations `loc`$_i$.

The locations `loc`$_i$ must be one of the following:

- A variable name. In this case, the `setf` macro is equivalent to the `setq` function.
- A predefined locations, with one of the following forms: `(car x)`, `(cdr x)`, `(caar x)`, `(cadr x)`, `(cadar x)`, `(cddr x)`, `(getprop s i)`, `(vref v i)` and `(nth i l)`. The list can be extended by using the `defsetf` macro, which is about to be presented.
- One of the access functions of any defined structure.

`(setf (vref v i) x)` is equivalent to `(vset v i x)`.

`(defsetf access lparams store . body)` [*macro*]

Lets you define the assignment function associated with the form `(access . lparams)`. The `store` argument is a single-element list, the updated value is bound to the contents of `store`. The `body` argument is a macro body that must return the form that does the updating.

```
(defsetf (vref v i) (x) `(vset ,v ,i ,x))

(defsetf (cdr lst) (val)
    `(progn (rplacd ,lst ,val)
            ,val))   ; because rplacd does not return the right value
```

3.5 Variable functions

A function with a variable number of arguments that gives read and write access (`get`/`set`) to a LISP datum will be referred to as a *variable function*. Read access takes place by calling this function with `n-1` arguments. Write access takes place by calling it with `n` arguments. In the write call, the last argument is the new value of the variable function. The type of the LISP datum manipulated by these variable functions can be one of the following:

- An internal LE-LISP system type: for example, the numerical output conversion base, or input/output stamps.
- Specific to a particular program, in which case any LISP object whatever can be used.

A variable function has all the properties of a function. Indeed, it *is* a function. But it can also be *dynamically bound*—in the same way as LISP variables—by using the following function:

`(with l s`$_1$ ... `s`$_n$`)` [*special form*]

The argument `l` is a list of the form

`((symb`$_1$ `arg`$_{11}$ ... `arg`$_{1n}$`) ... (symb`$_n$ `arg`$_{n1}$ ... `arg`$_{nn}$`))`

where the `symb`$_i$ are the names of the variable functions, and `arg`$_{i1}$ ... `arg`$_{in}$ are their write-access arguments. `with` performs these writes after saving the values of calls to these variable functions in the read mode, then evaluating the expressions `s`$_1$ ... `s`$_n$ and returning the value of `s`$_n$ as its result. But, in the spirit of dynamic binding, `with` calls these function variables again in write mode, with the values of the variable functions previously saved as arguments. The reconstitution of these values is protected in the manner of the `protect` function.

In LE-LISP, `with` could be defined in the following manner:

```
(defmacro with (l . body)
     (let ((var
            (let ((n -1))
                 (mapcar (lambda (x)
                           (symbol 'with
                                (concat "arg" (incr n))))
                         l))))
         `(let (,@(mapcar
                    (lambda (var l)
                        (list var
                           (or (consp (firstn (1- (length l))
                                              l))
                               (error 'with 'errsxt l))))
                    var
                    l))
          (protect
            (progn ,@l ,@body)
            ,@(mapcar
                 (lambda (var l)
                    (append1 (firstn (1- (length l)) l)
                             var))
                 var
                 l)))))
```

For example, `obase` is the system variable-function for accessing the numeric base of numbers being printed out:

```
? (obase)
= 10   ? (prin 100)
100
= 100   ? (with ((obase 16)) (prin 100))
64
= 100   ? (obase)
= 10
```

`(with ((obase 16)) (prin 100))` is equivalent to

```
(let ((xxx (obase)))
     (protect (progn (obase 16)
                     (prin 100))
              (obase xxx)))
```

Here is another example:

```
(defun monindic (pl . n)
     (ifn (consp n)
          (getprop pl 'indic)
          (putprop pl (car n) 'indic)
          (car n)))

(monindic 'x '(a))                          ⟹   (a)
(monindic 'x)                               ⟹   (a)
(with ((monindic 'x 20)) (monindic 'x))     ⟹   20
(monindic 'x)                               ⟹   (a)
```

3.6 Basic control functions

All the functions described in this section allow the sequential chain of evaluations to be broken. That is why they are all of type `fsubr`. The arguments are not evaluated by `eval`, but by the functions themselves. Even then, this is not always the case.

The simplest control function is the conditional `if` function. This function will be used with the `while` function to describe all the other control functions in LISP, in the form of `fexpr`s.

(if s_1 s_2 s_3 ... s_n) [*special form*]

If the value of the evaluation of s_1 differs from `()`, `if` returns the value of the evaluation of the expression s_2. Otherwise, `if` evaluates the expressions s_3 ... s_n in turn, and returns the value of the last evaluation: s_n. `if` allows for the construction of a control structure of the form IF `s` THEN s_2 ELSE s_3 ... s_n.

```
(if t 1 2 3)    ⟹   1
(if () 1 2 3)   ⟹   3
```

Here is Ackermann's function described with the use of the `if` function:

```
(defun ack (x y)
    (if (= x 0)
        (1+ y)
        (ack (1- x)
             (if (= y 0)
                 1
                 (ack x (1- y))))))
```

(ifn s_1 s_2 s_3 ... s_n) [*special form*]

If the value of the evaluation of s_1 is equal to `()`, `ifn` returns the value of the evaluation of the expression s_2. Otherwise, `ifn` evaluates the expressions $s_3 \ldots s_n$ in turn, and returns the value of the last evaluation: that of s_n. `ifn` allows for the construction of a control structure of the form IF NOT s_1 THEN s_2 ELSE $s_3 \ldots s_n$.

`ifn` is therefore equivalent to `(if (not` s_1`)` s_2 $s_3 \ldots s_n$`)`.

In LE-LISP, `ifn` could be defined in the following manner:

```
(df ifn (test then . else)
    (if (null (eval test))
        (eval then)
        (eprogn else)))
```

```
(ifn t 1 2 3)   ⟹   3
(ifn () 1 2 3)  ⟹   1
```

(when s_1 s_2 ... s_n) [*special form*]

If the value of the evaluation of s_1 differs from `()`, `when` evaluates the expressions $s_2 \ldots s_n$ in turn, and returns the value of the last evaluation: that of s_n. If the value of s_1 is `()`, `when` returns `()`. `when` gives rise to a control structure of the form IF s_1 THEN $s_2 \ldots s_n$ ELSE `()`.

In LE-LISP, `when` could be defined in the following manner:

```
(df when (test . body)
    (if (eval test)
        (eprogn body)
        ()))
```

```
(when t 1 2 3)   ⟹   3
(when () 1 2 3)  ⟹   ()
```

(unless s_1 s_2 ... s_n) [*special form*]

If the value of the evaluation of s_1 is equal to `()`, `unless` evaluates the expressions $s_2 \ldots s_n$ in turn, and returns the value of the last evaluation: that of s_n. If the value of s_1 differs from `()`, `unless` returns `()`. `unless` allows for the construction of a control structure of the form IF s_1 THEN `()` ELSE $s_2 \ldots s_n$.

In LE-LISP, `unless` could be defined in the following manner:

```
(df unless (test . body)
    (if (eval test)
        ()
        (eprogn body)))
```

```
(unless t 1 2 3)   ⟹   ()
(unless () 1 2 3)  ⟹   3
```

(or s_1 ... s_n) [*special form*]

Evaluates the expressions s_1 ... s_n, in turn, until one of these evaluations has a value different than `()`. `or` returns this value. If no expression is given as argument, this function returns `()`.

In LE-LISP, `or` could be defined in the following manner:

```
(df or l
    (letn oreval ((l l))
          (if (null (cdr l))
              (eval (car l))
              (let ((resul (eval (car l))))
                   (if resul
                       resul
                       (oreval (cdr l)))))))
```

```
(or)           ⟹   ()
(or ())        ⟹   ()
(or 1 2)       ⟹   1
(or () () 2 3) ⟹   2
```

(and s_1 ... s_n) [*special form*]

Evaluates the expressions s_1 ... s_n, in turn, until the value of one of these evaluations is equal to `()`, and then returns `()`. If this does not occur—that is, if all the expressions, when evaluated, differ from `()`—`and` returns the value of the last evaluation: that of s_n. If no expressions are given as arguments, this function returns `t`. `and` allows for the construction of control structures of the form IF s_1 THEN IF s_2 THEN IF ... THEN s_n.

In LE-LISP, `and` could be defined in the following manner:

```
(df and l
    (if (null l)
        t
        (letn andeval ((l l))
              (if (null (cdr l))
                  (eval (car l))
                  (if (eval (car l))
                      (andeval (cdr l))
                      ())))))
```

```
(and)          ⟹   t
(and ())       ⟹   ()
(and 1 2 3 4)  ⟹   4
(and 1 2 () 4) ⟹   ()
```

(cond l_1 ... l_n) [*special form*]

The famous `cond` primitive is the most general conditional function in LISP. The arguments $\texttt{l}_1 \ldots \texttt{l}_n$ are lists, called *clauses*, each of which has the structure `(ss` $\texttt{s}_1 \ldots \texttt{s}_n$`)`.

`ss` is any expression, and $\texttt{s}_1 \ldots \texttt{s}_n$ are the body of the clause. `cond` evaluates the body of the first clause whose first element, `ss`, evaluates to something other than `()`. `cond` evaluates sequentially the expressions of the body, $\texttt{s}_1 \ldots \texttt{s}_n$, and returns the value of the last evaluation: that of $\texttt{s}_n$. If the clause in question has only one element, `ss`, `cond` returns the value of `ss`: that is, the value that caused the clause to be chosen for evaluation. `cond` allows for the construction of control structures of the form IF ... THEN ... ELSE IF ... THEN ...

If no clause is chosen for evaluation, `cond` returns `()`. In order to force the selection of the last clause, it is customary to use the symbolic constant `t` as the selector, since its value is always different to `()`. (See the section on predicates.)

In LE-LISP, `cond` could be defined in the following manner:

```
(df cond l
    (letn condeval ((l l))
          (when l
                (if (or (atom l) (atom (car l)))
                    (error 'cond 'errsxt l)
                    (let ((select (eval (caar l))))
                         (if select
                             (if (cdar l)
                                 (eprogn (cdar l))
                                 select)
                             (condeval (cdr l))))))))
```

The form

```
(cond
  (p1 e11 e12 e13)
  (p2 e21 e22)
  (p3)
  (p4 e41))
```

is therefore equivalent to

```
(cond
  (p1 (progn e11 e12 e13))
  (p2 (progn e21 e22))
  ((let ((aux p3)) aux))
  (p4 e41))

(cond (() 1 2) (t 3 4 5))  ⟹  5
```

(selectq s $\texttt{l}_1 \ldots \texttt{l}_n$) [*special form*]

Like the `cond` function, `selectq` chooses one of the clauses $\texttt{l}_1 \ldots \texttt{l}_n$. The selector of these clauses is the value of the evaluation of `s`, the selection being accomplished by the comparison of this evaluation with one of the following:

- The unevaluated `car` of the clause if this is an atom. This comparison uses the `equal` predicate.
- The elements of the unevaluated `car` of the clause if this is a list. In this case, the comparison uses the `member` search function. This option allows for the selection to take place between `s` and an arbitrary number of objects.

Upon the selection of a clause, `selectq` evaluates the rest of the clause in order, and returns the value of the last evaluation.

If none of the clauses $l_1 \ldots l_n$ is selected, `selectq` returns `()`.

`selectq` therefore allows for the construction of switches on constant values.

As in the `cond` function, it is possible to force the selection of the last clause by using the symbolic constant `t` as its first element.

In LE-LISP, `selectq` could be defined in the following manner:

```
(df selectq (sel . cl)
    (letn sel1 ((sel (eval sel)) (cl cl))
       (cond
          ((null cl)
             ; no more clauses
             ())
          ((or (atom cl) (atom (car cl)))
             ; bad clause
             (error 'selectq 'errsxt cl))
          ((eq (caar cl) t)
             ; selection of the "always true" clause
             (eprogn (cdar cl)))
          ((and (atomp (caar cl)) (equal sel (caar  cl)))
             ; simple selection
             (eprogn (cdar cl)))
         ((member sel (caar  cl))
             ; general selection
             (eprogn (cdar cl)))
         (t (sel1 sel (cdr cl))))))
```

```
? (selectq 'red
?  (green 'hope)
?  (red 'ok)
?  (t 'no))
= ok    ? (selectq 'red
?  ((blue green red) 'color)
?  ((rose iris) 'flower)
?  (t 'dunno))
= color
```

(while s s_1 ... s_n) [*special form*]

As long as the value of `s` differs from `()`, `while` evaluates the expressions `s` s_1 ... `s` s_n in turn. `while` always returns `()` as its value. This is the value of the last evaluation of `s`. The loop terminates at this point. This function allows for the construction of infinite loops with the form `(while t ...)`, since the symbolic constant `t` always has a value different to `()`.

In LE-LISP, `while` could be defined in the following manner:

```
(df while (test . body)
    (letn while1 ()
          (if (null (eval test))
              ()
              (eprogn body)
              (while1))))
```

`while` is equivalent to the following LISP form:

`(letn while () (ifn s () s`$_1$ ... `s`$_n$ `(while)))`

Here is an example of this function:

```
? (setq s '(a b c d))
= (a b c d)   ? (while s (prin (nextl s)))
abcd
= ()
```

(until s s_1 ... s_n) [*special form*]

As long as the value of the evaluation of `s` is equal to `()`, `until` evaluates all the expressions `s` s_1 ... `s` s_n in turn. `until` returns the value of the first evaluation of `s` that differs from `()`. This is the value that causes the exit from the `until` loop. Like the preceding function, `while`, `until` provides a very convenient way to construct conditional loops, as well as infinite loops with the form `(until () ... )`.

In LE-LISP, `until` could be defined in the following manner:

```
(df until (test . body)
    (letn until1 ()
          (or (eval test)
              (progn (eprogn body) (until1)))))
```

`until` is equivalent—up to the return value—to the following LISP form:

`(while (not s) s`$_1$ ... `s`$_n$`)`

Here is an example of this function:

```
? (setq s '(a b c d))
= (a b c d)   ? (until (null s) (prin (nextl s)))
```

```
abcd
= t
```

(repeat m s_1 ... s_n) [*special form*]

repeat evaluates the expression m. The resulting value must be a whole number. repeat then performs m evaluations of the expressions s_1 ... s_n. If the value of m is not a number, the errnia error is raised. If m is not strictly greater than zero, the loop is not executed. In any case, repeat returns t as its value.

In LE-LISP, repeat could be defined in the following manner:

```
(df repeat (n . l)
    (let ((n (eval n)))
         (if (fixp n)
             (while (>= n 0) (eprogn l) (decr n))
             (error 'repeat 'errnia n)))
    t)
```

```
? (repeat 10 (prin '*))
**********
= t   ? (repeat 0 (prin '*))
= t
```

(for (var in ic ntl e_1 ... e_m) s_1 ... s_n) [*special form*]

Allows nostalgic programmers to implement a loop of the Algol variety. The argument var must be a symbol. It is the numeric control variable for the loop. The in argument is the initial numeric value of var at the beginning of the loop, and ic is the numeric increment added to var each time round the loop. Finally, ntl is the maximal value—or minimal value, depending on the sign of ic—that the control variable can receive. It is the limit value of var. The for function returns the value of the evaluation of (progn e_1 ... e_m).

In LE-LISP, for could be defined in the following manner:

```
(defmacro for ((var init step end . res) . body)
     (unless (variablep var)
          (error 'for 'errnva var))
     (unless (and init step end)
          (error 'for 'errsxt (list var init step end)))
     (let ((endvar (concat 'for (gensym)))
           (test (if (numberp step)
                     (if (> step 0) '<= '>=)
                     `(if (> ,step 0) '<=
                          (if (< ,step 0) '>=
                              (error 'for "increment nul" 0))))))
          `(let ((,var ,init)
```

```
                (,endvar ,end))
               (while (,test ,var ,endvar)
                      ,@body
                      (incr ,var ,step))
               ,@res)))
```

```
? (for (i 0 1 9 'ok) (prin i))
0123456789
= ok
```

3.7 Lexical control functions

LE-LISP version 15.22 introduces the notion of lexical, or textual, control. The control structures described in this section have a strictly lexical scope. Incorrect use of one of them can raise the `errnab` or `errxia` errors, which have the default screen displays:

```
** <fn> : no lexical scope : <e>

** <fn> : inactive lexical scope : <e>
```

3.7.1 Primitive lexical control forms

(block `symb` e_1 ... e_n**)** [*special form*]

`symb` is the unevaluated symbolic name of a lexical block that evaluates the expressions e_1 ... e_n. By default, `block` returns the value of the last evaluation: that is, that of e_n. However, these sequential evaluations can be interrupted by the next function.

(return-from `symb e`**)** [*special form*]

`symb` must be the name of a lexical block, of type `block`, and it is not evaluated. `return-from` exits from this lexical block and returns the value of the evaluation of `e`. If this function occurs outside the lexical scope of the block in question, the `errnab` error is raised.

(return `e`**)** [*special form*]

This function is the same as the following form:

```
(return-from () e)
```

(tagbody ec_1 ... ec_n**)** [*special form*]

ec_1 ... ec_n are either symbols considered as labels, or lists that are forms to be evaluated. `tagbody` evaluates the forms in turn, and returns `()` by default. The sequential evaluation of these forms can be interrupted by the next function.

(go symb) [*special form*]

symb is the name of a label in the current **tagbody**. The **go** function, called with **symb** as its argument, allows you to restart the sequential evaluation from the point at which **symb** appears in the **tagbody**. If **symb** is not a label in a lexical **tagbody**, the **errnab** error is raised.

Here are examples of the use of lexical control functions:

```
? (block commonality 1 (return-from commonality 2) 3)
= 2    ? (block () 1 (block portability 2 (return 3) 4) 5)
= 3    ? (block consistency (eval '(return-from consistency 10)))
** return-from : no lexical scope : consistency
? (block expressiveness
? (let ((y #'(lambda () (return-from expressiveness 10))))
?            (funcall y)))
= 10    ? (let (y) (block compatibility
? (setq y #'(lambda () (return-from compatibility 10))))
?           (funcall y))
** return-from : inactive lexical scope : compatibility
? (let ((n 5) l)
?      (tagbody
?         efficiency (if (<= n 0)
?                        (go power)
?                        (newl l (decr n))
?                        (go efficiency))
?         power)
?      l)
= (0 1 2 3 4)
```

The following example is inspired by a well-known source [Steele 84]:

```
(defun cex (f g x)
    (if (= x 0)
        (funcall f)
        (block here
               (+ 5 (cex g
                         #'(lambda () (return-from here 4))
                         (- x 1))))))

(cex () () 2)  ⟹   4
```

3.7.2 Iteration functions of the prog kind

(prog l ec_1 ... ec_n) [*macro*]

Allows you to combine the **block**, **let** and **tagbody** functions.

In LE-LISP, **prog** could be defined in the following manner:

```
(defmacro prog (l . body)
```

```
        `(block ()
                (let ,l
                     (tagbody ,@body))))
```

(prog* l ec_1 ... ec_n) [*macro*]

`prog*` is equivalent to the preceding function, `prog`, except that the variable bindings take place in sequence—not in parallel—due to the use of the `let*` form.

In LE-LISP, `prog*` could be defined in the following manner:

```
(defmacro prog* (l . body)
     `(block ()
             (let* ,l
                   (tagbody ,@body))))
```

3.7.3 Iteration functions of the `do` kind

(do lv lr ec_1 ... ec_n) [*macro*]

This macro, borrowed from MacLisp, is a very general iteration function with one or several control variables. It, too, is a combination of the `let`, `block` and `tagbody` functions, enriched by an iteration structure.

- `lv` is a list of triples, each of which describes a control variable as `(symb val inc)`.
- `symb` is the name of the control variable.
- The expression `val` is its initial value, before entering the loop. By default, it is `()`.
- `inc` is the expression indicating its increment value, added after each iteration. If this expression is not supplied, the variable does not change values during loop execution. `lr`, the second argument of a `do`, is a list of the form

  `(test` e_1 ... e_n`)`

  where `test` is the end-of-iteration expression and e_1 ... e_n constitute the value to return at the end of the iterations. This argument is analogous to a `cond` clause. If the value of `test` is equal to `()`, the body of the `do` is executed. If not, the expressions e_1 ... e_n are evaluated, and the value of the evaluation of e_n is the return value of the `do`.
- ec_1 ... ec_n is the body of the `do`: the analog of the `tagbody` function.

The evaluation of a `do` proceeds in the following manner:

- The expressions val_1 ... val_n of the triples are evaluated, and are used to bind the control variables sym_1 ... sym_n. This is done in parallel, in the style of `let` or `psetq`.
- `test` is evaluated. If its value is different to `()`, the expressions e_1 ... e_n are evaluated, and `do` returns the value of e_n.

- If the test returns `()`, the `do` body $\texttt{ec}_1 \ldots \texttt{ec}_n$ is executed, as in a `prog`.
- At the end of the `do` body, the expressions $\texttt{inc}_1 \ldots \texttt{inc}_n$ in the triples are evaluated and are used—if they exist—to bind the control variables $\texttt{sym}_1 \ldots \texttt{sym}_n$. The process then continues at the second step.

In LE-LISP, `do` could be defined in the following manner:

```
(defmacro do (lv (test . resul) . body)
     `(block ()
            (let ,(mapcar (lambda (x) (list (car x) (cadr x)))
                          lv)
                 (until ,test
                     (tagbody ,@body)
                     (psetq ,@(mapcan (lambda (x)
                                        (when (consp (cddr x))
                                            (list (car x) (caddr x))))
                                      lv)))
                 ,@resul)))
```

Notice that this definition is not optimized for empty blocks or `tagbody` forms.

Let us look at some examples of the use of `do` without a body. Here is a way of writing the `length` function:

```
(defun lngt (l)
    (do ((x l (cdr x))
         (j 0 (1+ j)))
        ((atom x) j)))
```

This function is transformed as follows:

```
(defun lngt (l)
    (let ((x l) (j 0))
         (until (atom x)
                (psetq x (cdr x) j (1+ j)))
         j))
```

Here is a of writing the `reverse` function:

```
(defun rev (l)
    (do ((x l (cdr x))
         (y () (cons (car x) y)))
        ((atom x) y)))
```

This function is transformed as follows:

```
(defun rev (l)
    (let ((x l) (y ()))
         (until (atom x)
                (psetq x (cdr x) y (cons (car x) y)))
         y))
```

(do* lv lr ec_1 ... ec_n) [*macro*]

`do*` is identical to the preceding function, `do`, except that the bindings take place sequentially and not in parallel. The relationship between `do*` and `do` is the same as that between `let*` and `let`.

In LE-LISP, `do*` could be defined in the following manner:

```
(defmacro do* (lv (test . resul) . body)
     `(block ()
            (let* ,(mapcar (lambda (x) (list (car x) (cadr x)))
                           lv)
                 (until ,test
                     (tagbody ,@body)
                     (setq ,@(mapcan (lambda (x)
                                       (when (consp (cddr x))
                                           (list (car x) (caddr x))))
                                     lv)))
                ,@resul)))
```

Notice that this definition is not optimized for empty blocks or `tagbody` forms.

Here is an example of an exponentiation function:

```
(defun expt (m n)
    (do* ((result m (* m result))
          (exponent n (1- exponent))
          (counter (1- exponent) (1- exponent)))
         ((= counter 0) result)))
```

This function is transformed as follows:

```
(defun expt (m n)
    (let* ((result m)
           (exponent n)
           (counter (1- exponent)))
          (until (= counter 0)
                 (setq result (* m result)
                       exponent (1- exponent)
                       counter (1- exponent)))
          result))
```

3.8 Dynamic, non-local control functions

The following control functions are powerful and rapid, both in compiled and interpreted code. They are necessary components of more sophisticated control structures, and their use is strongly recommended.

A dynamically-defined label is referred to as an *escape*. LE-LISP associates symbols with these escapes. These associations initialize neither the values of symbols, nor the functions associated with them.

(tag symb s_1 ... s_n) [*special form*]

This lets you define an escape named symb. This name is not evaluated, but s_1 ... s_n are evaluated in turn. If there is no exit function call executed during these evaluations, tag returns the value of the last evaluation: that is, the evaluation of s_n. On the other hand, if a call of the form (exit symb s_1 ... s_n) is encountered among s_1 ... s_n, the evaluations inside the tag are terminated, the expressions e_1 ... e_n are evaluated in turn, and tag returns the value of the evaluation of e_n as its value.

```
(defun present (l e)
    (tag find
         (letn auxfn ((l l))
               (cond ((null l) ())
                     ((eq l e) (exit find l))
                     ((consp l)
                        (auxfn (car l))
                        (auxfn (cdr l)))
                     (t ()))))))

(present '(1 (2 . 3) 4) 3)  ⟹  3
(present '(1 (2 . 3) 4) 5)  ⟹  ()
```

(evtag s s_1 ... s_n) [*special form*]

evtag is similar to the preceding function, tag, except that the name of the escape is evaluated. evtag therefore provides the means to define computed escapes.

(exit symb s_1 ... s_n) [*special form*]

Can be used to leave escapes. After evaluating the expressions s_1 ... s_n in the environment of the exit, it returns to the last dynamically defined (tag symb ...) or (evtag symb ...).

If symb is not the name of a dynamically-accessible escape, the exit from the escape can be captured by a lock form. It is up to this form to raise the errudt error in certain cases. This error has the screen display:

```
** <fn> : undefined tag : <symb>
```

where symb is the name of the escape and fn is the type of the lock form that caught the escape. By default, the main interaction loop has a lock form and raises the following error:

```
** toplevel : undefined tag : <symb>
```

(evexit s s_1 ... s_n) [*special form*]

evexit is similar to exit, except that the name of the escape is evaluated.

(unexit symb s_1 ... s_n) [*special form*]

Can be used to leave escapes. It returns to the last dynamically-defined `(tag symb ...)` or `(evtag symb ...)`. It also evaluates the expressions $s_1 \ldots s_n$ in the environment of the corresponding `tag` or `evtag`.

```
(let ((i 10)) (tag out (let ((i 20)) (exit out i))))     ⟹   20
(let ((i 10)) (tag out (let ((i 20)) (evexit 'out i)))) ⟹   20
(let ((i 10)) (tag out (let ((i 20)) (unexit out i))))   ⟹   10
```

(untilexit symb e_1 ... e_n) [*special form*]

Provides the means to create loops that are controlled by escapes. `symb` is the name of an escape. The expressions $e_1 \ldots e_n$ are repeated indefinitely until the occurrence of an explicit call to exit the escape `symb`, or an escape for which `symb` lies in its dynamic scope.

The form `(untilexit symb` $s_1 \ldots s_n$`)` therefore corresponds to the following call:

`(tag symb (while t` $s_1 \ldots s_n$`))`

(lock fn s_1 ... s_n) [*special form*]

First, `lock` evaluates the argument `fn`, which must return a two-argument function as its value. This function is called the *lock function*.

Then, `lock` evaluates the expressions $s_1 \ldots s_n$ as a `progn`. If an escape occurs in the course of these evaluations, the lock function is automatically called with two arguments. The first is the name of the current escape, and the second is the value which this escape returns. The value returned by this lock function `fn`, if it returns one, becomes the value of the `lock` function. Even if the evaluation of the body of the `lock` terminates normally, the lock function is called, but with `()` as the name of the escape and with the value of the evaluation of s_n as its second argument. In this case, too, the value returned by `lock` is the value returned by the lock function `fn`.

Here is a `lock` function that is useless, because it reactivates an escape with the same name and the same value:

```
(lambda (tag val) (if tag (evexit tag val) val))
```

Here is a `lock` function that stops all escapes and raises an error:

```
(lambda (tag val)
        (if tag
            (error 'lock 'errudt tag)
            val))
```

(protect s_1 s_2 ... s_n) [*special form*]

Evaluates the expressions $s_1 \ldots s_n$ and returns the value of the first evaluation: that of s_1. In this simple case, it behaves like the `prog1` function. However, if an escape occurs during the evaluation

of s_1, the expressions s_2 ... s_n are evaluated during the restoration of the exit context of the escape. This function—needed for the construction of LISP sub-systems—allows for the automatic restoration of complex environments in the case of exceptional exits or errors.

In LE-LISP, **protect** could be defined in the following manner:

```
(df protect (e1 . l)
    (lock (lambda (tag val)
                (eprogn l)
                (if tag
                    (evexit tag val)
                    val))
          (eval e1)))
```

Let us look at a function that prints the hexadecimal value of a number. The number is shifted left by two bits. That is, it is multiplied by four:

```
? (defun prinhex (n)
?    (protect (progn (obase 16)
?                    (print (* n 4)))
?             (obase 10)))
= prinhex   ? (prinhex 100)
190
= 400   ? (prinhex 'foo)
** * : not a number : foo
? 100
= 100   The output base has been restored.
```

(unwind n s_1 ... s_n) [*special form*]

After having unstacked **n** activation blocks from the dynamic execution stack (see the **cstack** function), **unwind** evaluates the expressions s_1 ... s_n and returns the value of the last evaluation: that is, of s_n. These evaluations are performed, therefore, in a previous environment. If there are not at least **n** activation blocks still on the dynamic execution stack, the fatal **errfsud** is raised, and the following default screen is displayed:

```
***** Fatal error : stack underflow.
```

This function cannot be described in LE-LISP in a simple manner.

```
? (let ((x 10)) (let ((x '(a))) (unwind 1 x)))
= 10   ? (let ((x 10))
?      (protect (let ((x '((a)))) (unwind 2 x))
?               (print 'ok)))
ok
= 10
```

(catch-all-but l s_1 ... s_n) [*special form*]

l is a list of escape names. During the evaluation of the expressions s_1 ... s_n, if there is an escape call—that is, a call to exit or to evexit—that causes an exit from the catch-all-but, then the function searches the list l for an occurrence of the escape name in question. If it occurs in l, the escape takes place normally. If not, the errudt error—undefined escape—is raised. This function therefore provides a way of *filtering* escapes.

In LE-LISP, catch-all-but could be defined in the following manner:

```
(defmacro catch-all-but (l . body)
     `(lock (lambda (tag val)
              (cond ((null tag) val)
                    ((memq tag ',l) (evexit tag val))
                    (t (error 'catch-all-but
                              'errudt
                              tag))))
            ,@body)))
```

(backtrack symb l e_1 ... e_n) [*special form*]

Provides a means of executing *backup* control structures. symb is a symbol that has the backtrack name as its value. l is a list of variables to be guarded. e_1 ... e_n are the various actions of the backtrack. Unless interrupted, backtrack evaluates e_1 and returns its value. During this evaluation, if an escape with the name symb occurs, the evaluation of e_1 is abandonned, the values of the variables in the list l are restored, and evaluation of e_2 is undertaken. It too can effect an escape, by performing an exit or evexit with symb as its first argument. Evaluation of e_3 is then undertaken, after the variables are restored as described above. The same actions take place for e_3 ... e_n. To quit a backtrack structure exceptionally, use the form

(exit backtrack *return value*)

In LE-LISP, backtrack could be defined in the following manner:

```
(defmacro backtrack (name lvar . body)
     (unless (symbolp name)
             (error 'backtrack 'errnaa name))
     `(tag backtrack
           ,@(if (null lvar)
                 (mapcar (lambda (e)
                           `(tag ,name ,e
                                 (exit backtrack)))
                         body)
               `((let ((backtrack (list ,@lvar)))
                   ,@(mapcan
                       (lambda (e)
                         `((tag ,name ,e
```

```
                                    (exit backtrack))
                                (desetq ,lvar backtrack)))
                                body))))))))))

(pprint (macroexpand1
  '(backtrack re
              (s1 s2)
              (try1 s1)
              (try2 s2)
              (try3 s1 s2))))

(tag backtrack
     (let ((backtrack (list s1 s2)))
          (tag re (try1 s1) (exit backtrack))
          (desetq (s1 s2) backtrack)
          (tag re (try2 s2) (exit backtrack))
          (desetq (s1 s2) backtrack)
          (tag re (try3 s1 s2) (exit backtrack))
          (desetq (s1 s2) backtrack)))
```

3.9 Basic predicates

In LISP, the boolean value *false* is equivalent to the value `()`, and the boolean value *true* is equivalent to any value other than `()`. The latter fact allows you to choose any kind of representation.

Certain predicates that must return the boolean value *true* use the symbolic constant `t`. By definition, this constant is different to `()`, since the value of the symbol `t` is the symbol `t` itself.

All predicates which are false of the value `()` return their first argument if the predicate is affirmed of that argument, and `()` otherwise; all predicates which are true of the value `()` return `t` if the predicate is affirmed of its argument, and `()` otherwise.

The majority of the names of these functions end with the letter `p`, which is meant to stand for *predicate*.

This section only describes the basic predicates. Tests on integer and mixed-arithmetic expressions are described elsewhere in this manual.

(false e_1 ... e_n**)** [*function with a variable number of arguments*]

Returns the boolean value *false*—that is, `()`—regardless of the number or values of its arguments.

(true e_1 ... e_n**)** [*function with a variable number of arguments*]

Returns the boolean value *true*—that is, the symbolic constant `t`—regardless of the number or values of its arguments.

(null s) [*function with one argument*]

Tests whether `s` is equal to `()`. If so, `null` returns `t`. Otherwise, it returns `()`.

In LE-LISP, `null` could be defined in the following manner:

```
(defun null (s)
   (if (eq s ()) t ()))

(null ())   ⟹   t
(null t)    ⟹   ()
```

(not s) [*function with one argument*]

Inverts the boolean value of `s`. Due to the implementation of the boolean values in LISP, this function is identical to the function `null`. To make your programs more readable, it is preferable to use `null` to test whether a list is empty, and `not` to invert a logical value.

(atom s) [*function with one argument*]

(atomp s) [*function with one argument*]

These predicates test whether `s` is an atom: that is, either a symbol, a number or a character string. `atom` returns `t` if this is the case. Otherwise, it returns `()`. In order to handle the relation `(atom x) == (not (consp x))`, `atom` returns `t` when its argument is a vector of S-expressions. `atomp` behaves in the same manner; it is merely another name for `atom`.

```
(atom ())       ⟹   t
(atom 'argh)    ⟹   t
(atom #[1 2])   ⟹   t
(atom "oops")   ⟹   t
(atom 42)       ⟹   t
(atom '(a b))   ⟹   ()
```

(constantp s) [*function with one argument*]

Tests whether `s` is a constant: that is the special `nil` or an object that returns, when evaluated, the same object. Basic numbers, character strings, vectors of S-expressions and symbols of which `variablep` are not true are all constants. `constantp` returns `t` if the test is positive, and `()` if not.

In LE-LISP, `constantp` could be defined in the following manner:

```
(defun constantp (s)
    (if (or (fixp s)
            (floatp s)
            (vectorp s)
            (stringp s)
            (and (symbolp s) (not (variablep s))))
        t
        ()))
```

```
(constantp ())       ⟹   t
(constantp nil)      ⟹   t
(constantp 'nil)     ⟹   t
(constantp 123)      ⟹   t
(constantp 1.14)     ⟹   t
(constantp "barf")   ⟹   t
(constantp #[1 2])   ⟹   t
(constantp 'argh)    ⟹   ()
(constantp '(a b))   ⟹   ()
```

(symbolp s) [*function with one argument*]

Tests whether `s` is a symbol. `symbolp` returns `t` if so, and `()` if not.

```
(symbolp ())       ⟹   t
(symbolp 'argh)    ⟹   t
(symbolp #[1 2])   ⟹   ()
(symbolp "oops")   ⟹   ()
(symbolp 44)       ⟹   ()
(symbolp '(a b))   ⟹   ()
```

(variablep s) [*function with one argument*]

Tests whether `s` is a symbol that is not a symbolic constant: that is, whether it is a symbol different from `()`, `||`, `nil` and `t`. It returns `s` if this is so, and `()` if not.

```
(variablep ())       ⟹   ()
(variablep nil)      ⟹   ()
(variablep 'nil)     ⟹   ()
(variablep 'argh)    ⟹   argh
(variablep 123)      ⟹   ()
(variablep "barf")   ⟹   ()
(variablep #[1 2])   ⟹   ()
(variablep '(a b))   ⟹   ()
```

(numberp s) [*function with one argument*]

Tests whether `s` is a number. `numberp` returns the number `s` if the test is positive. Otherwise, it returns `()`. Tests on different types of numbers are described in chapter 4.

```
(numberp ())       ⟹   ()
(numberp 'argh)    ⟹   ()
(numberp #[1 2])   ⟹   ()
(numberp "oops")   ⟹   ()
(numberp 44)       ⟹   44
(numberp '(a b))   ⟹   ()
```

(vectorp s) [*function with one argument*]

Tests whether `s` is a vector. `vectorp` returns `s` if this is so, and `()` if not.

```
(vectorp ())       ⟹   ()
(vectorp 'argh)    ⟹   ()
(vectorp #[1 2])   ⟹   #[1 2]
(vectorp "oops")   ⟹   ()
(vectorp 44)       ⟹   ()
(vectorp '(a b))   ⟹   ()
```

(stringp s) [*function with one argument*]

Tests whether `s` is a character string. `stringp` returns `s` if this is so, and `()` if not.

```
(stringp ())       ⟹   ()
(stringp 'argh)    ⟹   ()
(stringp #[1 2])   ⟹   ()
(stringp "oops")   ⟹   "oops"
(stringp 44)       ⟹   ()
(stringp '(a b))   ⟹   ()
```

(consp s) [*function with one argument*]

Tests whether `s` is a non-empty list. `consp` returns `s` if the test is positive. Otherwise, it returns `()`. `consp` is the inverse predicate of `atom`.

```
(consp ())       ⟹   ()
(consp nil)      ⟹   ()
(consp 'nil)     ⟹   ()
(consp 'argh)    ⟹   ()
(consp #[1 2])   ⟹   ()
(consp "oops")   ⟹   ()
(consp 44)       ⟹   ()
(consp '(a b))   ⟹   (a b)
```

(listp s) [*function with one argument*]

Tests whether `s` is a list. The test includes the possibility of the empty list, written as `||`. `listp` returns the symbol `t` if the test is positive. Otherwise, it returns `()`.

The representation of the empty list as `()` or as a symbol—`||` poses the problem of the status of this symbol. `||` is simultaneuosly a symbol and the representation of the empty list. LE-LISP—like most post-MacLisp versions of LISP—resolves this ambiguity by calling upon two functions to test lists: `consp` and `listp`.

```
(listp ())       ⟹   t
(listp ||)       ⟹   t
(listp '||)      ⟹   t
```

```
(listp nil)      ⟹   t
(listp 'nil)     ⟹   ()
(listp 'argh)    ⟹   ()
(listp #[1 2])   ⟹   ()
(listp "oops")   ⟹   ()
(listp 44)       ⟹   ()
(listp '(a b))   ⟹   t
```

(nlistp s)

[function with one argument]

Tests whether `s` is neither a list nor the symbol for the empty list. In LE-LISP, the latter is written as the symbol `||`. If the test is positive, `nlistp` returns the argument `s`. Otherwise, it returns `()`. Up to the return value, this function is the inverse of the preceding one.

In LE-LISP, `nlistp` could be defined in the following manner:

```
(defun nlistp (x)
    (if (or (null x) (consp x))
        ()
        x))
```

```
(nlistp ())      ⟹   ()
(nlistp nil)     ⟹   ()
(nlistp 'nil)    ⟹   nil
(nlistp 'argh)   ⟹   argh
(nlistp #[1 2])  ⟹   #[1 2]
(nlistp "oops")  ⟹   "oops"
(nlistp 44)      ⟹   44
(nlistp '(a b))  ⟹   ()
```

(eq s1 s2)

[function with two arguments]

`eq` tests whether the two symbols `s1` and `s2` are equal, and returns `t` if this is the case. Otherwise, it returns `()`. When the arguments are not symbols, the `eq` function tests for the equality of the internal representations of `s1` and `s2`. In fact, `eq` tests for the equality of the physical addresses of its two arguments.

Since the internal representation of numbers, vectors and character strings varies with different implementations of the system, it is recommended that the function `=` (which is described with the numeric predicates) be used to test for the equality of numeric values. Similarly, the `eqvector` function should be used for the same test on vectors, and `eqstring` should be used to test for equality between two strings.

```
(eq () nil)          ⟹   t
(eq nil 'nil)        ⟹   ()
(eq 'a (car '(a)))   ⟹   t
(eq #[1 2] #[1 2])   ⟹   () or t   (depending on the implementation)
(eq "strr" "strr")   ⟹   () or t   (depending on the implementation)
```

```
(eq (1+ 119) 120)    ==>   t or ()    (depending on the implementation)
(eq '(a b) '(a b))   ==>   ()
(setq l '(x y))      ==>   (x y)
(eq l l)             ==>   t
```

(neq s1 s2) [*function with two arguments*]

`neq` is equivalent to `(not (eq s1 s2))`.

In LE-LISP, `neq` could be defined in the following manner:

```
(defun neq (s1 s2)
   (if (eq s1 s2) () t))
```

(equal s1 s2) [*function with two arguments*]

This functions carries out the most general kind of comparison that is available in LE-LISP, and must be used if the exact type of the objects to be compared is not known. Up to the return value, when `s1` and `s2` are symbols, `equal` is identical to the `eq` function. `equal` is identical to `eqn` in the case of integer numbers, and to `feqn` in the case of floating-point numbers. If `s1` and `s2` are vectors, `equal` is identical to the `eqvector` function. If the arguments are character strings, `equal` is identical to the `eqstring` function. If they are lists, `equal` tests whether they have the same structure: that is, whether they have the same elements. In all cases, if the test is positive, `equal` returns `t`. Otherwise, it returns `()`.

To compare extended arithmetic numbers, the `=` function must be used.

In LE-LISP, `equal` could be defined in the following manner:

```
(defun equal (s1 s2)
    (tag no (if (equalaux s1 s2) t ())))

(defun equalaux (s1 s2)
    (cond ((symbolp s1)
              (if (symbolp s2)
                  (eq s1 s2)
                  (exit no ())))
          ((or (fixp s1) (floatp s1))
             (if (or (fixp s2) (floatp s2))
                  (= s1 s2)
                  (exit no ())))
          ((vectorp s1)
             (if (vectorp s2)
                  (eqvector s1 s2)
                  (exit no ())))
          ((stringp s1)
             (if (stringp s2)
                  (eqstring s1 s2)
```

```
                (exit no ()))))
          (t (if (consp s2)
                 (and (equalaux (car s1) (car s2))
                      (equalaux (cdr s1) (cdr s2)))
                 (exit no ()))))))
```

```
(equal () ())                          ⟹   t
(equal 'a '|a|)                        ⟹   t
(equal 1214 (1+ 1213))                 ⟹   t
(equal 10 10.)                         ⟹   t
(equal #[1 2] #[1 2])                  ⟹   t
(equal #[1 2 3] #[1 2])                ⟹   ()
(equal "foo bar" "foo bar")            ⟹   t
(equal '(a (b . c) d) '(a (b . c) d))  ⟹   t
(equal '(a b c) '(a b c d))            ⟹   ()
```

John McCarthy posed an interesting problem. Can we find non-trivial solutions for `s` such that `(equal s (eval s))`? Here are three proposed solutions, due to Christian Queinnec:

Solution #1:

```
(setq s1 '((lambda (s) (list (car s)
                             (subst s '= (cadr s))))
           '((lambda (s)
                (list (car s)
                      (subst s '= (cadr s))))
             '=)))

(equal s1 (eval s1))  ⟹   t
```

Solution #2:

```
(setq s2 '((lambda (s) (list (car s) (kwote s)))
           '((lambda (s) (list (car s) (kwote s))))))

(equal s2 (eval s2))  ⟹   t
```

Solution #3:

```
(setq s3 '((flambda (s) `(,s ,s))
           (flambda (s) `(,s ,s))))

(equal s3 (eval s3))  ⟹   t
```

`(nequal s1 s2)` *[function with two arguments]*

The form `(nequal s1 s2)` is equivalent to `(not (equal s1 s2))`.

In LE-LISP, `nequal` could be defined in the following manner:

```
(defun nequal (s1 s2)
    (if (equal s1 s2) () t))
```

3.10 Functions on lists

3.10.1 Search functions on lists

All these functions accept a single list `l` as their argument. It can be the empty list `()`. If any other type or number of arguments is given, the `errnla` error is raised.

(car l) [*function with one argument*]

If `l` is a list, `car` returns its first element. If `l` equals `()`, `car` returns `()`. The `car` of a symbol other than `||`, of a number, a vector or a character string is undefined, and therefore raises the `errnla` error. This error is not generated by compiled code.

```
(car '(a b c))    ⟹   a
(setq x '(u p))   ⟹   (u p)
(car x)           ⟹   u
(car 'x)          ⟹      ** car : not a list : x
```

(cdr l) [*function with one argument*]

If `l` is a list, `cdr` returns this list without its first element. The `cdr` of `()` is, by definition, `()`. The `cdr` of a symbol other than `||`, of a number, a vector or a character string is undefined, and raises—as in the case of the previous function, `car`—the `errnla` error under the interpreter. In compiled code, this error does not occur for bad arguments to `cdr`.

```
(cdr '(a b c))    ⟹   (b c)
(cdr '(x . y))    ⟹   y
(cdr ())          ⟹   ()
```

(c----r l) [*functions of one argument*]

There are overlapping combinations of `car` and `cdr`, with up to four letters between the `c` and the `r`. This gives rise to 28 different functions.

- `(cadr l)` is equivalent to `(car (cdr l))`.
- `(cdaar l)` is equivalent to `(cdr (car (car l)))`.
- `(cadaar l)` is equivalent to `(car (cdr (car (car l))))`.

(memq symb l) [*function with two arguments*]

If the symbol `symb` is an element of the list `l`, `memq` returns the sublist of `l` that begins at the first occurrence of the symbol `symb`. Otherwise, it returns `()`. This function uses the `eq` predicate to test for the presence of `symb` in the list `l`. This function is often used as a predicate to test for the presence of an element in a given list.

In LE-LISP, `memq` could be defined in the following manner:

```
(defun memq (at l)
    (cond ((atom l) ())
          ((eq at (car l)) l)
          (t (memq at (cdr l)))))
```

```
(memq 'c '(a b c d e))  ⟹  (c d e)
(memq 'z '(a b c d e))  ⟹  ()
(memq 'f '(a . f))      ⟹  ()
```

(member s l)

[*function with two arguments*]

If the expression `s` is an element of the list `l`, `member` returns the sublist of `l` that begins at the first occurrence of `symb` in `l`. Otherwise, it returns `()`. This function uses the `equal` predicate, and so facilitates searches for elements of any type whatsoever in a list.

In LE-LISP, `member` could be defined in the following manner:

```
(defun member (s l)
    (cond ((atom l) ())
          ((equal s (car l)) l)
          (t (member s (cdr l)))))
```

```
(member 'c '(a b c d e))      ⟹  (c d e)
(member 'z '(a b c d e))      ⟹  ()
(member 'f '(a . f))          ⟹  ()
(member '(a b) '(a (a b) c))  ⟹  ((a b) c)
```

(tailp s l)

[*function with two arguments*]

If `s` is one of the `cdr`s of `l`, `tailp` returns `s`. Otherwise, it returns `()`.

In LE-LISP, `tailp` could be defined in the following manner:

```
(defun tailp (s l)
    (cond ((atom l) ())
          ((eq s l) s)
          (t (tailp s (cdr l)))))
```

```
(setq l '(a b c d) ll (cddr l))  ⟹  (c d)
(tailp ll l)                     ⟹  (c d)
(tailp '(c d) '(a b c d))        ⟹  ()    tailp uses eq
```

(nthcdr n l)

[*function with two arguments*]

Returns the sublist of `l` that begins with its n^{th} `cdr`. If `l` is not a list, or if `(length l)` is less than `n`, `nthcdr` returns `()`. If `n` is less than or equal to zero, `nthcdr` returns the entire list `l`.

In LE-LISP, `nthcdr` could be defined in the following manner:

```
(defun nthcdr (n l)
    (if (<= n 0)
        l
        (nthcdr (1- n) (cdr l))))
```

```
(nthcdr 3 '(a b c d e))  ⟹   (d e)
(nthcdr 0 '(a b))        ⟹   (a b)
(nthcdr 10 '(a b))       ⟹   ()
```

(nth n l) *[function with two arguments]*

Returns the n^{th} element of the list `l`, counting the `car` of `l` as its zeroth element. `nth` is equivalent to

```
(car (nthcdr n l))
```

In LE-LISP, `nth` could be defined in the following manner:

```
(defun nth (n l)
    (if (<= n 0)
        (car l)
        (nth (1- n) (cdr l))))
```

```
(nth 3 '(a b c d e f))  ⟹   d
(nth 10 '(a b c))       ⟹   ()
```

(last s) *[function with one argument]*

If `s` is a list, `last` returns the list composed of its final element. If `s` is not a list, `last` returns `s`.

In LE-LISP, `last` could be defined in the following manner:

```
(defun last (s)
    (if (or (atom s) (atom (cdr s)))
        s
        (last (cdr s))))
```

```
(last '(a b c d e))  ⟹   (e)
(last '(a b c . d))  ⟹   (c . d)
(last 120)           ⟹   120
```

(length s) *[function with one argument]*

If `s` is a list, `length` returns the number of elements in `s`. If `s` is not a list—that is, if `s` is an atom—`length` returns zero.

In LE-LISP, `length` could be defined in the following manner:

```
(defun length (s)
```

```
(if (atom s)
    0
    (1+ (length (cdr s)))))
```

```
(length '(a (b c) d e))  ⟹  4
(length '(a b . c))      ⟹  2
(length "esar")          ⟹  0
```

3.10.2 List creation functions

All the functions that are described in this subsection build new lists. The management of memory allocated to lists is dynamic and automatic. (See the section on the garbage collector.)

(cons s1 s2) *[function with two arguments]*

Builds a list that has `s1` as its first element and `s2` as the rest. If `s2` is an atom, `cons` builds the dotted pair `(s1 . s2)`.

If we suppose that the variable `free` contains the list of available memory cells, `cons` could be described in LE-LISP in the following manner:

```
(defun cons (x y)
    (rplaca free x)
    (rplacd (prog1 free (setq free (cdr free))) y))
```

```
(cons 'a '(b c))                ⟹  (a b c)
(cons 1 'x)                     ⟹  (1 . x)
(setq s '(x y z))               ⟹  (x y z)
(equal (cons (car s) (cdr s)) s)  ⟹  t
```

(xcons s1 s2) *[function with two arguments]*

This function is equivalent to `(cons s2 s1)`. The order of the arguments is simply reversed.

```
(xcons 1 2)              ⟹  (2 . 1)
(setq x 10)              ⟹  10
(xcons (incr x) (incr x))  ⟹  (12 . 11)
```

(ncons s) *[function with one argument]*

This function is the equivalent of `(cons s ())` or `(list s)`.

(mcons s_1 ... s_n) *[function with a variable number of arguments]*

This function performs a multiple `cons`. The form

`(mcons` s_1 s_2 ... s_{n-1} s_n`)`

is equivalent to the form

(cons s_1 (cons s_2 ... (cons s_{n-1} s_n) ...))

In LE-LISP, mcons could be defined in the following manner:

```
(df mcons l
    (cons (eval (car l))
          (eval (if (null (cddr l))
                    (cadr l)
                    (cons 'mcons (cdr l))))))
```

mcons has been defined here in the form of an fexpr, assuming the presence of at least two arguments. Alternatively, it could be defined in the form of a macro:

```
(dm mcons (mcons . l)
    (list 'cons (cadr l)
          (if (null (cdddr l))
              (caddr l)
              (cons 'mcons (cddr l)))))
```

```
(mcons)          ⟹   ()
(mcons 'a)       ⟹   a
(mcons 'a 'b)    ⟹   (a . b)
(mcons 'a 'b 'c) ⟹   (a b . c)
```

(list s_1 ... s_n) [*function with a variable number of arguments*]

Returns the list of the values of the expressions s_1 ... s_n. In terms of cons, the call

(list s_1 s_2 ... s_{n-1} s_n)

is equivalent to

(cons s_1 (cons s_2 ... (cons s_{n-1} (cons s_n ())) ...))

In LE-LISP, list could be defined in the following manner:

```
(defun list l l)
```

list has been defined here in the form of an expr. Alternatively, we could define it in the form of an fexpr:

```
(df list l
    (let ((r))
         (while l (newr r (eval (nextl l))))
         r))
```

```
(list 'a 'b 'c) ⟹   (a b c)
(list)          ⟹   ()
```

(kwote s) *[function with one argument]*

Builds a two-element list that has the symbol `quote` as its first element, and the value of `s` as its second argument.

In LE-LISP, `kwote` could be defined in the following manner:

```
(defun kwote (s)
    (list 'quote s))
```

```
(kwote 'a)              ⟹   'a
(kwote (cdr '(a . b)))  ⟹   'b
```

(makelist n s) *[function with two arguments]*

Returns a list of `n` elements, each one having the value of `s` as its own value. This function is useful for building lists of a given length.

In LE-LISP, `makelist` could be defined in the following manner:

```
(defun makelist (n s)
    (if (<= n 0)
        ()
        (cons s (makelist (1- n) s))))
```

```
(makelist 4 'a)  ⟹   (a a a a)
```

(append l_1 ... l_n) *[function with a variable number of arguments]*

Returns the concatenation of a copy of the first level of all the lists l_1 ... l_{n-1} to the list l_n. Arguments that are not lists are ignored, except that l_n, if atomic, becomes the last `cdr` of the resulting list. To copy the first level of a list `l`, use the form `(append l ())`.

In LE-LISP, `append` could be defined in the following manner:

```
(defun append l
    (append2 (car l)
             (if (null (cddr l))
                 (cadr l)
                 (apply 'append (cdr l)))))

(defun append2 (l1 l2)
    (if (null l1)
        l2
        (cons (car l1) (append2 (cdr l1) l2))))
```

```
(append '(a b c) '(x y))      ⟹   (a b c x y)
(append () '(x y) '(z))       ⟹   (x y z)
(append '(a) 1 '(b) 2 '(c))   ⟹   (a b c)
(append '(a) 1 '(b) 2 'c)     ⟹   (a b . c)
```

```
(setq l1 '(a b c))          ⟹   (a b c)
(eq l1 (append l1))         ⟹   t
(setq l3 (append l1 ()))    ⟹   (a b c)
(eq l1 (append l1 ()))      ⟹   ()
```

(append1 l s)

[*function with two arguments*]

The form `(append1 l s)` is equivalent to `(append l (list s))`.

This allows you to copy the first level of the list `l` and add the element `s` to its tail.

In LE-LISP, `append1` could be defined in the following manner:

```
(defun append1 (l s)
    (append l (list s)))
```

```
(append1 '(a b) 'c)  ⟹   (a b c)
```

(reverse s)

[*function with one argument*]

Returns a reversed copy of the first level of the list `s`.

In LE-LISP, `reverse` could be defined in the following manner:

```
(defun reverse (s)
    (letn rev2 ((s s) (r ()))
          (if (atom s)
              r
              (rev2 (cdr s) (cons (car s) r)))))
```

```
(reverse '(a (b c) d))  ⟹   (d (b c) a)
```

It is sometimes claimed that the following function also reverses its argument:

```
(defun rev (l)
    (if (null (cdr l))
        l
        (cons (car (rev (cdr l)))
              (rev (cons (car l)
                         (rev (cdr (rev (cdr l))))))))))
```

(copylist l)

[*function with one argument*]

Builds a new copy of `l` in its entirety. In this function, the copying takes place at every level of nesting within `l`. To copy character strings, the cells of labelled lists and vectors of S-expressions, use the next function to be described.

In LE-LISP, `copylist` could be defined in the following manner:

```
(defun copylist (s)
```

```
(if (atom s)
    s
    (cons (copylist (car s)) (copylist (cdr s)))))
```

```
(copylist 'a)               ⟹   a
(copylist '(a (b (c (d))))) ⟹   (a (b (c (d))))
(copylist '#(a b . #(d)))   ⟹   (a b d)
```

This function does not handle circular or shared lists. To solve this problem, use the following function:

```
(defun circopylist (l)
    (let ((d) (e))
         (letn circ1 ((l l))
               (cond ((atom l) l)
                     ((cdr (assq l d)))
                     (t (setq e (cons () ())
                              d (acons l e d))
                        (rplac e
                               (circ1 (car l))
                               (circ1 (cdr l))))))))
```

(copy s) *[function with one argument]*

Returns a distinct copy of the object `s`. This is true for character strings, cells of labelled lists and vectors of S-expressions.

In LE-LISP, `copy` could be defined in the following manner:

```
(defun copy (s)
    (cond ((atom s)
           (cond
            ((stringp s) (substring s 0))
            ((vectorp s)
               (let ((v (makevector (vlength s) ())))
                    (for (i 0 1 (1- (vlength s)))
                         (vset v i
                             (copy (vref s i))))
                    v))
            (t s)))
          ((tconsp s)
             (tcons (copy (car s)) (copy (cdr s))))
          (t (cons (copy (car s)) (copy (cdr s))))))
```

```
(setq strg "gdy jest")    ⟹   "gdy jest"
(eq strg strg)            ⟹   t
(eq strg (copy strg))     ⟹   ()
(equal strg (copy strg))  ⟹   t
```

```
(setq vect #[1 2])          ⟹   #[1 2]
(eq vect vect)              ⟹   t
(eq vect (copy vect))       ⟹   ()
(equal vect (copy vect))    ⟹   t
(copy '#(a b . #(d)))       ⟹   #(a b . #(d))
```

(firstn n l)

[function with two arguments]

Returns a copy of the first `n` elements of the list `l`. It might happen that `l` contains fewer than `n` elements. In other words, `n` is greater than `(length l)`. In this case, `firstn` returns a copy of the entire list `l`.

In LE-LISP, `firstn` could be defined in the following manner:

```
(defun firstn (n l)
    (cond ((null l) ())
          ((<= n 0) ())
          (t (cons (car l) (firstn (1- n) (cdr l))))))
```

```
(firstn 3 '(a b c d e f))   ⟹   (a b c)
(firstn 5 '(a b c d))       ⟹   (a b c d)
(firstn 0 '(a b c))         ⟹   ()
```

(lastn n l)

[function with two arguments]

Returns a copy of the last `n` elements of the list `l`. It might happen that the list contains fewer than the number of elements requested. In other words, `(length l)` is less than `n`. In this case, `lastn` returns a copy of the entire list `l`.

In LE-LISP, `lastn` could be defined in the following manner:

```
(defun lastn (n l)
    (reverse (firstn n (reverse l))))
```

```
(lastn 2 '(a b c d e))   ⟹   (d e)
(lastn 10 '(a b c))      ⟹   (a b c)
(lastn 0 '(a))           ⟹   ()
```

(subst s1 s2 s)

[function with three arguments]

Builds a new copy of the expression `s`, substituting the expression `s1` for each occurrence of the expression `s2` in `s`. The substitution takes place at every level of `s`. This function uses the `equal` predicate to perform the test, and builds a new expression by sharing the greatest possible number of list cells with the original expression `s`.

Use the form `(nsubst s1 s2 (copy s))` to obtain a distinct copy of the expression. By this, we mean a copy that shares no cells whatsoever with the original list `s`.

In LE-LISP, `subst` could be defined in the following manner:

```
(defun subst (new old s)
    (cond ((equal s old) new)
          ((atom s) s)
          (t (let ((car (subst new old (car s)))
                   (cdr (subst new old (cdr s))))
                  (if (and (eq car (car s))
                           (eq cdr (cdr s)))
                      s
                      (cons car cdr)))))))
```

```
(subst '(x y z) 'a '(a c (d a)))  ⟹  ((x y z) c (d (x y z)))
```

(remq symb l) [*function with two arguments*]

Returns a copy of `l` in which occurrences of the symbol `symb` have been removed. This modification is effected only at the first level of `l`.

In LE-LISP, `remq` could be defined in the following manner:

```
(defun remq (symb l)
    (cond ((atom l) l)
          ((eq symb (car l)) (remq symb (cdr l)))
          (t (cons (car l) (remq symb (cdr l))))))
```

```
(remq 'a '(a b a (c a b) d a s))  ⟹  (b (c a b) d s)
(remq 'a '(a . b))                ⟹  b
(remq 'a '(a b . a))              ⟹  (b . a)
(remq 'a '(a a a))                ⟹  ()
```

(remove s l) [*function with two arguments*]

This function is identical to the last one described, except that it uses `equal` to test for occurrences of `s` in `l`.

In LE-LISP, `remove` could be defined in the following manner:

```
(defun remove (s l)
    (cond ((atom l) l)
          ((equal s (car l)) (remove s (cdr l)))
          (t (cons (car l) (remove s (cdr l))))))
```

```
(remove '(a) '(a b (a) (c a b) (a) s))  ⟹  (a b (c a b) s)
(remove '(a) '((a) (a) (a)))            ⟹  ()
```

3.10.3 Functions on cells of labelled lists

These functions use an *invisible bit* that can be set and tested for each list cell. By default, the external representation of a labelled list cell—one whose invisible bit is set—is `#(car . cdr)` for

input and output functions. However, it is possible to effect the evaluation and the printing of these labelled list cells by means of user functions. (See the section on extended types in chapter 2.)

(tconsp s) *[function with one argument]*

Returns `s` if `s` is a labelled `cons`, and `()` if this is not the case. This function cannot be described in LISP.

(tconsmk s) *[function with one argument]*

Sets the invisible bit for the `cons` referred to as `s`. The final letters in the name of this function, `mk`, stand for 'mark': a synonym of 'set'. This function cannot be described in LISP.

```
(setq x '(a . b))  ⟹   (a . b)
(tconsmk x)        ⟹   #(a . b)
x                  ⟹   #(a . b)
```

(tconscl s) *[function with one argument]*

Clears the invisible bit for the `cons` referred to as `s`. This function cannot be described in LISP.

```
(setq x '#(a . b))  ⟹   #(a . b)
(tconscl x)         ⟹   (a . b)
x                   ⟹   (a . b)
```

(tcons s1 s2) *[function with two arguments]*

Creates a labelled `cons` with `s1` as its `car` and `s2` as its `cdr`.

In LE-LISP, `tcons` could be defined in the following manner:

```
(defun tcons (car cdr)
    (tconsmk (cons car cdr)))

(setq x (tcons 'a '(b c)))  ⟹   #(a b c)
(tcons 'd (cons 'e x))      ⟹   #(d e . #(a b c))
```

3.10.4 Physical modification functions

All the functions presented in this sub-section must be used with caution, and exactly as described. Otherwise, you might run into serious problems. These functions enable you to physically modify LISP structures. Be especially careful whenever you set out to physically modify *shared* lists. This possibility of operating upon the internal representations of lists means that LISP offers the same power that you find in machine languages.

(rplaca l s) *[function with two arguments]*

Replaces the `car` of the list `l` by `s`. It returns the modified list `l` as its value. If the argument `l` is not a list, the `errnla` error occurs.

```
(rplaca '(a b c) '(x y))  ==>  ((x y) b c)
(rplaca 'x 'foo)          ==>    ** rplaca : not a list : x
```

(rplacd l s) *[function with two arguments]*

Replaces the `cdr` of the list `l` with `s`. It returns the new list `l` as its value. If the argument `l` is not a list, the `errnla` error occurs.

```
(rplacd '(a b c) '(x y z))  ==>  (a x y z)
(rplacd () 't)              ==>    ** rplacd : not a list : ()
```

(rplac l s1 s2) *[function with three arguments]*

Replaces the `car` of the list `l` with `s1` and its `cdr` with `s2`, and returns the modified list. If the argument `l` is not a list, the `errnla` error occurs.

In LE-LISP, `rplac` could be defined in the following manner:

```
(defun rplac (l s1 s2)
    (if (atom l)
        (error 'rplac 'errnla l)
        (rplaca l s1)
        (rplacd l s2)
        l))
```

```
(setq l1 '(a b) l2 l1)  ==>  (a b)
(rplac l1 1 '(2))       ==>  (1 2)
l1                      ==>  (1 2)
l2                      ==>  (1 2)
```

(displace l ln) *[function with two arguments]*

Replaces the `car` of the list `l` with the `car` of `ln`, and the `cdr` of `l` with the `cdr` of `ln`. If `ln` is not a list, it is transformed into `(progn ln)`. This function is often used in macros, to physically modify the `macro` call itself.

In LE-LISP, `displace` could be defined in the following manner:

```
(defun displace (l ln)
    (if (atom l)
        (error 'displace 'errnla l)
        (if (atom ln)
            (rplac l 'progn (list ln))
            (rplac l (car ln) (cdr ln)))))
```

```
(setq l1 '(a b c))    ==>  (a b c)
```

```
(setq l2 l1)          ==>   (a b c)
(displace l1 '(x y))  ==>   (x y)
l2                    ==>   (x y)
(displace l1 'z)      ==>   (progn z)
l2                    ==>   (progn z)
```

(placdl l s)

[function with two arguments]

Lets you attach a new cell to the `cdr` of the list `l`. This new cell has `s` as its `car`. `placdl` returns this new cell as its value. This function is used extensively to build lists—in the right order—in one single non-recursive pass.

In LE-LISP, `placdl` could be defined in the following manner:

```
(defun placdl (l s)
    (cdr (rplacd l (cons s ()))))
```

(evlis l)

[function with one argument]

Returns a list composed of the values of the evaluations of all the elements of `l`.

In LE-LISP, `evlis` could be defined in the following manner:

```
(defun evlis (l)
    (cond ((null l) ())
          ((null (cdr l)) (list (eval (car l))))
          (t (let ((head (list (eval (car l)))))
                  (evlisaux (cdr l) head)
                  head)))))

(defun evlisaux (rest current)
    (when rest
          (evlisaux (cdr rest)
          (placdl current (eval (car rest))))))
```

(nconc l_1 ... l_n)

[function with a variable number of arguments]

Physically concatenates all the lists l_i. In other words, it places a pointer to the list l_i in the `cdr` of the last element of the list l_{i-1}. It returns the new list l_1 as its value. If two of the l_i lists are the same physical pointers, `nconc` builds a circular list by forcing the last `cdr` of the list to point to the first element of this same list. Arguments that are not lists are ignored, except for the final argument, l_n. If this last argument is atomic, it becomes the final `cdr` of the resulting list.

In LE-LISP, `nconc` could be defined in the following manner:

```
(defun nconc2 (l1 l2) ; Assume that l1 and l2 are well-formed lists
    (let ((l1 l1))
```

```
        (while (consp (cdr l1)) (nextl l1))
        (rplacd l1 l2))
    l1)
```

```
(setq x1 '(a b c) x2 x1)  ==>  (a b c)
(nconc x1 '(d e) '(f g))  ==>  (a b c d e f g)
x2                        ==>  (a b c d e f g)
(nconc () '(a) () '(b))   ==>  (a b)
(setq x1 '(a b c))        ==>  (a b c)
(nconc x1 x1)             ==>  (a b c a b c a b ...
```

(nconc1 l s)

[function with two arguments]

Physically adds the new element `s` to the end of the list `l`.

In LE-LISP, `nconc1` could be defined in the following manner:

```
(defun nconc1 (l a)
    (nconc l (list a)))
```

```
(setq l1 '(a b c) l2 l1)  ==>  (a b c)
(nconc1 l1 'd)            ==>  (a b c d)
l2                        ==>  (a b c d)
```

(cirlist e_1 ... e_n)

[function with a variable number of arguments]

Builds a circular list composed of the elements e_1 ... e_n. Since this function builds a new list, it does not modify the elements. Among other things, `cirlist` provides the means to build invariants in `map`-type functions.

In LE-LISP, `cirlist` could be defined in the following manner:

```
(defun cirlist l
    (nconc l l))
```

```
(cirlist 'a)       ==>  (a a a a a a ...
(cirlist 'a 'b 'c) ==>  (a b c a b c a b c ...
```

(nreverse l)

[function with one argument]

Reverses, physically and rapidly, the list `l`. This function must be used with caution, since it physically modifies the whole list. This brings about a disaster if the list was shared.

In LE-LISP, `nreverse` could be defined in the following manner:

```
(defun nreverse (l)
    (if (consp l)
        (letn nr ((l l) (r))
              (if (null (cdr l))
```

```
                (rplacd l r)
                (nr (cdr l) (rplacd l r))))
        l))
```

```
(setq l1 '(a b c d e))  ==>  (a b c d e)
(setq l2 (cdr l1))      ==>  (b c d e)
(setq l3 (last l1))     ==>  (e)
(nreverse l1)           ==>  (e d c b a)
l1                      ==>  (a)
l2                      ==>  (b a)
l3                      ==>  (e d c b a)
```

(nreconc l s)

[function with two arguments]

Reverses, physically and rapidly, the list `l`, and adds—by means of a physical `nconc`—the list `s`. This function therefore corresponds to `(nconc (nreverse l) s)`.

This function must be used with caution, for it physically modifies LE-LISP structures. Operations on shared structures can be particularly confusing. On the other hand, this function is far more rapid than the conventional `reverse`.

In LE-LISP, `nreconc` could be defined in the following manner:

```
(defun nreconc (l s)
    (if (consp l)
        (nconc (nreverse l) s)
        s))
```

```
(setq l1 '(a b c d e))  ==>  (a b c d e)
(setq l2 (cdr l1))      ==>  (b c d e)
(setq l3 (last l1))     ==>  (e)
(nreconc l1 '(x y))     ==>  (e d c b a x y)
l1                      ==>  (a x y)
l2                      ==>  (b a x y)
l3                      ==>  (e d c b a x y)
```

(nsubst s1 s2 l)

[function with three arguments]

Physically modifies the list `l` by substituting the expression `s1` for each occurrence of the expression `s2` in `l`. This function uses the `equal` predicate to test for occurences of `s2` in `l`. To obtain a copy of the modified list, instead of changing `l` physically, use the `subst` function.

In LE-LISP, `nsubst` could be defined in the following manner:

```
(defun nsubst (new old s)
    (cond ((equal s old) new)
          ((atom s) s)
          (t (rplac s
```

```
                    (nsubst new old (car s))
                    (nsubst new old (cdr s)))))))
```

```
(setq l '(a c (d a)))   ⟹   (a c (d a))
(nsubst '(x y z) 'a l)  ⟹   ((x y z) c (d (x y z)))
l                       ⟹   ((x y z) c (d (x y z)))
```

(delq symb l)

[*function with two arguments*]

Physically removes all occurrences of the symbol `symb` from the first level of the list `l`, and returns the modified list. If this list was shared—for example, if a variable pointed at or into it—the values in pointers that previously effected this sharing will be possibly invalid. This function uses the `eq` predicate. In order to get a physically distinct copy of such a modified list, use the `remq` function.

In LE-LISP, `delq` could be defined in the following manner:

```
(defun delq (symb l)
    (cond ((atom l) l)
          ((eq symb (car l)) (delq symb (cdr l)))
          (t (rplacd l (delq symb (cdr l))))))
```

```
(setq l '(a b c b b d))  ⟹   (a b c b b d)
(setq l (delq 'b l))     ⟹   (a c d)
(delq 'a l)              ⟹   (c d)
l                        ⟹   (a c d)
(setq l (delq 'a l))     ⟹   (c d)
l                        ⟹   (c d)
```

(delete s l)

[*function with two arguments*]

Physically removes all occurrences of the expression `s` from the first level of the list `l`, and returns the modified list. If this list was shared—for example, if a variable pointed at or into it—the values in pointers that previously effected this sharing will be possibly invalid. This function uses the `equal` predicate. In order to get a physically distinct copy of such a modified list, use the `remove` function.

In LE-LISP, `delete` could be defined in the following manner:

```
(defun delete (s l)
    (cond ((atom l) l)
          ((equal s (car l)) (delete s (cdr l)))
          (t (rplacd l (delete s (cdr l))))))
```

```
(setq l '(a (b) c b (b) d))  ⟹   (a (b) c b (b) d)
(setq l (delete '(b) l))     ⟹   (a c b d)
l                            ⟹   (a c b d)
```

3.10.5 Functions on A-lists

The term *A-lists* stands for **association lists**, which are tables—in the SNOBOL 4 sense—that have the following structure:

((key$_1$. val$_1$) (key$_2$. val$_2$) ... (key$_n$. val$_n$))

Each element of an A-list is a pair composed of a *key* and a *value*. The key is the `car` of the table element, and the corresponding value is its `cdr`. A value is accessed by means of its key.

In all the functions to be described here, the argument `al` is an A-list. For A-list access functions that use the `eq` predicate, the keys must be symbols. For access functions that use `equal`, these keys must be S-expressions. All elements of an A-list that are not of a `cons` nature are ignored.

(acons s1 s2 al) *[function with three arguments]*

Adds an element to the A-list `al`. The element is composed of the key `s1`, and its value is `s2`. `acons` returns the newly-modified A-list as its value.

In LE-LISP, `acons` could be defined in the following manner:

```
(defun acons (s1 s2 al)
    (cons (cons s1 s2) al))
```

```
(acons 'a 10 '((b . 11) (z . 40)))  ⟹  ((a . 10) (b . 11) (z . 40))
```

(pairlis l1 l2 al) *[function with three arguments]*

Here, `l1` is a list of keys, and `l2` is a list of values. `pairlis` returns a new A-list composed of the keys in `l1` and the associated values in `l2`. If a third argument, `al`, is supplied, it is added to the end of the newly-created A-list.

In LE-LISP, `pairlis` could be defined in the following manner:

```
(defun pairlis (l1 l2 al)
    (if (and (consp l1) (listp l2))
        (acons (car l1)
               (car l2)
               (pairlis (cdr l1) (cdr l2) al))
        al))
```

```
(pairlis '(the big tree) '(le grand arbre) ())  ⟹  ((the . le) (big . grand) (tree . arbre))
(pairlis '(x y z) '(a (b)) '((a . x) (b . y)))  ⟹  ((x . a) (y b) (z) (a . x) (b . y))
```

(assq symb al) *[function with two arguments]*

Returns the element of the A-list `al` whose key—that is, whose `car`—is equal to the symbol `symb`. If there is no such element, `assq` returns `()`.

In LE-LISP, `assq` could be defined in the following manner:

```
(defun assq (symb al)
    (cond ((atom al) ())
          ((and (consp (car al))
                (eq (caar al) symb))
             (car al))
          (t (assq symb (cdr al)))))
```

```
(assq 'b '((a) (b 1) (c d e)))  ⟹  (b 1)
```

(cassq symb al) [*function with two arguments*]

`cassq` is identical to `assq`, except that it returns only the value—or `cdr`—of the element of the A-list `al` whose key matches `symb`.

So, `(cassq symb al)` is equivalent to `(cdr (assq symb al))`.

Warning: The value `()` could be returned as the successful result of a match on a key with which it is associated in `al`. On the other hand, this same `()` value could be returned because there was no match on the given key value, `symb`. It is not possible to distinguish between these two cases.

```
(cassq 'c '((a) (b 1) (c d e)))  ⟹  (d e)
```

(rassq symb al) [*function with two arguments*]

Returns the element of the A-list `al` whose value—or `cdr`—is equal to the symbol `symb`. If there is no such element, `rassq` returns `()`.

In LE-LISP, `rassq` could be defined in the following manner:

```
(defun rassq (symb al)
    (cond ((atom al) ())
          ((and (consp (car al))
                (eq (cdar al) symb))
             (car al))
          (t (rassq symb (cdr al)))))
```

```
(rassq 1 '((a) (b . 1) (c d e)))  ⟹  (b . 1)
```

(assoc s al) [*function with two arguments*]

This function is identical to `rassq`, except that it uses the `equal` predicate to test for matches between `s` and the keys of the elements in `al`. These keys can therefore be of any type whatsoever.

In LE-LISP, `assoc` could be defined in the following manner:

```
(defun assoc (s al)
   (cond
      ((atom al) ())
      ((and (consp (car al))
```

```
            (equal s (caar al)))
          (car al))
        (t (assoc s (cdr al)))))
```

```
(assoc '(b) '((a) ((b) 1) (c d e)))  ==>  ((b) 1)
```

(cassoc s al) [*function with two arguments*]

This function is identical to the `cassq` function, except that it uses the `equal` predicate to test for matches between `s` and the keys of the elements in `al`. These keys can therefore be of any type whatsoever.

`(cassoc s l)` is therefore equivalent to `(cdr (assoc s l))`.

```
(cassoc '(c) '((a) (b 1) ((c) d e)))  ==>  (d e)
```

(rassoc s al) [*function with two arguments*]

This function is equivalent to the `rassq` function, except that it uses the `equal` predicate to test for matches between `s` and the values of the elements in `al`.

In LE-LISP, `rassoc` could be defined in the following manner:

```
(defun rassoc (s al)
    (cond ((atom al) ())
          ((and (consp (car al))
                (equal s (cdar al)))
           (car al))
          (t (rassoc a (cdr al)))))
```

```
(rassoc '(d e) '((a) ((b) 1) (c d e)))  ==>  (c d e)
```

(sublis al s) [*function with two arguments*]

`sublis` returns a copy of the expression `s` in which all occurrences of the keys of the A-list `al` have been replaced by their associated values. The returned copy shares—as far as possible—the cells of the initial expression. This function uses the `assq` predicate.

In LE-LISP, `sublis` could be defined in the following manner:

```
(defun sublis (al s)
    (if (atom s)
        (let ((x (assq s al)))
             (if x (cdr x) s))
        (let ((car (sublis al (car s)))
              (cdr (sublis al (cdr s))))
             (if (and (eq car (car s))
                      (eq cdr (cdr s)))
                 s
                 (cons car cdr)))))
```

```
(sublis '((a . z) (b 2 3)) '(a (b a c) d b . b))  ⟹  (z ((2 3) z c) d (2 3) 2 3)
```

3.10.6 Sorting functions

sort *[feature]*

This feature indicates whether the sorting functions are loaded into memory.

(sort fn l) *[function with two arguments]*

Physically sorts the list `l`. The elements of the list are compared by using the two-argument function `fn`. If this list was shared—for example, if a variable pointed at or into it—the values in pointers that previously effected this sharing will be possibly invalid. `sort` returns the modified list `l` as its value. The sort effected by `sort` is *unstable*. Identical elements in the original list might not be in the same physical location in the resulting list.

In LE-LISP, `sort` could be defined in the following manner:

```
(defun sort (fn l)
    (if (null (cdr l))
        l
        (let ((l1 l) (l2))
             (setq l  (nthcdr (1- (div (length l) 2)) l)
                   l2 (cdr l))
             (rplacd l ())
             (ffusion (sort fn l1)
                      (sort fn l2)))))

(defun ffusion (l1 l2)    ; Physically merge two sorted lists
    (unless (funcall fn (car l1) (car l2))
            (psetq l1 l2 l2 l1))
    (prog1 l1
           (while (and (cdr l1) l2)
                  (when (funcall fn (car l2) (cadr l1))
                        (rplacd l1
                           (prog1 l2
                              (setq l2 (cdr l1)))))
                  (nextl l1))
           (when l2 (rplacd l1 l2))))
```

(sortl l) *[function with one argument]*

Performs a physical alphabetic sort of the list `l`.

In LE-LISP, `sortl` could be defined in the following manner:

```
(defun sortl (l) (sort 'alphalessp l))
```

```
? (mapcar 'sortl
?         '((requiem aeternam dona eis domine)
?           (et lux perpetua luceat eis)
?           (in memoria aeterna eris justus)
?           (ab auditione mala non timebit)))
= ((aeternam domine dona eis requiem)
(eis et luceat lux perpetua)
(aeterna eris in justus memoria)
(ab auditione mala non timebit))
```

(sortp l) *[function with one argument]*

Performs a physical sort of the symbols in the list `l`, including packages.

In LE-LISP, `sortp` could be defined in the following manner:

```
(defun sortp (l) (sort 'pkgcmp l))

(defun pkgcmp (s1 s2)
    (if (eq (packagecell s1) (packagecell s2))
        (alphalessp s1 s2)
        (pkgcmp (packagecell s1) (packagecell s2))))
```

```
(sortp '(a #:b:a z #:b:c:a y))  ⟹  (a y z #:b:a #:b:c:a)
```

(sortn l) *[function with one argument]*

Performs a physical numeric sort of the list `l`.

In LE-LISP, `sortn` could be defined in the following manner:

```
(defun sortn (l) (sort '< l))
```

```
(sortn '(6 4 8 6 5 8 7))  ⟹  (4 5 6 6 7 8 8)
```

3.11 Functions on symbols

3.11.1 Functions that access symbol values

The value cells of a symbol are usually accessed by `eval`, when it evaluates the symbol. `eval` always checks whether a symbol possesses a value. If not, it raises the `errudv` error. The following two special functions also allow access to symbol values.

(boundp symb) *[function with one argument]*

This functions checks whether the argument `symb` is a symbol that has a value. If so, `boundp` returns `t`. Otherwise, it returns `()`. With `boundp`, you can avoid the raise of the `errudv` error for

an undefined variable. If the argument `symb` is not a symbol, `boundp` returns `()`. This function cannot be represented in LISP, which has no means for representing undefined values.

```
(boundp t)          ⟹   t
(boundp ())         ⟹   t     because () is identical to ||
(boundp 'foofoo)    ⟹   ()    if foofoo has no value
```

(symeval symb) [*function with one argument*]

`symeval` returns the value of the symbol `symb`. This function is therefore equivalent to a call to `eval`, but it is a lot more efficient, particularly for the compiler. `symeval` has the same behavior as `eval`, and it raises the same `errudv` error if its argument is undefined.

```
(setq foo 'bar)     ⟹   bar
(boundp 'foo)       ⟹   t
(symeval 'foo)      ⟹   bar
(symeval 'niema)    ⟹      ** symeval : undefined variable : niema
```

In the last example, it is assumed that `niema` has no value.

3.11.2 Functions that modify symbol values

(defvar symb e) [*special form*]

Even though variables do not need to be declared, it is good practice to define global variables with this function. It assigns the value of `e` to the value of `symb`. The expression `e` is evaluated, but not `symb`. `defvar` returns `symb` as its value. As with all other definition functions, if the value of the `#:system:loaded-from-file` symbol is not `()`, it is added to the P-list of the symbol `symb` under the `#:system:loaded-from-file` property.

#:system:loaded-from-file [*variable*]

In LE-LISP, `defvar` could be defined in the following manner:

```
(df defvar (var val)
    (set var (eval val))
    (when #:system:loaded-from-file
          (putprop var
                   #:system:loaded-from-file
                   '#:system:loaded-from-file))
    var)

(defvar foo 100)    ⟹   foo
foo                 ⟹   100
```

(makunbound symb) [*function with one argument*]

Changes the value of the symbol `symb` in such a way that all future attempts to access its value will raise the `errudv` error, indicating an undefined variable. `makunbound` returns `s` as its value.

```
(setq x 10)        ==>   10
x                  ==>   10
(makunbound 'x)    ==>   x
x                  ==>      ** eval : undefined variable : x
```

(set symb s) [*function with two arguments*]

Changes the value of the symbol `symb` into `s`. `set` returns `s` as its value.

```
(set 'x (1+ 5))    ==>   6
x                  ==>   6
```

(setq sym_1 s_1 ... sym_n s_n) [*special form*]

sym_1 ... sym_n are symbols that are not evaluated.

s_1 ... s_n are expressions that *are* evaluated. `setq` is the initialization function that is used most often in LISP. Each symbol sym_i is initialized with the value of the corresponding expression s_i. The initializations and evaluations of the expressions s_i are carried out *in sequence*. `setq` returns the result of the evaluation of s_n as its value.

```
(setq l1 '(a b c))      ==>   (a b c)
(setq l2 l1)            ==>   (a b c)
(setq l3 l2 l4 'foo)    ==>   foo
l3                      ==>   (a b c)
```

(setqq sym_1 e_1 ... sym_n e_n) [*special form*]

`setqq` is identical to `setq`, except that the values assigned to the symbols—that is, the expressions e_i—are not evaluated.

```
(setqq a 10 b (x y z) c b)   ==>   b
b                            ==>   (x y z)
c                            ==>   b
```

(psetq sym_1 s_1 ... sym_n s_n) [*special form*]

`psetq` is identical to `setq`, except that the assignments take place *in parallel*. It is very useful for permuting variable values. `psetq` returns s_1 as its value.

In LE-LISP, `psetq` could be defined in the following manner:

```
(df psetq l
    (let ((lvar) (lval))
         (while l (newl lvar (nextl l))
                  (newl lval (eval (nextl l))))
```

```
        (while lvar (set (nextl lvar) (nextl lval)))))
```

```
(setq x 10 y 11 z 12)  ⟹   12
(psetq x y y z z x)    ⟹   11
x                      ⟹   11
y                      ⟹   12
z                      ⟹   10
```

(deset l1 l2) [*function with two arguments*]

`deset` implements the *destructuring set* of the NIL system [White 79]. The argument `l1` is a tree of variables, and `l2` is a list of values. `deset` assigns the values in the tree of values to the corresponding variables in the tree of variables.

`deset` uses the same algorithm as the evaluator for binding a tree of parameters to a list of values. `deset` always returns `t` as its value.

In LE-LISP, `deset` could be defined in the following manner:

```
(defun deset (l1 l2)
    (cond ((null l1)
           (or (null l2)
               (error 'deset 'errwna l2)))
          ((variablep l1) (set l1 l2) t)
          ((atom l1) (error 'deset 'errbpa l1))
          ((and (consp l1) (consp l2))
           (deset (car l1) (car l2))
           (deset (cdr l1) (cdr l2)))
          (t (error 'deset 'errilb (list l1 l2)))))
```

```
(deset '(a (b . c)) '((1 2) (3 4)))  ⟹   t
a                                    ⟹   (1 2)
b                                    ⟹   3
c                                    ⟹   (4)
```

(desetq l1 l2) [*special form*]

`desetq` is equivalent to the preceding function, except that the first argument is not evaluated ... as in `setq`.

The form

```
(desetq l1 l2)
```

is therefore equivalent to

```
(deset (quote l1) l2)
```

(nextl sym1 sym2) [*special form*]

`sym1` must be a symbol, and its value must be a list. It is not evaluated. `nextl` returns the `car` of this list as its value, and assigns the `cdr` of this list to the value cell of `sym1`. If the second argument, `sym2`, is supplied, it too must be a symbol, which is not evaluated. In this case, the return value of `nextl`—the `car` of `sym1`—is assigned to `sym2`.

This function is useful for 'moving down through' a list that is the value of a certain symbol.

With a single argument, this function corresponds to

```
(prog1 (car symb) (setq symb (cdr symb)))
```

With two arguments, it corresponds to

```
(prog1 (setq sym2 (car sym1))
       (setq sym1 (cdr sym1)))
```

```
(setq a '(x y z))  ⟹  (x y z)
(nextl a)          ⟹  x
a                  ⟹  (y z)
(nextl a b)        ⟹  y
a                  ⟹  (z)
b                  ⟹  y
```

(newl symb s) [*special form*]

`symb` must be a symbol, and its value must be a list. It is not evaluated. `newl` puts the value of `s` at the head of this list, and returns the resulting list as its value. The combined use of the `nextl` and `newl` functions makes it easy to implement stacks as lists.

In LE-LISP, `newl` could be defined in the following manner:

```
(defmacro newl (symb s)
     `(setq ,symb (cons ,s ,symb)))
```

```
(setq a '(x y z))  ⟹  (x y z)
(newl a 'w)        ⟹  (w x y z)
a                  ⟹  (w x y z)
```

(newr symb s) [*special form*]

`symb` must be a symbol, and its value must be a list. It is not evaluated. `newr` adds the value of `s` to the tail of this list, and returns the resulting list as its value. If the value of `symb` is not a list, `newr` creates the list `(s)`, and assigns it to the value cell of `symb`.

In LE-LISP, `newr` could be defined in the following manner:

```
(defmacro newr (symb s)
     `(setq ,symb (nconc ,symb (ncons ,s))))
```

```
(setq a '(x y z))  ⟹  (x y z)
(newr a 'w)        ⟹  (x y z w)
```

```
a                ⟹   (x y z w)
(setq b ())      ⟹   ()
(newr b 'z)      ⟹   (z)
b                ⟹   (z)
```

(incr symb n) [*special form*]

`symb` must be the name of a symbol that has a numeric value. If the argument `n` is supplied, `incr` increments the value of `symb` by the value of the expression `n`. Otherwise, it increments `symb` by one. This function uses generic arithmetic. `incr` raises an error if `symb` is not a variable.

`(incr symb n)` is equivalent to `(setq symb (+ symb n))`.

`(incr symb)` is equivalent to `(setq symb (1+ symb))`.

In LE-LISP, `incr` could be defined in the following manner:

```
(df incr (var . val)
    (if (variablep var)
        (set var
             (if (consp val)
                 (+  (symeval var) (eval (car val)))
                 (1+ (symeval var))))
       (error 'incr 'errnva var)))
```

```
(setq x 10)   ⟹   10
(incr x)      ⟹   11
x             ⟹   11
(incr x 3)    ⟹   14
(incr x 2.5)  ⟹   16.5
(incr x)      ⟹   17.5
x             ⟹   17.5
```

(decr symb n) [*special form*]

`symb` must be the name of a symbol that has a numeric value. If the argument `n` is supplied, `decr` decrements the value of `symb` by the value of the expression `n`. Otherwise, it decrements `symb` by one. This function uses generic arithmetic. `decr` raises an error if `symb` is not a variable.

`(decr symb n)` is equivalent to `(setq symb (- symb n))`.

`(decr symb)` is equivalent to `(setq symb (1- symb))`.

In LE-LISP, `decr` could be defined in the following manner:

```
(df decr (var . val)
    (if (variablep var)
        (set var
             (if (consp val)
                 (-  (symeval var) (eval (car val)))
```

```
                (1- (symeval var))))
        (error 'decr 'errnva var)))
```

```
(setq x 10)    ⟹   10
(decr x)       ⟹   9
x              ⟹   9
(decr x 3)     ⟹   6
(decr x 0.5)   ⟹   5.5
(decr x)       ⟹   4.5
```

3.11.3 Functions on P-lists

The term *P-lists* stands for **property lists**, which are composed of indicators and values that have the following structure:

(`indic`$_1$ `val`$_1$ `indic`$_2$ `val`$_2$ ... `indic`$_n$ `val`$_n$)

Each indicator `indic`$_i$ is followed immediately in the P-list by its associated value, `val`$_i$. Searches on P-lists therefore take place in a two-elements-at-a-time fashion.

Each symbol has its own P-list, which is initialized to the value `()` at the creation of the symbol.

The arguments of functions on P-lists can be described as follows:

- `pl` is a symbol whose P-list is to be accessed. Functions on P-lists raise the `errnaa` or `errnva` errors if this argument is not a symbol, or if it is equal to `||`.
- `ind` is an indicator. It must be a symbol, since searches on indicators are carried out with the `eq` predicate.
- `pval` can be any expression.

(plist pl l) [*function with one or two arguments*]

If the argument `l` is not supplied, `plist` returns the P-list associated with the symbol `pl`. If `pl` has no P-list, `plist` returns `()`. If the argument `l` is supplied, it becomes the new value of the P-list of the symbol `pl`, and is returned by `plist` as its value.

`plist` cannot be described in LE-LISP in a simple manner.

```
(plist 'computer '(tool scientific toy expensive))  ⟹   (tool scientific toy expensive)
(plist 'computer)                                   ⟹   (tool scientific toy expensive)
```

(getprop pl ind) [*function with two arguments*]

(get pl ind) [*function with two arguments*]

These functions return the value in the P-list of the symbol `pl` that is associated with the indicator `ind`. If there is no such indicator in the P-list, `getprop` and `get`—which are identical—return `()`.

Warning: A value of `()` would be returned if it is the value associated with an indicator in a P-list. On the other hand, this same value might be returned because `getprop` did not find a given indicator in the P-list. There is no way of distinguishing between these two cases. (Concerning this problem, see the following function.)

In LE-LISP, `getprop` could be defined in the following manner:

```
(defun getprop (pl ind)
    (letn get1 ((pl (plist pl)))
          (cond ((atom pl) ())
                ((eq (car pl) ind) (cadr pl))
                (t (when (consp (cdr pl))
                         (get1 (cddr pl)))))))
```

```
(plist 'computer)                 ⟹   (tool scientific toy expensive)
(getprop 'computer 'tool)         ⟹   scientific
(getprop 'computer 'intelligence) ⟹   ()
```

(getl pl l) *[function with two arguments]*

`l` must be a list of indicators. `getl` determines whether one of the indicators in `l` exists in the P-list of the symbol `pl`. If so, `getl` returns the sub-list of the P-list beginning with this indicator. This being the case, `getl` provides the means to resolve the potential ambiguity arising with the use of the previous function.

In LE-LISP, `getl` could be defined in the following manner:

```
(defun getl (pl l)
    ; pl = a p-list
    ; l  = a list of indicators
    (letn getl1 ((pl (plist pl)))
          (cond
             ((atom pl) ())
             ((memq (car pl) l) pl)
             (t (when (consp (cdr pl))
                      (getl1 (cddr pl)))))))
```

```
(plist 'rose '(noun common nature flower))  ⟹   (noun common nature flower)
(getl 'rose '(nature noun))                 ⟹   (noun common nature flower)
(getl 'rose '(size nature))                 ⟹   (nature flower)
(getl 'rose '(type size))                   ⟹   ()
```

(addprop pl pval ind) *[function with three arguments]*

`addprop` adds the indicator `ind` and its associated value `pval` to the head of the P-list of the symbol `pl`. It returns `pval` as its value.

In LE-LISP, `addprop` could be defined in the following manner:

```
(defun addprop (pl pval ind)
    (plist pl (mcons ind pval (plist pl))))
```

```
(plist 'plt '(i1 a i2 b))  ⟹   (i1 a i2 b)
(addprop 'plt 'c 'i1)      ⟹   c
(plist 'plt)               ⟹   (i1 c i1 a i2 b)
```

(putprop pl pval ind) [*function with three arguments*]

If the indicator `ind` already appears in the P-list of the symbol `pl`, its associated value is changed to `pval`. Otherwise, the indicator `ind` and the value `pval` are added—in that order—to the head of the P-list ... in the same manner as for `addprop`. `putprop` returns `pval` as its value.

In LE-LISP, `putprop` could be defined in the following manner:

```
(defun putprop (pl pval ind)
    (letn put1 ((p (plist pl)))
          (cond ((atom p) (addprop pl pval ind))
                ((eq (car p) ind)
                   (rplaca (cdr p) pval) pval)
                (t (put1 (when (consp (cdr p))
                               (cddr p))))))
    pval)
```

```
(plist 'plt '(i1 a i2 b))  ⟹   (i1 a i2 b)
(putprop 'plt 'c 'i1)      ⟹   c
(plist 'plt)               ⟹   (i1 c i2 b)
(putprop 'plt 0 'i9)       ⟹   0
(plist 'plt)               ⟹   (i9 0 i1 c i2 b)
```

(defprop pl pval ind) [*special form*]

`defprop` is the `fsubr` version of the preceding function. In other words, it is the same as `defprop` except for the fact that it does not evaluate its arguments. It is useful in the case of arguments that are all constants.

```
(plist 'pink ())          ⟹   ()
(defprop pink 123 price)  ⟹   123
(defprop pink red color)  ⟹   red
(plist 'pink)             ⟹   (color red price 123)
```

(remprop pl ind) [*function with two arguments*]

`remprop` removes the indicator `ind` and its associated value from the P-list associated with the symbol `pl`. This removal only takes place, of course, if `ind` was present in the P-list. In this case, `remprop` returns the sublist of the original P-list beginning with `ind`. Otherwise, it returns `()`.

The combined use of the `remprop` and `addprop` functions facilitates the use of P-lists as property-value stacks.

In LE-LISP, `remprop` could be defined in the following manner:

```
(defun remprop (pl ind)
    (letn rem1 ((pl1 pl) (pl2 (plist pl)))
          (cond ((atom pl2) ())
                ((eq (car pl2) ind)
                   (if (atom pl1)
                       (plist pl1
                              (when (consp (cdr pl2))
                                    (cddr pl2)))
                       (rplacd pl1
                               (when (consp (cdr pl2))
                                     (cddr pl2)))))
                (t (rem1 pl2
                         (when (consp (cdr pl2))
                               (cddr pl2)))))))
```

```
(plist 'plt '(i1 a i2 b))  ==>  (i1 a i2 b)
(remprop 'plt 'i2)         ==>  (i2 b)
(plist 'plt)               ==>  (i1 a)
```

3.11.4 Access to function definitions

LE-LISP manages function definitions by means of two intrinsic properties associated with each symbol:

- `ftype`: the function *type*.
- `fval`: the function *value*.

The functions in this section are independent of packages. For an explanation of how to access a function in a particular package, see the entries for the `getfn1` and `getfn` functions.

(typefn symb) [*function with one argument*]

`typefn` returns the type of the function associated with the symbol `symb`, or `()` if the symbol has no function definition. The type of a function can be one of the following symbols: `expr`, `fexpr`, `macro`, `dmacro`, `subr0`, `subr1`, `subr2`, `subr3`, `nsubr`, `fsubr`, `msubr` or `dmsubr`. This function provides a means of determining whether the error `errudf` will be raised when the symbol `symb` is used as a function. `typefn` cannot be represented in LISP, which is incapable of representing undefined functions.

```
(typefn 'cond)    ==>  fsubr
(typefn 'foofoo)  ==>  ()   if foofoo has no function value
```

(valfn symb) [*function with one argument*]

`valfn` returns the function value associated with the symbol `symb`, or `()` if it has none. This value is an address for `subr`- or `fsubr`-type functions, and a list in the case of `exprs`, `fexprs`, `macros`, and `dmacros`.

(setfn symb ftype fval) [*function with three arguments*]

`setfn` allows you to initialize the function type, `ftype`, and the function value, `fval`, associated with the symbol `symb`. `setfn` is used as a definition function in the case of computed function names.

Warning: `setfn` performs only limited validity testing on its arguments. Whenever possible, it is preferable to use true definition functions that verify the validity of the function.

```
(setfn 'foo 'expr '((x) (+ x x)))  ⟹  foo
(typefn 'foo)                      ⟹  expr
(valfn 'foo)                       ⟹  ((x) (+ x x))
```

(resetfn symb ftype fval) [*function with three arguments*]

`resetfn` allows you to change the type, `ftype`, and the value, `fval`, of the function associated with the symbol `symb`. It also assures that they are both compatible with the previous function definition. If this is not the case, and if the `#:system:redef-flag` flag is equal to `()`, which is the default value, the following message is printed:

```
** resetfn : incompatible function : <symb>
```

`resetfn` is used by the standard definition functions to assure the compatibility of calls between compiled and interpreted functions.

```
(resetfn 'foo 'expr '((x) (+ x x)))  ⟹  foo
(typefn 'foo)                        ⟹  expr
(valfn 'foo)                         ⟹  ((x) (+ x x))
```

(findfn s) [*function with one argument*]

`findfn` searches the entire `oblist` for the symbol whose function value is equal to `s`. If found, it returns this symbol. Otherwise, it returns `()`.

In LE-LISP, `findfn` could be defined in the following manner:

```
(defun findfn (s)
    (tag found
         (mapoblist
              (lambda (symb)
                      (when (eq (valfn symb) s)
                            (exit found symb))))))

(defun foo (x) (+ x 2))  ⟹  foo
```

```
(valfn 'foo)              ⟹   ((x) (+ x 2))
(findfn (valfn 'foo))     ⟹   foo
```

(remfn symb) [*function with one argument*]

remfn removes the function definition associated with the symbol **symb**. It returns **symb** as its value.

```
(defun foo (x) (+ x x))  ⟹   foo
(foo 10)                 ⟹   20
(remfn 'foo)             ⟹   foo
(foo 10)                 ⟹      ** eval : undefined function : foo
```

(makedef symb ftyp fval) [*function with three arguments*]

makedef builds the definition of the function named **symb**, using the **ftyp** and **fval** components within a definition—that is, within a call to one of the **de**, **df**, **dm**, **defmacro** or **ds** functions.

In LE-LISP, **makedef** could be defined in the following manner:

```
(defun makedef (x typefn valfn)
    (selectq (typefn x)
        ((subr0 subr1 subr2 subr3 nsubr fsubr msubr dmsubr)
           (list 'ds x typefn valfn))
        (expr   (mcons 'defun    x valfn))
        (fexpr  (mcons 'df       x valfn))
        (macro  (mcons 'dm       x valfn))
        (dmacro (mcons 'defmacro x valfn))
        (t ())))
```

```
(makedef 'f 'expr '((x) (+ x x)))  ⟹   (defun f (x) (+ x x))
```

(getdef symb) [*function with one argument*]

getdef allows you to retrieve the definition of the function associated with the symbol **symb** in the form of its definition: that is, in the form of a call to one of the **defun**, **df**, **dm**, **defmacro**, or **ds** functions.

In LE-LISP, **getdef** could be defined in the following manner:

```
(defun getdef (x)
    (if (symbolp x)
        (makedef x (typefn x) (valfn x))
        (error 'getdef 'errsym x)))
```

```
(defun bar (n) (+ n n))  ⟹   bar
(typefn 'bar)            ⟹   expr
(getdef 'bar)            ⟹   (defun bar (n) (+ n n))
```

(revert symb) [*function with one argument*]

`revert` allows you to retrieve the former definition associated with a symbol `symb`, if this definition was previously saved by a static definition function. (See the description of these functions.)

In LE-LISP, `revert` could be defined in the following manner:

```
(defun revert (symb)
    (let ((oldef (getprop symb '#:system:previous-def)))
         (when oldef (eval oldef))))
```

(synonym sym1 sym2) [*function with two arguments*]

`synonym` lets you assign to the symbol `sym1` the type and value of the function associated with the symbol `sym2`.

In LE-LISP, `synonym` could be defined in the following manner:

```
(defun synonym (at1 at2)
    (let ((ftype (typefn at2)) (fval (valfn at2)))
         (if ftype
             (setfn at1 ftype fval)
             (error 'synonym 'errudf at2))))
```

```
(synonym 'kons 'cons)  ⟹  kons
(kons 'a 'b)           ⟹  (a . b)
```

(synonymq sym1 sym2) [*special form*]

This function is the `fsubr` form of the preceding function. In other words, it is identical except for the fact that it does not evaluate its arguments.

In LE-LISP, `synonymq` could be defined in the following manner:

```
(df synonymq (sym1 sym2)
    (synonym sym1 sym2))
```

There is an alternative definition:

```
(defmacro synonymq (sym1 sym2)
     (list 'synonym sym1 sym2))
```

`(synonymq foo bar)` is equivalent to `(synonym 'foo 'bar)`.

3.11.5 Access to symbol special fields

(objval symb s) [*function with one or two arguments*]

`objval` retrieves the `o-val` field of the symbol `symb`. If the second argument, `s`, is supplied, it becomes the new value of this field. This function cannot be described in LE-LISP in an easy manner.

```
(objval 'gee)          ⟹   ()
(objval 'gee 'haugh)   ⟹   haugh
(objval 'gee)          ⟹   haugh
```

(packagecell symb pkgc) [*function with one or two arguments*]

If the second argument is not supplied, this function reads the package field, `pkgc`, of the symbol `symb`. If this argument *is* supplied, it becomes the new package of the symbol. This function cannot be described in LE-LISP in an easy manner.

```
(defvar x '#:sator:arepo:tenet:opera:rotas)  ⟹   x
(packagecell x)                              ⟹   #:sator:arepo:tenet:opera
(packagecell (packagecell x))                ⟹   #:sator:arepo:tenet
(packagecell (packagecell (packagecell x)))  ⟹   #:sator:arepo
(packagecell x '#:en:to:pan)                 ⟹   #:en:to:pan
x                                            ⟹   #:en:to:pan:rotas
```

(getfn1 pkgc symb) [*function with two arguments*]

`getfn1` returns the symbol `#:pkgc:symb` if the symbol `symb` has a function definition in the package `pkgc`.

In LE-LISP, `getfn1` could be defined in the following manner:

```
(defun getfn1 (pkgc symb)
    (let ((nom (symbol pkgc symb)))
         (if (typefn nom)
             nom
             ())))
```

```
(getfn1 () 'car)       ⟹   car
(defun #:foo:bar ())   ⟹   #:foo:bar
(getfn1 'foo 'bar)     ⟹   #:foo:bar
(getfn1 'gee 'bar)     ⟹   ()
```

(getfn pkgc symb lastpkgc) [*function with two or three arguments*]

`getfn` searches in the package `pkgc` for a symbol named `symb` that has an associated function definition. It it finds none, it continues searching through the package hierarchy up to, but excluding, the package `lastpkgc`, or to the top of the hierarchy—to the global package `()`—if `lastpkgc` is not specified. `getfn` returns the name of this function if it was found, or `()` if not. If found, the function name could then be used as the first argument to the `apply` or `funcall` functions, for example. `getfn` is used inside the interpreter to manage programmable interrupts, the starting up of `#`-macros, and the control of the virtual terminal.

In LE-LISP, `getfn` could be defined in the following manner:

```
(defun getfn (pkgc symb . lastpkgc)
    (let ((name (symbol pkgc symb)))
         (cond ((typefn name) name)
               ((null pkgc) ())
               ((and (consp lastpkgc)
                     (eq (packagecell pkgc)
                         (car lastpkgc)))
                ())
               (t (getfn (packagecell pkgc)
                         symb lastpkgc)))))
```

```
(defun foo ())                    ==>   foo
(defun #:bar:foo ())              ==>   #:bar:foo
(defun #:bar:gee:buz:foo ())      ==>   #:bar:gee:buz:foo
(getfn '#:bar:gee:buz 'foo)       ==>   #:bar:gee:buz:foo
(getfn '#:bar:gee 'foo)           ==>   #:bar:foo
(getfn 'bar 'foo)                 ==>   #:bar:foo
(getfn () 'foo)                   ==>   foo
(getfn '#:potop:teraz 'foo)       ==>   foo
(getfn '#:bar:gee 'foo ())        ==>   #:bar:foo
(getfn 'bar 'foo ())              ==>   #:bar:foo
(getfn 'gee 'foo ())              ==>   ()
(getfn '#:bar:gee:buz 'foo 'bar)  ==>   #:bar:gee:buz:foo
(getfn '#:bar:gee 'foo 'bar)      ==>   ()
```

3.11.6 Symbol creation functions

Symbols are managed by means of a hash table shared among all the packages. The external representation of a symbol outside the current package is `#:package:symbol`. (See the information on input/output functions.)

(symbol pkgc strg) *[function with two arguments]*

`symbol` creates a new symbol named, `strg`, in the package named `pkgc`. If the `pkgc` argument is `()`, the root package is used. This function cannot be described in LE-LISP in an easy manner.

```
(symbol 'foo "bar")  ==>   #:foo:bar
(symbol () "fuu")    ==>   fuu
```

(concat str_1 ... str_n) *[function with a variable number of arguments]*

Creates a new symbol whose `p-name` is built by concatenating all the strings str_1 ... str_n.

In LE-LISP, `concat` could be defined in the following manner:

```
(defun concat lpname
    (symbol () (mapcan 'pname lpname)))
```

```
(concat 'foo (1+ 5) () 'bar)    ⟹   foo6bar
(concat "foo" nil '#"bar" -2)   ⟹   foobar-2
```

(gensym) [*function with no arguments*]

Each time it is called, `gensym` returns a new symbol of type *Gxxx*. In this type name, *G* is a string, stored in the `#:system:gensym-string` system variable, and *xxx* is a number that is incremented by one after each call to `gensym`. This number is the value of the `gensym` counter, which is stored in the `#:system:gensym-counter` system variable. When the interpreter starts up, *G* and *xxx* have the respective values of `g` and `100`.

In LE-LISP, `gensym` could be defined in the following manner:

```
(defvar #:system:gensym-string "g")
(defvar #:system:gensym-counter 100)

(defun gensym ()
    (concat #:system:gensym-string
            (incr #:system:gensym-counter)))
```

```
(gensym)                                          ⟹   g101
(gensym)                                          ⟹   g102
(gensym)                                          ⟹   g103
(let ((#:system:gensym-string "etiq")) (gensym))  ⟹   etiq104
```

3.11.7 Symbol management functions

(oblist pkgc symb) [*function with zero, one or two arguments*]

If `oblist` is called with no arguments, it returns the lengthy list of all the symbols present in the system. At system initialization, this list contains the names of all predefined functions and variables. Since this list is very long (often of the order of two thousand symbols), you are advised to use the `mapoblist`, `mapcoblist` or `maploblist` functions to sequentially access these symbols. The two optional arguments provide means of filtering the `oblist`.

If the first argument, `pkgc`, is supplied, it must be a symbol that designates the package from which the symbols are to be extracted. If the second argument, `symb`, is supplied, it too must be a symbol that designates the name of the symbols, since there could be multiple instances of `symb` in the `oblist`, in different packages. A call such as `(oblist pkgc symb)` is therefore an effective means of testing whether a symbol is present in a certain package, without creating it if it was not already there. It is not possible to define this function in LISP. To do so, we would need a means of accessing the `a-link` and `p-name` intrinsic properties of the symbols, and only the functions `loc` and `memory` provide such a means.

```
(oblist)  ⟹
```

```
(assoc sin cadadr funcall typecn eqn tracend plist
#:system:var var nom symbolp get addadr load letvq
curread false plength cddadr fileout pairlis ex*
                        .....
                        .....
                        .....
libload #:system:length-float quo inbuf untilexit xdef)
```

```
(oblist 'sharp)  ⟹
```

```
(#:sharp:" #:sharp:value #:sharp:$ #:sharp:% #:sharp:(
#:sharp:+ #:sharp:- #:sharp:. #:sharp:/ #:sharp::
#:sharp:[ #:sharp:\\ #:sharp:^ #:sharp:|)
```

(lhoblist strg)

[*function with one argument*]

`lhoblist` returns the list of all the symbols that contain `strg` as a substring in their name. This function therefore provides a way of filtering the symbols of the `oblist`. It is useful for finding the exact spelling of a function name.

In LE-LISP, `lhoblist` could be defined in the following manner:

```
(defun lhoblist (pname)
    (maploblist (lambda (symb) (index pname symb 0))))
```

```
(lhoblist "apc")     ⟹   (mapc mapcon mapcan mapcar mapcoblist)
(lhoblist "string")  ⟹
```

```
(string duplstring makestring fillstring eqstring
prinstring spanstring readstring bltstring substring
#:system:gensym-string scanstring stringp)
```

(boblist n)

[*function with an optional argument*]

If the argument `n` is not present, `boblist` returns the list of *buckets* of the global symbol hash table of the system. If the argument `n` is provided, `boblist` returns the n^{th} bucket. This function cannot be described easily in LISP.

- To obtain the size of the hash table—that is, the number of buckets—type
  `(length (boblist))`
- To obtain a list of the sizes of all the buckets, type
  `(mapcar 'length (boblist))`
- To obtain the size of the largest bucket, type
  `(apply 'max (mapcar 'length (boblist)))`
- To obtain a copy of the bucket where a symbol lives, type
  `(boblist (hash symb))`

`(remob symb)` [*function with one argument*]

`remob` removes the values of all the intrinsic properties of the symbol `symb`, so that it be destroyed during the next garbage collection ... provided that there are no other pointers to `symb`.

In LE-LISP, `remob` could be defined in the following manner:

```
(defun remob (symb)
    (makunbound symb)
    (plist symb ())
    (remfn symb)
    (objval symb ())
    symb)
```

To recover all the symbols of a package, enter

`(mapc 'remob (oblist 'pkgc))`.

The `mapoblist`, `mapcoblist` and `maploblist` functions are described in the section on application functions.

3.12 Functions on character strings

A character string is a collection of characters accessible by their index, which is a number. Indexes of character strings begin at zero. Each character string also possesses its own symbolic type. By default, this type is `string`, but it can be changed with the `typestring` function.

The external representation of a character string of type `string` is `"xxxxxxxxx"`. The external representation of a character string of any other type is `#:type:"xxxxxxxxx"`.

For all these functions, the `strg` argument must be of the type known as *character string*. If it is not of this type, it will be automatically converted to an object of this type by the `string` function. This conversion does *not* occur in the case of the basic string-manipulation functions: `slen`, `sref`, `sset`, `typestring` and `exchstring`.

If the conversion is not possible, the `errnsa` error is raised. Its default screen display is

```
** <fn> : non string argument : <s>
```

where the faulty object `s` is displayed, along with the name of the function, `fn`, that raised the error.

The functions `makestring`, `fillstring`, `set`, `chset` and `chrpos`, used for the manipulation of character strings, take as argument a character in the range `[0,256[`. In other words, a modulo `256` operation is performed automatically.

Since the size of a string is limited to `32k` (32,767) characters, the `catenate` and `duplstring` string-creation functions might raise the `errstl` error, which has the following default screen display:

```
** <fn> : string too long : <s>
```

Here, the maximum size, `s`, is displayed along with the name of the function, `fn`, that raised the error.

In the case of the basic functions, attempts to access a character beyond the string boundary—its maximum legal index value—raises the `erroob` error, which has the following default screen display:

```
** <fn> : out of bounds : <s>
```

where the faulty index, `s`, is printed along with the name of the function, `fn`, that raised the error.

The character-string test predicate `stringp` is described in the section on basic predicates.

To describe these functions in LISP, we shall use the basic `pname` function, which provides a way of converting from the internal representation of a `p-name` to its representation in the form of internal character codes. These descriptions do not reflect the real implementation details of the functions, which obviously involve no list cells.

3.12.1 Basic manipulation functions

These functions are the basic primitives for use on character strings. They are sufficient to define all other functions on character strings.

(slen strg) *[function with one argument]*

`slen` returns the length of the character string `strg`. The type test on the argument is not performed after compilation in 'open' mode. For reasons of efficiency, `slen` compiles into a single LLM3 instruction. This function cannot be described easily in LISP.

```
(slen "abc")  ⟹   3
(slen "")     ⟹   0
(slen t)      ⟹      ** slen : non string argument : t
```

(sref strg n) *[function with two arguments]*

`sref` returns the n^{th} character in the string `strg`. If `n` is beyond the limit of `strg`, the `erroob` error is raised. This test for overflow, as well as the type test, are not performed, however, after code is compiled in 'open' mode. For reasons of efficiency, `sref` compiles into a single LLM3 instruction. This function cannot be described easily in LISP.

```
(setq x "abcdef")  ⟹   "abcdef"
(sref x 0)         ⟹   97 that is, #/a
(sref x 5)         ⟹   102 that is, #/f
(sref t 1)         ⟹      ** sref : non string argument : t
(sref x -2)        ⟹      ** sref : argument out of bounds : -2
(sref x 6)         ⟹      ** sref : argument out of bounds : 6
```

(sset strg n cn) *[function with three arguments]*

`sset` sets the value if the n^{th} character in the string `strg` to `cn`. If `n` is beyond the limit of the string `strg`, the `erroob` error is raised. This test for overflow, as well as the type test, are not performed, however, after code is compiled in 'open' mode. For reasons of efficiency, `sset` compiles into a single LLM3 instruction. This function cannot be described in LE-LISP in an easy manner.

```
(setq x "abcdef")  ⟹   "abcdef"
(sset x 1 #/y)     ⟹   121 that is, #/y
(sset x 3 #/z)     ⟹   122 that is, #/z
x                  ⟹   "ayczef"
(sset x -3 #/t)    ⟹      ** sset : argument out of bounds : -3
(sset x 6 #/t)     ⟹      ** sset : argument out of bounds : 6
```

(typestring strg symb) [*function with one or two arguments*]

The second argument, if supplied, becomes the new type of the character string `strg`. In general, this `symb` type is a symbol. But it could be a list of symbols, allowing you to carry out multiple inheritance. For further details, look up the topic of object-oriented programming. The `typestring` function returns the new type of `strg`.

The `type-of` function, applied to a string, returns the value of `typestring` applied to the string. The `typestring` function cannot be described easily in LISP.

```
(setq s "abc")           ⟹   "abc"
(typestring s)           ⟹   string
(type-of s)              ⟹   string
(typestring s 'foo)      ⟹   foo
(typestring s)           ⟹   foo
s                        ⟹   #:foo:"abc"
(type-of s)              ⟹   foo
(typestring t)           ⟹      ** typestring : non string argument : t
(typestring "v" "bar")   ⟹      ** typestring : not a symbol : bar
```

(exchstring strg1 strg2) [*function with two arguments*]

This rather esoteric function provides a means of physically exchanging the values and types of the strings `strg1` and `strg2`. `exchstring` returns the modified `strg1` string: that is, the one that was initially called `strg2`. This function cannot be described easily in LISP.

```
(setq v "abc" w "de")  ⟹   "de"
(typestring v 'foo)    ⟹   foo
(setq y v z w)         ⟹   "de"
(exchstring v w)       ⟹   "de"
v                      ⟹   "de"
w                      ⟹   #:foo:"abc"
y                      ⟹   "de"
z                      ⟹   #:foo:"abc"
```

3.12.2 Character string conversion

(string s) [*function with one argument*]

`string` converts the argument `s` into a character string. The sequence of characters of this string is determined by the following rules:

- If `s` is the empty list—that is, `()`—the sequence is empty.
- If `s` is a symbol, the sequence is nothing other than the sequence of characters of the symbol's external name: its `p-name`.
- If `s` is a character string, the sequence is just this sequence of characters.
- If `s` is a list, it is assumed to be a list of internal character codes, and must not be more than 1024 in length.

If one of these conditions is not met, the `errnsa` error is raised.

```
(string ())                ⟹   ""
(string '||)               ⟹   ""
(string '|foo|)            ⟹   "foo"
(string "bar")             ⟹   "bar"
(string -345)              ⟹   "-345"
(string 2.3)               ⟹   "2.3"
(string '#"abc")           ⟹   "abc"
(string (makelist 5 #/x))  ⟹   "xxxxx"
```

(pname strg) [*function with one argument*]

`pname` returns the string `strg` in the form of a list of internal character codes. This function, which depends on internal LISP representations, cannot be simply described (in LISP).

```
(pname ())         ⟹   ()
(pname nil)        ⟹   ()
(pname 'nil)       ⟹   (110 105 108)
(pname 'foobar)    ⟹   (102 111 111 98 97 114)
(pname -123)       ⟹   (45 49 50 51)
(pname "abcdef")   ⟹   (97 98 99 100 101 102)
(pname '#"abc")    ⟹   (97 98 99)
```

(plength strg) [*function with one argument*]

(slength strg) [*function with one argument*]

These functions return the number of characters in the string `strg` as their values. There is no difference between the two.

In LE-LISP, `plength` could be defined in the following manner:

```
(defun plength (strg)
    (length (pname strg)))
```

```
(plength ())          ⟹   0
(plength nil)         ⟹   0
(plength 'nil)        ⟹   3
(plength 'foobar)     ⟹   6
(plength -123)        ⟹   4
(plength "sobota")    ⟹   6
(plength '#"dzien")   ⟹   5
```

(hash strg) *[function with one argument]*

Returns the internal hash algorithm key for the string `strg`. The performance of this algorithm can be measured by using the `boblist` function. The following description is only given by way of explanation, and the actual implementation is much speedier.

In LE-LISP, `hash` could be defined in the following manner:

```
(defun hash (strg)
    (rem (logand (hashaux (pname strg)) #$7fff)
         (length (boblist))))

(defun hashaux (pn)
    (if (<= (length pn) 6)
        (hashcount (length pn) pn (length pn))
        (hashcount 6
                   (lastn 6 pn)
                   (hashcount 6
                              (firstn 6 pn)
                              (length pn)))))

(defun hashcount (count pn val)
    (setq pn (reverse pn))
    (repeat count
            (setq val (add (logshift val 1) (nextl pn))))
    val)
```

```
(hash ())                  ⟹   0
(hash 'nil)                ⟹   776
(hash 'foobar)             ⟹   157
(hash -123)                ⟹   815
(hash "abcdef")            ⟹   771
(hash "galamantdelareine") ⟹   105
(hash '#"abcd")            ⟹   556
```

3.12.3 Comparison of character strings

(eqstring `str1 str2`**)** [*function with two arguments*]

Tests whether the two arguments—of type *character string*—are equal. If they are, `eqstring` returns `str1`. If not, it returns `()`.

In LE-LISP, `eqstring` could be defined in the following manner:

```
(defun eqstring (str1 str2)
    (and (= (slength str1) (slength str2))
         (eq (typestring str1) (typestring str2))
         (equal (pname str1) (pname str2))))
```

```
(eqstring "foo" "bar")                      ⟹   ()
(eqstring "foo" "foo")                      ⟹   "foo"
(eqstring "Foo" "foo")                      ⟹   ()
(eqstring (string (1+ 11)) (catenate 1 2))  ⟹   "12"
```

(alphalessp `str1 str1`**)** [*function with two arguments*]

Returns `t` if the `p-name` of `str1` is lexicographically less than or equal to the `p-name` of `str2`. Otherwise, it returns `()`. This function is used to perform alphabetic sorts. (See the `sortl` function.)

In LE-LISP, `alphalessp` could be defined in the following manner:

```
(defun alphalessp (str1 str2)
    (letn alphal1 ((lst1 (pname str1)) (lst2 (pname str2)))
          (cond ((null lst1) t)
                ((null lst2) ())
                ((= (car lst1) (car lst2))
                    (alphal1 (cdr lst1) (cdr lst2)))
                (t (if (< (car lst1) (car lst2)) t ())))))
```

```
(alphalessp 'a 'a)       ⟹   t
(alphalessp 'b 'a)       ⟹   ()
(alphalessp 'a 'b)       ⟹   t
(alphalessp 'zzz 'zzzz)  ⟹   t
```

3.12.4 Character-string creation functions

(catenate $\texttt{str}_1 \ldots \texttt{str}_n$**)** [*function with a variable number of arguments*]

`catenate` builds a new string, the result of the concatenation of the strings $\texttt{str}_1 \ldots \texttt{str}_n$.

The newly-formed string cannot be longer than 32k (32,767) characters, or the `errstl` error will be raised.

This function is not to be confused with the `concat` function, which creates a symbol.

In LE-LISP, `catenate` could be defined in the following manner:

```
(defun catenate lst
    (string (mapcan 'pname lst)))
```

```
(catenate "foo" () 'bar (1+ 10) '#"()")  ==>  "foobar11()"
(catenate -34 0 12)                      ==>  "-34012"
(catenate "" || () nil)                  ==>  ""
```

(makestring n cn) [*function with two arguments*]

Returns a string of `n` characters, each of which has `cn` as its internal character code.

In LE-LISP, `makestring` could be defined in the following manner:

```
(defun makestring (n cn)
    (string (makelist n cn)))
```

```
(makestring 0 #/a)   ==>  ""
(makestring 4 #/a)   ==>  "aaaa"
(makestring -1 #/a)  ==>  ""
```

(substring strg n1 n2) [*function with two or three arguments*]

Returns a copy of the substring of `strg` that starts at position `n1`—starting with count zero—and has length `n2`. If this last argument is not supplied, the returned substring is the maximal substring beginning at position `n1`. That is, it extends to the end of `strg`.

In LE-LISP, `substring` could be defined in the following manner:

```
(defun substring (strg n1 . n2)
    (string (firstn (- (if (consp n2)
                           (car n2)
                           (plength strg))
                       n1)
                    (nthcdr n1 (pname strg)))))
```

```
(substring "abcde" 0 3)  ==>  "abc"
(substring "abcde" 1 2)  ==>  "bc"
(substring "abcde" 2)    ==>  "cde"
(substring "abcde" 9 2)  ==>  ""
```

(duplstring n strg) [*function with two arguments*]

Makes a string of `n` copies of the string `strg`.

The resulting string cannot be longer than 32k (32,767) characters long, otherwise the `errstl` error is raised.

In LE-LISP, `duplstring` could be defined in the following manner:

```
(defun duplstring (n strg)
    (if (<= n 0)
        ""
        (catenate strg (duplstring (1- n) strg))))
```

```
(duplstring 3 "ab")   ⟹   "ababab"
(duplstring 3 "")     ⟹   ""
(duplstring 1 "abc")  ⟹   "abc"
(duplstring 0 "abc")  ⟹   ""
```

3.12.5 Character-string access functions

These functions provide means to manipulate single characters within a character string.

(chrpos cn strg pos) *[function with two or three arguments]*

`chrpos` returns the position of the internal character code `cn` in the string `strg`. If the third argument is not supplied, the position of the first character in `strg` is presumed to be zero. Otherwise, it is presumed to be `pos`. If the code `cn` does not occur in the string, `chrpos` returns `()`. `chrpos` can therefore be used as a predicate for testing the presence of a character in a string.

In LE-LISP, `chrpos` could be defined in the following manner:

```
(defun chrpos (cn strg . pos)
    (letn chrp ((lcn (pname strg))
                (n (if (consp pos)
                       (car pos)
                       0)))
          (cond ((null lcn) ())
                ((eq (car lcn) cn) n)
                (t (chrp (cdr lcn) (1+ n))))))
```

```
(chrpos #/y "otyoty")              ⟹   2
(chrpos #/n "oty")                 ⟹   ()
(chrpos #/d '#"0123456789abcdef")  ⟹   13
(chrpos #/a 'abc 1)                ⟹   ()
```

(chrnth n strg) *[function with two arguments]*

`chrnth` returns the internal character code of the n^{th} character in the string `strg`. The position, or index, of the first character is zero, not one. So, there is no n^{th} character in the string when `n` >= `(length strg)` or `n` < `0`. In such a case, `chrnth` returns `()`.

In LE-LISP, `chrnth` could be defined in the following manner:

```
(defun chrnth (n strg)
    (nth n (pname strg)))
```

```
(chrnth 0 "Fob")                ⟹   70   that is, #/F
(chrnth 2 "FoB")                ⟹   66   that is, #/B
(chrnth 3 "fob")                ⟹   ()
(chrnth -1 "fob")               ⟹   ()
(chrnth 10 "0123456789ABcdE")   ⟹   65   that is, #/A
```

(chrset n strg cn) *[function with three arguments]*

`chrset` provides a means of changing the n^{th} character in the string `strg` to the character with internal character code `cn`. This internal character code is returned by `chrset` as its value. The position, or index, of the first character in a string is zero. This function, which physically modifies the string, cannot be described in LISP.

```
(setq x "abc")     ⟹   "abc"
(chrset 0 x #/A)   ⟹   65   that is, #/A
x                  ⟹   "Abc"
```

3.12.6 Functions that physically modify strings

These physical-modification functions can be described in LISP using the `chrset` function.

(bltstring str1 n1 str2 n2 n3) *[function with four or five arguments]*

Copies `n3` characters from the string `str2`, beginning at position `n2`, into the string `str1`, beginning at position `n1`. All indexes are relative to zero. If the `n3` argument is not included, the whole of `str2`, beginning at position `n3`, will be transferred. This happens up to the limit of `str1`, which is never extended by this function. `bltstring` returns the modified `str1` string. `str1` and `str2` can be the same physical string. So, it is possible to write over the contents of a string with the string itself, using any combination of valid index values.

In LE-LISP, `bltstring` could be defined in the following manner:

```
(defun bltstring (str1 n1 str2 n2 n3)
    ; Does not treat the shared strings nor limit cases.
    (repeat (min (if (consp n3)
                     (car n3)
                     (- (slength str2) n2))
                 (- (slength str1) n1))
            (chrset n1 str1 (chrnth n2 str2))
            (incr n1)
            (incr n2))
    str1)
```

```
(bltstring "foobar" 1 "xyz" 2 1)   ⟹   "fzobar"
(bltstring "foobar" 1 "xyz" 0)     ⟹   "fxyzar"
(bltstring "foobar" 1 "toto" 0 6)  ⟹   "ftotor"
(bltstring "foobar" 3 "toto" 0 8)  ⟹   "footot"
```

```
(setq strt "abcdefghij")          ⟹   "abcdefghij"
(bltstring strt 1 strt 3 4)       ⟹   "adefgfghij"
(bltstring strt 6 strt 0 2)       ⟹   "adefgfadij"
(bltstring strt 0 strt 4 4)       ⟹   "gfadgfadij"
```

(fillstring strg n1 cn n2) *[function with three or four arguments]*

Fills the string `strg` with `n2` characters having internal character code `cn`, beginning at the $n1^{th}$ character. If the argument `n2` is not given, or if `n1` + `n2` is longer than the length of the string, the string will be filled only through its last character position, and not extended. `fillstring` returns the modified string as its value.

In LE-LISP, `fillstring` could be defined in the following manner:

```
(defun fillstring (strg n1 cn . n2)
    (repeat (if (consp n2)
                (min (car n2) (- (slength strg) n1)
                (- (slength strg) n1))
            (chrset n1 strg cn) (incr n1))))
```

```
(fillstring "foobar" 1 #/x 2)    ⟹   "fxxbar"
(fillstring "foobar" 0 #/y 3)    ⟹   "yyybar"
(fillstring "foobar" 2 #/x)      ⟹   "foxxxx"
(fillstring "foobar" 2 #/x 10)   ⟹   "foxxxx"
```

3.12.7 Search functions on character strings

(index str1 str2 n) *[function with two or three arguments]*

Determines whether the argument `str1` is a substring of the tail of `str2` that begins at character position `n`. If so, `index` returns the index (counting from zero) in `str2` of the first instance of this substring. Otherwise, it returns `()`.

If the `n` argument is not supplied, the search starts with zero.

In LE-LISP, `index` could be defined in the following manner:

```
(defun index (str1 str2 . n)
    (letn inxaux1 ((p1 (pname str1))
                   (p2 (nthcdr (or (car n) 0)
                               (pname str2)))
                   (n n))
          (cond ((< (length p2) (length p1)) ())
                ((<> (car p1) (car p2))
                   (inxaux1 p1 (cdr p2) (1+ n)))
                (t (letn inxaux2 ((pp1 (cdr p1))
                                  (pp2 (cdr p2)))
```

```
                        (cond ((null pp1) n)
                              ((= (car pp1) (car pp2))
                                   (inxaux2 (cdr pp1)
                                            (cdr pp2)))
                              (t (inxaux1 p1 (cdr p2)
                                          (1+ n))))))))))
```

```
(index "foo" "foobar")      ⇒   0
(index "bar" "foobar")      ⇒   3
(index "foo" "xfoobar")     ⇒   1
(index "foo" "xfoobar" 2)   ⇒   ()
(index "foo" "xfoobar" 1)   ⇒   1
(index "foo" "" 0)          ⇒   ()
(index "" "foo" 0)          ⇒   0
```

(substring-equal size strg1 pos1 strg2 pos2) *[function with five arguments]*

Compares the substring of length `size` beginning at position `pos1` in `strg1` with the substring of the same length beginning at position `pos2` in `strg2`. This comparison is carried out without creating a new string. `substring-equal` returns `size` if the comparison succeeds, or `()` if it does not. `substring-equal` can raise the `erroob`, `errnia` and `errnsa` errors.

In LE-LISP, `substring-equal` could be defined in the following manner:

```
(defun substring-equal (size strg1 pos1 strg2 pos2)
    ; this description includes no tests for value validity..
    (letn substrgeq ((count size))
          (cond ((eq count 0) size)
                ((eq (sref strg1 pos1) (sref strg2 pos2))
                 (substrgeq (sub1 count) strg1 (add1 pos1)
                                         strg2 (add1 pos2)))
                (t ()))))
```

```
(substring-equal 0 "fobar" 0 "gezu" 0)     ⇒   0
(substring-equal 5 "fobar" 0 "fobar" 0)    ⇒   5
(substring-equal 5 "fobara" 0 "fobar" 0)   ⇒   5
(substring-equal 5 "fobar" 0 "fobara" 0)   ⇒   5
(substring-equal 2 "fobar" 0 "afob" 1)     ⇒   2
(substring-equal 2 "fobar" 0 "afab" 1)     ⇒   ()
(substring-equal 2 "fobar" 3 "afar" 2)     ⇒   2
```

(scanstring str1 str2 n) *[function with two or three arguments]*

Searches from character position `n` (relative to zero, the default value of this argument) in the string `str1` for characters occurring in the string `str2`. `scanstring` returns the index—again, relative to zero—of the first character in `str1` that matches any of those in `str2`, or `()` if no such thing exists.

```
(scanstring "abcd" "sbe")     ⇒   1
```

```
(scanstring "abcd" "efg")     ⟹   ()
(scanstring "abcd" ".a" 1)    ⟹   ()
(scanstring "abc" "defghc")   ⟹   2
(scanstring "foo" "")         ⟹   ()
```

(spanstring str1 str2 n) *[function with two or three arguments]*

Searches from character position `n`—relative to zero, the default value of this argument—in the string `str1` for a character not occurring in the string `str2`. `spanstring` returns the index—again, relative to zero—of the first character of `str1` that does not occur in `str2`, or `()` if no such thing exists.

```
(spanstring "abcb" "ab")         ⟹   2
(spanstring "abcd" "aabbccdd")   ⟹   ()
(spanstring "abcd" "bcd" 1)      ⟹   ()
(spanstring "foo" "")            ⟹   0
```

3.13 Functions on characters

These functions manipulate internal character codes. Throughout this section, an internal character code is denoted by `cn`. Character coding used in LE-LISP is mainly `ascii`.

(ascii cn) *[function with one argument]*

Returns the character whose `ascii` code is `cn`, modulo `256`. A character is a LE-LISP object with a single-character `p-name`.

```
(ascii 67)       ⟹   C
(ascii 99)       ⟹   c
(1+ (ascii 49))  ⟹   2
```

(cascii ch) *[function with one argument]*

Returns the `ascii` code of the character `ch`.

```
(cascii 'c)   ⟹   99
(cascii 1)    ⟹   49
```

(uppercase cn) *[function with one argument]*

If `cn` is the internal character code of a lower-case character, then `uppercase` returns the code of the corresponding upper-case character. If not, the function returns `cn`, unchanged.

```
(uppercase #/a)   ⟹   65 that is, #/A
(uppercase #/A)   ⟹   65 that is, #/A
(uppercase #/2)   ⟹   50 that is, #/2
```

(lowercase cn) [*function with one argument*]

If `cn` is the internal character code of a upper-case character, then `lowercase` returns the code of the corresponding lower-case character. If not, the function returns `cn`, unchanged.

```
(lowercase #/3)  ⟹   51 that is, #/3
(lowercase #/A)  ⟹   97 that is, #/a
(lowercase #/z)  ⟹   122 that is, #/z
```

(asciip cn) [*function with one argument*]

Tests whether `cn` is in fact an internal character code: that is, a small integer in the range `0-255` inclusive. If so, it returns the code itself. If not, it returns `()`.

```
(asciip 0)    ⟹   0
(asciip 127)  ⟹   127
(asciip -1)   ⟹   ()
(asciip 128)  ⟹   128
(asciip 256)  ⟹   ()
```

(digitp cn) [*function with one argument*]

Tests whether `cn` is the internal character code of a decimal numeral. If so, it returns the code. If not, it returns `()`.

```
(digitp #/0)  ⟹   48 that is, #/0
(digitp #/9)  ⟹   57 that is, #/9
(digitp #//)  ⟹   ()
(digitp #/:)  ⟹   ()
```

(letterp cn) [*function with one argument*]

Tests whether `cn` is the internal character code of a letter. If so, it returns the code. If not, it returns `()`.

```
(letterp #/a)  ⟹   97 that is, #/a
(letterp #/z)  ⟹   122 that is, #/z
(letterp #/A)  ⟹   65 that is, #/A
(letterp #/Z)  ⟹   90 that is, #/Z
(letterp #sp)  ⟹   ()
(letterp #/.)  ⟹   ()
(letterp #/])  ⟹   ()
```

3.14 Functions on vectors

The final type of LE-LISP to be presented is the vector of S-expressions.

A vector is a collection of LISP objects accessible by their index, or object number. Vector indexes begin at zero.

Each vector has its own symbolic type. By default, this type is `vector`, but it can be changed by means of the `typevector` function.

By default, this type is `vector`, but it can be changed by using the `typevector` function.

The external representation of a vector of any other type is

`#:type:#[`s_1 s_2 ... s_n`]`

In the following descriptions of functions, the arguments designated by `vect` must be of the vector type. Otherwise, the `errvec` error is raised, and the following default error screen will be displayed:

```
** <fn> : not a vector : <v>
```

Here, `v` is the faulty object, and `fn` is the name of the function that raised the error.

Out-of-bounds access attempts raise the `erroob` error, which has the following default screen display:

```
** <fn> : out of bounds : <v>
```

Here, `v` is the faulty index, and `fn` is the name of the function that raised the error.

The vector test predicate, `vectorp`, is described in the section on basic predicates.

(makevector n s)

[*function with two arguments*]

Creates a new vector, of type `vector`, comprising `n` elements. Each of these elements is initialized with the value `s`. If `n` is not a positive integer, the `errnia` and `erroob` errors are raised. `makevector` returns the newly-created vector as its value. This function cannot be described satisfactorily in LISP.

```
(makevector 5 ())       ==>   #[() () () () ()]
(makevector 3 '(a b))   ==>   #[(a b) (a b) (a b)]
(makevector 0 'a)       ==>   #[]
(makevector -1 ())      ==>      ** makevector : argument out of bounds : -1
(makevector t ())       ==>      ** makevector : not a fixnum : t
```

(vector s_1 ... s_n)

[*function with a variable number of arguments*]

Creates a new vector, of type `vector`, comprising n elements, where n is the number of arguments provided in the function call. The elements are initialized with the values $s_1 \ldots s_n$. `vector` returns this newly-created vector as its value.

In LE-LISP, `vector` could be defined in the following manner:

```
(defun vector lst
    (let ((vect (makevector (length lst) ())))
```

```
(for (i 0 1 (1- (vlength vect)))
     (vset vect i (nextl lst)))
vect))
```

```
(vector 0 1 2 3 4)                ⟹   #[0 1 2 3 4]
(vector)                          ⟹   #[]
(apply 'vector '(a b c))          ⟹   #[a b c]
(vector 1 #[1 2] "foo" 'a '(b c)) ⟹   #[1 #[1 2] "foo" a (b c)]
```

(vlength vect) *[function with one argument]*

Returns the number of elements in the vector `vect`. This function cannot be satisfactorily described in LISP. The type test on the argument, which is performed by the interpreter, is not carried out when code is compiled in 'open' mode. For reasons of efficiency, `vlength` compiles into a single LLM3 instruction.

```
(vlength #[])       ⟹   0
(vlength #[1 2 3])  ⟹   3
(vlength t)         ⟹      ** vlength : not a vector : t
```

(vref vect n) *[function with two arguments]*

Returns the n^{th} element of the vector `vect`. If `n` goes beyond the boundary of `vect`, the `erroob` error is raised. This boundary check is not carried out, however, once the function has been compiled in 'open' mode. For reasons of efficiency, `vref` compiles into a single LLM3 instruction. This function cannot be satisfactorily described in LISP.

```
(setq x #[a b c d e f])  ⟹   #[a b c d e f]
(vref x 0)               ⟹   a
(vref x 5)               ⟹   f
(vref t 1)               ⟹      ** vref : not a vector : t
(vref x -2)              ⟹      ** vref : argument out of bounds : -2
(vref x 6)               ⟹      ** vref : argument out of bounds : 6
```

(vset vect n e) *[function with three arguments]*

Sets the value of the n^{th} element of the vector `vect` to the object `e`. If `n` goes beyond the boundary of `vect`, the `erroob` error is raised. This boundary check will not be performed, however, when the function has been compiled in 'open' mode. For reasons of efficiency, `vref` compiles into a single LLM3 instruction . `vset` returns the new value of the element: that is, `e`. This function cannot be described in LE-LISP in a simple manner.

```
(setq x #[a b c d e f])  ⟹   #[a b c d e f]
(vset x 1 #[x y])        ⟹   #[x y]
(vset x 3 '(h i))        ⟹   (h i)
x                        ⟹   #[a #[x y] c (h i) e f]
(vset x -3 #[1])         ⟹      ** vset : argument out of bounds : -3
```

```
(vset x 6 #[1])          ==>      ** vset : argument out of bounds : 6
```

(typevector vect symb)

[function with one or two arguments]

The second argument, if it is supplied, becomes the new type of the vector **vect**. In general, this **symb** type is a symbol. But it could be a list of symbols, allowing you to carry out multiple inheritance. For further details, look up the topic of object-oriented programming. The **typevector** function returns the new type of **vect**.

The **type-of** function, applied to a vector, returns the value of **typevector** applied to the vector.

This function cannot be satisfactorily described in LISP.

```
(setq v #[a b c])      ==>   #[a b c]
(typevector v)         ==>   vector
(type-of v)            ==>   vector
(typevector v 'foo)    ==>   foo
(typevector v)         ==>   foo
v                      ==>   #:foo:#[a b c]
(type-of v)            ==>   foo
(typevector t)         ==>      ** typevector : not a vector : t
(typevector v "bar")   ==>      ** typevector : not a symbol : bar
```

(eqvector vect1 vect2)

[function with two arguments]

Tests whether the two arguments—both of which are vectors—are equal. That is, it tests whether they both have the same type and the same number of elements, and whether their elements are equal according to the **equal** predicate.

In LE-LISP, **eqvector** could be defined in the following manner:

```
(defun eqvector (vect1 vect2)
    (cond ((not (vectorp vect1))
           (error 'eqvector 'errvec vect1))
          ((not (vectorp vect2))
           (error 'eqvector 'errvec vect2))
          ((and (= (vlength vect1) (vlength vect2))
                (eq (typevector vect1) (typevector vect2)))
           (tag no
                (for (i 0 1 (1- (vlength vect1)))
                     (unless (equal (vref vect1 i)
                                    (vref vect2 i))
                             (exit no ())))
                vect1))
          (t ())))
```

```
(eqvector #[1 2 3] #[1 2 3])      ==>   #[1 2 3]
(eqvector #[1 2 3] #[1 2])        ==>   ()
(eqvector #:foo:#[1 2] #[1 2])    ==>   ()
```

```
(eqvector #:foo:#[1] #:foo:#[1])  ⟹  #:foo:#[1]
```

(bltvector vect1 n1 vect2 n2 n3) *[function with five arguments]*

Copies `n3` elements of the vector `vect2`, beginning at position `n2`, into the vector `vect1`, beginning at position `n1`. Both index values start at zero. The vector `vect1` is never extended. `bltvector` returns the modified `vect1`.

In LE-LISP, `bltvector` could be defined in the following manner:

```
(defun bltvector (vect1 n1 vect2 n2 n3)
    ; Does not treat the shared vector nor limit cases.
    (repeat n3
            (vset vect1 n1 (vref vect2 n2))
            (incr n1)
            (incr n2))
    vect1)
```

```
(bltvector #[f o o b a r] 1 #[x y z] 2 1)        ⟹  #[f z o b a r]
(bltvector #[f o o b a r] 1 #[t o t o] 0 6)       ⟹  #[f t o t o r]
(bltvector #[f o o b a r] 3 #[t o t o t a] 0 8)   ⟹  #[f o o t o t]
(bltvector (setq v #[a b c d]) 2 v 0 2)          ⟹  #[c d c d]
v                                                ⟹  #[c d c d]
```

(fillvector vect n1 e n2) *[function with three or four arguments]*

Fills the vector `vect`, beginning at position `n1`, counting from zero, with `n2` expressions `e` of whatever description. If `n2` is not supplied, or if the sum of `n1` and `n2` is larger than the length of the vector, `fillvector` fills up to the end of `vect`, and does not extend it. `fillvector` returns the modified `vect` as its value.

In LE-LISP, `fillvector` could be defined in the following manner:

```
(defun fillvector (vect n1 s . n2)
    (repeat (if (consp n2)
                (min (car n2) (- (vlength vect) n1)
                (- (vlength vect) n1))
            (vset vect n1 s) (incr n1))))
```

```
(fillvector #[f o o b a r] 1 'x 2)   ⟹  #[f x x b a r]
(fillvector #[f o o b a r] 0 'y 3)   ⟹  #[y y y b a r]
(fillvector #[f o o b a r] 2 'x)     ⟹  #[f o x x x x]
(fillvector #[f o o b a r] 2 'x 10)  ⟹  #[f o x x x x]
```

(exchvector vect1 vect2) *[function with two arguments]*

This rather esoteric function provides a way of physically exchanging the values and the types of the vectors `vect1` and `vect2`. It returns the modified value of `vect1`: that is, the previous value of `vect2`.

```
(setq v #[a b c] w #[d e])  ⟹   #[d e]
(typevector v 'foo)         ⟹   foo
v                           ⟹   #:foo:#[a b c]
(setq y v z w)              ⟹   #[d e]
(exchvector v w)            ⟹   #[d e]
v                           ⟹   #[d e]
w                           ⟹   #:foo:#[a b c]
y                           ⟹   #[d e]
z                           ⟹   #:foo:#[a b c]
```

The `mapvector` function is described in the section on application functions.

3.15 Functions on arrays

`array` [*feature*]

This feature indicates whether the functions on array are loaded into memory.

It is easy to implement multi-dimensional arrays with the functions of the preceding section.

`(makearray` a_1 ... a_n `s)` [*function with a variable number of arguments*]

Makes an array of $a_1 \times a_1 \times \ldots \times a_{n-1} \times a_n$ elements, all initialized to the value `s`.

In LE-LISP, `makearray` could be defined in the following manner:

```
(defun makearray (arg1 . args)
    (if (consp args)
        (let ((result (makevector arg1 ())))
             (for (i 0 1 (1- arg1))
                  (vset result i (apply 'makearray args)))
             result)
        arg1))
```

`(aref array` i_1 ... i_n`)` [*function with a variable number of arguments*]

Returns the value of the array `array` that has the index $i_1 \ldots i_n$. Each index is relative to zero.

In LE-LISP, `aref` could be defined in the following manner:

```
(defmacro aref (inst . args)
     (cond ((null args) inst)
           ((atom args) (error 'aref 'errwna args))
           (t `(vref (aref ,inst ,@(nreverse (cdr (reverse args))))
                     ,(car (last args))))))
```

(aset array i_1 ... i_n e) [*function with a variable number of arguments*]

Provides a means of setting the value of the element of the array `array` to the value `e`. The index values are relative to zero. `aset` returns `e` as its value.

In LE-LISP, `aset` could be defined in the following manner:

```
(defmacro aset (inst . args)
     (cond ((atom args) (error 'aref 'errwna args))
           ((consp (cdr args))
              `(vset (aref ,inst
                           ,.(nreverse (cddr (reverse args))))
                     ,(cadr (reverse args))
                     ,(car (last args))))
           (t inst)))
```

```
(setq x (makearray 3 4 0))  ==>  #[#[0 0 0 0] #[0 0 0 0] #[0 0 0 0]]
(aset x 1 2 -1)             ==>  -1
(aref x 1 2)                ==>  -1
x                           ==>  #[#[0 0 0 0] #[0 0 -1 0] #[0 0 0 0]]
```

3.16 Hash tables

hash [*feature*]

This feature indicates whether the functions on hash tables are loaded into memory.

LE-LISP version 15.2 incorporates a new data structure, called *hash tables*, for processing pairs composed of a *key* and a *value*. These pairs are managed in a more efficient manner than in the case of association lists. Association lists (cf. section 3.10.5) use a higher number of keys. With the hash table implementation, search time is not a function of the number of elements in the table, but rather of a constant that depends only on the state of the hash table. These hash tables are adaptive. They automatically minimize the search time constant as a function of the number of currently active associations, and at the same time optimize the use of the memory space dedicated to storing key/value associations.

There are two kinds of hash tables, according to the function that is used to perform key lookups:

- Tables using the `eq` predicate.
- Hash tables using the `equal` predicate.

The value of a hash table-type object is the hash table itself, so there is no need to 'quote' it.

The external representation of a hash table depends on the `#:system:print-for-read` flag. If this flag is *false*—equal to `()`—the printed image of a table is

`#entriesHelements`

where `entries` is the number of table entries, and `elements` is the number of associations actually stored in the table. If the flag is *true*—not equal to `()`—the printed image of a hash table is

`#entriesH(type (key`$_1$ `value`$_1$`) ... (key`$_n$ `value`$_n$`))`

This image—which explicitly represents every association in the table and can in some cases be very long—allows for the saving of a hash table object so that it can be subsequently read.

In the description of each function presented in this section, the `ht` argument must be of the hash-table type, or else the `errnht` error will be raised. The default screen display of this latter is

```
** <fn> : not an hash table : <s>
```

where `s` is the faulty argument and `fn` is the name of the function that raised the error.

Similarly, if the type of the hash table is incorrect, the `errbht` error is raised. Its default screen display is

```
** <fn> : bad type for an hash table : <s>
```

where `s` is the faulty object and `fn` is the name of the function that raised the error.

3.16.1 Hash table creation functions

There are currently two kinds of hash tables, which differ according to how keys are retrieved. One kind uses the built-in `eq` predicate. The other kind uses the `equal` predicate.

(make-hash-table-eq) [*function with no arguments*]

Returns a new hash table that tests keys using the `eq` predicate.

(make-hash-table-equal) [*function with no arguments*]

Returns a new hash table that tests keys using the `equal` predicate.

(hash-table-p `obj`**)** [*function with one argument*]

This predicate returns `obj` if it is a hash-table object of any type. Otherwise, it returns `()`.

3.16.2 Hash table access functions

(gethash `key ht default`**)** [*function with two or three arguments*]

Returns the value associated with the `key` in the hash table `ht`. If no key in `ht` matches `key`, `gethash` returns the value `default`, if it is provided, or `()` if not.

(puthash `key ht value`**)** [*function with three arguments*]

Adds the key/value pair composed of `key` and `value` to the hash table `ht`. If the `key` existed already in `ht`, its value is changed to `value`. `puthash` always returns `value`.

(remhash key ht) [*function with two arguments*]

Removes the `key` and its associated value from the hash table `ht`. The function returns `t` if the key existed in `ht`, and `()` if not.

(maphash fnt ht) [*function with two arguments*]

Applies the function `fnt` to all the pairs in the hash table `ht`. This function, `fnt`, expects to receive two arguments. The first is a hash-pair key, and the second is its associated value.

(clrhash ht) [*function with one argument*]

Removes the entire contents of the hash table `ht`.

(hash-table-count ht) [*function with one argument*]

Returns the number of key/value pairs stored in the table `ht`.

```
? (setq ht (make-hash-table-eq))
= #17H<0>   ? (puthash 'k1 ht 'v1)
= v1   ? (gethash 'k1 ht)
= v1   ? (gethash 'k2 ht)
= ()   ? (gethash 'k2 ht 'ko)
= ko   ? ht
= #17H<1>
```

3.17 Mathematical sets

sets [*feature*]

This feature indicates whether the functions on sets are loaded into memory.

LE-LISP includes functions that let you handle lists as if they were **sets**. They enable you to perform operations of the following kinds:

- Add an element to a set.
- Obtain the union of two sets.
- Obtain the intersection of two sets.
- Obtain the difference between two sets.
- Obtain the symmetric difference between two sets.
- Compare two sets to see if they are identical, or if one is a sub-set of the other.

In a list argument that is meant to be handled as a set, the order of the elements is of no significance. When such a list happens to be returned as a function value, the order of its elements is, in fact, unpredictable.

All the functions described below use the `eq` predicate, but they allow you to specify—by means of an optional argument—another kind of equality test. If speed is important for you, it is preferable not to use this facility. Using `eq`, execution of tests takes place directly, without the need for a `funcall`. The implementation also calls upon rapid functions such as `memq` and `delq`. If you are forced to use the optional argument, it is better to choose the `equal` predicate rather than, say, `eqstring`, since the implementation optimizes execution time by avoiding calls to `funcall` and by using the appropriate functions: `member` and `delete`.

Any function whose name starts with `n` physically modifies its arguments. While its effects are destructive for the arguments, such a function has the advantage of never calling upon `cons`.

3.17.1 Operations on sets

(adjoin `item list eq-func`**)** [*function with two or three arguments*]

If `item` is not already a member of the list called `list`, it is added to this list. If this addition takes place, `item` appears at the head of the new list. The optional argument `eq-func` allows you to choose a different equality operator to the default primitive, which is `eq`.

```
(adjoin 'a '(b q f a g))              ⟹   (b q f a g)
(adjoin 'c '(b q f a g))              ⟹   (c b q f a g)
(adjoin "abc" '("dw" "abc"))          ⟹   ("abc" "dw" "abc")
(adjoin "abc" '("dw" "abc") 'equal)   ⟹   ("dw" "abc")
```

(union `list1 list2 eq-func`**)** [*function with two or three arguments*]

Takes a pair of lists and produces a new list in which each element belongs to one or other, or maybe both, of the initial lists. The optional argument `eq-func` allows you to choose a different equality operator to the default primitive, which is `eq`.

```
(union '(a b c) '(f b h a q))   ⟹   (c f b h a q)
```

(nunion `list1 list2 eq-func`**)** [*function with two or three arguments*]

This is the destructive version of the previous function. It returns a new list that is built directly from the cells of `list1` and `list2`, which can no longer be used as before. The optional argument `eq-func` allows you to choose a different equality operator to the default primitive, which is `eq`.

(intersection `list1 list2 eq-func`**)** [*function with two or three arguments*]

Takes a pair of lists and returns a new list containing all the elements that appear in both of the initial lists. The optional argument `eq-func` allows you to choose a different equality operator to the default primitive, which is `eq`.

```
(intersection '(a b c) '(f b h a q))  ⟹  (b a)
```

(nintersection list1 list2 eq-func) [*function with two or three arguments*]

This is the destructive version of the previous function. It returns a new list that is built by using the cells of `list1`. On the other hand, `list2` is not altered by this function. The optional argument `eq-func` allows you to choose a different equality operator to the default primitive, which is `eq`.

(set-difference list1 list2 eq-func) [*function with two or three arguments*]

Returns a list comprising all the elements of `list1` that do not appear in `list2`. The optional argument `eq-func` allows you to choose a different equality operator to the default primitive, which is `eq`.

```
(set-difference '(a b c) '(f h a q))  ⟹  (c b)
```

(nset-difference list1 list2 eq-func) [*function with two or three arguments*]

This is the destructive version of the previous function. It returns a new list that is built by using the cells of `list1`. On the other hand, `list2` is not altered by this function. The optional argument `eq-func` allows you to choose a different equality operator to the default primitive, which is `eq`.

(set-exclusive-or list1 list2 eq-func) [*function with two or three arguments*]

Returns the symmetric difference between `list1` and `list2`. This is a list containing all elements that occur in only one of the initial lists. The optional argument `eq-func` allows you to choose a different equality operator to the default primitive, which is `eq`.

```
(set-exclusive-or '(a b c) '(f h a q))          ⟹  (q h f b c)
(set-exclusive-or '(a b c) '(f g h))            ⟹  (h g f a b c)
(set-exclusive-or '(3 s 2 3.5) '(3.50 8 b s))   ⟹  (b 8 3 2)
```

(nset-exclusive-or list1 list2 eq-func) [*function with two or three arguments*]

This is the destructive version of the previous function. It returns a new list that is built directly from the cells of `list1` and `list2`, which can no longer be used as before. The optional argument `eq-func` allows you to choose a different equality operator to the default primitive, which is `eq`.

(power-set set) [*function with one argument*]

Creates the power set of `set`.

```
(power-set ())        ⟹  ()
(power-set '(a b c))  ⟹  (() (c) (b) (b c) (a) (a c) (a b) (a b c))
```

(cartesian-product set1 set2) [*function with two arguments*]

Creates the Cartesian product of `set1` and `set2`. The elements of the resulting set are pointed pairs built from the elements of `set1` and `set2`.

```
(cartesian-product '(a b) ())       ⟹   ()
(cartesian-product '(a b) '(c d))   ⟹   ((a . c) (a . d) (b . c) (b . d))
```

3.17.2 Comparisons on sets

(subsetp `list1 list2 eq-func`**)** *[function with two or three arguments]*

Returns `t` if `list1` is a sub-set of `list2`. For this to be the case, every element of `list1` must also appear in `list2`. If this is not the case, the function returns `()`. The optional argument `eq-func` allows you to choose a different equality operator to the default primitive, which is `eq`.

```
(subsetp '(a b) '(b c f))                  ⟹   ()
(subsetp '(a b) '(b f a q))                ⟹   t
(subsetp '("asd") '("xhg" "asd"))          ⟹   ()
(subsetp '("asd") '("xhg" "asd") 'equal)   ⟹   t
```

(set-equal `list1 list2 eq-func`**)** *[function with two or three arguments]*

Returns `t` if `list1` and `list2` are identical lists. The optional argument `eq-func` allows you to choose a different equality operator to the default primitive, which is `eq`.

In LE-LISP, `set-equal` could be defined in the following manner:

```
(defun set-equal (list1 list2)
   (and (subsetp list1 list2) (subsetp list2 list1)))
```

```
(set-equal '(a b c) '(b a c))                          ⟹   t
(set-equal '((a b) (c d)) '((c d) (a b)))              ⟹   ()
(set-equal '((a b) (c d)) '((c d) (a b)) 'equal)       ⟹   t
(set-equal '((a b) (c d)) '((a b) (d c)) 'equal)       ⟹   ()
(set-equal '((a b) (c d)) '((a b) (d c)) 'set-equal)   ⟹   t
```

3.17.3 Transitive closure

Consider `f` to be an arbitrary single-argument function that returns, as its value, a set. The unique argument of `f` is a set element. The *transitive closure* of a set `E` by the `f` function is defined to be a minimal sub-set of `E`, called `F`, such that, for each element `x` in `F`, `f(x)` is a sub-set of `F`.

Transitive closure can be a useful operation when you need to search, say, for all the descendents of a summit in a tree or a graph.

(transitive-closure `fn list eq-func`**)** *[function with two or three arguments]*

Returns a list that represents the transitive closure of `list` by the function `fn`. The optional argument `eq-func` allows you to choose a different equality operator to the default primitive, which is `eq`.

```
(defun test1 (elt)
  (cassq elt '((b a f c) (d e b) (c q))))

(transitive-closure 'test1 '(a))    ⟹   (a)
(transitive-closure 'test1 '(b))    ⟹   (a f q c b)
(transitive-closure 'test1 '(d g))  ⟹   (g e a f q c b d)
```

The following example shows a possible use for `transitive-closure` :

```
(defun submodules (mod)
     (getdefmodule (readdefmodule mod) 'import))

(submodules 'complice)                          ⟹   (files module loader)
(submodules 'files)                             ⟹   (path)
(submodules 'module)                            ⟹   (files)
(submodules 'loader)                            ⟹   (files)
(transitive-closure 'submodules '(complice))    ⟹   (path files module loader complice)
```

The next example demonstrates the way in which a bad choice for the equality-test predicate can cause a function to loop:

```
(defun test2 (s)
   (list (substring s 0 (sub1 (slen s)))))

(transitive-closure 'test2 '("hip" "hop") 'eqstring)  ⟹   ("ho" "hop" "" "h" "hi" "hip")

(transitive-closure 'test2 '("hip" "hop"))  ⟹
                                    ***** Fatal error : no room for strings.
```

The `transitive-closure` function could be implemented—but not very successfully—in the following manner:

```
(defun transitive-closure (fn list)
   (let ((old-list list)
                (new-list list))
      (while list
        (setq new-list (union new-list (funcall fn (nextl list)))))
      (if (set-equal old-list new-list)
                 old-list
                 (transitive-closure fn new-list))))
```

Table of contents

3 Predefined functions **3-1**

3.1 Evaluation functions ... 3-2

3.2 Application functions ... 3-6

3.2.1 Simple application functions ... 3-7

3.2.2 Application functions of the `map` type ... 3-7

3.2.3 Other application functions ... 3-10

3.3 Environment manipulation functions ... 3-13

3.4 Function-definition functions ... 3-17

3.4.1 Static function definitions ... 3-17

3.4.2 Advanced use of `macro` functions ... 3-20

3.4.3 Definition of closures ... 3-22

3.4.4 Definition of dynamic functions ... 3-23

3.4.5 Generalized assignment ... 3-23

3.5 Variable functions ... 3-24

3.6 Basic control functions ... 3-26

3.7 Lexical control functions ... 3-33

3.7.1 Primitive lexical control forms ... 3-33

3.7.2 Iteration functions of the `prog` kind ... 3-34

3.7.3 Iteration functions of the `do` kind ... 3-35

3.8 Dynamic, non-local control functions ... 3-37

3.9 Basic predicates ... 3-42

3.10 Functions on lists ... 3-49

3.10.1 Search functions on lists ... 3-49

3.10.2 List creation functions ... 3-52

3.10.3 Functions on cells of labelled lists ... 3-58

3.10.4 Physical modification functions 3-59
3.10.5 Functions on A-lists 3-65
3.10.6 Sorting functions 3-68
3.11 Functions on symbols 3-69
3.11.1 Functions that access symbol values 3-69
3.11.2 Functions that modify symbol values 3-70
3.11.3 Functions on P-lists 3-75
3.11.4 Access to function definitions 3-78
3.11.5 Access to symbol special fields 3-81
3.11.6 Symbol creation functions 3-83
3.11.7 Symbol management functions 3-84
3.12 Functions on character strings 3-86
3.12.1 Basic manipulation functions 3-87
3.12.2 Character string conversion 3-89
3.12.3 Comparison of character strings 3-91
3.12.4 Character-string creation functions 3-91
3.12.5 Character-string access functions 3-93
3.12.6 Functions that physically modify strings 3-94
3.12.7 Search functions on character strings 3-95
3.13 Functions on characters 3-97
3.14 Functions on vectors 3-98
3.15 Functions on arrays 3-103
3.16 Hash tables 3-104
3.16.1 Hash table creation functions 3-105
3.16.2 Hash table access functions 3-105
3.17 Mathematical sets 3-106
3.17.1 Operations on sets 3-107
3.17.2 Comparisons on sets 3-109
3.17.3 Transitive closure 3-109

Function Index

(eval s env) [*function with one or two arguments*] 3-2
(evlis l) [*function with one argument*] 3-2
(eprogn l) [*function with one argument*] 3-3
(prog1 s_1 ... s_n) [*special form*] 3-3
(prog2 s_1 s_2 ... s_n) [*special form*] 3-4
(progn s_1 ... s_n) [*special form*] 3-4
(quote s) [*special form*] 3-5
(function fn) [*special form*] 3-5
(arg n) [*function with an optional argument*] 3-5
(identity s) [*function with one argument*] 3-5
(comment e_1 ... e_n) [*special form*] 3-6
(lambda l s_1 ... s_n) [*special form*] 3-6
(flambda l s_1 ... s_n) [*special form*] 3-6
(mlambda l s_1 ... s_n) [*special form*] 3-6
(apply fn s_1 ... s_n l) [*function with a variable number of arguments*] 3-7
(funcall fn s_1 ... s_n) [*function with a variable number of arguments*] 3-7
(allcar l) [*function with one argument*] 3-8
(mapl fn l_1 ... l_n) [*function with a variable number of arguments*] 3-8
(map fn l_1 ... l_n) [*function with a variable number of arguments*] 3-8
(mapc fn l_1 ... l_n) [*function with a variable number of arguments*] 3-8
(maplist fn l_1 ... l_n) [*function with a variable number of arguments*] 3-9
(mapcar fn l_1 ... l_n) [*function with a variable number of arguments*] 3-9
(mapcon fn l_1 ... l_n) [*function with a variable number of arguments*] 3-9
(mapcan fn l_1 ... l_n) [*function with a variable number of arguments*] 3-10
(every fn l_1 ... l_n) [*function with a variable number of arguments*] 3-11
(any fn l_1 ... l_n) [*function with a variable number of arguments*] 3-11

(mapvector fn vect) [*function with two arguments*] 3-11
(mapoblist fn) [*function with one argument*] 3-12
(mapcoblist fn) [*function with one argument*] 3-12
(maploblist fn) [*function with one argument*] 3-12
(let lv s_1 ... s_n) [*special form*] 3-13
(letv lvar lval s_1 ... s_n) [*special form*] 3-14
(letvq lvar lval s_1 ... s_n) [*special form*] 3-15
(lets lv s_1 ... s_n) [*special form*] 3-15
(slet lv s_1 ... s_n) [*special form*] 3-15
(let* lv s_1 ... s_n) [*special form*] 3-15
(letn symb lv s_1 ... s_n) [*special form*] 3-16
#:system:previous-def-flag [*variable*] 3-17
#:system:previous-def [*property*] 3-17
#:system:redef-flag [*variable*] 3-17
#:system:loaded-from-file [*variable*] 3-18
#:system:loaded-from-file [*property*] 3-18
(defun symb lvar s_1 ... s_n) [*special form*] 3-18
(de symb lvar s_1 ... s_n) [*special form*] 3-18
(df symb lvar s_1 ... s_n) [*special form*] 3-18
(dm symb lvar s_1 ... s_n) [*special form*] 3-19
(defmacro symb lvar s_1 ... s_n) [*special form*] 3-19
(dmd symb lvar s_1 ... s_n) [*special form*] 3-19
(ds symb type adr) [*special form*] 3-20
(macroexpand1 s) [*function with one argument*] 3-20
(macroexpand s) [*function with one argument*] 3-21
(closure lvar fn) [*function with two arguments*] 3-22
(flet l s_1 ... s_n) [*special form*] 3-23
(setf loc_1 val_1 ... loc_n val_n) [*macro*] 3-24
(defsetf access lparams store . body) [*macro*] 3-24
(with l s_1 ... s_n) [*special form*] 3-25
(if s_1 s_2 s_3 ... s_n) [*special form*] 3-26
(ifn s_1 s_2 s_3 ... s_n) [*special form*] 3-27
(when s_1 s_2 ... s_n) [*special form*] 3-27
(unless s_1 s_2 ... s_n) [*special form*] 3-27

(or s_1 ... s_n) [*special form*] 3-28
(and s_1 ... s_n) [*special form*] 3-28
(cond l_1 ... l_n) [*special form*] 3-29
(selectq s l_1 ... l_n) [*special form*] 3-29
(while s s_1 ... s_n) [*special form*] 3-31
(until s s_1 ... s_n) [*special form*] 3-31
(repeat m s_1 ... s_n) [*special form*] 3-32
(for (var in ic ntl e_1 ... e_m) s_1 ... s_n) [*special form*] 3-32
(block symb e_1 ... e_n) [*special form*] 3-33
(return-from symb e) [*special form*] 3-33
(return e) [*special form*] 3-33
(tagbody ec_1 ... ec_n) [*special form*] 3-33
(go symb) [*special form*] 3-34
(prog l ec_1 ... ec_n) [*macro*] 3-34
(prog* l ec_1 ... ec_n) [*macro*] 3-35
(do lv lr ec_1 ... ec_n) [*macro*] 3-35
(do* lv lr ec_1 ... ec_n) [*macro*] 3-37
(tag symb s_1 ... s_n) [*special form*] 3-38
(evtag s s_1 ... s_n) [*special form*] 3-38
(exit symb s_1 ... s_n) [*special form*] 3-38
(evexit s s_1 ... s_n) [*special form*] 3-38
(unexit symb s_1 ... s_n) [*special form*] 3-39
(untilexit symb e_1 ... e_n) [*special form*] 3-39
(lock fn s_1 ... s_n) [*special form*] 3-39
(protect s_1 s_2 ... s_n) [*special form*] 3-39
(unwind n s_1 ... s_n) [*special form*] 3-40
(catch-all-but l s_1 ... s_n) [*special form*] 3-41
(backtrack symb l e_1 ... e_n) [*special form*] 3-41
(false e_1 ... e_n) [*function with a variable number of arguments*] 3-42
(true e_1 ... e_n) [*function with a variable number of arguments*] 3-42
(null s) [*function with one argument*] 3-42
(not s) [*function with one argument*] 3-43
(atom s) [*function with one argument*] 3-43
(atomp s) [*function with one argument*] 3-43

`(constantp s)` [*function with one argument*] 3-43
`(symbolp s)` [*function with one argument*] 3-44
`(variablep s)` [*function with one argument*] 3-44
`(numberp s)` [*function with one argument*] 3-44
`(vectorp s)` [*function with one argument*] 3-45
`(stringp s)` [*function with one argument*] 3-45
`(consp s)` [*function with one argument*] 3-45
`(listp s)` [*function with one argument*] 3-45
`(nlistp s)` [*function with one argument*] 3-46
`(eq s1 s2)` [*function with two arguments*] 3-46
`(neq s1 s2)` [*function with two arguments*] 3-47
`(equal s1 s2)` [*function with two arguments*] 3-47
`(nequal s1 s2)` [*function with two arguments*] 3-48
`(car l)` [*function with one argument*] 3-49
`(cdr l)` [*function with one argument*] 3-49
`(c----r l)` [*functions of one argument*] 3-49
`(memq symb l)` [*function with two arguments*] 3-49
`(member s l)` [*function with two arguments*] 3-50
`(tailp s l)` [*function with two arguments*] 3-50
`(nthcdr n l)` [*function with two arguments*] 3-50
`(nth n l)` [*function with two arguments*] 3-51
`(last s)` [*function with one argument*] 3-51
`(length s)` [*function with one argument*] 3-51
`(cons s1 s2)` [*function with two arguments*] 3-52
`(xcons s1 s2)` [*function with two arguments*] 3-52
`(ncons s)` [*function with one argument*] 3-52
`(mcons` s_1 `...` s_n`)` [*function with a variable number of arguments*] 3-52
`(list` s_1 `...` s_n`)` [*function with a variable number of arguments*] 3-53
`(kwote s)` [*function with one argument*] 3-54
`(makelist n s)` [*function with two arguments*] 3-54
`(append` l_1 `...` l_n`)` [*function with a variable number of arguments*] 3-54
`(append1 l s)` [*function with two arguments*] 3-55
`(reverse s)` [*function with one argument*] 3-55
`(copylist l)` [*function with one argument*] 3-55

`(copy s)` [*function with one argument*] 3-56
`(firstn n l)` [*function with two arguments*] 3-57
`(lastn n l)` [*function with two arguments*] 3-57
`(subst s1 s2 s)` [*function with three arguments*] 3-57
`(remq symb l)` [*function with two arguments*] 3-58
`(remove s l)` [*function with two arguments*] 3-58
`(tconsp s)` [*function with one argument*] 3-59
`(tconsmk s)` [*function with one argument*] 3-59
`(tconscl s)` [*function with one argument*] 3-59
`(tcons s1 s2)` [*function with two arguments*] 3-59
`(rplaca l s)` [*function with two arguments*] 3-60
`(rplacd l s)` [*function with two arguments*] 3-60
`(rplac l s1 s2)` [*function with three arguments*] 3-60
`(displace l ln)` [*function with two arguments*] 3-60
`(placdl l s)` [*function with two arguments*] 3-61
`(evlis l)` [*function with one argument*] 3-61
`(nconc` l_1 ... l_n`)` [*function with a variable number of arguments*] 3-61
`(nconc1 l s)` [*function with two arguments*] 3-62
`(cirlist` e_1 ... e_n`)` [*function with a variable number of arguments*] 3-62
`(nreverse l)` [*function with one argument*] 3-62
`(nreconc l s)` [*function with two arguments*] 3-63
`(nsubst s1 s2 l)` [*function with three arguments*] 3-63
`(delq symb l)` [*function with two arguments*] 3-64
`(delete s l)` [*function with two arguments*] 3-64
`(acons s1 s2 al)` [*function with three arguments*] 3-65
`(pairlis l1 l2 al)` [*function with three arguments*] 3-65
`(assq symb al)` [*function with two arguments*] 3-65
`(cassq symb al)` [*function with two arguments*] 3-66
`(rassq symb al)` [*function with two arguments*] 3-66
`(assoc s al)` [*function with two arguments*] 3-66
`(cassoc s al)` [*function with two arguments*] 3-67
`(rassoc s al)` [*function with two arguments*] 3-67
`(sublis al s)` [*function with two arguments*] 3-67
`sort` [*feature*] 3-68

`(sort fn l)` [*function with two arguments*] 3-68
`(sortl l)` [*function with one argument*] 3-68
`(sortp l)` [*function with one argument*] 3-69
`(sortn l)` [*function with one argument*] 3-69
`(boundp symb)` [*function with one argument*] 3-69
`(symeval symb)` [*function with one argument*] 3-70
`(defvar symb e)` [*special form*] 3-70
`#:system:loaded-from-file` [*variable*] 3-70
`(makunbound symb)` [*function with one argument*] 3-71
`(set symb s)` [*function with two arguments*] 3-71
`(setq` $\mathtt{sym}_1$ $\mathtt{s}_1$ ... $\mathtt{sym}_n$ $\mathtt{s}_n$`)` [*special form*] 3-71
`(setqq` $\mathtt{sym}_1$ $\mathtt{e}_1$ ... $\mathtt{sym}_n$ $\mathtt{e}_n$`)` [*special form*] 3-71
`(psetq` $\mathtt{sym}_1$ $\mathtt{s}_1$ ... $\mathtt{sym}_n$ $\mathtt{s}_n$`)` [*special form*] 3-71
`(deset l1 l2)` [*function with two arguments*] 3-72
`(desetq l1 l2)` [*special form*] 3-72
`(nextl sym1 sym2)` [*special form*] 3-73
`(newl symb s)` [*special form*] 3-73
`(newr symb s)` [*special form*] 3-73
`(incr symb n)` [*special form*] 3-74
`(decr symb n)` [*special form*] 3-74
`(plist pl l)` [*function with one or two arguments*] 3-75
`(getprop pl ind)` [*function with two arguments*] 3-75
`(get pl ind)` [*function with two arguments*] 3-75
`(getl pl l)` [*function with two arguments*] 3-76
`(addprop pl pval ind)` [*function with three arguments*] 3-76
`(putprop pl pval ind)` [*function with three arguments*] 3-77
`(defprop pl pval ind)` [*special form*] 3-77
`(remprop pl ind)` [*function with two arguments*] 3-77
`(typefn symb)` [*function with one argument*] 3-78
`(valfn symb)` [*function with one argument*] 3-79
`(setfn symb ftype fval)` [*function with three arguments*] 3-79
`(resetfn symb ftype fval)` [*function with three arguments*] 3-79
`(findfn s)` [*function with one argument*] 3-79
`(remfn symb)` [*function with one argument*] 3-80

(makedef symb ftyp fval) [*function with three arguments*] 3-80
(getdef symb) [*function with one argument*] 3-80
(revert symb) [*function with one argument*] 3-81
(synonym sym1 sym2) [*function with two arguments*] 3-81
(synonymq sym1 sym2) [*special form*] 3-81
(objval symb s) [*function with one or two arguments*] 3-82
(packagecell symb pkgc) [*function with one or two arguments*] 3-82
(getfn1 pkgc symb) [*function with two arguments*] 3-82
(getfn pkgc symb lastpkgc) [*function with two or three arguments*] 3-82
(symbol pkgc strg) [*function with two arguments*] 3-83
(concat str_1 ... str_n) [*function with a variable number of arguments*] 3-83
(gensym) [*function with no arguments*] 3-84
(oblist pkgc symb) [*function with zero, one or two arguments*] 3-84
(lhoblist strg) [*function with one argument*] 3-85
(boblist n) [*function with an optional argument*] 3-85
(remob symb) [*function with one argument*] 3-86
(slen strg) [*function with one argument*] 3-87
(sref strg n) [*function with two arguments*] 3-87
(sset strg n cn) [*function with three arguments*] 3-88
(typestring strg symb) [*function with one or two arguments*] 3-88
(exchstring strg1 strg2) [*function with two arguments*] 3-88
(string s) [*function with one argument*] 3-89
(pname strg) [*function with one argument*] 3-89
(plength strg) [*function with one argument*] 3-89
(slength strg) [*function with one argument*] 3-89
(hash strg) [*function with one argument*] 3-90
(eqstring str1 str2) [*function with two arguments*] 3-91
(alphalessp str1 str1) [*function with two arguments*] 3-91
(catenate str_1 ... str_n) [*function with a variable number of arguments*] 3-91
(makestring n cn) [*function with two arguments*] 3-92
(substring strg n1 n2) [*function with two or three arguments*] 3-92
(duplstring n strg) [*function with two arguments*] 3-92
(chrpos cn strg pos) [*function with two or three arguments*] 3-93
(chrnth n strg) [*function with two arguments*] 3-93

`(chrset n strg cn)` [*function with three arguments*] 3-94
`(bltstring str1 n1 str2 n2 n3)` [*function with four or five arguments*] 3-94
`(fillstring strg n1 cn n2)` [*function with three or four arguments*] 3-95
`(index str1 str2 n)` [*function with two or three arguments*] 3-95
`(substring-equal size strg1 pos1 strg2 pos2)` [*function with five arguments*] 3-96
`(scanstring str1 str2 n)` [*function with two or three arguments*] 3-96
`(spanstring str1 str2 n)` [*function with two or three arguments*] 3-97
`(ascii cn)` [*function with one argument*] 3-97
`(cascii ch)` [*function with one argument*] 3-97
`(uppercase cn)` [*function with one argument*] 3-97
`(lowercase cn)` [*function with one argument*] 3-98
`(asciip cn)` [*function with one argument*] 3-98
`(digitp cn)` [*function with one argument*] 3-98
`(letterp cn)` [*function with one argument*] 3-98
`(makevector n s)` [*function with two arguments*] 3-99
`(vector` s_1 `...` s_n`)` [*function with a variable number of arguments*] 3-99
`(vlength vect)` [*function with one argument*] 3-100
`(vref vect n)` [*function with two arguments*] 3-100
`(vset vect n e)` [*function with three arguments*] 3-100
`(typevector vect symb)` [*function with one or two arguments*] 3-101
`(eqvector vect1 vect2)` [*function with two arguments*] 3-101
`(bltvector vect1 n1 vect2 n2 n3)` [*function with five arguments*] 3-102
`(fillvector vect n1 e n2)` [*function with three or four arguments*] 3-102
`(exchvector vect1 vect2)` [*function with two arguments*] 3-102
`array` [*feature*] 3-103
`(makearray` a_1 `...` a_n `s)` [*function with a variable number of arguments*] 3-103
`(aref array` i_1 `...` i_n`)` [*function with a variable number of arguments*] 3-103
`(aset array` i_1 `...` i_n `e)` [*function with a variable number of arguments*] 3-104
`hash` [*feature*] 3-104
`(make-hash-table-eq)` [*function with no arguments*] 3-105
`(make-hash-table-equal)` [*function with no arguments*] 3-105
`(hash-table-p obj)` [*function with one argument*] 3-105
`(gethash key ht default)` [*function with two or three arguments*] 3-105
`(puthash key ht value)` [*function with three arguments*] 3-105

`(remhash key ht)` [*function with two arguments*] 3-106
`(maphash fnt ht)` [*function with two arguments*] 3-106
`(clrhash ht)` [*function with one argument*] 3-106
`(hash-table-count ht)` [*function with one argument*] 3-106
`sets` [*feature*] 3-106
`(adjoin item list eq-func)` [*function with two or three arguments*] 3-107
`(union list1 list2 eq-func)` [*function with two or three arguments*] 3-107
`(nunion list1 list2 eq-func)` [*function with two or three arguments*] 3-107
`(intersection list1 list2 eq-func)` [*function with two or three arguments*] 3-107
`(nintersection list1 list2 eq-func)` [*function with two or three arguments*] 3-108
`(set-difference list1 list2 eq-func)` [*function with two or three arguments*] 3-108
`(nset-difference list1 list2 eq-func)` [*function with two or three arguments*] 3-108
`(set-exclusive-or list1 list2 eq-func)` [*function with two or three arguments*] 3-108
`(nset-exclusive-or list1 list2 eq-func)` [*function with two or three arguments*] 3-108
`(power-set set)` [*function with one argument*] 3-108
`(cartesian-product set1 set2)` [*function with two arguments*] 3-109
`(subsetp list1 list2 eq-func)` [*function with two or three arguments*] 3-109
`(set-equal list1 list2 eq-func)` [*function with two or three arguments*] 3-109
`(transitive-closure fn list eq-func)` [*function with two or three arguments*] 3-110

Chapter 4

Arithmetic functions

Le-Lisp uses several types of numbers:

- Fixed-precision numbers.
- Integers of 16 bits.
- Floating-point numbers with 31, 32, 48 or 64 bits, depending on the system. In this manual, we often refer to floating-point numbers as *floats*.
- Arbitrary-precision numbers supported by a Le-Lisp library.

Le-Lisp also proposes five types of arithmetic:

- Generic arithmetic.
- Integer arithmetic.
- Extended integer arithmetic.
- Floating-point arithmetic.
- Mixed arithmetic.

Functions described in this chapter only treat 16-bit integers and floating-point numbers. Chapter 10 describes rational and complex arithmetic, which use extended numeric types.

Division by zero invariably raises the `err0dv` error, which has the following default screen display:

`**` *function* `: division by zero`

where *function* is the name of the function that raised the error.

4.1 Generic arithmetic

Arguments to the functions described here can be integer numbers, floats or numbers of an extended type. If one of the arguments is a float, the result will also be a float. If all the arguments are integers, the result is an integer. If a calculation overflows, or if one of the function's arguments is neither an integer nor a float, the `genarith` interrupt is called.

While it is advisable to use the functions described here, due to their generality, they are not particularly fast in execution. If efficiency is important, specialized arithmetic functions can be used on integers or on floats.

4.1.1 Generic arithmetic interrupt

(genarith) [*arithmetic interrupt*]

This interrupt is called if a calculation cannot be performed by a generic arithmetic function. It invokes a function whose name is the value of the following expression:

```
(or (getfn (type-of arg1) fn ())
    (getfn #:sys-package:genarith fn ()))
```

where `fn` is the name of the generic function that cannot perform the calculation, and $\texttt{arg}_1$ is the value of its first argument. This function is called with one or two arguments, depending on its type (see section 4.1.7). If the function does not exist, the `errgen` error is raised. Its default screen display is

** *function* : `can't compute` : *argument-list*

where *function* is the name of the function that could not be found, and *argument-list* is the list of evaluated arguments of the unsuccessful function call. This escape mechanism provides a way to easily build interfaces to other arithmetic packages written entirely in LISP.

#:sys-package:genarith [*variable*]

This variable holds the name of the package in which generic functions will be searched for, in the case that a function associated with the type of the first argument supplied is not found. By default, its value is `genarith`.

4.1.2 Type tests

Only the first function described here, `numberp`, calls the `genarith` interrupt on error.

(numberp x) [*function with one argument*]

`numberp` determines whether `x` is a number. If this is so, it returns `x`. If not, it returns `()`.

```
(numberp 44)    ⟹   44
(numberp 3.14)  ⟹   3.14
```

(fixp x) [*function with one argument*]

If `x` is a fixed-precision integer number, `fixp` returns `x`. If not, it returns `()`.

```
(fixp 120)      ⟹   120
(fixp 10.256)   ⟹   ()
```

(floatp x) [*function with one argument*]

If `x` is a fixed-precision floating-point number, `floatp` returns `x`. If not, it returns `()`.

```
(floatp 120)      ⟹   ()
(floatp 10.256)   ⟹   10.256
```

4.1.3 Numeric conversions

Even though certain numeric functions do their own argument conversion, it is also possible to explicitly convert numeric values by means of the following three functions.

(truncate x) [*function with one argument*]

(fix x) [*function with one argument*]

If the argument `x` is an integer, `truncate` returns it as its value. If `x` is a floating-point number, it will be converted into an integer if possible, and this value will be returned. If the conversion cannot be carried out, `truncate` calls the `genarith` interrupt. The conversion is a truncation. The result is an integer with the same sign as the argument `x`. Its absolute value is equal to the greatest integer less than or equal to the absolute value of the argument. `fix` is simply another name—retained for reasons of compatibility—for the same function.

```
(truncate 10.)      ⟹   10
(truncate 10.4)     ⟹   10
(truncate 10.6)     ⟹   10
(truncate -10.)     ⟹   -10
(truncate -10.4)    ⟹   -10
(truncate -10.6)    ⟹   -10
(truncate 32766.2)  ⟹   32766
(truncate 32768.)   ⟹      ** call to genarith **
```

(floor n) [*function with one argument*]

If the argument `n` is an integer, `floor` returns it. If `n` is a float, it is converted, if possible, into an integer. If not, `floor` raises the `genarith` interrupt. This conversion is a 'downwards' truncation. The result is an integer with the same sign as the `n` argument. Its value is the largest integer that is smaller than `n`.

```
(floor 10.)     ⟹   10
(floor 10.4)    ⟹   10
(floor 10.6)    ⟹   10
(floor -10.)    ⟹   -10
(floor -10.4)   ⟹   -11
(floor -10.6)   ⟹   -11
```

(ceiling n) [*function with one argument*]

If the argument `n` is an integer, `ceiling` returns it. If `n` is a float, it is converted, if possible, into an integer. If not, `ceiling` raises the `genarith` interrupt. This conversion is an 'upwards' truncation. The result is an integer with the same sign as the `n` argument. Its value is the smallest integer that is larger than `n`.

```
(ceiling 10.)    ⟹   10
(ceiling 10.4)   ⟹   11
(ceiling 10.4)   ⟹   11
(ceiling -10.)   ⟹   -10
(ceiling -10.4)  ⟹   -10
(ceiling -10.4)  ⟹   -10
```

(round n d) [*function with two arguments*]

Carries out a division of `n` by `d`. The function returns the integer that is closest to the result of the division. When `d` is one, `round` simply returns the integer that is closest to `n`.

```
(round 10. 1)    ⟹   10
(round 10.4 1)   ⟹   10
(round 10.6 1)   ⟹   11
(round 5 2)      ⟹   2
(round 5.2 -2)   ⟹   -3
(round -5 2)     ⟹   -2
```

(float x) [*function with one argument*]

If the argument `x` is a floating-point number, `float` returns it as its value. If `x` is an integer, it is converted into a float, and this value is returned. Otherwise, `float` calls the `genarith` interrupt.

```
(float 123)  ⟹   123.
```

4.1.4 Generic arithmetic functions

(+ x_1 ... x_n) [*function with a variable number of arguments*]

This function, referred to as *plus*, returns the value of the sum $x_1 + x_2 + \ldots + x_n$. If no argument is given, it returns the additive identity, `0`.

```
(+)                          ⟹   0
(+ 8)                        ⟹   8
(+ 5 6)                      ⟹   11
(+ -5 -6 1)                  ⟹   -10
(+ 32000 32000)              ⟹     ** call to genarith **
(+ 32000. 32000. 1)          ⟹   64001.
```

```
(+ 100. 1000. 10000. 100000.)  ⟹   111100.
```

(1+ x)

[*function with one argument*]

The form `(1+ x)` is equivalent to `(+ x 1)`.

(- x_1 ... x_n)

[*function with a variable number of arguments*]

This function, referred to as *minus*, returns the value of the difference $x_1 - x_2 - \ldots - x_n$. If no argument is given, it returns the additive identity, `0`. If a single argument is given, this function returns the additive inverse of its argument: that is, `-x`.

```
(-)                  ⟹   0
(- 20)               ⟹   -20
(- 123.)             ⟹   -123.
(- 6 2)              ⟹   4
(- 12 -7)            ⟹   19
(- 10000. 1000. 100) ⟹   8900.
(- #$8000 1 1)       ⟹     ** call to genarith **
```

(1- x)

[*function with one argument*]

The form `(1- x)` is equivalent to `(- x 1)`.

(abs x)

[*function with one argument*]

The `abs` function returns the absolute value of its argument `x`: that is, `|x|`.

```
(abs 10)      ⟹   10
(abs -10)     ⟹   10
(abs 10.23)   ⟹   10.23
(abs -10.23)  ⟹   10.23
```

To calculate the square of `x` without changing its sign, use: `(* x (abs x))`.

(* x_1 ... x_n)

[*function with a variable number of arguments*]

The `*` function returns the value of the product $x_1 \times x_2 \times \ldots \times x_n$. If no argument is given, `*` returns the multiplicative identity: that is, `1`.

```
(*)            ⟹   1
(* 5)          ⟹   5
(* 10 4)       ⟹   40
(* 2 -3)       ⟹   -6
(* -100 -100)  ⟹   10000
(* 1000 1000)  ⟹     ** call to genarith **
(* 2.2 3 2)    ⟹   13.2
```

(/ x y) [*function with one or two arguments*]

(// x y) [*function with one or two arguments*]

These identical functions return the value of the division of `x` by `y`. If a single argument is given, the inverse `1/x` is calculated and returned. If two integer arguments are given and their quotient is not exact, a call to `genarith` takes place, and `genarith` attempts to produce a rational number as result. To produce a floating-point result, the `divide` function must be used. To produce a rounded integer result, the `quotient` function should be used. If the argument `y` is zero, the `err0dv` error is raised.

```
(/ 40.)          ⟹    .025
(/ 5)            ⟹      ** call to genarith **
(/ 20 5)         ⟹    4
(/ 21 5)         ⟹      ** call to genarith **
(/ 40 -4)        ⟹    -10
(/ 123.45 6.7)   ⟹    18.42537
```

(quo x y) [*function with two arguments*]

(quotient x y) [*function with two arguments*]

These two identical functions return the value of the whole division of `x` by `y`. More precisely, they return an integer value that we shall refer to here as `q`. If `n` and `d` are two integers, then the quotient `q` and the remainder `r` satisfy the relation `n = q * d + r`, where `r` lies between zero and the absolute value of `d`. If the argument `y` is zero, the `err0dv` error is raised.

```
(quotient 17 7)         ⟹    2
(quotient 17 -7)        ⟹    -2
(quotient -17 7)        ⟹    -3
(quotient -17 -7)       ⟹    3
(quotient 12. 5)        ⟹    2
(quotient 123.45 6.7)   ⟹    18
```

(modulo x y) [*function with two arguments*]

Returns the value of the remainder of the integer division of `x` by `y`. If one of the arguments is not an integer, the `genarith` interrupt is raised. The remainder of an integer division—represented by the formula in the description of the preceding function—is always positive.

The result affects the `#:ex:mod` variable.

```
(modulo 17 7)     ⟹    3
(modulo 17 -7)    ⟹    3
(modulo -17 7)    ⟹    4
(modulo -17 -7)   ⟹    4
```

```
(modulo -8 2)     ⟹   0
(modulo 123 0)    ⟹      ** call to genarith **
(modulo 12.4 2)   ⟹      ** call to genarith **
```

(quomod x y) [*function with two arguments*]

#:ex:mod [*variable*]

This function combines the functionality of the two preceding functions. It returns the same value as `quotient` and writes the result of applying the `modulo` function to `x` and `y` into the `#:ex:mod` global variable.

In LE-LISP, `quomod` could be defined in the following manner:

```
(defun quomod (n1 n2)
    ; The actual implementation is much more efficient
    (setq #:ex:mod (modulo n1 n2))
    (quotient n1 n2))
```

(min x_1 ... x_n) [*function with a variable number of arguments*]

The symbol `min` stands for *minimum*. This function returns the smallest numeric value from among $x_1 \ldots x_n$.

```
(min 10)          ⟹   10
(min 10 20)       ⟹   10
(min -10 -20)     ⟹   -20
(min 1 3. 2 -7)   ⟹   -7
(min -2. 3 0 7)   ⟹   -2.
```

(max x_1 ... x_n) [*function with a variable number of arguments*]

The symbol `max` stands for *maximum*. This function returns the numerically largest value from among $x_1 \ldots x_n$.

```
(max 10)          ⟹   10
(max 10 20)       ⟹   20
(max -10 -20)     ⟹   -10
(max 1 3. 2 -7)   ⟹   3.
(max -2 3. 0 7)   ⟹   7
```

4.1.5 Predicates of generic arithmetic

These functions generally accept a variable number of arguments, which can be numbers of any type: integers, floats or extended numerics. If one of the arguments is a float, these functions do

floating-point comparisons. If all the arguments are integers, they do integer comparisons. There is automatic conversion between the arguments, with priority given to floating-point numbers.

All these functions return their first argument if the predicate holds true, and `()` if not.

In the event of a call to the `genarith` interrupt, the `<?>` function must be defined, since all the other functions are defined as functions of `<?>`.

(<?> x y)

[*function with two arguments*]

The `<?>` function returns `-1` if `x` is strictly less than `y`, `0` if `x` equals `y`, and `1` if `x` is strictly greater than `y`.

```
(<?> 100 100)   ⟹   0
(<?> 100 200.)  ⟹   -1
(<?> 200. 100)  ⟹   1
```

(= x_1 x_2 ... x_n)

[*function with a variable number of arguments*]

If x_1 is equal to x_2, and x_2 is equal to x_3, and ... and x_{n-1} is equal to x_n, then the `=` function returns x_1. Otherwise, it returns `()`.

```
(= 10 10)       ⟹   10
(= -3 -2)       ⟹   ()
(= 10 10.)      ⟹   10
(= 10 10. 10)   ⟹   10
(= 10 9 10)     ⟹   ()
```

(<> x y)

[*function with two arguments*]

(/= x y)

[*function with two arguments*]

These two functions are identical. If the value of `x` is different to that of `y`, they return `x`. Otherwise, they return `()`. This function is binary, since the function 'not equal to' is not transitive.

```
(<> 11 10)   ⟹   11
(<> -3 -3)   ⟹   ()
(<> 10. 10)  ⟹   ()
```

(>= x_1 x_2 ... x_n)

[*function with a variable number of arguments*]

If the value of x_1 is greater than or equal to that of x_2, and the value of x_2 is greater than or equal to that of x_3, and ... and the value of x_{n-1} is greater than or equal to that of x_n, then the `>=` function returns x_1 as its value. If such is not the case, the function returns `()`. This function can be used to test intervals.

```
(>= 3 7)        ⟹   ()
(>= 7 7)        ⟹   7
```

```
(>= 9. 9)       ⟹   9.
(>= 11 11 10)   ⟹   11
(>= 10 11 10)   ⟹   ()
```

(> x_1 x_2 ... x_n)

[*function with a variable number of arguments*]

If the value of x_1 is strictly greater than the value of x_2, and the value of x_2 is strictly greater than the value of x_3, and ... and the value of x_{n-1} is strictly greater than the value of x_n, then the `>` function returns the value of x_1. If this is not the case, it returns `()`.

```
(> 5 5)         ⟹   ()
(> 7 4)         ⟹   7
(> 9. 9)        ⟹   ()
(> 11 10 10)    ⟹   ()
(> 10 11 12)    ⟹   ()
```

(<= x_1 x_2 ... x_n)

[*function with a variable number of arguments*]

If the value of x_1 is less than or equal to that of x_2, and the value of x_2 is less than or equal to that of x_3, and ... and the value of x_{n-1} is less than or equal to that of x_n, then the `<=` function returns the value of x_1. If this is not the case, it returns `()`.

```
(<= 5 5)        ⟹   5
(<= 4 6)        ⟹   4
(<= 9. 9)       ⟹   9.
(<= 9 10 10)    ⟹   9
(<= 10 9 10)    ⟹   ()
```

(< x_1 x_2 ... x_n)

[*function with a variable number of arguments*]

If the value of x_1 is less than that of x_2, and the value of x_2 is less than that of x_3, and ... and the value of x_{n-1} is less than that of x_n, the `<` function returns the value of x_1. If this is not the case, it returns `()`.

```
(< 5 5)      ⟹   ()
(< 4 5)      ⟹   4
(< 9. 9)     ⟹   ()
(< 8 9 10)   ⟹   8
(< 8 9 9)    ⟹   ()
```

(zerop x)

[*function with one argument*]

`(zerop x)` is equivalent to `(= x 0)`.

(plusp x)

[*function with one argument*]

`(plusp x)` is equivalent to `(>= x 0)`.

(minusp x) *[function with one argument]*

`(minusp x)` is equivalent to `(< x 0)`.

4.1.6 Circular and mathematical functions

All these functions expect floating-point or integer arguments. If an argument of some other type is passed to one of them, the `genarith` interrupt is raised.

(sin x) *[function with one argument]*

(cos x) *[function with one argument]*

(asin x) *[function with one argument]*

(acos x) *[function with one argument]*

(atan x) *[function with one argument]*

```
(setq pi 3.1415926535)  ⟹   3.1415926535
(setq pi/2 (/ pi 2.))   ⟹   1.570795
(setq pi/4 (/ pi 4.))   ⟹   0.7853975
(sin 0)                 ⟹   0.
(sin pi/4)              ⟹   0.707107
(sin pi/2)              ⟹   1.
(asin (sin 1.2))        ⟹   1.2
(cos 0)                 ⟹   1.
(cos pi/4)              ⟹   0.707107
(cos pi/2)              ⟹   0.
(acos (cos 1.2))        ⟹   1.2
(* 4 (atan 1))          ⟹   3.141593
```

```
(defun fnt (x y)
    (* nf (+ (cos (/ x nd)) (cos (/ y nd)) 2)))

(defun pattern (nl nf nd)
    (let ((x) (y 30))
         (repeat nl
                 (setq x -30 y (1- y))
                 (repeat 59
                    (princn
                      (chrnth (fix (fnt (incr x) y))
                              "0 1 2 3 4 5 6 7 8 9 ")))
                 (terpri))))
```

The evaluation of `(pattern 13 3.5 5.5)` produces

```
55 44  33       33 44  5   66666   5  44 33       33  44 55
5 44  33  2222   33 44  55       55  44 33   2222  33  44 5
 44  3  22222222  33 44  555   555  44 33  22222222  3  44
44  3  22      22  33 44  5555555  44 33  22      22  3  44
4  3  22   11    22 33 44         44 33 22    11   22  3  4
  3  22  1111111  22 33  44     44  33 22  1111111  22  3
 3  22 111    111  22 33  4444444  33 22  111    111 22  3
33 22 11        11  2  33   444   33  2  11        11 22 33
3 22  1   0000   11 22  33       33  22 11   0000   1  22 3
  2  11  000000   1  22 333     333 22  1   000000  11  2
 22  1  00000000  11 22  333   333  22 11  00000000  1  22
 22 11  00000000  11  2  3333 3333  2  11  00000000  11 22
 22 11  00000000  11  2  3333 3333  2  11  00000000  11 22
```

(exp x) [*function with one argument*]

(log x) [*function with one argument*]

(log10 x) [*function with one argument*]

(power x y) [*function with two arguments*]

(sqrt x) [*function with one argument*]

```
(power 10 0)                                  ⟹   1.
(power 10 1)                                  ⟹   10.
(power 10 10)                                 ⟹   1e+10
(power 10 -1)                                 ⟹   .1
(power 10 -2)                                 ⟹   .01
(power 123.45 6.7)                            ⟹   1.030362e+14
(exp 0)                                       ⟹   1.
(exp 1)                                       ⟹   2.718282
(log 1)                                       ⟹   0.
(log 2.718282)                                ⟹   1.
(log (exp 1))                                 ⟹   1.
(log10 1)                                     ⟹   0.
(log10 10)                                    ⟹   1.
(log10 20)                                    ⟹   1.30103
(log10 100)                                   ⟹   2.
(sqrt 100)                                    ⟹   10.
(sqrt 1000000.)                               ⟹   1000.
(setq x 1.2)                                  ⟹   1.2
(sqrt (+ (power (sin x) 2) (power (cos x) 2)))  ⟹   1.
```

4.1.7 Extensions to generic arithmetic

To create new generic arithmetic facilities, the following functions should be defined in the extended arithmetic package, called `ext` in the list that follows. These functions are all unary or binary, and the system handles multiple arguments.

```
(#:ext:numberp s)
(#:ext:float s)
(#:ext:truncate s)
(#:ext:+ s1 s2)
(#:ext:- s1 s2)
(#:ext:0- s)
(#:ext:/ s1 s2)
(#:ext:1/ s1)
(#:ext:quomod s1 s2)
(#:ext:abs n)
(#:ext:<?> s1 s2)
(#:ext:sin s)
(#:ext:cos s)
(#:ext:asin s)
(#:ext:acos s)
(#:ext:atan s)
(#:ext:exp s)
(#:ext:log s)
(#:ext:log10 s)
(#:ext:power s)
(#:ext:sqrt s)
```

To define a new extended type, the following two functions are needed:

```
(#:ext:eval n)
(#:ext:prin n)
```

An example of the creation of extended arithmetic functions is given in chapter 10, section 3.

4.2 Integer arithmetic

Integer arithmetic functions use operands `n`— in the following descriptions that follow—that must necessarily be 16-bit fixed-precision integer-type objects. These functions only do type validity checking in the interpreted mode, and never do overflow checks. If the arguments received by one of them are not integers, the results will be unpredictable, and can even cause errors in the host operating system or hardware. All these functions are almost always compiled into a single LLM3 instruction.

In the interpreter, if one of the arguments is not an integer, the `errnia` error is raised. Its default screen is

** *function* : not a fix : *argument*

where *function* is the name of the function that raised the error, and *argument* is the faulty argument.

4.2.1 Integer arithmetic functions

(add1 x) [*function with one argument*]

Returns the value x + 1.

```
(add1 6)   ⟹   7
(add1 -3)  ⟹   -2
```

(sub1 x) [*function with one argument*]

Returns the value x − 1.

```
(sub1 7)   ⟹   6
(sub1 -3)  ⟹   -4
```

(add x y) [*function with two arguments*]

Returns the value of the sum x + y.

```
(add 5 6)  ⟹   11
```

(sub x y) [*function with two arguments*]

Returns the value of the difference x − y.

```
(sub 6 2)  ⟹   4
```

(mul x y) [*function with two arguments*]

Returns the value of the product x × y.

```
(mul 10 4)  ⟹   40
```

(div x y) [*function with two arguments*]

Returns the integer value of the quotient x ÷ y, truncated to the integer value. div does an integer division on integer arguments. If the argument y is zero, the err0dv error is raised.

```
(div 20 5)  ⟹   4
(div 21 5)  ⟹   4
```

(rem x y) [*function with two arguments*]

Returns the value of the remainder of the integer division of `x` by `y`.

```
(rem 11 3)    ⟹    2
(rem 14 22)   ⟹    14
```

(scale x y z) [*function with three arguments*]

Returns the value of the calculation of $x \times y \div z$.

So, `(scale x y z)` is equivalent to `(div (mul x y) z)`.

The intermediate value is stored in double-precision integer memory (depending somewhat on the host machine, but in at least 32 bits of storage) in order to avoid overflow in the intermediate calculation. The final result is an integer, truncated—if necessary—to 16 bits.

```
(scale 1000 1000 1000)    ⟹    1000
(scale -100 2000 -1000)   ⟹    200
(scale 1000 1000 3000)    ⟹    333
```

4.2.2 Fixed-precision integer comparison functions

These functions have either one or two arguments, written as `x` and `y`, which must be integer number expressions. The functions carry out type checking only in the interpreted mode. If the arguments given in a call to one of them are not integers, the results are meaningless in compiled code. All of these functions are almost always compiled into a single LLM3 instruction.

Each of these functions returns its first argument if the test is true of the arguments. If not, `()` is returned.

(evenp x) [*function with one argument*]

If the value of the argument `x` is even, `evenp` returns it as its value. Otherwise, it returns `()`.

(oddp x) [*function with one argument*]

If the value of the argument `x` is odd, `oddp` returns it as its value. Otherwise, it returns `()`.

(eqn x y) [*function with two arguments*]

If the value of `x` is equal to the value of `y`, `eqn` returns the value of `x`. Otherwise, it returns `()`.

```
(eqn 10 10)   ⟹    10
(eqn -3 3)    ⟹    ()
```

(neqn x y) [*function with two arguments*]

If the value of `x` is not equal to the value of `y`, `neqn` returns the value of `x`. Otherwise, it returns `()`.

```
(neqn 11 10)  ⟹  11
(neqn -3 -3)  ⟹  ()
```

`(ge x y)` [*function with two arguments*]

If the value of `x` is greater than or equal to the value of `y`, `ge` returns the value of `x`. Otherwise, it returns `()`.

```
(ge 3 7)  ⟹  ()
(ge 7 7)  ⟹  7
```

`(gt x y)` [*function with two arguments*]

If the value of `x` is greater than the value of `y`, `gt` returns the value of `x`. Otherwise, it returns `()`.

```
(gt 5 5)  ⟹  ()
(gt 7 4)  ⟹  7
```

`(le x y)` [*function with two arguments*]

If the value of `x` is less than or equal to the value of `y`, `le` returns the value of `x`. Otherwise, it returns `()`.

```
(le 5 5)  ⟹  5
(le 4 6)  ⟹  4
```

`(lt x y)` [*function with two arguments*]

If the value of `x` is less than the value of `y`, `lt` returns the value of `x`. Otherwise, it returns `()`.

```
(lt 5 5)  ⟹  ()
(lt 4 5)  ⟹  4
```

`(imin x y)` [*function with two arguments*]

If the value of `x` is less than or equal to the value of `y`, `imin` returns the value of `x`. Otherwise, it returns the value of `y`.

In LE-LISP, `imin` could be defined in the following manner:

```
(defun imin (x y)
    (if (le x y)
        x
        y))

(imin 10 10)    ⟹  10
```

```
(imin 10 11)     ⟹   10
(imin 11 10)     ⟹   10
(imin -10 -10)   ⟹   -10
(imin -10 -11)   ⟹   -11
(imin -11 -10)   ⟹   -11
```

(imax x y) *[function with two arguments]*

If the value of `x` is greater than or equal to the value of `y`, `imax` returns the value of `x`. Otherwise, it returns the value of `y`.

In LE-LISP, `imax` could be defined in the following manner:

```
(defun imax (x y)
    (if (ge x y)
        x
        y))
```

```
(imax 10 10)     ⟹   10
(imax 10 11)     ⟹   11
(imax 11 10)     ⟹   11
(imax -10 -10)   ⟹   -10
(imax -10 -11)   ⟹   -10
(imax -11 -10)   ⟹   -10
```

4.2.3 Boolean functions

In all the function descriptions that follow, the arguments `x` and `y` must be integers. These functions only work on 16-bit words, do no type conversion, and never cause numeric overflow exceptions. They are all compiled from single LLM3 instructions.

(lognot x) *[function with one argument]*

Returns the logical complement of its argument `n`.

The form `(lognot x)` is equivalent to `(logxor x #$ffff)`.

```
(lognot 0)    ⟹   -1 That is, hexadecimal #$ffff
(lognot -2)   ⟹   1
```

(logand x y) *[function with two arguments]*

Returns the logical *AND* of the values of `x` and `y`.

```
(logand #$36 #$25)   ⟹   #$24
```

To determine whether `x` is a power of 2, evaluate

```
(= x (logand x (- x)))
```

(logor x y) [*function with two arguments*]

Returns the logical *OR* of the values of `x` and `y`.

```
(logor #$15 #$17)  ⟹  #$17
```

(logxor x y) [*function with two arguments*]

Returns the logical exclusive *OR* of the values of `x` and `y`.

```
(logxor 5 3)  ⟹  6
```

(logshift x y) [*function with two arguments*]

Logically shifts the value `x` by `y` bit positions. If the value of `y` is positive, the shift is towards the left: that is, towards the most significant bit—the *msb*—performing in effect a multiplication on `x`. If the value of `y` is negative, the shift is rightwards: towards the least significant bit—the *lsb*—performing in effect a division on `x`.

In LE-LISP, `logshift` could be defined in the following manner:

```
(defun logshift (n nb)    ; This definition assumes n positive
    (if (= nb 0)
        n
        (if (< nb 0)
            (logshift (div n 2) (1+ nb))
            (logshift (mul n 2) (1- nb)))))
```

```
(logshift 1 0)   ⟹  1
(logshift 1 1)   ⟹  2
(logshift 1 3)   ⟹  8
(logshift 1 15)  ⟹  #$8000
(logshift 8 -3)  ⟹  1
```

4.2.4 Functions on bit fields

All these functions work on 16-bit words, and provide various means to manipulate arbitrary bit fields within these words. Throughout this section, a bit field is described by two numbers:

- `np` is the number of the first bit of the field, counting from zero.
- `nl` is the length of the bit field.

(2** n) [*function with one argument*]

Returns the value of two raised to the power `n`. This is the value that corresponds to a word with only the n^{th} bit set.

In LE-LISP, `2**` could be defined in the following manner:

```
(defun 2** (n)
    (logshift 1 n))
```

```
(2** 0)   ⟹   1
(2** 4)   ⟹   16
(2** 15)  ⟹   #$8000
```

(mask-field n np nl) *[function with three arguments]*

Returns the bit field of length `nl` beginning at bit `np` in the word `n`.

In LE-LISP, `mask-field` could be defined in the following manner:

```
(defun mask-field (n np nl)
    (logand (logshift (sub1 (2** nl)) np) n))
```

```
(mask-field #$f0 2 4)  ⟹   #$30
```

(load-byte n np nl) *[function with three arguments]*

Extracts the bit field of length `nl`, beginning at the bit `np`, from the word `n`.

In LE-LISP, `load-byte` could be defined in the following manner:

```
(defun load-byte (n np nl)
    (logshift (mask-field n np nl) (sub 0 np)))
```

```
(load-byte #$f0 2 4)  ⟹   #$c
```

(deposit-byte n np nl m) *[function with four arguments]*

Returns the value of the word `n` after the bit field of length `nl` beginning at bit `np` has been replaced by the value `m`.

In LE-LISP, `deposit-byte` could be defined in the following manner:

```
(defun deposit-byte (n np nl m)
    (let ((n1 (mask-field (logshift m np) np nl))
          (n2 (logand n (sub1 (2** np))))
          (n3 (logand n (lognot (sub1 (2** (add np nl)))))))
         (logor n1 (logor n2 n3))))
```

```
(deposit-byte #$f0 2 4 #$3)  ⟹   #$cc
```

(deposit-field n np nl m) *[function with four arguments]*

Returns the value of the word `n` after the bit-field of length `nl` beginning at bit `np` has been replaced by the corresponding bit field in the word `m`.

In LE-LISP, `deposit-field` could be defined in the following manner:

```
(defun deposit-field (n np nl m)
    (let ((n1 (mask-field m np nl))
          (n2 (logand n (sub1 (2** np))))
          (n3 (logand n (lognot (sub1 (2** (add np nl)))))))
         (logor n1 (logor n2 n3))))
```

```
(deposit-field #$f0 2 4 #$c)  ⟹  #$cc
```

(load-byte-test n np nl) *[function with three arguments]*

The `load-byte-test` function tests whether the bit field of length `nl` beginning at position `np` is non-zero. If so, the bit field is returned as value. Otherwise, `load-byte-test` returns `()`.

In LE-LISP, `load-byte-test` could be defined in the following manner:

```
(defun load-byte-test (n np nl)
    (neqn (load-byte n np nl) 0))
```

```
(load-byte-test #$f0 2 2)  ⟹  ()
(load-byte-test #$f0 2 4)  ⟹  #$c
```

4.2.5 Pseudo-random functions

(srandom x) *[function with an optional argument]*

Initializes the internal accumulator— the seed— of the pseudo-random number generator with the number `x`. If the argument is not provided, `srandom` returns the current accumulator value.

(random x y) *[function with two arguments]*

Returns a number from the interval `[x ... y[`. This function uses the method of linear congruence described in [Knuth vol 2]. The internal accumulator, `g`, is calculated according to the formula

```
g(n+1) = (a * g(n) + c) modulo m
```

with `m = 7*7*7*7*13`, `a = (7 * 13) + 1` and `c = 2731` (prime).

This gives a sequence of accumulators with period `m` (`= 31213`), ranging across the segment `[0 ... m-1]`. The random number is then calculated by means of the following formula

```
r = n1 + ((g * (n2 - n1)) / m)
```

This avoids the uneven distribution of the least significant figures of the results of the accumulator, modulo 7 or 13.

4.3 Extended integer arithmetic

In order to implement arbitrary-precision numbers, a group of extended fixed-precision integer functions is available. For all these functions, arguments—expressed here as `x`, `y` or `z`—are 16-bit

integers. All calculations are carried out upon unsigned 16-bit quantities. For reasons of efficiency, these functions do not test the type(s) of their argument(s), nor do they raise any errors. The carry-over of return values, when these are longer than 16 bits, is always stored in the system variable `#:ex:regret`.

`#:ex:regret` [*variable*]

This variable always contains the most significant bytes of extended fixed-precision integer functions, in the form of a 16-bit integer.

`(ex+ x y)` [*function with two arguments*]

Returns the least-significant bytes (a total of 16 bits) of the sum

```
x + y + #:ex:regret
```

The new overflow is found in the `#:ex:regret` variable.

```
(setq #:ex:regret 0)  ⟹  0
(ex+ 100 200)         ⟹  300
#:ex:regret           ⟹  0
(setq #:ex:regret 1)  ⟹  1
(ex+ 100 200)         ⟹  301
#:ex:regret           ⟹  0
(setq #:ex:regret 0)  ⟹  0
(ex+ #$ffff #$ffff)   ⟹  -2
#:ex:regret           ⟹  1
(setq #:ex:regret 1)  ⟹  1
(ex+ #$ffff #$ffff)   ⟹  -1
#:ex:regret           ⟹  1
```

`(ex1+ x)` [*function with one argument*]

Returns the least-significant bytes (a total of 16 bits) of the sum

```
x + 1 + #:ex:regret
```

It is therefore equivalent to `(ex+ x 1)`.

The new overflow is found in the `#:ex:regret` variable.

```
(setq #:ex:regret 0)  ⟹  0
(ex1+ 10)             ⟹  11
#:ex:regret           ⟹  0
(ex1+ #$ffff)         ⟹  0
#:ex:regret           ⟹  1
```

`(ex- x)` [*function with one argument*]

Returns the unsigned 16-bit inverse of `x`.

```
(ex- -2)   ⟹   1
(ex- -1)   ⟹   0
(ex- 0)    ⟹   -1
(ex- 1)    ⟹   -2
(ex- 2)    ⟹   -3
```

(ex* x y z) *[function with three arguments]*

Returns the least-significant bytes (16 bits total) of the calculation

`x * y + z + #:ex:regret`

The most-significant bytes (16 bits total) are found in the `#:ex:regret` variable.

```
(setq #:ex:regret 0)   ⟹   0
(ex* 100 100 10)       ⟹   10010
#:ex:regret            ⟹   0
(setq #:ex:regret 0)   ⟹   0
(ex* -1 -1 0)          ⟹   1
#:ex:regret            ⟹   -2
(setq #:ex:regret -1)  ⟹   -1
(ex* -1 -1 0)          ⟹   0
#:ex:regret            ⟹   -1
```

(ex/ x y) *[function with two arguments]*

The `#:ex:regret` variable should contain the most-significant bytes of the dividend, and `x` should contain the least-significant bytes of this same dividend. `ex/` returns the least-significant bytes (16 bits total) of the division of this dividend by the divisor `y`, and the remainder is stored in the `#:ex:regret` variable.

`#:ex:regret|x / y` is returned as the value of `ex/`.

`#:ex:regret` gets set to `#:ex:regret|x rem y`.

```
(setq #:ex:regret 0)   ⟹   0
(ex/ 100 5)            ⟹   20
#:ex:regret            ⟹   0
(setq #:ex:regret 1)   ⟹   1
(ex/ 100 5)            ⟹   13127
#:ex:regret            ⟹   1
(setq #:ex:regret -2)  ⟹   -2
(ex/ 0 -1)             ⟹   -2
#:ex:regret            ⟹   -2
(setq #:ex:regret -3)  ⟹   -3
(ex/ 3 -1)             ⟹   -2
#:ex:regret            ⟹   1
```

`(ex? x y)` [*function with two arguments*]

Returns `-1` if the value of `x` is strictly less than the value of `y`, `0` if these values are equal, and `1` if the value of `x` is strictly greater than the value of `y`. The comparisons are carried out on 16-bit unsigned quantities.

```
(ex? 0 0)     ⟹   0
(ex? 2 3)     ⟹   -1
(ex? 3 2)     ⟹   1
(ex? -1 -2)   ⟹   1
(ex? -1 1)    ⟹   1
(ex? -1 -1)   ⟹   0
(ex? -2 -1)   ⟹   -1
```

4.4 Floating-point arithmetic

Functions described here take arguments—`x` and `y` in the descriptions that follow—that must be floats. They perform type checking only in the interpreted mode. If the arguments given in a call to one of them are not floating-point numbers, the results will be undefined and might cause errors in the host operating system or hardware. All these functions are compiled into single LLM3 instructions.

In the interpreter, if one of the arguments is not a floating-point number, the `errnfa` error is raised. Its default screen display is

`**` *function* `: not a float :` *argument*

where *function* is the name of the function that raised the error, and *argument* is the faulty argument.

4.4.1 Floating-point arithmetic functions

`(fadd x y)` [*function with two arguments*]

Returns the value of the sum $x + y$.

```
(fadd 5. 6.)   ⟹   11.
```

`(fsub x y)` [*function with two arguments*]

Returns the value of the difference $x - y$.

```
(fsub 6. 2.)   ⟹   4.
```

`(fmul x y)` [*function with two arguments*]

Returns the value of the product $x \times y$.

```
(fmul 10. 4.)  ⟹  40.
```

(fdiv x y) [*function with two arguments*]

Returns the quotient `x` ÷ `y`. If the argument `y` is `0.`, the `err0dv` error is raised.

```
(fdiv 11. 2.)  ⟹  5.5
(fdiv 10. 2.)  ⟹  5.
```

4.4.2 Floating-point arithmetic comparisons

Each of these functions takes two arguments—`x` and `y` in the descriptions that follow—which must be floating-point numbers. They only perform type checks on their arguments in the interpreted mode. If the arguments passed to these functions are not floating-point numbers, the results are meaningless for compiled functions.

All these functions return their first argument if the test succeeds, and `()` if not.

(feqn x y) [*function with two arguments*]

If the value of `x` is equal to the value of `y`, `feqn` returns `x`. Otherwise, it returns `()`.

```
(feqn 10. 10.)  ⟹  10.
(feqn -3. 3.)   ⟹  ()
```

(fneqn x y) [*function with two arguments*]

If the value of `x` is not equal to the value of `y`, `fneqn` returns `x`. Otherwise, it returns `()`.

```
(fneqn 11. 10.)  ⟹  11.
(fneqn -3. -3.)  ⟹  ()
```

(fge x y) [*function with two arguments*]

If the value of `x` is greater than or equal to the value of `y`, `fge` returns `x`. Otherwise, it returns `()`.

```
(fge 3. 7.)  ⟹  ()
(fge 7. 7.)  ⟹  7.
```

(fgt x y) [*function with two arguments*]

If the value of `x` is greater than the value of `y`, `fgt` returns `x`. Otherwise, it returns `()`.

```
(fgt 5. 5.)  ⟹  ()
(fgt 7. 4.)  ⟹  7.
```

(fle x y) [*function with two arguments*]

If the value of `x` is less than or equal to the value of `y`, `fle` returns `x`. Otherwise, it returns `()`.

```
(fle 5. 5.)  ⟹  5.
(fle 4. 6.)  ⟹  4.
```

(flt x y) [*function with two arguments*]

If the value of `x` is less than the value of `y`, `flt` returns `x`. Otherwise, it returns `()`.

```
(flt 5. 5.)  ⟹  ()
(flt 4. 5.)  ⟹  4.
```

(fmin x y) [*function with two arguments*]

If the value of `x` is less than or equal to the value of `y`, `fmin` returns `x`. Otherwise, it returns `n2`.

In LE-LISP, `fmin` could be defined in the following manner:

```
(defun fmin (x y)
    (if (fle x y)
        x
        y))
```

```
(fmin 10. 10.)    ⟹  10.
(fmin 10. 11.)    ⟹  10.
(fmin 11. 10.)    ⟹  10.
(fmin -10. -10.)  ⟹  -10.
(fmin -10. -11.)  ⟹  -11.
(fmin -11. -10.)  ⟹  -11.
```

(fmax x y) [*function with two arguments*]

If the value of `x` is greater than or equal to the value of `y`, `fmax` returns `x`. Otherwise, it returns `y`.

In LE-LISP, `fmax` could be defined in the following manner:

```
(defun fmax (x y)
    (if (fge x y)
        x
        y))
```

```
(fmax 10. 10.)    ⟹  10.
(fmax 10. 11.)    ⟹  11.
(fmax 11. 10.)    ⟹  11.
(fmax -10. -10.)  ⟹  -10.
(fmax -10. -11.)  ⟹  -10.
(fmax -11. -10.)  ⟹  -10.
```

4.5 Mixed arithmetic

Arguments to these functions can be either integers or floating-point numbers. If one of their arguments is a float, or if an integer arithmetic operation overflows, the result is a float. If all the arguments are integers, the result is an integer. Therefore, the functions carry out automatic argument conversion, giving priority to floating-point numbers. If one of the arguments is neither an integer nor a floating-point number, the `errnna` error is raised. Its default screen display is

** *function* : not a number : *argument*

where *function* is the name of the function that raised the error, and *argument* is the faulty argument.

There are no special comparison functions for this kind of arithmetic. The generic comparisons are sufficient.

Mixed arithmetic is just a kind of floating-point arithmetic in which integers can be used.

(plus x_1 ... x_n) [*function with a variable number of arguments*]

Returns the value of the sum $x_1 + x_2 + \ldots + x_n$. If no argument is given, `plus` returns the additive identity: `0`.

```
(plus)                          ⟹   0
(plus 10)                       ⟹   10
(plus 5 6)                      ⟹   11
(plus -5 -6 1)                  ⟹   -10
(plus 100. 1000. 10000. 100000.)  ⟹   111100.
(plus 32000 32000 1)            ⟹   64001.
```

(differ x_1 ... x_n) [*function with a variable number of arguments*]

(difference x_1 ... x_n) [*function with a variable number of arguments*]

These two identical functions return the value of the difference $x_1 - x_2 - \ldots - x_n$. If no argument is given, they return the (right) subtractive identity: `0`. Given a single argument, they return its additive inverse.

```
(difference)                  ⟹   0
(difference 20.)              ⟹   -20.
(difference 6 2)              ⟹   4
(difference 12 -7)            ⟹   19
(difference 10000. 1000. 100)  ⟹   8900.
(difference #$8000 1 1)       ⟹   -32770
```

(times x_1 ... x_n) [*function with a variable number of arguments*]

Returns the value of the product $x_1 \times x_2 \times \ldots \times x_n$. If no argument is given, `times` returns the

multiplicative identity: `1`.

```
(times)            ⟹   1
(times 5)          ⟹   5
(times 10 4)       ⟹   40
(times 2 -3)       ⟹   -6
(times -100 -100)  ⟹   10000
(times 2.2 3 2)    ⟹   13.2
(times 1000 1000)  ⟹   1.e+06
```

(divide x y)

[function with two arguments]

Returns the value of the quotient `x` ÷ `y`. If the two arguments are both integers and their quotient is not exact, `divide` returns a floating-point value. If the argument `y` is zero, the `err0dv` error is raised.

```
(divide 10 5)         ⟹   2
(divide 12 5)         ⟹   2.4
(divide 12. 5)        ⟹   2.4
(divide 40 -4)        ⟹   -10
(divide 123.45 6.7)   ⟹   18.42537
```

Table of contents

4 Arithmetic functions 4-1

4.1 Generic arithmetic 4-1

4.1.1 Generic arithmetic interrupt 4-2

4.1.2 Type tests 4-2

4.1.3 Numeric conversions 4-3

4.1.4 Generic arithmetic functions 4-4

4.1.5 Predicates of generic arithmetic 4-7

4.1.6 Circular and mathematical functions 4-10

4.1.7 Extensions to generic arithmetic 4-12

4.2 Integer arithmetic 4-12

4.2.1 Integer arithmetic functions 4-13

4.2.2 Fixed-precision integer comparison functions 4-14

4.2.3 Boolean functions 4-16

4.2.4 Functions on bit fields 4-17

4.2.5 Pseudo-random functions 4-19

4.3 Extended integer arithmetic 4-19

4.4 Floating-point arithmetic 4-22

4.4.1 Floating-point arithmetic functions 4-22

4.4.2 Floating-point arithmetic comparisons 4-23

4.5 Mixed arithmetic 4-25

Function Index

`(genarith)` [*arithmetic interrupt*] 4-2
`#:sys-package:genarith` [*variable*] 4-2
`(numberp x)` [*function with one argument*] 4-2
`(fixp x)` [*function with one argument*] 4-2
`(floatp x)` [*function with one argument*] 4-3
`(truncate x)` [*function with one argument*] 4-3
`(fix x)` [*function with one argument*] 4-3
`(floor n)` [*function with one argument*] 4-3
`(ceiling n)` [*function with one argument*] 4-4
`(round n d)` [*function with two arguments*] 4-4
`(float x)` [*function with one argument*] 4-4
`(+` x_1 ... x_n`)` [*function with a variable number of arguments*] 4-4
`(1+ x)` [*function with one argument*] 4-5
`(-` x_1 ... x_n`)` [*function with a variable number of arguments*] 4-5
`(1- x)` [*function with one argument*] 4-5
`(abs x)` [*function with one argument*] 4-5
`(*` x_1 ... x_n`)` [*function with a variable number of arguments*] 4-5
`(/ x y)` [*function with one or two arguments*] 4-6
`(// x y)` [*function with one or two arguments*] 4-6
`(quo x y)` [*function with two arguments*] 4-6
`(quotient x y)` [*function with two arguments*] 4-6
`(modulo x y)` [*function with two arguments*] 4-6
`(quomod x y)` [*function with two arguments*] 4-7
`#:ex:mod` [*variable*] 4-7
`(min` x_1 ... x_n`)` [*function with a variable number of arguments*] 4-7
`(max` x_1 ... x_n`)` [*function with a variable number of arguments*] 4-7

(<?> x y) [*function with two arguments*] 4-8

(= x_1 x_2 ... x_n) [*function with a variable number of arguments*] 4-8

(<> x y) [*function with two arguments*] 4-8

(/= x y) [*function with two arguments*] 4-8

(>= x_1 x_2 ... x_n) [*function with a variable number of arguments*] 4-8

(> x_1 x_2 ... x_n) [*function with a variable number of arguments*] 4-9

(<= x_1 x_2 ... x_n) [*function with a variable number of arguments*] 4-9

(< x_1 x_2 ... x_n) [*function with a variable number of arguments*] 4-9

(zerop x) [*function with one argument*] 4-9

(plusp x) [*function with one argument*] 4-9

(minusp x) [*function with one argument*] 4-10

(sin x) [*function with one argument*] 4-10

(cos x) [*function with one argument*] 4-10

(asin x) [*function with one argument*] 4-10

(acos x) [*function with one argument*] 4-10

(atan x) [*function with one argument*] 4-10

(exp x) [*function with one argument*] 4-11

(log x) [*function with one argument*] 4-11

(log10 x) [*function with one argument*] 4-11

(power x y) [*function with two arguments*] 4-11

(sqrt x) [*function with one argument*] 4-11

(add1 x) [*function with one argument*] 4-13

(sub1 x) [*function with one argument*] 4-13

(add x y) [*function with two arguments*] 4-13

(sub x y) [*function with two arguments*] 4-13

(mul x y) [*function with two arguments*] 4-13

(div x y) [*function with two arguments*] 4-13

(rem x y) [*function with two arguments*] 4-14

(scale x y z) [*function with three arguments*] 4-14

(evenp x) [*function with one argument*] 4-14

(oddp x) [*function with one argument*] 4-14

(eqn x y) [*function with two arguments*] 4-14

(neqn x y) [*function with two arguments*] 4-14

(ge x y) [*function with two arguments*] 4-15

`(gt x y)` [*function with two arguments*] 4-15
`(le x y)` [*function with two arguments*] 4-15
`(lt x y)` [*function with two arguments*] 4-15
`(imin x y)` [*function with two arguments*] 4-15
`(imax x y)` [*function with two arguments*] 4-16
`(lognot x)` [*function with one argument*] 4-16
`(logand x y)` [*function with two arguments*] 4-16
`(logor x y)` [*function with two arguments*] 4-17
`(logxor x y)` [*function with two arguments*] 4-17
`(logshift x y)` [*function with two arguments*] 4-17
`(2** n)` [*function with one argument*] 4-17
`(mask-field n np nl)` [*function with three arguments*] 4-18
`(load-byte n np nl)` [*function with three arguments*] 4-18
`(deposit-byte n np nl m)` [*function with four arguments*] 4-18
`(deposit-field n np nl m)` [*function with four arguments*] 4-18
`(load-byte-test n np nl)` [*function with three arguments*] 4-19
`(srandom x)` [*function with an optional argument*] 4-19
`(random x y)` [*function with two arguments*] 4-19
`#:ex:regret` [*variable*] 4-20
`(ex+ x y)` [*function with two arguments*] 4-20
`(ex1+ x)` [*function with one argument*] 4-20
`(ex- x)` [*function with one argument*] 4-20
`(ex* x y z)` [*function with three arguments*] 4-21
`(ex/ x y)` [*function with two arguments*] 4-21
`(ex? x y)` [*function with two arguments*] 4-22
`(fadd x y)` [*function with two arguments*] 4-22
`(fsub x y)` [*function with two arguments*] 4-22
`(fmul x y)` [*function with two arguments*] 4-22
`(fdiv x y)` [*function with two arguments*] 4-23
`(feqn x y)` [*function with two arguments*] 4-23
`(fneqn x y)` [*function with two arguments*] 4-23
`(fge x y)` [*function with two arguments*] 4-23
`(fgt x y)` [*function with two arguments*] 4-23
`(fle x y)` [*function with two arguments*] 4-24

`(flt x y)` [*function with two arguments*] 4-24
`(fmin x y)` [*function with two arguments*] 4-24
`(fmax x y)` [*function with two arguments*] 4-24
`(plus` x_1 ... x_n`)` [*function with a variable number of arguments*] 4-25
`(differ` x_1 ... x_n`)` [*function with a variable number of arguments*] 4-25
`(difference` x_1 ... x_n`)` [*function with a variable number of arguments*] 4-25
`(times` x_1 ... x_n`)` [*function with a variable number of arguments*] 4-25
`(divide x y)` [*function with two arguments*] 4-26

Chapter 5

Object-oriented programming

This chapter first describes functions that provide ways to create structured objects in LISP. These objects are analogous to Pascal *records*. In addition, LISP allows you to create a hierarchy of structure types.

The second section is a description of the basic LISP type hierarchy and the means provided for extending it: typed vectors, typed strings and tagged pairs. Since they are implemented as typed vectors, structures provide the most immediate way to extend the LE-LISP type scheme.

The third section explains how to use the primitive `oblist` search functions and the typing functions to implement an object-oriented language of the SmallTalk variety.

The fourth section presents an abbreviation facility for describing typed objects.

The final section describes a minimal kernel on which object-oriented extensions can be built. This kernel, called **MicroCeyx**, benefits from experience acquired in the creation at INRIA of earlier object-oriented products called Ceyx and Alcyone.

5.1 Structures

The `defstruct` primitive provides the means to define new types of structured objects (structures) analogous to Pascal records. The `new` function facilitates the creation of *instances* of these types of objects.

Structured objects are made up of a number of named fields. Each field can contain any kind of LISP object. These fields are not typed. You can examine and modify the contents of the fields of a structured object by means of field-access functions, which are defined along with structures.

You can define a structure as a sub-structure of a previously defined structure. In this case, in addition to the fields explicitly declared within its definition, the sub-structure will inherit all the fields of the original parent structure.

`defstruct` [*feature*]

This feature indicates that functions on structures are loaded into the system.

5.1.1 Structure definition

(defstruct `struct field`$_1$ ... `field`$_n$**)** *[macro]*

Defines a structure named `struct`, which is a symbol, containing the fields `field`$_1$ ... `field`$_n$. `struct` becomes a new type of LISP object. (See the `type-of` function.)

> `field`$_1$ ... `field`$_n$ are the names of the structure's fields. They are either symbols or two-element lists. In the second case, the first element in the list is the field name, and the second element is a LISP form that is evaluated to assign an initial value to the field when instances are created. This evaluation takes place each time an instance of the structure is created.
>
> When a structure is defined, read-access and write-access functions, as well as a specific instance creation function, are defined simultaneously. These functions are described further on in this document.
>
> If `struct` is of the form `#:super-structure:structure`, where `super-structure` is the name of a structure previously defined in the system, `struct` will have, in addition to the fields `field`$_1$ ... `field`$_n$, all the fields of `super-structure`. In this case, the type `#:super-structure:structure` is a sub-type of the `super-structure` type.

#:system:defstruct-all-access-flag *[variable]*

> The `defstruct` macro examines this flag to determine whether access functions defined for inherited fields should be made available for use with the structure currently being defined. By default, this flag is *true*. In other words, it has the value `t`. If it is false, then the only access functions defined and available for use with this structure will be those for fields that are supplied explicitly as arguments to the relevant `defstruct` macro.

5.1.2 Instance creation

(new `struct`**)** *[function with one argument]*

(#:struct:make) *[function with no arguments]*

These two functions create a new instance of the `struct` structure, and return it as their value. The `#:struct:make` function is the creation function—*constructor*—for the `struct` structure. It is automatically defined by the `defstruct` macro. Objects created with this constructor will have type `struct`. (See the `type-of` function.)

The fields of the object so created are (optionally) initialized with the initialization forms included in the structure definition. These forms are evaluated when the object is created. The order of evaluation of the initialization forms is not fully determined. Fields for which no initialization forms are provided are initialized with the value `()`.

If the argument to the `new` function is not the name of a structure already defined in the system, the `errstc` error is raised, and this screen is displayed:

```
** <fn> : not a structure : <s>
```

where `fn` is the name of the function that raised the error (in this case, `new`), and `s` is the bad argument.

In LE-LISP, `new` could be defined in the following manner:

```
(defun new (type)
    (if (structurep type)
        (apply (symbol type 'make) ())
        (error 'new 'errstc type)))
```

5.1.3 Access to fields

(#:struct:field o e) *[function with one or two arguments]*

`struct` is the name of a structure, and `field` is the name of one of the structure's fields. The `#:struct:field` functions are the access functions to the fields of instances of the structure `struct`, and they are automatically defined by the `defstruct` function.

If the object `o` is not an instance of the `struct` structure (see the `typep` function), the `errstc` error is raised. Given a single argument, an access function returns the value of the field `field` of the object `o`. If a second argument is given, its value is stored in the field `field` of the object `o` and is also returned as the value of the access function.

Suppose that the LE-LISP compiler is active when the structure is defined. Consequently, the form `(featurep 'compiler)` is true. In this case, field-access functions are defined to be compiler macros. (See the `defmacro-open` function.) Compilation of calls to these functions produces extremely efficient code. After being compiled, access functions will not do type checks on their first arguments.

5.1.4 Type tests

(structurep o) *[function with one argument]*

This predicate is true if the object `o` is an instance of a defined structure. (See also the `typep` function.)

Here, for example, is the definition of a structure, `person`, with two fields: `age` and `weight`. The second field is initialized with the value `123`:

```
? (defstruct person
?  age
?  (weight 123))
= person
```

Creation of an instance:

```
? (setq person1 (#:person:make))
= #:person:#[() 123]
```

Assign the value `36` to the `age` field of the `person1` object:

```
? (#:person:age person1 36)
= 36
```

Extract the `weight` field:

```
? (#:person:weight person1)
= 123
```

Use the `structurep` function:

```
? (structurep person1)
= t
```

Next, we look at an example of a sub-structure. Extend the `person` structure into a `researcher` structure:

```
? (defstruct #:person:researcher
?  project
?  articles)
= #:person:researcher
```

Creation of a `researcher` and automatic initialization of a field:

```
? (setq r1 (#:person:researcher:make))
= #:person:researcher:#[() 123 () ()]    ? (#:person:researcher:articles r1
'("conf-proc" "thesis"))
= (conf-proc thesis)
```

The `researcher` inherits the fields of the `person` structure:

```
? (#:person:weight r1)
= 123    ? (#:person:age r1 26)
= 26    ? (#:person:researcher:weight r1)
= 123    ? (#:person:researcher:age r1)
= 26
```

5.1.5 Implementation of structures

Instances of a structure are implemented as typed vectors whose type is the name of the structure. The values of the object's fields are stored in vector cells.

5.2 Le-Lisp typology

LE-LISP object types are represented by symbols. The LISP package hierarchy provides a way to organize these types. The symbol `#:person:researcher` represents the `researcher` sub-type of the `person` type.

It is not strictly necessary to explicitly *create* new types. Simply invoking the symbol `#:foo:bar` results in a successful *mention* of the type `foo` and its sub-type `bar`. Any LISP symbol can therefore be interpreted as being a type.

In practice, the interesting types are those capable of being returned as values of the `type-of` function, either a predefined system type, or a *user type*: that is, the type of a typed vector, a typed string or a tagged pair.

The basic LISP types are predefined. In other words, they are 'already present' at system start-up. They all have the empty type (the type `||`), which is the universal type.

The basic LISP types are:

- `fix`: small integers
- `float`: floating-point numbers
- `symbol`: symbols
- `null`: the end-of-list marker, `()`
- `string`: character strings
- `vector`: vectors of pointers
- `cons`: list cells

(type-of s) [*function with one argument*]

This function determines the type of any LISP object. If the argument is a string or a vector, `type-of` returns the type of the object in question. In these cases, `type-of` is equivalent to `typestring` or `typevector`, respectively. If the argument is a tagged pair, `type-of` returns its `car`, provided that this is an atom or a list whose `car` is an atom. In all other cases—except the error condition associated with tagged pairs (see below)—`type-of` returns the LISP type of the object: `fix`, `float`, `cons`, etc.

If the `car` of a tagged pair argument is neither a symbol nor the symbol of a list, the result of `type-of` is undefined.

In LE-LISP, `type-of` could be defined in the following manner:

```
(defun type-of (s)
    (cond ((fixp s) 'fix)
          ((floatp s) 'float)
          ((null s) 'null)
          ((symbolp s) 'symbol)
          ((consp s)
           (if (tconsp s)
               (when (or (atom (car s)) (symbolp (caar s)))
                     (car s))
               'cons))
          ((vectorp s) (typevector s))
```

```
        ((stringp s) (typestring s))
        (t ())))
```

```
? (type-of 12)
= fix   ? (type-of 12.2)
= float   ? (type-of ())
= null   ? (type-of 'asd)
= symbol   ? (type-of "assasa")
= string   ? (type-of #[1 12 3])
= vector   ? (type-of #:foo:bar:"hello mamma")
= #:foo:bar   ? (type-of #:person:#[() 75])
= person   ? (type-of '(1))
= cons   ? (type-of '#(foo . a))
= foo
```

(subtypep type1 type2) *[function with two arguments]*

`type1` and `type2` are two symbols, representing LISP types. The predicate `subtypep` is true if `type1` is a sub-type of `type2`, and false otherwise.

In LE-LISP, `subtypep` could be defined in the following manner:

```
(defun subtypep (type1 type2)
    (or (eq type1 type2)
        (ifn (packagecell type1)
             (null type2)
             (subtypep (packagecell type1) type2))))
```

```
? (subtypep '#:person:researcher 'person)
= t   ? (subtypep 'symbol 'symbol)
= t   ? (subtypep '#:person:researcher '#:person:discoverer)
= ()   ? (subtypep 'fix '||)
= t
```

(typep s type) *[function with two arguments]*

This predicate is true if the type of the object `s` is a sub-type of `type`.

In LE-LISP, `typep` could be defined in the following manner:

```
(defun typep (s type)
    (subtypep (type-of s) type))
```

```
? (typep (#:person:researcher:make) 'person)
= t   ? (typep 'foo 'symbol)
= t   ? (typep (#:person:researcher:make) '#:person:discoverer)
= ()   ? (typep 12 '||)
= t
```

5.3 Object-oriented programming

The LE-LISP package hierarchy is used to encode the hierarchy of data types. In an object-oriented language of the SmallTalk variety—such as Ceyx or Alcyone, for example—one wants to be able to attach *methods* to object types.

In LE-LISP, these methods are LISP functions, defined in the packages to whose types they are attached. So, the function `#:person:describe` is considered to be the `describe` method attached to the `person` type.

The goal is for the sub-types of a given type to *inherit* the methods attached to this type. For example, the objects of type `#:person:researcher` should inherit the `describe` method of the `person` object. The predefined `getfn` function provides a way to do this method search, accounting for inheritance.

Sometimes it is desirable to have an object type inherit methods from several LISP types. This is multiple inheritance. The `getfn` function searches for a function in a list of packages, by first doing a depth-first search in the parent (superior) packages of those in the list.

Finally, you sometimes need methods associated with a mixture of several types of objects. For instance, to implement a generic arithmetic package, methods would have to be defined that allow you to add together different types of numbers such as `float`, `fix`, `bignum` and `rational`. In LE-LISP, the methods of the *product* of types `t1` and `t2` are located in the *product package* designated as `(t1 . t2)`. An addition method of the desired kind would be written as `#:(t1 . t2):add`. The predefined `getfn2` function provides a means of searching for product methods.

5.3.1 Searching for methods

The following functions are predefined search functions that operate on methods.

(getfn1 `pkgc fn`**)** [*function with two arguments*]

Searches for a function named `fn` in the `pkgc` package. If this function exists, it is returned as the value of `getfn1`. If not, the function returns `()`.

In LE-LISP, `getfn1` could be defined in the following manner:

```
(defun getfn1 (pkgc fn)
    (let ((nom (symbol pkgc fn)))
         (if (typefn nom)
             nom
             ())))
```

```
(getfn1 () 'car)       ⟹   car
(defun #:foo:bar ())   ⟹   #:foo:bar
(getfn1 'foo 'bar)     ⟹   #:foo:bar
(getfn1 'gee 'bar)     ⟹   ()
```

(getfn pkgc1 fn pkgc2) [*function with two or three arguments*]

Searches for a function named `fn` in the `pkgc1` package, or in one of the packages that is superior to `pkgc1` in the package hierarchy. If the optional `pkgc2` argument is given, it is excluded from the search. On the search up the package tree, if `pkgc2` is encountered, the search is terminated immediately. `pkgc2` is not searched. If the function is found, it is returned as the value of `getfn`. If not, `getfn` returns `()`.

If `pkgc1` is a list of packages, the search begins in the first package in the list and in all its parent packages, then in the second package in the list and in all its parent packages, and so on, until either the function is found or the list of packages names is exhausted. This *depth-search* also takes place if one of the parents of `pkgc1` is a list. This mechanism can be used to implement *multiple inheritance* of methods. In this case, the third argument, if present, provides a way to stop the search for each package in the list.

In LE-LISP, `getfn` could be defined in the following manner:

```
(defun getfn (pkgc symb . lastpkgc)
    (let ((nom (symbol pkgc symb)))
         (cond ((typefn nom) nom)
               ((null pkgc) ())
               ((and (consp lastpkgc)
                     (eq (packagecell pkgc)
                         (car lastpkgc)))
                ())
               (t (getfn (packagecell pkgc)
                         symb lastpkgc)))))
```

Let us define some functions:

```
(defun foo ())                    ⟹   foo
(defun #:bar:foo ())              ⟹   #:bar:foo
(defun #:bar:gee:buz:foo ())      ⟹   #:bar:gee:buz:foo
(getfn '#:bar:gee:buz 'foo)       ⟹   #:bar:gee:buz:foo
(getfn '#:bar:gee 'foo)           ⟹   #:bar:foo
(getfn 'bar 'foo)                 ⟹   #:bar:foo
(getfn () 'foo)                   ⟹   foo
(getfn '#:potop:teraz 'foo)       ⟹   foo
(getfn '#:bar:gee 'foo ())        ⟹   #:bar:foo
(getfn 'bar 'foo ())              ⟹   #:bar:foo
(getfn 'gee 'foo ())              ⟹   ()
(getfn '#:bar:gee:buz 'foo 'bar)  ⟹   #:bar:gee:buz:foo
(getfn '#:bar:gee 'foo 'bar)      ⟹   ()
```

(getfn2 pkgc1 pkgc2 fn) [*function with three arguments*]

Allows you to search for the function `fn` in the product package `(pkgc1 . pkgc2)`, or in one of the product packages that is superior to this one. The product package hierarchy is built from the

package hierarchy by imposing a lexicographic ordering on the combinations of possible pairs. The climb up the product package hierarchy is always stopped before reaching the root package for one of the two members of the product.

The packages superior to the package `(#:a:b . #:c:d)` are, in order

```
(#:a:b . #:c:d) < (#:a:b . c) < (a . #:c:d) < (a . c)
```

5.3.2 Invoking methods

The combination of type-determinating functions and method search functions provides a way to implement message transmission functions.

(send `symb o` $p_1 \ldots p_n$**)** [*function with a variable number of arguments*]

(send-error `symb rest`**)** [*function with two arguments*]

The `send` function searches for and invokes the method `symb` asssociated with the type of the object `o`. This method must be a LISP function that takes $n + 1$ arguments. It is called with the list of arguments formed by the object `o` and the n arguments $p_1 \ldots p_n$.

The intended method is the LISP function named `symb` in the package that is the type of the object `o`, or in one of its parent packages, up to and excluding the root package. If no such method is found, `send` calls the `send-error` function. By default, this function raises the `errudm` error and displays the following error screen:

```
** <fn> : undefined method : <l>
```

where `l` is the list of arguments supplied to `send`, and `fn` is the name of the function that raises the error: here, `send`.

In LE-LISP, `send` could be defined in the following manner:

```
(defun send (m o . larg)
    (let ((fn (getfn (type-of o) m '||)))
         (if fn
             (apply fn o larg)
             (funcall 'send-error m (cons o larg)))))
```

In LE-LISP, `send-error` could be defined in the following manner:

```
(defun send-error (m rest)
    (error 'send 'errudm (cons m rest)))
```

Here is a definition of the `hello` method for objects of type `fix`:

```
? (defun #:fix:hello (n)
?  (repeat n (print "hello")))
= #:fix:hello
```

Search for this method and invoke it:

```
? (send 'hello 3)
hello
hello
hello
= t
```

And here is a definition of the `hello` method for objects of the type `person`:

```
? (defun #:person:hello (pers)
?  (print "I am a person")
?  (print "I weigh " (#:person:weight pers) " pounds"))
= #:person:hello   ? (send 'hello (#:person:make))
I am a person
I weigh 123 pounds
=  pounds   ? (send 'hello (#:person:researcher:make))
I am a person
I weigh 123 pounds
=  pounds
```

(send-super type symb o p_1 ... p_n) [*function with a variable number of arguments*]

The function behaves very much like the `send` function, except that the method is sought beginning with (but excluding) the package `type`, and not beginning with the type of the object `o`. `type` must, however, be the name of one of the parent types—that is, parent packages—of the type (or package) of object `o`. If not, an error will be raised.

`send-super` is used primarily inside methods to invoke a method of the same name but defined at a higher level in the type hierarchy.

In the following example, the `describe` method is defined for objects of type `person`. It is redefined for objects of the sub-type `#:person:researcher`, so that it invokes the `describe` method at the level immediately above `#:person:researcher` in the type hierarchy, and then does some processing specific to objects of the `#:person:researcher` type.

```
? (defun #:person:describe (p)
?  (print "A person " (#:person:age p) " years old, who weighs "
?  (#:person:weight p) " pounds"))
= #:person:describe   ?
? (defun #:person:researcher:describe (r)
?  (send-super '#:person:researcher 'describe r)
?  (print "who has written "
?  (length (#:person:researcher:articles r))
?  " articles"))
= #:person:researcher:describe   ?
? (send 'describe (#:person:researcher:make))
A person () years old, who weighs 123 pounds
who has written 0 articles
```

```
= articles
```

(csend fndef symb o p_1 ... p_n) [*function with a variable number of arguments*]

This function is similar to the `send` function. If the search for the method fails, instead of rising an error, `csend` calls the `fndef` function with all the arguments of `csend` except the first.

This function is particularly useful for redefining the search for methods. One can define methods by default in a special package, for instance, or implement another multiple inheritance.

In LE-LISP, `csend` could be defined in the following manner:

```
(defun csend (default s o . largs)
    (let ((fun (getfn (type-of o) s '||)))
         (if fun
             (apply fun o largs)
             (apply default s o largs))))
```

Use the `*` package as a default:

```
? (defun default (m . para)
?  (apply (getfn '* m '||) para))
= default   ?
? (defun #:*:describe (o)
?  (print "An unknown object"))
= #:*:describe   ?
? (csend 'default 'describe 12)
An unknown object
= An unknown object   ? (csend 'default 'describe (#:person:make))
A person () years old, who weighs 123 pounds
=  pounds
```

(send2 symb o1 o2 p_1 ... p_n) [*function with a variable number of arguments*]

This function searches for and invokes the method `symb` associated with the product type of the objects `o1` and `o2`. Searching is done by the `getfn2` function (see its description). The function, if it is found, is invoked with `o1`, `o2` and $p_1 \ldots p_n$ as arguments.

Here are some definitions of methods for product types:

```
? (defun #:(a . a):foo (o1 o2)
?  'aa)
= #:(a . a):foo   ?
? (defun #:(a . #:a:b):foo (o1 o2)
?  'ab)
= #:(a . #:a:b):foo   ?
? (defun #:(#:a:b . a):foo (o1 o2)
?  'ba)
= #:(#:a:b . a):foo   ?
```

```
? (defun #:(a . #:a:c):foo (o1 o2)
?  'ac)
= #:(a . #:a:c):foo
```

We can invoke these methods. The objects here are tagged pairs:

```
? (send2 'foo '#(a) '#(a))
= aa    ? (send2 'foo '#(a) '#(#:a:b))
= ab    ? (send2 'foo '#(#:a:b) '#(a))
= ba    ? (send2 'foo '#(#:a:b) '#(#:a:b))
= ba    ? (send2 'foo '#(#:a:c) '#(#:a:b))
= ab    ? (send2 'foo '#(#:a:b) '#(#:a:c))
= ba    ? (send2 'foo '#(#:a:c) '#(#:a:c))
= ac
```

5.3.3 Predefined methods

LE-LISP uses the `send` function internally to print and evaluate LISP objects.

Commands to print LISP objects that are implemented as typed vectors, typed character strings, and tagged pairs cause a `prin` message to be sent to these objects. This message should cause the object to be printed onto the output channel, without any end-of-line character. If no print method exists for the object, the default print format is used.

Print methods are used by all the printing functions such as `print`, `prin`, `prinflush` and `explode`. They will be used, for instance, when the parameters of a traced function, entire functions, or error messages are printed.

The following warning applies in the last of these cases. If the print method raises an error, it is very probable the error procedure reuses the same print method and raises ... the same error! You therefore run the risk of getting the famous looping 'error in the error handler' error. It is therefore suggested that the print methods for an object be implemented under a name other than `prin`. Name them `prin` only when they seem to have been completely debugged.

The evaluation methods are used for typed vectors. The evaluation of a typed vector causes the `eval` message to be sent to the vector. If no `eval` method exists for the vector, the evaluation of the vector returns the vector itself as its value. Otherwise, if an `eval` method exists, it is the result of the vector's `eval` method that is returned as a value.

In the following example, the objects of `person` type will be printed as `#P(<age> <weight>)`. The `#`-macro provides a means of rereading this form of output:

```
? (defun #:person:prin (p)
?  (prin "#m(" (#:person:age p) " " (#:person:weight p) ")"))
= #:person:prin   ?
? (new 'person)
= #m(() 123)   ?
? (defsharp |m| ()
?  (let ((age-weight (read))
?  (person (new 'person)))
```

```
?  (#:person:age person (car age-weight))
?  (#:person:weight person (cadr age-weight))
?  (list person)))
= h   ?
? #m(25 123)
= #m(25 123)
```

5.4 Abbreviation functions

This section describes a set of functions that allow the programmer to use abbreviations while addressing the LE-LISP reader. Once instantiated, these abbreviations are used by the LE-LISP reader if the `#:system:print-with-abbrev-flag` flag is set.

In the worst of cases, this feature can raise the following errors:

`errsxtacc`, which has the following default screen display:

```
** <fn> : bad {} abbreviation : <e>
```

`errsxtclosingacc`, which has the following default screen display:

```
** <fn> : } not within {} : <e>
```

`errnotanabbrev`, which has the following default screen display:

```
** <fn> : not an abbreviation : <e>
```

abbrev [*feature*]

This flag indicates that the abbreviation package has been loaded into the system.

(put-abbrev sym1 sym2) [*function with two arguments*]

(defabbrev sym1 sym2) [*macro*]

These two functions define the symbol `sym1` to be an abbreviation of the symbol `sym2`. After calling one of these functions, the occurrence of the symbol `sym1` between braces—the characters `{` and `}`—is exactly equivalent, for the LISP reader, to the occurrence of the symbol `sym2`. If a previous abbreviation existed, it is changed.

`defabbrev` is identical to `put-abbrev`, except that it does not evaluate its arguments.

Consider, for instance, the following definition.

```
? (defabbrev foo #:bar:gee)
= foo
```

```
{foo}          is read as  #:bar:gee
{foo}:muu      is read as  #:bar:gee:muu
{foo}:#[1 2]   is read as  #:bar:gee:#[1 2]
```

(get-abbrev symb) [*function with one argument*]

Returns the symbol that has `symb` as its abbreviation. If `symb` is not an abbreviation, it raises the `errnotanabbrev` error.

```
(get-abbrev 'foo)  ⟹   #:bar:gee
(get-abbrev 'bar)  ⟹     ** get-abbrev : not an abbreviation : bar
```

(abbrevp symb) [*function with one argument*]

Returns `t` if `symb` is an abbreviation. If not, it returns `()`.

```
(abbrevp 'foo)  ⟹   t
(abbrevp 'bar)  ⟹   ()
```

(has-an-abbrev symb) [*function with one argument*]

Returns the abbreviation of the symbol `symb`.

```
(has-an-abbrev '#:bar:gee)  ⟹   foo
```

(rem-abbrev symb) [*function with one argument*]

Removes the abbreviation associated with the symbol `symb`, and returns `symb` as its value.

```
(rem-abbrev 'foo)  ⟹   foo
```

{symb} [*macro character*]

The macro characters `{` and `}` facilitate the reading of abbreviations. A symbol between braces is read as though it were the symbol it abbreviates. If `symb` is not an abbreviation, then `symb` is read as `||`, or `()`. If `symb` is not a symbol, or if there are several LISP objects between a single pair of braces, the `errsxtacc` error is raised.

#:system:print-with-abbrev-flag [*variable*]

If this flag is set—that is, if it is true—the standard printer unabbreviates abbreviated symbols on output. By default, this flag has the value *true*.

Here is a definition of the print method for symbols with abbreviations:

```
(defun #:symbol:prin (symb)
    (if (and #:system:print-with-abbrev-flag
```

```
        (has-an-abbrev (packagecell symb)))
    (progn (let ((#:system:print-for-read ()))
                (pratom '|{|))
           (pratom (has-an-abbrev (packagecell symb)))
           (let ((#:system:print-for-read ()))
                (pratom '|}:|))
           (let ((#:system:print-package-flag ()))
                (pratom symb)))
    (pratom symb)))
```

5.5 MicroCeyx

This section describes the MicroCeyx object layer, which is a smaller version of the original Ceyx.

All the MicroCeyx structures are sub-types of a root class called `tclass`, which will be mentioned frequently in the rest of this chapter.

`microceyx` [*feature*]

This feature indicates whether or not MicroCeyx is loaded into the system.

5.5.1 Specific errors

All the MicroCeyx functions check the type and form of their arguments. They can raise one of the errors listed below.

```
(defvar errnotafield
          "argument not a field of a microceyx tclass")

(defvar errnotatclass
          "argument not a name of a microceyx tclass")

(defvar errtclassabbrev
          "abbrev of microceyx tclass already defined")

(defvar errrecordabbrev
          "abbrev of microceyx record already defined")

(defvar errbadfield
          "syntax error in field")

(defvar errrecordtoolong
          "16 fields maximum per record")

(defvar errnotarecordoratclass
```

```
"argument not a name of microceyx tclass or record")
```

5.5.2 Definition of structures

(deftclass `symbol field`$_1$ ... `field`$_n$**)** [*macro*]

This function is identical to the `defstruct` function, the only difference being that the type it defines is a sub-type of `tclass` ... if this was not already the case. Moreover, the symbol `symbol` of the root package is defined as being the abbreviation for `symbol` in its entirety.

The form

```
? (deftclass foo a (b 2) c)
= #:tclass:foo
```

corresponds to

```
(progn (let ((#:system:defstruct-all-access-flag ()))
            (defstruct #:tclass:foo a (b 2) c))
       (defabbrev foo #:tclass:foo))
```

Likewise, the form

```
? (deftclass {foo}:bar (d 4))
= {foo}:bar
```

corresponds to

```
(progn (let ((#:system:defstruct-all-access-flag ()))
            (defstruct #:tclass:foo:bar (d 4)))
       (defabbrev bar #:tclass:foo:bar))
```

(defrecord `symbol field`$_1$ ... `field`$_n$**)** [*macro*]

This function is similar to the `defstruct` function, except that the instances it creates are implemented in the form of balanced binary trees of lists cells. Moreover, the instances contain no type information. They only satisfy the `consp` and `listp` predicates.

`symbol` must be in the root package. There is no inheritance for the records. For consistency with the `deftclass` function, the abbreviation `symbol`—standing for `symbol` itself—is defined.

A record cannot have more than 16 fields, otherwise the `errrecordtoolong` error is raised.

```
? (defrecord recfoo a b)
= recfoo
```

(tclass-namep `symbol`**)** [*function with one argument*]

This predicate is true if its argument is the name of a class defined by the `deftclass` function.

```
(tclass-namep '#:tclass:foo)  ⟹  t
(tclass-namep '{bar})         ⟹  t
(tclass-namep 'foothebarre)   ⟹  ()
(tclass-namep '(foo bar))     ⟹  ()
```

(record-namep symbol) [*function with one argument*]

This predicate is true if its argument is the name of a record defined by `defrecord`.

```
(record-namep 'recfoo)   ⟹  t
(record-namep 'bar)      ⟹  ()
(record-namep '(1 2 3))  ⟹  ()
```

(field-list symbol) [*function with one argument*]

Returns the list of the field names of instances of the `tclass` referred to as `symbol`, which must be a valid `tclass` name, and not an abbreviation.

```
(field-list '{foo})             ⟹  (a b c)
(field-list '#:tclass:foo:bar)  ⟹  (a b c d)
```

5.5.3 Tclass and record instances

(defmake type fn (field$_1$ ... field$_n$)) [*macro*]

Defines the function `fn` to be an instance constructor function for the `tclass` or record referred to as `type`. The function `fn` takes n arguments, which are the values of the fields named `field`$_i$ of the instance that is created.

```
? (defmake {foo} make-foo (c a))
= make-foo   ? (setq x (make-foo 12 34))
= #:tclass:foo:#[34 2 12]   ? ({foo}:a x)
= 34   ? ({foo}:c x)
= 12
```

(omakeq type field$_1$ val$_1$ ... field$_n$ val$_n$) [*macro*]

Returns an instance of the `tclass` referred to as `type`. Within this instance, each `field`$_i$ has the value `val`$_i$. Neither `type` nor the `field`$_i$ are evaluated.

```
(omakeq {bar})      ⟹  #:tclass:foo:bar:#[() 2 () 4]
(omakeq {bar} a 3)  ⟹  #:tclass:foo:bar:#[3 2 () 4]
```

(ogetq tclass-or-record field obj) [*macro*]

(oputq `tclass-or-record field obj val`**)** [*macro*]

(omatchq `tclass obj`**)** [*macro*]

These functions are identical to the corresponding Ceyx functions.

```
(ogetq {foo} b (omakeq {foo}))              ⟹   2
(ogetq {recfoo} b (omakeq {recfoo} b 48))   ⟹   48
(omatchq {foo} (omakeq {bar}))              ⟹   t
```

5.5.4 Methods and message sending

(demethod `{tclass}:name (obj` p_1 `...` p_n`) (`f_1 `...` f_n`) . body`**)** [*macro*]

f_1 ... f_n are fields of `tclass`.

```
(demethod {foo}:gee (obj x y z) (a b)
    <body>)
```

is equivalent to

```
(defun {foo}:gee (obj x y z)
    (let ((a ({foo}:a obj))
          (b ({foo}:b obj)))
         <body>))
```

(send `message objet` par_1 `...` par_n**)** [*function with a variable number of arguments*]

(sendq `message object` par_1 `...` par_n**)** [*macro*]

This function is similar to the standard `send` function, except that in MicroCeyx it searches for methods up to but excluding the root package. In the case of no match, it searches in the `*` package, and then in the `||` package.

`sendq` is like `send`, except that it does not evaluate its first argument.

In LE-LISP, `sendq` could be defined in the following manner:

```
(defmacro sendq (message . rest)
   `(send ',message ,@rest))
```

The search for methods carried out by the `send` function is implemented by redefining the `send-error` function in the following manner:

```
(defun send-error (sem argslist)
    (let ((fun (getfn '* sem)))
         (if fun
```

```
        (apply fun argslist)
        (error 'send 'errudm (cons sem argslist)))))
```

(sendf message par_1 ... par_n) [*macro*]

(sendfq message par_1 ... par_n) [*macro*]

Generates a one-argument function that transmits to its argument the message `message` with the parameters par_i. This function is especially useful for mapping the `send` function.

`sendfq` is like `sendf`, except that it does not evaluate its first argument.

```
? (defun {foo}:goo (o a)
?  (print "{foo}:goo " ({foo}:b o) " " a))
= {foo}:goo   ? (mapc (sendf 'goo 12)
?  (list (omakeq {foo} b 14) (omakeq {foo} b 20)))
{foo}:goo 14 12
{foo}:goo 20 12
= ()
```

Table of contents

5 Object-oriented programming **5-1**

5.1 Structures 5-1

5.1.1 Structure definition 5-2

5.1.2 Instance creation 5-2

5.1.3 Access to fields 5-3

5.1.4 Type tests 5-3

5.1.5 Implementation of structures 5-4

5.2 Le-Lisp typology 5-4

5.3 Object-oriented programming 5-7

5.3.1 Searching for methods 5-7

5.3.2 Invoking methods 5-9

5.3.3 Predefined methods 5-13

5.4 Abbreviation functions 5-14

5.5 MicroCeyx 5-16

5.5.1 Specific errors 5-16

5.5.2 Definition of structures 5-17

5.5.3 Tclass and record instances 5-18

5.5.4 Methods and message sending 5-19

Function Index

`defstruct` [*feature*] 5-1

`(defstruct struct field`$_1$ `...` `field`$_n$`)` [*macro*] 5-2

`#:system:defstruct-all-access-flag` [*variable*] 5-2

`(new struct)` [*function with one argument*] 5-2

`(#:struct:make)` [*function with no arguments*] 5-2

`(#:struct:field o e)` [*function with one or two arguments*] 5-3

`(structurep o)` [*function with one argument*] 5-3

`(type-of s)` [*function with one argument*] 5-5

`(subtypep type1 type2)` [*function with two arguments*] 5-6

`(typep s type)` [*function with two arguments*] 5-6

`(getfn1 pkgc fn)` [*function with two arguments*] 5-7

`(getfn pkgc1 fn pkgc2)` [*function with two or three arguments*] 5-8

`(getfn2 pkgc1 pkgc2 fn)` [*function with three arguments*] 5-8

`(send symb o p`$_1$ `...` `p`$_n$`)` [*function with a variable number of arguments*] 5-9

`(send-error symb rest)` [*function with two arguments*] 5-9

`(send-super type symb o p`$_1$ `...` `p`$_n$`)` [*function with a variable number of arguments*] 5-10

`(csend fndef symb o p`$_1$ `...` `p`$_n$`)` [*function with a variable number of arguments*] 5-12

`(send2 symb o1 o2 p`$_1$ `...` `p`$_n$`)` [*function with a variable number of arguments*] 5-12

`abbrev` [*feature*] 5-14

`(put-abbrev sym1 sym2)` [*function with two arguments*] 5-14

`(defabbrev sym1 sym2)` [*macro*] 5-14

`(get-abbrev symb)` [*function with one argument*] 5-15

`(abbrevp symb)` [*function with one argument*] 5-15

`(has-an-abbrev symb)` [*function with one argument*] 5-15

`(rem-abbrev symb)` [*function with one argument*] 5-15

`{symb}` [*macro character*] 5-15

`#:system:print-with-abbrev-flag` [*variable*] 5-15
`microceyx` [*feature*] 5-16
`(deftclass symbol field`$_1$ `...` `field`$_n$`)` [*macro*] 5-17
`(defrecord symbol field`$_1$ `...` `field`$_n$`)` [*macro*] 5-17
`(tclass-namep symbol)` [*function with one argument*] 5-17
`(record-namep symbol)` [*function with one argument*] 5-18
`(field-list symbol)` [*function with one argument*] 5-18
`(defmake type fn (field`$_1$ `...` `field`$_n$`))` [*macro*] 5-18
`(omakeq type field`$_1$ `val`$_1$ `...` `field`$_n$ `val`$_n$`)` [*macro*] 5-18
`(ogetq tclass-or-record field obj)` [*macro*] 5-18
`(oputq tclass-or-record field obj val)` [*macro*] 5-19
`(omatchq tclass obj)` [*macro*] 5-19
`(demethod {tclass}:name (obj p`$_1$ `...` `p`$_n$`) (f`$_1$ `...` `f`$_n$`) . body)` [*macro*] 5-19
`(send message objet par`$_1$ `...` `par`$_n$`)` [*function with a variable number of arguments*] 5-19
`(sendq message object par`$_1$ `...` `par`$_n$`)` [*macro*] 5-19
`(sendf message par`$_1$ `...` `par`$_n$`)` [*macro*] 5-20
`(sendfq message par`$_1$ `...` `par`$_n$`)` [*macro*] 5-20

Chapter 5

Object-oriented programming

This chapter first describes functions that provide ways to create structured objects in LISP. These objects are analogous to Pascal *records*. In addition, LISP allows you to create a hierarchy of structure types.

The second section is a description of the basic LISP type hierarchy and the means provided for extending it: typed vectors, typed strings and tagged pairs. Since they are implemented as typed vectors, structures provide the most immediate way to extend the LE-LISP type scheme.

The third section explains how to use the primitive `oblist` search functions and the typing functions to implement an object-oriented language of the SmallTalk variety.

The fourth section presents an abbreviation facility for describing typed objects.

The final section describes a minimal kernel on which object-oriented extensions can be built. This kernel, called **MicroCeyx**, benefits from experience acquired in the creation at INRIA of earlier object-oriented products called Ceyx and Alcyone.

5.1 Structures

The `defstruct` primitive provides the means to define new types of structured objects (structures) analogous to Pascal records. The `new` function facilitates the creation of *instances* of these types of objects.

Structured objects are made up of a number of named fields. Each field can contain any kind of LISP object. These fields are not typed. You can examine and modify the contents of the fields of a structured object by means of field-access functions, which are defined along with structures.

You can define a structure as a sub-structure of a previously defined structure. In this case, in addition to the fields explicitly declared within its definition, the sub-structure will inherit all the fields of the original parent structure.

`defstruct` [*feature*]

This feature indicates that functions on structures are loaded into the system.

5.1.1 Structure definition

(defstruct `struct field`$_1$ ... `field`$_n$**)** *[macro]*

Defines a structure named `struct`, which is a symbol, containing the fields `field`$_1$ ... `field`$_n$. `struct` becomes a new type of LISP object. (See the `type-of` function.)

> `field`$_1$ ... `field`$_n$ are the names of the structure's fields. They are either symbols or two-element lists. In the second case, the first element in the list is the field name, and the second element is a LISP form that is evaluated to assign an initial value to the field when instances are created. This evaluation takes place each time an instance of the structure is created.
>
> When a structure is defined, read-access and write-access functions, as well as a specific instance creation function, are defined simultaneously. These functions are described further on in this document.
>
> If `struct` is of the form `#:super-structure:structure`, where `super-structure` is the name of a structure previously defined in the system, `struct` will have, in addition to the fields `field`$_1$ ... `field`$_n$, all the fields of `super-structure`. In this case, the type `#:super-structure:structure` is a sub-type of the `super-structure` type.

#:system:defstruct-all-access-flag *[variable]*

> The `defstruct` macro examines this flag to determine whether access functions defined for inherited fields should be made available for use with the structure currently being defined. By default, this flag is *true*. In other words, it has the value `t`. If it is false, then the only access functions defined and available for use with this structure will be those for fields that are supplied explicitly as arguments to the relevant `defstruct` macro.

5.1.2 Instance creation

(new `struct`**)** *[function with one argument]*

(#:struct:make) *[function with no arguments]*

These two functions create a new instance of the `struct` structure, and return it as their value. The `#:struct:make` function is the creation function—*constructor*—for the `struct` structure. It is automatically defined by the `defstruct` macro. Objects created with this constructor will have type `struct`. (See the `type-of` function.)

The fields of the object so created are (optionally) initialized with the initialization forms included in the structure definition. These forms are evaluated when the object is created. The order of evaluation of the initialization forms is not fully determined. Fields for which no initialization forms are provided are initialized with the value `()`.

If the argument to the `new` function is not the name of a structure already defined in the system, the `errstc` error is raised, and this screen is displayed:

```
** <fn> : not a structure : <s>
```

where `fn` is the name of the function that raised the error (in this case, `new`), and `s` is the bad argument.

In LE-LISP, `new` could be defined in the following manner:

```
(defun new (type)
    (if (structurep type)
        (apply (symbol type 'make) ())
        (error 'new 'errstc type)))
```

5.1.3 Access to fields

(#:struct:field o e) *[function with one or two arguments]*

`struct` is the name of a structure, and `field` is the name of one of the structure's fields. The `#:struct:field` functions are the access functions to the fields of instances of the structure `struct`, and they are automatically defined by the `defstruct` function.

If the object `o` is not an instance of the `struct` structure (see the `typep` function), the `errstc` error is raised. Given a single argument, an access function returns the value of the field `field` of the object `o`. If a second argument is given, its value is stored in the field `field` of the object `o` and is also returned as the value of the access function.

Suppose that the LE-LISP compiler is active when the structure is defined. Consequently, the form `(featurep 'compiler)` is true. In this case, field-access functions are defined to be compiler macros. (See the `defmacro-open` function.) Compilation of calls to these functions produces extremely efficient code. After being compiled, access functions will not do type checks on their first arguments.

5.1.4 Type tests

(structurep o) *[function with one argument]*

This predicate is true if the object `o` is an instance of a defined structure. (See also the `typep` function.)

Here, for example, is the definition of a structure, `person`, with two fields: `age` and `weight`. The second field is initialized with the value `123`:

```
? (defstruct person
?  age
?  (weight 123))
= person
```

Creation of an instance:

```
? (setq person1 (#:person:make))
= #:person:#[() 123]
```

Assign the value `36` to the `age` field of the `person1` object:

```
? (#:person:age person1 36)
= 36
```

Extract the `weight` field:

```
? (#:person:weight person1)
= 123
```

Use the `structurep` function:

```
? (structurep person1)
= t
```

Next, we look at an example of a sub-structure. Extend the `person` structure into a `researcher` structure:

```
? (defstruct #:person:researcher
?  project
?  articles)
= #:person:researcher
```

Creation of a `researcher` and automatic initialization of a field:

```
? (setq r1 (#:person:researcher:make))
= #:person:researcher:#[() 123 () ()]    ? (#:person:researcher:articles r1
'("conf-proc" "thesis"))
= (conf-proc thesis)
```

The `researcher` inherits the fields of the `person` structure:

```
? (#:person:weight r1)
= 123    ? (#:person:age r1 26)
= 26    ? (#:person:researcher:weight r1)
= 123    ? (#:person:researcher:age r1)
= 26
```

5.1.5 Implementation of structures

Instances of a structure are implemented as typed vectors whose type is the name of the structure. The values of the object's fields are stored in vector cells.

5.2 Le-Lisp typology

LE-LISP object types are represented by symbols. The LISP package hierarchy provides a way to organize these types. The symbol `#:person:researcher` represents the `researcher` sub-type of the `person` type.

It is not strictly necessary to explicitly *create* new types. Simply invoking the symbol `#:foo:bar` results in a successful *mention* of the type `foo` and its sub-type `bar`. Any LISP symbol can therefore be interpreted as being a type.

In practice, the interesting types are those capable of being returned as values of the `type-of` function, either a predefined system type, or a *user type*: that is, the type of a typed vector, a typed string or a tagged pair.

The basic LISP types are predefined. In other words, they are 'already present' at system start-up. They all have the empty type (the type `||`), which is the universal type.

The basic LISP types are:

- `fix`: small integers
- `float`: floating-point numbers
- `symbol`: symbols
- `null`: the end-of-list marker, `()`
- `string`: character strings
- `vector`: vectors of pointers
- `cons`: list cells

(type-of s) [*function with one argument*]

This function determines the type of any LISP object. If the argument is a string or a vector, `type-of` returns the type of the object in question. In these cases, `type-of` is equivalent to `typestring` or `typevector`, respectively. If the argument is a tagged pair, `type-of` returns its `car`, provided that this is an atom or a list whose `car` is an atom. In all other cases—except the error condition associated with tagged pairs (see below)—`type-of` returns the LISP type of the object: `fix`, `float`, `cons`, etc.

If the `car` of a tagged pair argument is neither a symbol nor the symbol of a list, the result of `type-of` is undefined.

In LE-LISP, `type-of` could be defined in the following manner:

```
(defun type-of (s)
    (cond ((fixp s) 'fix)
          ((floatp s) 'float)
          ((null s) 'null)
          ((symbolp s) 'symbol)
          ((consp s)
           (if (tconsp s)
               (when (or (atom (car s)) (symbolp (caar s)))
                     (car s))
               'cons))
          ((vectorp s) (typevector s))
```

```
        ((stringp s) (typestring s))
        (t ())))
```

```
? (type-of 12)
= fix   ? (type-of 12.2)
= float   ? (type-of ())
= null   ? (type-of 'asd)
= symbol   ? (type-of "assasa")
= string   ? (type-of #[1 12 3])
= vector   ? (type-of #:foo:bar:"hello mamma")
= #:foo:bar   ? (type-of #:person:#[() 75])
= person   ? (type-of '(1))
= cons   ? (type-of '#(foo . a))
= foo
```

(subtypep type1 type2) *[function with two arguments]*

`type1` and `type2` are two symbols, representing LISP types. The predicate `subtypep` is true if `type1` is a sub-type of `type2`, and false otherwise.

In LE-LISP, `subtypep` could be defined in the following manner:

```
(defun subtypep (type1 type2)
    (or (eq type1 type2)
        (ifn (packagecell type1)
             (null type2)
             (subtypep (packagecell type1) type2))))
```

```
? (subtypep '#:person:researcher 'person)
= t   ? (subtypep 'symbol 'symbol)
= t   ? (subtypep '#:person:researcher '#:person:discoverer)
= ()   ? (subtypep 'fix '||)
= t
```

(typep s type) *[function with two arguments]*

This predicate is true if the type of the object `s` is a sub-type of `type`.

In LE-LISP, `typep` could be defined in the following manner:

```
(defun typep (s type)
    (subtypep (type-of s) type))
```

```
? (typep (#:person:researcher:make) 'person)
= t   ? (typep 'foo 'symbol)
= t   ? (typep (#:person:researcher:make) '#:person:discoverer)
= ()   ? (typep 12 '||)
= t
```

5.3 Object-oriented programming

The LE-LISP package hierarchy is used to encode the hierarchy of data types. In an object-oriented language of the SmallTalk variety—such as Ceyx or Alcyone, for example—one wants to be able to attach *methods* to object types.

In LE-LISP, these methods are LISP functions, defined in the packages to whose types they are attached. So, the function `#:person:describe` is considered to be the `describe` method attached to the `person` type.

The goal is for the sub-types of a given type to *inherit* the methods attached to this type. For example, the objects of type `#:person:researcher` should inherit the `describe` method of the `person` object. The predefined `getfn` function provides a way to do this method search, accounting for inheritance.

Sometimes it is desirable to have an object type inherit methods from several LISP types. This is multiple inheritance. The `getfn` function searches for a function in a list of packages, by first doing a depth-first search in the parent (superior) packages of those in the list.

Finally, you sometimes need methods associated with a mixture of several types of objects. For instance, to implement a generic arithmetic package, methods would have to be defined that allow you to add together different types of numbers such as `float`, `fix`, `bignum` and `rational`. In LE-LISP, the methods of the *product* of types `t1` and `t2` are located in the *product package* designated as `(t1 . t2)`. An addition method of the desired kind would be written as `#:(t1 . t2):add`. The predefined `getfn2` function provides a means of searching for product methods.

5.3.1 Searching for methods

The following functions are predefined search functions that operate on methods.

(getfn1 `pkgc fn`**)** [*function with two arguments*]

Searches for a function named `fn` in the `pkgc` package. If this function exists, it is returned as the value of `getfn1`. If not, the function returns `()`.

In LE-LISP, `getfn1` could be defined in the following manner:

```
(defun getfn1 (pkgc fn)
    (let ((nom (symbol pkgc fn)))
         (if (typefn nom)
             nom
             ())))
```

```
(getfn1 () 'car)       ⟹   car
(defun #:foo:bar ())   ⟹   #:foo:bar
(getfn1 'foo 'bar)     ⟹   #:foo:bar
(getfn1 'gee 'bar)     ⟹   ()
```

(getfn pkgc1 fn pkgc2) [*function with two or three arguments*]

Searches for a function named `fn` in the `pkgc1` package, or in one of the packages that is superior to `pkgc1` in the package hierarchy. If the optional `pkgc2` argument is given, it is excluded from the search. On the search up the package tree, if `pkgc2` is encountered, the search is terminated immediately. `pkgc2` is not searched. If the function is found, it is returned as the value of `getfn`. If not, `getfn` returns `()`.

If `pkgc1` is a list of packages, the search begins in the first package in the list and in all its parent packages, then in the second package in the list and in all its parent packages, and so on, until either the function is found or the list of packages names is exhausted. This *depth-search* also takes place if one of the parents of `pkgc1` is a list. This mechanism can be used to implement *multiple inheritance* of methods. In this case, the third argument, if present, provides a way to stop the search for each package in the list.

In LE-LISP, `getfn` could be defined in the following manner:

```
(defun getfn (pkgc symb . lastpkgc)
    (let ((nom (symbol pkgc symb)))
         (cond ((typefn nom) nom)
               ((null pkgc) ())
               ((and (consp lastpkgc)
                     (eq (packagecell pkgc)
                         (car lastpkgc)))
                ())
               (t (getfn (packagecell pkgc)
                         symb lastpkgc)))))
```

Let us define some functions:

```
(defun foo ())                    ⟹   foo
(defun #:bar:foo ())              ⟹   #:bar:foo
(defun #:bar:gee:buz:foo ())      ⟹   #:bar:gee:buz:foo
(getfn '#:bar:gee:buz 'foo)       ⟹   #:bar:gee:buz:foo
(getfn '#:bar:gee 'foo)           ⟹   #:bar:foo
(getfn 'bar 'foo)                 ⟹   #:bar:foo
(getfn () 'foo)                   ⟹   foo
(getfn '#:potop:teraz 'foo)       ⟹   foo
(getfn '#:bar:gee 'foo ())        ⟹   #:bar:foo
(getfn 'bar 'foo ())              ⟹   #:bar:foo
(getfn 'gee 'foo ())              ⟹   ()
(getfn '#:bar:gee:buz 'foo 'bar)  ⟹   #:bar:gee:buz:foo
(getfn '#:bar:gee 'foo 'bar)      ⟹   ()
```

(getfn2 pkgc1 pkgc2 fn) [*function with three arguments*]

Allows you to search for the function `fn` in the product package `(pkgc1 . pkgc2)`, or in one of the product packages that is superior to this one. The product package hierarchy is built from the

package hierarchy by imposing a lexicographic ordering on the combinations of possible pairs. The climb up the product package hierarchy is always stopped before reaching the root package for one of the two members of the product.

The packages superior to the package `(#:a:b . #:c:d)` are, in order

```
(#:a:b . #:c:d) < (#:a:b . c) < (a . #:c:d) < (a . c)
```

5.3.2 Invoking methods

The combination of type-determinating functions and method search functions provides a way to implement message transmission functions.

(send `symb o` $p_1 \ldots p_n$**)** [*function with a variable number of arguments*]

(send-error `symb rest`**)** [*function with two arguments*]

The `send` function searches for and invokes the method `symb` asssociated with the type of the object `o`. This method must be a LISP function that takes $n + 1$ arguments. It is called with the list of arguments formed by the object `o` and the n arguments $p_1 \ldots p_n$.

The intended method is the LISP function named `symb` in the package that is the type of the object `o`, or in one of its parent packages, up to and excluding the root package. If no such method is found, `send` calls the `send-error` function. By default, this function raises the `errudm` error and displays the following error screen:

```
** <fn> : undefined method : <l>
```

where `l` is the list of arguments supplied to `send`, and `fn` is the name of the function that raises the error: here, `send`.

In LE-LISP, `send` could be defined in the following manner:

```
(defun send (m o . larg)
    (let ((fn (getfn (type-of o) m '||)))
         (if fn
             (apply fn o larg)
             (funcall 'send-error m (cons o larg)))))
```

In LE-LISP, `send-error` could be defined in the following manner:

```
(defun send-error (m rest)
    (error 'send 'errudm (cons m rest)))
```

Here is a definition of the `hello` method for objects of type `fix`:

```
? (defun #:fix:hello (n)
?  (repeat n (print "hello")))
= #:fix:hello
```

Search for this method and invoke it:

```
? (send 'hello 3)
hello
hello
hello
= t
```

And here is a definition of the `hello` method for objects of the type `person`:

```
? (defun #:person:hello (pers)
?  (print "I am a person")
?  (print "I weigh " (#:person:weight pers) " pounds"))
= #:person:hello   ? (send 'hello (#:person:make))
I am a person
I weigh 123 pounds
=  pounds   ? (send 'hello (#:person:researcher:make))
I am a person
I weigh 123 pounds
=  pounds
```

(send-super type symb o p_1 ... p_n) [*function with a variable number of arguments*]

The function behaves very much like the `send` function, except that the method is sought beginning with (but excluding) the package `type`, and not beginning with the type of the object `o`. `type` must, however, be the name of one of the parent types—that is, parent packages—of the type (or package) of object `o`. If not, an error will be raised.

`send-super` is used primarily inside methods to invoke a method of the same name but defined at a higher level in the type hierarchy.

In the following example, the `describe` method is defined for objects of type `person`. It is redefined for objects of the sub-type `#:person:researcher`, so that it invokes the `describe` method at the level immediately above `#:person:researcher` in the type hierarchy, and then does some processing specific to objects of the `#:person:researcher` type.

```
? (defun #:person:describe (p)
?  (print "A person " (#:person:age p) " years old, who weighs "
?  (#:person:weight p) " pounds"))
= #:person:describe   ?
? (defun #:person:researcher:describe (r)
?  (send-super '#:person:researcher 'describe r)
?  (print "who has written "
?  (length (#:person:researcher:articles r))
?  " articles"))
= #:person:researcher:describe   ?
? (send 'describe (#:person:researcher:make))
A person () years old, who weighs 123 pounds
who has written 0 articles
```

```
= articles
```

(csend fndef symb o p_1 ... p_n) [*function with a variable number of arguments*]

This function is similar to the `send` function. If the search for the method fails, instead of rising an error, `csend` calls the `fndef` function with all the arguments of `csend` except the first.

This function is particularly useful for redefining the search for methods. One can define methods by default in a special package, for instance, or implement another multiple inheritance.

In LE-LISP, `csend` could be defined in the following manner:

```
(defun csend (default s o . largs)
    (let ((fun (getfn (type-of o) s '||)))
         (if fun
             (apply fun o largs)
             (apply default s o largs))))
```

Use the `*` package as a default:

```
? (defun default (m . para)
?  (apply (getfn '* m '||) para))
= default   ?
? (defun #:*:describe (o)
?  (print "An unknown object"))
= #:*:describe   ?
? (csend 'default 'describe 12)
An unknown object
= An unknown object   ? (csend 'default 'describe (#:person:make))
A person () years old, who weighs 123 pounds
=  pounds
```

(send2 symb o1 o2 p_1 ... p_n) [*function with a variable number of arguments*]

This function searches for and invokes the method `symb` associated with the product type of the objects `o1` and `o2`. Searching is done by the `getfn2` function (see its description). The function, if it is found, is invoked with `o1`, `o2` and $p_1 \ldots p_n$ as arguments.

Here are some definitions of methods for product types:

```
? (defun #:(a . a):foo (o1 o2)
?  'aa)
= #:(a . a):foo   ?
? (defun #:(a . #:a:b):foo (o1 o2)
?  'ab)
= #:(a . #:a:b):foo   ?
? (defun #:(#:a:b . a):foo (o1 o2)
```

```
?  'ba)
= #:(#:a:b . a):foo   ?
? (defun #:(a . #:a:c):foo (o1 o2)
?  'ac)
= #:(a . #:a:c):foo
```

We can invoke these methods. The objects here are tagged pairs:

```
? (send2 'foo '#(a) '#(a))
= aa   ? (send2 'foo '#(a) '#(#:a:b))
= ab   ? (send2 'foo '#(#:a:b) '#(a))
= ba   ? (send2 'foo '#(#:a:b) '#(#:a:b))
= ba   ? (send2 'foo '#(#:a:c) '#(#:a:b))
= ab   ? (send2 'foo '#(#:a:b) '#(#:a:c))
= ba   ? (send2 'foo '#(#:a:c) '#(#:a:c))
= ac
```

5.3.3 Predefined methods

LE-LISP uses the `send` function internally to print and evaluate LISP objects.

Commands to print LISP objects that are implemented as typed vectors, typed character strings, and tagged pairs cause a `prin` message to be sent to these objects. This message should cause the object to be printed onto the output channel, without any end-of-line character. If no print method exists for the object, the default print format is used.

Print methods are used by all the printing functions such as `print`, `prin`, `prinflush` and `explode`. They will be used, for instance, when the parameters of a traced function, entire functions, or error messages are printed.

The following warning applies in the last of these cases. If the print method raises an error, it is very probable the error procedure reuses the same print method and raises ... the same error! You therefore run the risk of getting the famous looping 'error in the error handler' error. It is therefore suggested that the print methods for an object be implemented under a name other than `prin`. Name them `prin` only when they seem to have been completely debugged.

The evaluation methods are used for typed vectors. The evaluation of a typed vector causes the `eval` message to be sent to the vector. If no `eval` method exists for the vector, the evaluation of the vector returns the vector itself as its value. Otherwise, if an `eval` method exists, it is the result of the vector's `eval` method that is returned as a value.

In the following example, the objects of `person` type will be printed as `#P(<age> <weight>)`. The `#`-macro provides a means of rereading this form of output:

```
? (defun #:person:prin (p)
?  (prin "#m(" (#:person:age p) " " (#:person:weight p) ")"))
= #:person:prin   ?
? (new 'person)
= #m(() 123)   ?
? (defsharp |m| ()
```

```
? (let ((age-weight (read))
? (person (new 'person)))
? (#:person:age person (car age-weight))
? (#:person:weight person (cadr age-weight))
? (list person)))
= h   ?
? #m(25 123)
= #m(25 123)
```

5.4 Abbreviation functions

This section describes a set of functions that allow the programmer to use abbreviations while addressing the LE-LISP reader. Once instantiated, these abbreviations are used by the LE-LISP reader if the `#:system:print-with-abbrev-flag` flag is set.

In the worst of cases, this feature can raise the following errors:

`errsxtacc`, which has the following default screen display:

```
** <fn> : bad {} abbreviation : <e>
```

`errsxtclosingacc`, which has the following default screen display:

```
** <fn> : } not within {} : <e>
```

`errnotanabbrev`, which has the following default screen display:

```
** <fn> : not an abbreviation : <e>
```

abbrev [*feature*]

This flag indicates that the abbreviation package has been loaded into the system.

(put-abbrev sym1 sym2) [*function with two arguments*]

(defabbrev sym1 sym2) [*macro*]

These two functions define the symbol `sym1` to be an abbreviation of the symbol `sym2`. After calling one of these functions, the occurrence of the symbol `sym1` between braces—the characters `{` and `}`—is exactly equivalent, for the LISP reader, to the occurrence of the symbol `sym2`. If a previous abbreviation existed, it is changed.

`defabbrev` is identical to `put-abbrev`, except that it does not evaluate its arguments.

Consider, for instance, the following definition.

```
? (defabbrev foo #:bar:gee)
= foo
```

```
{foo}          is read as  #:bar:gee
{foo}:muu      is read as  #:bar:gee:muu
{foo}:#[1 2]   is read as  #:bar:gee:#[1 2]
```

(get-abbrev symb) [*function with one argument*]

Returns the symbol that has `symb` as its abbreviation. If `symb` is not an abbreviation, it raises the `errnotanabbrev` error.

```
(get-abbrev 'foo)  ⟹   #:bar:gee
(get-abbrev 'bar)  ⟹      ** get-abbrev : not an abbreviation : bar
```

(abbrevp symb) [*function with one argument*]

Returns `t` if `symb` is an abbreviation. If not, it returns `()`.

```
(abbrevp 'foo)  ⟹   t
(abbrevp 'bar)  ⟹   ()
```

(has-an-abbrev symb) [*function with one argument*]

Returns the abbreviation of the symbol `symb`.

```
(has-an-abbrev '#:bar:gee)  ⟹   foo
```

(rem-abbrev symb) [*function with one argument*]

Removes the abbreviation associated with the symbol `symb`, and returns `symb` as its value.

```
(rem-abbrev 'foo)  ⟹   foo
```

{symb} [*macro character*]

The macro characters `{` and `}` facilitate the reading of abbreviations. A symbol between braces is read as though it were the symbol it abbreviates. If `symb` is not an abbreviation, then `symb` is read as `||`, or `()`. If `symb` is not a symbol, or if there are several LISP objects between a single pair of braces, the `errsxtacc` error is raised.

#:system:print-with-abbrev-flag [*variable*]

If this flag is set—that is, if it is true—the standard printer unabbreviates abbreviated symbols on output. By default, this flag has the value *true*.

Here is a definition of the print method for symbols with abbreviations:

```
(defun #:symbol:prin (symb)
    (if (and #:system:print-with-abbrev-flag
```

```
        (has-an-abbrev (packagecell symb)))
    (progn (let ((#:system:print-for-read ()))
                (pratom '|{|))
           (pratom (has-an-abbrev (packagecell symb)))
           (let ((#:system:print-for-read ()))
                (pratom '|}:|))
           (let ((#:system:print-package-flag ()))
                (pratom symb)))
    (pratom symb)))
```

5.5 MicroCeyx

This section describes the MicroCeyx object layer, which is a smaller version of the original Ceyx.

All the MicroCeyx structures are sub-types of a root class called `tclass`, which will be mentioned frequently in the rest of this chapter.

`microceyx` [*feature*]

This feature indicates whether or not MicroCeyx is loaded into the system.

5.5.1 Specific errors

All the MicroCeyx functions check the type and form of their arguments. They can raise one of the errors listed below.

```
(defvar errnotafield
          "argument not a field of a microceyx tclass")

(defvar errnotatclass
          "argument not a name of a microceyx tclass")

(defvar errtclassabbrev
          "abbrev of microceyx tclass already defined")

(defvar errrecordabbrev
          "abbrev of microceyx record already defined")

(defvar errbadfield
          "syntax error in field")

(defvar errrecordtoolong
          "16 fields maximum per record")

(defvar errnotarecordoratclass
```

```
"argument not a name of microceyx tclass or record")
```

5.5.2 Definition of structures

(deftclass `symbol field`$_1$ ... `field`$_n$**)** [*macro*]

This function is identical to the `defstruct` function, the only difference being that the type it defines is a sub-type of `tclass` ... if this was not already the case. Moreover, the symbol `symbol` of the root package is defined as being the abbreviation for `symbol` in its entirety.

The form

```
? (deftclass foo a (b 2) c)
= #:tclass:foo
```

corresponds to

```
(progn (let ((#:system:defstruct-all-access-flag ()))
            (defstruct #:tclass:foo a (b 2) c))
       (defabbrev foo #:tclass:foo))
```

Likewise, the form

```
? (deftclass {foo}:bar (d 4))
= {foo}:bar
```

corresponds to

```
(progn (let ((#:system:defstruct-all-access-flag ()))
            (defstruct #:tclass:foo:bar (d 4)))
       (defabbrev bar #:tclass:foo:bar))
```

(defrecord `symbol field`$_1$ ... `field`$_n$**)** [*macro*]

This function is similar to the `defstruct` function, except that the instances it creates are implemented in the form of balanced binary trees of lists cells. Moreover, the instances contain no type information. They only satisfy the `consp` and `listp` predicates.

`symbol` must be in the root package. There is no inheritance for the records. For consistency with the `deftclass` function, the abbreviation `symbol`—standing for `symbol` itself—is defined.

A record cannot have more than 16 fields, otherwise the `errrecordtoolong` error is raised.

```
? (defrecord recfoo a b)
= recfoo
```

(tclass-namep `symbol`**)** [*function with one argument*]

This predicate is true if its argument is the name of a class defined by the `deftclass` function.

```
(tclass-namep '#:tclass:foo)  ⟹  t
(tclass-namep '{bar})         ⟹  t
(tclass-namep 'foothebarre)   ⟹  ()
(tclass-namep '(foo bar))     ⟹  ()
```

(record-namep symbol) [*function with one argument*]

This predicate is true if its argument is the name of a record defined by `defrecord`.

```
(record-namep 'recfoo)   ⟹  t
(record-namep 'bar)      ⟹  ()
(record-namep '(1 2 3))  ⟹  ()
```

(field-list symbol) [*function with one argument*]

Returns the list of the field names of instances of the `tclass` referred to as `symbol`, which must be a valid `tclass` name, and not an abbreviation.

```
(field-list '{foo})             ⟹  (a b c)
(field-list '#:tclass:foo:bar)  ⟹  (a b c d)
```

5.5.3 Tclass and record instances

(defmake type fn (field$_1$ ... field$_n$)) [*macro*]

Defines the function `fn` to be an instance constructor function for the `tclass` or record referred to as `type`. The function `fn` takes n arguments, which are the values of the fields named `field`$_i$ of the instance that is created.

```
? (defmake {foo} make-foo (c a))
= make-foo   ? (setq x (make-foo 12 34))
= #:tclass:foo:#[34 2 12]   ? ({foo}:a x)
= 34   ? ({foo}:c x)
= 12
```

(omakeq type field$_1$ val$_1$ ... field$_n$ val$_n$) [*macro*]

Returns an instance of the `tclass` referred to as `type`. Within this instance, each `field`$_i$ has the value `val`$_i$. Neither `type` nor the `field`$_i$ are evaluated.

```
(omakeq {bar})      ⟹  #:tclass:foo:bar:#[() 2 () 4]
(omakeq {bar} a 3)  ⟹  #:tclass:foo:bar:#[3 2 () 4]
```

(ogetq tclass-or-record field obj) [*macro*]

(oputq `tclass-or-record field obj val`**)** [*macro*]

(omatchq `tclass obj`**)** [*macro*]

These functions are identical to the corresponding Ceyx functions.

```
(ogetq {foo} b (omakeq {foo}))            ⟹   2
(ogetq {recfoo} b (omakeq {recfoo} b 48)) ⟹   48
(omatchq {foo} (omakeq {bar}))            ⟹   t
```

5.5.4 Methods and message sending

(demethod {`tclass`}`:name (obj` p_1 ... p_n`) (`f_1 ... f_n`) . body`**)** [*macro*]

f_1 ... f_n are fields of `tclass`.

```
(demethod {foo}:gee (obj x y z) (a b)
    <body>)
```

is equivalent to

```
(defun {foo}:gee (obj x y z)
    (let ((a ({foo}:a obj))
          (b ({foo}:b obj)))
         <body>))
```

(send `message objet par`$_1$ ... `par`$_n$**)** [*function with a variable number of arguments*]

(sendq `message object par`$_1$ ... `par`$_n$**)** [*macro*]

This function is similar to the standard `send` function, except that in MicroCeyx it searches for methods up to but excluding the root package. In the case of no match, it searches in the `*` package, and then in the `||` package.

`sendq` is like `send`, except that it does not evaluate its first argument.

In LE-LISP, `sendq` could be defined in the following manner:

```
(defmacro sendq (message . rest)
   `(send ',message ,@rest))
```

The search for methods carried out by the `send` function is implemented by redefining the `send-error` function in the following manner:

```
(defun send-error (sem argslist)
    (let ((fun (getfn '* sem)))
         (if fun
```

```
        (apply fun argslist)
        (error 'send 'errudm (cons sem argslist)))))
```

(sendf message par_1 ... par_n) [*macro*]

(sendfq message par_1 ... par_n) [*macro*]

Generates a one-argument function that transmits to its argument the message `message` with the parameters par_i. This function is especially useful for mapping the `send` function.

`sendfq` is like `sendf`, except that it does not evaluate its first argument.

```
? (defun {foo}:goo (o a)
?  (print "{foo}:goo " ({foo}:b o) " " a))
= {foo}:goo    ? (mapc (sendf 'goo 12)
?  (list (omakeq {foo} b 14) (omakeq {foo} b 20)))
{foo}:goo 14 12
{foo}:goo 20 12
= ()
```

Table of contents

5 Object-oriented programming **5-1**

5.1 Structures 5-1

5.1.1 Structure definition 5-2

5.1.2 Instance creation 5-2

5.1.3 Access to fields 5-3

5.1.4 Type tests 5-3

5.1.5 Implementation of structures 5-4

5.2 Le-Lisp typology 5-4

5.3 Object-oriented programming 5-7

5.3.1 Searching for methods 5-7

5.3.2 Invoking methods 5-9

5.3.3 Predefined methods 5-13

5.4 Abbreviation functions 5-14

5.5 MicroCeyx 5-16

5.5.1 Specific errors 5-16

5.5.2 Definition of structures 5-17

5.5.3 Tclass and record instances 5-18

5.5.4 Methods and message sending 5-19

Function Index

`defstruct` [*feature*] 5-1

`(defstruct struct field`$_1$ `...` `field`$_n$`)` [*macro*] 5-2

`#:system:defstruct-all-access-flag` [*variable*] 5-2

`(new struct)` [*function with one argument*] 5-2

`(#:struct:make)` [*function with no arguments*] 5-2

`(#:struct:field o e)` [*function with one or two arguments*] 5-3

`(structurep o)` [*function with one argument*] 5-3

`(type-of s)` [*function with one argument*] 5-5

`(subtypep type1 type2)` [*function with two arguments*] 5-6

`(typep s type)` [*function with two arguments*] 5-6

`(getfn1 pkgc fn)` [*function with two arguments*] 5-7

`(getfn pkgc1 fn pkgc2)` [*function with two or three arguments*] 5-8

`(getfn2 pkgc1 pkgc2 fn)` [*function with three arguments*] 5-8

`(send symb o p`$_1$ `...` `p`$_n$`)` [*function with a variable number of arguments*] 5-9

`(send-error symb rest)` [*function with two arguments*] 5-9

`(send-super type symb o p`$_1$ `...` `p`$_n$`)` [*function with a variable number of arguments*] 5-10

`(csend fndef symb o p`$_1$ `...` `p`$_n$`)` [*function with a variable number of arguments*] 5-12

`(send2 symb o1 o2 p`$_1$ `...` `p`$_n$`)` [*function with a variable number of arguments*] 5-12

`abbrev` [*feature*] 5-14

`(put-abbrev sym1 sym2)` [*function with two arguments*] 5-14

`(defabbrev sym1 sym2)` [*macro*] 5-14

`(get-abbrev symb)` [*function with one argument*] 5-15

`(abbrevp symb)` [*function with one argument*] 5-15

`(has-an-abbrev symb)` [*function with one argument*] 5-15

`(rem-abbrev symb)` [*function with one argument*] 5-15

`{symb}` [*macro character*] 5-15

`#:system:print-with-abbrev-flag` [*variable*] 5-15
`microceyx` [*feature*] 5-16
`(deftclass symbol field`$_1$ `...` `field`$_n$`)` [*macro*] 5-17
`(defrecord symbol field`$_1$ `...` `field`$_n$`)` [*macro*] 5-17
`(tclass-namep symbol)` [*function with one argument*] 5-17
`(record-namep symbol)` [*function with one argument*] 5-18
`(field-list symbol)` [*function with one argument*] 5-18
`(defmake type fn (field`$_1$ `...` `field`$_n$`))` [*macro*] 5-18
`(omakeq type field`$_1$ `val`$_1$ `...` `field`$_n$ `val`$_n$`)` [*macro*] 5-18
`(ogetq tclass-or-record field obj)` [*macro*] 5-18
`(oputq tclass-or-record field obj val)` [*macro*] 5-19
`(omatchq tclass obj)` [*macro*] 5-19
`(demethod {tclass}:name (obj p`$_1$ `...` `p`$_n$`) (f`$_1$ `...` `f`$_n$`) . body)` [*macro*] 5-19
`(send message objet par`$_1$ `...` `par`$_n$`)` [*function with a variable number of arguments*] 5-19
`(sendq message object par`$_1$ `...` `par`$_n$`)` [*macro*] 5-19
`(sendf message par`$_1$ `...` `par`$_n$`)` [*macro*] 5-20
`(sendfq message par`$_1$ `...` `par`$_n$`)` [*macro*] 5-20

Chapter 6

Input and output

In LE-LISP, basic input/output—referred to, throughout this chapter, as I/O—operates on sequential streams at either a character level, a line level or an S-expression level.

6.1 Introduction

This introduction presents the principal ideas used in the description of I/O functions.

6.1.1 Characters

Characters can be represented in two ways:

- In the form of an integer number, which represents the internal character code of the character, often written `cn`.
- In the form of an atom (symbol, string or integer), `ch`. If the atom is a symbol or a string, its first character is used. If the atom is a number, the first digit of the representation of the value of the number, in the current output base (see the function `obase`), is used.

LE-LISP uses either of these two representations. Functions that use the first form generally have the suffix `cn`, and those which use the second form have the suffix `ch`. The internal character code used depends on the host system. In general, the ASCII code is used. Internal character codes range between 0–255 (inclusive).

It is always more efficient and more space-conserving to use the internal character code representation. Morever, the use of the `#`-macros `#/`, `#^` and `#\` makes this representation almost as readable as the other, single-character atom representation.

6.1.2 Character strings

Character strings can be represented in three different forms:

- Strings of characters, represented in the descriptions as `string`.

- Lists of internal character codes, represented in the descriptions as `lcn`.
- Lists of single-character atoms, represented in the descriptions as `lch`.

Functions that use the first of these representations have the suffix `string`. They are generally more efficient in performance and in memory use than the others.

6.1.3 Lines

Usually, read and write functions are used to manipulate S-expressions. In these cases, the format is free and the physical end-of-line character can occur anywhere. It is nevertheless sometimes necessary to read or write character by character or line by line. Then the position of the *end-of-line* characters is important. When reading a character at a time (using the `readcn/readch` or `peekcn/peekch` functions), LE-LISP delimits the end of a line with the two consecutive characters `#\cr` and `#\lf`. This is true for terminals as well as for files.

When reading a line at a time (using the `readstring` or `readline` functions), the physical end-of-line is not included in the result.

When printing, end-of-line marks are explicitly inserted by the user (by calling the `print`, `terpri`, and `tynewline` functions), and automatically by the system when the right margin of the output buffer is reached. Just which marks are inserted depends on the system and the type of terminal being used. They do not necessarily result in additional output characters (in the case of output to a `bitmapped display`, or output to record files, etc.).

The right margin can be positioned beyond the end of the buffer. In this case, the system does not automatically add the end-of-line mark (see the `rmargin` function).

6.1.4 Channels

All I/O takes place in a context of sequential streams that are connected to terminals or files. LE-LISP's basic architecture includes the current input and output streams, which are set by the two functions named `inchan` and `outchan`. All reading is done on the current input stream, and all writing is done on the current output stream.

6.1.5 Programmable I/O interrupts

The most often-used I/O functions—such as `read` and `print`—read and write characters to and from I/O buffers. They do not physically access the terminal or the disk.

The lowest-level functions provide means to manage the I/O buffers. These are the programmable interrupts `bol`, `eol` and `flush`. They are easily redefinable by the user, through the programmable interrupt management mechanism. One can thereby extend the I/O system to read from a list of character strings, write to several files simultaneously, etc. While these functions are very powerful, they are seldom used explicitly.

6.2 Basic input functions

(read) *[function with no arguments]*

Reads the next S-expression (of any type, atom or list) from the current input stream, and returns it as its value. `read` is the main read function, and provides the means to read any LISP object, of any size whatsoever. Its behavior is given in detail in the next section.

If a syntax error is detected, the LISP error `errsxt` is raised. This gives you the name of the function that caused the error, `fn`, and a message indicating the type of error:

```
** <fn> : syntax error : <msg>
```

Here is the list of possible messages:

```
 1   list too short
 2   string too long
 3   symbol too long
 4   bad beginning of expression
 5   special symbol too long
 6   bad package
 7   bad dotted pair construction
 8   reread list contains non-characters
 9   bad argument list
10   bad splice-macro
11   EOF during READ
12   bad use of BACKQUOTE
```

More often than not, this error is due to a bad use of dotted pairs; to a special `p-name` that is too long (more than 128 characters), as a result of the omission of the `|` character; or to a character string that is too long, as a result of the omission of a `"` (double-quote) character.

(stratom n strg i) *[function with three arguments]*

Performs a partial read of `n` characters from the character string `strg`. `stratom` returns a symbol or a number if the `i` flag is equal to `()`, and always returns a symbol if this flag is not equal to `()`. This function gives access to the internal part of the LE-LISP reader that does numerical conversions and searches for symbols in the oblist, and so cannot be easily described in LE-LISP. It is used primarily to write new readers in LE-LISP.

```
(stratom 3 "abcdef" ())     ⟹   abc
(stratom 3 "01234" ())      ⟹   12
(stratom 3 "01234" t)       ⟹   |012|
(stratom 4 " () " ())       ⟹   | () |
(stratom 5 "00012.34" ())   ⟹   12
(stratom 6 "00012.34" ())   ⟹   12.
(stratom 7 "00012.34" ())   ⟹   12.3
```

(readstring) [*function with no arguments*]

Reads the next line from the input stream and returns it in the form of a character string. This string contains no end-of-line delimiters that might have been present in the input. `readstring` provides a way to do portable reads at the input line level. It is used to build efficient and easily-implemented interfaces between programs and users.

Warning: This function does not convert upper-case letters into their lower-case equivalents.

In LE-LISP, `readstring` could be defined in the following manner:

```
(defun readstring ()
    (let ((l) (c))
         (while (neq (setq c (readcn)) #\cr) ; read up to #\cr
                (newl l c))
         (readcn)                       ; skip #\lf
         (string (nreverse  l))))
```

(readline) [*function with no arguments*]

Like the previous function, `readline` reads the next line from the input stream, but it returns this line in the form of a list of internal character codes. This list contains no end-of-line delimiters that might have been present in the input. This function is more expensive (in terms of memory resources) than the previous one, but sometimes provides a more practical interface.

In LE-LISP, `readline` could be defined in the following manner:

```
(defun readline ()
    (explode (readstring)))
```

```
? (progn (readline)         ; terminate the current line,
?  (mapcar 'ascii (readline))) ; then read another line.
? the gay nightingale
= (t h e g a y n i g h t i n g a l e )
```

To build a sequence of words from a line that contains no special characters (,), ., [, or], use the following.

```
? (progn (readline)
?  (implode (append '#"(" (readline) '#")")))
? the singing mockingbird
= (the singing mockingbird)
```

(readcn) [*function with no arguments*]

Reads and returns the value of the next character of the input stream. The value returned is the character's internal character code—that is, a number. This function does not do automatic conversion from upper-case to lower-case character codes. It cannot be easily described in LISP.

(readch) [*function with no arguments*]

`readch` is like the preceding function, except that it returns a single-character atom, and not a character code.

In LE-LISP, `readch` could be defined in the following manner:

```
(defun readch ()
    (ascii (readcn)))
```

(peekcn) [*function with no arguments*]

This function returns the next character in the input stream in the same way that the `readcn` function does, except that the character is not really read. It is merely *examined*. As a result, consecutive calls to `peekcn` always return the same result. `peekcn` is used for look-ahead operations on the input stream.

In LE-LISP, `peekcn` could be defined in the following manner:

```
(defun peekcn ()
    (let ((cn (readcn)))
         (reread (list cn))))
```

(peekch) [*function with no arguments*]

`peekch` is like the preceding function, except that it returns a single-character atom, and not a character code.

In LE-LISP, `peekch` could be defined in the following manner:

```
(defun peekch ()
    (ascii (peekcn)))
```

(reread lcn) [*function with one argument*]

Pushes the list of characters `lcn` onto the front of the input stream. These characters will be read by the next call to one of the preceding read functions (`read`, `readstring`, `readcn`, ...) before it continues with the actual input stream. This function is particularly powerful if it is used inside a `read` or a macro character. `reread` returns the list of characters that were added onto the head of the input stream as its value.

```
? (progn (reread (list #/' #/a #\sp)) (read))
= 'a
```

To print the list of characters to reread, type `(print (reread ()))`.

(read-delimited-list cn) *[function with one argument]*

Reads a list of S-expressions up to the occurrence of a character whose internal character code is `cn`. The type of the character `cn` must be `cmacro`, `csplice`, or `cmsymb` (see the section on character types.) This function is particularly useful inside a macro to read a group of S-expressions.

In LE-LISP, `read-delimited-list` could be defined in the following manner:

```
(defun read-delimited-list (cn)
    (let ((res ()))
         (until (eq (validcn) cn)
                (newl res (read)))
         (readcn)          ; read the character cn
         (nreverse res)))

(defun validcn ()
    (selectq (typecn (peekcn))
        ((csep cecom)
              (readcn)
              (validcn))
        (cbcom
              (readcn)
              (until (eq (typecn (readcn)) 'cecom))
              (validcn))
        (t (peekcn))))
```

Let us define two macro characters:

```
? (dmc |[| ()
?  (cons 'list (read-delimited-list #/])))
= [   ? (dmc |]| ()
?  (error '|]| 'errsxt "must appear after a [")
= ]
```

From this point on, expressions containing braces will be read in the following manner:

```
[1 2 3 4]                 is read as      (list 1 2 3 4)

['print [a b c] 'foo]     is read as      (list 'print (list a b c) 'foo)

[1 2 ; Comments ignored
   3
   ]                      is read as      (list 1 2 3)
```

(teread) [*function with no arguments*]

Flushes all the characters waiting in the input buffer of the current input channel. The first read immediately following a call to this function will trigger the programmable interrupt `bol`. This function is particularly useful for flushing the separation characters (delimiters) present in the reader's internal buffers just after the action of the `read` function. Since the reader's internal buffers cannot be easily described in LISP, this function has no LISP description.

```
? (defun foo ()
?  (print "read->" (read))
?  (print "readstring->" (readstring))
?  t)
= foo   ? (foo)
? hello
read->hello
readstring->
= t   ? (defun bar ()
?  (print "read->" (read))
?  (teread)
?  (print "readstring->" (readstring))
?  t)
= bar   ? (bar)
? hello
read->hello
? hello world
readstring->hello world
= t
```

6.2.1 Inside `read`

The variable and the function described next are only meaningful inside a `read`. They will only be used in a macro character or in the processing of the programmable interrupts `bol` or `eof`.

#:system:in-read-flag [*variable*]

This internal variable, which is maintained by the `read` function, indicates whether the LISP reader is in the middle of a `read` (that is, if a LISP object of any type is in the process of being read).

This variable is used for example in the `eof` function to test whether an end-of-file appears while a LISP object is being read (see the `eof` function).

(curread) [*function with no arguments*]

Returns either the list currently being read or `()` if the reader is not in the process of reading a list. The returned list contains an additional element at its head: the `curread` symbol.

Let us define a macro character as follows:

```
(dms |%| ()
     (print (curread))
     ())
```

Reading `'(a b % c (% d %) % e %)` will produce the following results:

```
(curread a b)
(curread)
(curread d)
(curread a b c (d))
(curread a b c (d) e)
```

Now let us redefine the percentage sign as an infix binary operator:

```
(dms |%| ()
     (if (consp (cdr (curread)))
         (rplacd (curread) (cons '|%| (cdr (curread))))
         (error '|%| 'errsxt ()))
     ())
```

Reading `'(a % (b % c))` will produce `(% a (% b c))`.

6.3 Use of the terminal for input

Before each request for input from the terminal, the system prints a prompt string (`?` by default). (This can be modified by way of the `prompt` function.) It then reads a line of terminal input. The user terminates the line by typing either the `#\cr` or `#\lf` character. The currently active programming interrupt `bol` manages the dialog with the user (see the section on programmable I/O interrupts).

LE-LISP echos each character as it is read. ASCII control characters (characters entered by holding down the CONTROL key and pressing a printable character key at the same time) are printed with a `^` character followed by the printable character.

A minimal line editor is also activated. It understands the following character keys with the meaning indicated:

- RUBOUT, DELETE or BACKSPACE: Erase the last character entered.
- `^U` or `^X`: Completely erase the current line.

A full line editor called `edlin` is provided with the system. It has a command-history facility and command-name completion. `edlin` is activated by evaluating the `edlin` function (see chapter 17). The ESCAPE-`?` command gives on-line help for `edlin`.

The global system variables `#:system:real-terminal-flag` and `#:system:line-mode-flag` can be used to supress the character echoing and the minimal line editor functions.

`#:system:real-terminal-flag` [*variable*]

If this flag is set (the default depends upon the system), the line editor and character echo functions are activated. If it is cleared, both these functions are deactivated, and the system assumes that another process performs these tasks.

`#:system:line-mode-flag` [*variable*]

If this flag is set (the default option depends upon the system), the line editor assumes that reading can be carried out one line at a time. In this case, the behavior of the `tyi` and `tys` functions is unpredictable, and the system uses the `tyinstring` function to read a line from the terminal.

`(prompt strg)` [*function with an optional argument*]

The `prompt` variable holds the *prompt string* printed by the system when it is ready to receive input from the terminal. This function can be used with the `with` control structure to change the prompt dynamically.

By default this string has the value `?`.

```
? ()                          ; the default prompt character string
= ()
? (prompt "> ")
= >
> ()                          ; it has been changed
= ()
> (with ((prompt "hello>"))   ; with the prompt string "hello>"
>       (toplevel))           ; call the toplevel (interpreter)
hello>(cons 1 2)
= (1 . 2)                     ; value of the cons
= (1 . 2)                     ; value of the call to toplevel
>                             ; return to the string "> "
```

6.4 Standard reader

The `read` function reads LISP S-expressions in *free format*: that is, each syntactic element can be surrounded by one or more delimiters (spaces, end-of-line characters or comments).

The LISP reader is parameterized by a *read table*, which indicates which are the special characters (list delimiters, comments string delimiters, macro characters, etc.). All other characters are characters of type `pname`, or "normal" characters.

6.4.1 Reading symbols

A symbol is a sequence of characters of type `pname` which does not represent a number (integer or floating-point).

To include special characters in a symbol, enclose the entire symbol within a pair of symbol delimiter characters (the vertical, or "absolute value", bar by default).

The reader will not accept symbols longer than 128 characters. Very long symbols can nevertheless be made, by using the `concat` function.

Symbol packages are specified in three ways, which are described further on:

- the #-macro written as `#:`
- the macro character writen as a `:` sign
- a character of type *package-delimiter*.

`#:system:read-case-flag` [*variable*]

This variable provides a way to regulate the automatic conversion of upper-case characters into their lower-case equivalents. If the value of this variable is `()`, which is its default value, the conversion is performed automatically. If not, no conversion is done.

Here are several examples of equivalences when this variable has its default value of `()`:

`car`	*corresponds to the symbol*	`car`
`LONGVERYLONGATOM`	" " " "	`longverylongatom`
`|Foo Bar|`	" " " "	`Foo Bar`
`#:Foo:Bar`	" " " "	`#:foo:bar`
`#:foo:|Bar()|:gee`	" " " "	`#:foo:Bar():gee`

Suppose that we read the symbols in the following manner:

```
(let ((#:system:read-case-flag t))
     (read))
```

In that case, the following results are obtained:

`CdR`	*corresponds to the symbol*	`CdR`
`|AcH|`	" " " "	`AcH`

6.4.2 Reading character strings

Character strings are written between pairs of string delimiter characters (the double quote character " by default). If this character is to be included in a string, it should be introduced into the string twice in succession. Any character can be included in a string. If there is an end-of-line in the middle of a string, this latter will contain the two characters `#\cr` and `#\lf` in the position of the end-of-line. Currently a string cannot exceed 256 characters on input; the `makestring` function allows for the creation of longer strings. Uppercase to lowercase conversion is never done during the reading of a string.

A type can be assigned to a string by means of the `#`-macro written as the `#:` signs.

```
"hello"                 contains the characters      hello
"with ""dblquotes"""        "     "      "          with "dblquotes"
#:schiz:"parano"        is a typed string of type    schiz
                        containing the characters    parano
```

6.4.3 Reading integer and rational numbers

Integer numbers are represented by a string of digits of any length, optionally preceded by a `+` or `-` sign. Rational numbers are represented by an integer immediately followed by the `/` character, which is itself immediately followed by a second integer.

The construction of numbers from their textual representation is done with generic arithmetic. This means that if a numeral given to the system is not the representation of a small (16-bit) integer, its value will depend on the generic arithmetic being used. The standard generic arithmetic converts representations of integers whose values are outside the range of 16-bit storage, and those of rationals which are not reducible to integers, to floating-point numbers.

Here are some examples of integers:

```
12      corresponds to the integer      12
-1          "       "   "    "          -1
+0          "       "   "    "           0
12/4        "       "   "    "           3
12/-4       "       "   "    "          -3
```

Here are some examples of standard generic arithmetic:

```
 32767    corresponds to the integer       32767
-32767        "       "   "    "          -32767
 32768    corresponds to floating-point    32768.
-32769        "       "   "    "         -32769.
-32768        "       "   "    "         -32768.
 12/5         "       "   "    "            2.4
-12/5         "       "   "    "           -2.4
 12/-5        "       "   "    "           -2.4
-12/-5        "       "   "    "            2.4
```

And here are some examples of rational arithmetic, as described in chapter 10:

```
30424324324     corresponds to the integer      30424324324
12/5            corresponds to the rational     12/5
45574512/56         "       "   "    "         5696814/7
```

`(ibase n)` [*function with an optional argument*]

`ibase` (for input `base`) called without an argument returns the input conversion base of the system. If the argument `n` is provided, `ibase` changes the input conversion base to this value. The input base affects only the input of integer and rational numbers, since the input of floating-point numbers always takes place in base 10. Values for the input base must be between 2 and 36, inclusive. This function is the counterpart to the `obase` function.

```
? (ibase)
= 10    ? (ibase 16)
= 16    ? fff ; this is no longer a symbol
= 4095    ? (ibase a) ; to go back to base 10
= 10    ? (ibase 36) ; gag!
= 36    ? ibase ; this is a number! so are all other symbols containing
= 23902         ; only letters and numerals.    ?  ; so -- how to get the input base back to
10 ?!?
? (|ibase| a) ; simple ..
= 10
```

There are other ways to read numbers. The `#-macros` written as `#$`, `#%` and `#<base>r` provide the means for describing numbers in any base.

6.4.4 Reading floating-point numbers

Floating-point numbers are represented by a sequence of decimal numerals (the integer or "whole" part), optionally followed by a decimal point ("period" character) and another sequence of decimal numerals (the fractional part), optionally followed by the character `E` (or `e`) followed by a signed integer number (the exponent part). The integer part or the fractional part can be missing, but both cannot be missing simultaneously. The exponent part can optionally be omitted altogether.

`1.50`	*corresponds to the number*	`1.50`
`1e+0`	" " " "	`1.`
`10e+1`	" " " "	`100.`
`-10e-1`	" " " "	`-1.`
`0e+1`	" " " "	`0.`
`.45`	" " " "	`.45`
`12.34e-3`	" " " "	`.01234`

`e-` *is a symbol!*

6.4.5 Reading lists

The representation of lists in LE-LISP is quite conventional. A list is represented by an opening `(` parenthesis sign followed by the elements of the list. These are separated by spaces, and followed by a closing `)` parenthesis sign. It is also possible to use the generalized dot notation:

```
( s-expr   .   s-expr )
```

```
e.g.   (a . (b . (c . d))) corresponds to (a b c . d)
       ((a) . (b))           "         "  ((a) b)
```

At the end of an expression read by the `read` function, there can be any number of closing parentheses which are ignored. This gives a sure way to close off S-expressions without having to count the exact number of closing parentheses required.

```
(defun foo (n) (+ n 2))))))   ; is read without error
```

6.4.6 Reading vectors

Vectors are represented by the two characters `#[` followed by the elements of the vector (which are separated by blanks), followed by the character `]`. Vectors are read using the `#`-macro written as `#[`. You can give vectors a type by using the `#`-macro designated by the `#:` signs.

```
#[a b c]            is a vector with three elements: a, b, c
#[]                 is an empty vector
#:type:#[12 23 40]  is a typed vector
```

6.4.7 Reading comments

It is possible to insert comments into the input stream, which are treated as delimiters. A comment is any sequence of characters, preceded by a character of type `begin-comment` and terminated by a character of type `end-comment`.

By default there is only one character of type begin-comment, the semi-colon (;), and a single character of type end-comment, the RETURN character. By default, then, all comments are terminated at the end-of-line.

```
(defun foo (n          ; the number
            l)         ; the list
    (if (= n 0)        ; nothing more to do
       ...
```

It is also possible to write multi-line comments, by using the `#`-macros `#|` and `|#`. These multi-line comments can be embedded.

```
(defun foo #|the name|# (n)
     #|  this comment
         goes on for several
         lines and contains #|
                    other comments |#
         which are very neatly embedded |#
     (+ n n))
```

6.4.8 Types of characters

The LE-LISP lexical analyzer (in other words, the `read` function) uses a *read table* in order to facilitate its analysis. This table associates with each character a symbolic type.

The read table is totally accessible to the user, who can therefore change it to be able to easily read new LISP dialects with differing syntaxes.

The available character types are:

`cnull` [*character type*]

All characters of this type are completely ignored by the reader. (Examples: the character *Advance-tape*, or the character `null` ...).

`cbcom` [*character type*]

A character of this type serves to indicate the beginning of a comment, which will be terminated at the occurrence of a character of the type described next. By default, there is only one character of this type, the semi-colon `;`.

`cecom` [*character type*]

A character of this type serves to indicate the end of a comment. By default there is only one character of this type, the `#\cr` (Carriage-Return) character.

`cquote` [*character type*]

A character of this type is used to *quote* any other character. This amounts to giving (this instance of the character concerned) the `cpname` type (the type of normal characters). By default, there are no characters of this type, but it is the custom to use either the slash `/` or the backslash `\` whenever you happen to need such a character.

`clpar` [*character type*]

A character of this type—the opening parenthesis `(` by default—serves to demarcate the beginning of a list.

`crpar` [*character type*]

A character of this type—the closing parenthesis `)` by default—serves to demarcate the end of a list.

`cdot` [*character type*]

A character of this type—the period `.` by default—is used to enter pointed pairs, when they occur surrounded by characters of the type `csep`.

csep [*character type*]

Characters of type `csep` are normal separators (examples: space, tab, ...).

cpkgc [*character type*]

Characters of this type play the rôle of package delimiter characters. By default there are no characters assigned to this type. It is however common practice to use the colon character (`:`). The use of `:` as a package delimiter character results in the following readings, which are more readable than the corresponding uses of the `#:` `#`-macro.

```
? (typecn #/: 'cpkgc)
= cpkgc   ? 'foo:bar
= #:foo:bar   ? 'gee:muu:wizz
= #:gee:muu:wizz   ? (setq #:sys-package:colon 'my:private:package)
= #:my:private:package   ? ':bizarre
= #:my:private:package:bizarre
```

csplice [*character type*]

A character with this type is a splice-macro. A function is associated with the symbol whose `p-name` is this character. This function is invoked automatically when this character is read in the input stream. (See the following section.) The classic example of such a character is the macro character "pound sign" (or "hash sign") `#`.

cmacro [*character type*]

A character with this type is a macro character. A function is associated with the symbol whose `p-name` is this character. This function is invoked automatically when this character is read in the input stream. (See the following section.)

cstring [*character type*]

Characters of this type play the rôle of character string delimiter characters. By default there is a single character of this type, namely the double quote character `"`.

cpname [*character type*]

Characters of this type are normal characters, and are thus capable of being used to build a `p-name`.

csymb [*character type*]

This type indicates that a character is a delimiter of special symbols. By default there is only one character of this type: the *vertical bar* sign written as `|` (also referred to as the *absolute value* sign).

cmsymb [*character type*]

Characters of this type must be read as mono-character symbols, and need not be delimited.

(typecn cn symb) [*function with one or two arguments*]

Provides a means to know (if `symb` is not supplied) or to modify (if `symb` is supplied) the type of the character `cn`. This function returns the current type of the character `cn`, after modification in the case the `symb` is present.

After the < and > characters have been redefined as follows:

```
(typecn #/< 'clpar)  ⟹   clpar
(typecn #/> 'crpar)  ⟹   crpar
```

the input `<cons '(a> '<b)>` will be read as `(cons '(a) '(b))`.

(typech ch symb) [*function with one or two arguments*]

`typech` is like the preceding function, except that the character is specified in the form of a mono-character symbol.

Here is a program that prints the table of character types:

```
(defun printablech ()
    ; print the table of characters' types
    (let ((i -1) (j 63)
          (al (mapcoblist (lambda (x)
                 (when (getprop x '#:sharp:value)
                       (list (cons (getprop x '#:sharp:value)
                             x)))))))
         (repeat 64
                 (printablech1 (incr i) 3  al)
                 (printablech1 (incr j) 34 al)
                 (terpri))))

(defun printablech1 (val pos al)
    ; print the type of a character
    (outpos pos)
    (with ((obase 10)) (prin val) (outpos (+ pos 5)))
    (with ((obase 16)) (prin "#$" val) (outpos (+ pos 10)))
    (cond ((assoc val al) (prin "#\" (cassoc val al)))
          ((< val 32) (prin "#^") (princn (+ val 64)))
          (t (prin "#/") (princn val)))
    (if (>= (outpos) (+ pos 17))
        (prin " ")
        (outpos (+ pos 17)))
    (prin (typecn val)))
```

Here is the printed character type table:

```
= t
   0    #$0  #\null cnull         64   #$40 #/@    cpname
   1    #$1  #^A    cmacro        65   #$41 #/A    cpname
```

```
2    #$2  #^B    cpname      66   #$42 #/B    cpname
3    #$3  #^C    cpname      67   #$43 #/C    cpname
4    #$4  #^D    cpname      68   #$44 #/D    cpname
5    #$5  #^E    cmacro      69   #$45 #/E    cpname
6    #$6  #^F    cmacro      70   #$46 #/F    cpname
7    #$7  #\bell cpname      71   #$47 #/G    cpname
8    #$8  #\bs   csep        72   #$48 #/H    cpname
9    #$9  #\tab  csep        73   #$49 #/I    cpname
10   #$A  #\lf   cecom       74   #$4A #/J    cpname
11   #$B  #^K    csep        75   #$4B #/K    cpname
12   #$C  #^L    cmacro      76   #$4C #/L    cpname
13   #$D  #\cr   cecom       77   #$4D #/M    cpname
14   #$E  #^N    cpname      78   #$4E #/N    cpname
15   #$F  #^O    cpname      79   #$4F #/O    cpname
16   #$10 #^P    cmacro      80   #$50 #/P    cpname
17   #$11 #^Q    cpname      81   #$51 #/Q    cpname
18   #$12 #^R    cpname      82   #$52 #/R    cpname
19   #$13 #^S    cpname      83   #$53 #/S    cpname
20   #$14 #^T    cpname      84   #$54 #/T    cpname
21   #$15 #^U    cpname      85   #$55 #/U    cpname
22   #$16 #^V    cpname      86   #$56 #/V    cpname
23   #$17 #^W    cpname      87   #$57 #/W    cpname
24   #$18 #^X    cpname      88   #$58 #/X    cpname
25   #$19 #^Y    cpname      89   #$59 #/Y    cpname
26   #$1A #^Z    cpname      90   #$5A #/Z    cpname
27   #$1B #\esc  cpname      91   #$5B #/[    cmacro
28   #$1C #^\    cpname      92   #$5C #/\    cpname
29   #$1D #^]    cpname      93   #$5D #/]    cmacro
30   #$1E #^^    cpname      94   #$5E #/^    cmacro
31   #$1F #^_    cpname      95   #$5F #/_    cpname
32   #$20 #\sp   csep        96   #$60 #/`    cmacro
33   #$21 #/!    cmacro      97   #$61 #/a    cpname
34   #$22 #/"    cstring     98   #$62 #/b    cpname
35   #$23 #/#    csplice     99   #$63 #/c    cpname
36   #$24 #/$    cpname      100  #$64 #/d    cpname
37   #$25 #/%    cpname      101  #$65 #/e    cpname
38   #$26 #/&    cpname      102  #$66 #/f    cpname
39   #$27 #/'    cmacro      103  #$67 #/g    cpname
40   #$28 #/(    clpar       104  #$68 #/h    cpname
41   #$29 #/)    crpar       105  #$69 #/i    cpname
42   #$2A #/*    cpname      106  #$6A #/j    cpname
43   #$2B #/+    cpname      107  #$6B #/k    cpname
44   #$2C #/,    cmacro      108  #$6C #/l    cpname
45   #$2D #/-    cpname      109  #$6D #/m    cpname
46   #$2E #/.    cdot        110  #$6E #/n    cpname
```

47	#$2F #//	cpname	111	#$6F #/o	cpname
48	#$30 #/0	cpname	112	#$70 #/p	cpname
49	#$31 #/1	cpname	113	#$71 #/q	cpname
50	#$32 #/2	cpname	114	#$72 #/r	cpname
51	#$33 #/3	cpname	115	#$73 #/s	cpname
52	#$34 #/4	cpname	116	#$74 #/t	cpname
53	#$35 #/5	cpname	117	#$75 #/u	cpname
54	#$36 #/6	cpname	118	#$76 #/v	cpname
55	#$37 #/7	cpname	119	#$77 #/w	cpname
56	#$38 #/8	cpname	120	#$78 #/x	cpname
57	#$39 #/9	cpname	121	#$79 #/y	cpname
58	#$3A #/:	cmacro	122	#$7A #/z	cpname
59	#$3B #/;	cbcom	123	#$7B #/{	cmacro
60	#$3C #/<	cpname	124	#$7C #/|	csymb
61	#$3D #/=	cpname	125	#$7D #/}	cmacro
62	#$3E #/>	cpname	126	#$7E #/~	cpname
63	#$3F #/?	cpname	127	#$7F #\del	cnull
			128 to 255		cpname

6.4.9 Macro characters

A macro character is any character to which a function is associated by the following macro definition functions. The associated function is automatically executed when this character is encountered in the input stream. There are two types of macro characters, `cmacro` and `csplice`. In the case of `cmacro`s, the value returned by the function associated with the macro character replaces the macro character read. In the case of `csplice`s, the value returned by the function associated with this macro character must be a list (which can be empty). This list will be added to the expression being read by means of the `nconc` function. If the value returned by this function is not a list, syntax error number `10` will be raised. All characters can be used as macro characters.

(dmc `ch () s`$_1$ ... `s`$_n$**)** *[special form]*

Used to define a macro character of the first (cmacro) type. The argument `ch` must be a mono-character symbol. `dmc` associates, with this character, a function with an empty parameter list (this empty list is required to occupy this position in the call) and a function body `s`$_1$ ... `s`$_n$. `dmc` has the same syntax as the other definition functions (`defun`, `df` and `dm`), and returns `ch` as its value.

In LE-LISP, `dmc` could be defined in the following manner:

```
(df dmc l
    ; construct the associated function
    (apply 'defun l))
    ; set the new type of this character
    (typech (car l) 'cmacro)
    ; and return the character as value
    (car l))
```

(dms ch () s_1 ... s_n) [*special form*]

Used to define a macro character of the second (csplice) type. The argument `ch` must be a mono-character symbol. `dms` associates, with this character, a function with an empty parameter list (this empty list is required to occupy this position in the call) and a function body s_1 ... s_n. `dms` has the same syntax as the other definition functions (`defun`, `df` and `defmacro`), and returns `ch` as its value.

In LE-LISP, `dms` could be defined in the following manner:

```
(df dms l
    ; build the associated function
    (apply 'defun l))
    ; set the character's new type
    (typech (car l) 'csplice)
    ; and return the character as value.
    (car l))
```

In order to destroy a macro character definition, the type of the character must be changed and the associated function must be destroyed, with for example the following function:

```
(defun remach (ch)
    ; [for  remove macro character]
    ; the character returns to normal
    (typech ch 'cpname)
    ; destruction of the function
    (remfn ch)
    ch))
```

Warning: Since the function associated with a macro character is of the same type as functions created by the user (of type `defun`), it is not possible for a mono-character atom to have a `defun`-type definition and a `dmc`- or `dms`-type definition at the same time.

There are 13 pre-defined macro characters:

- apostrophe macro character '
- grave accent macro character `
- comma macro character ,
- colon macro character :
- circumflex macro character ^
- open bracket macro character [
- pound sign macro character #
- bang macro character !
- load-file macro character ^L (CONTROL-L)

- load-module macro character ^A (CONTROL-A)
- edit-file macro character ^E (CONTROL-E)
- edit-function macro character ^F (CONTROL-F)
- paragraph macro character ^P (CONTROL-P)

Basic macro characters: colon, circumflex and open bracket

' [*macro character*]

The apostrophe or single quote ' is the best known and most used of the macro characters. Placed before any ordinary S-expression, it returns the list (quote S-expression).

In LE-LISP, ' could be defined in the following manner:

```
(dmc |'| ()
     (list 'quote (read)))

     e.g. '(a b)   is read as      (quote (a b))
          ''a           "          (quote (quote a))
```

^ [*macro character*]

This macro character lets mono-character symbols whose p-names are control characters, be read.

In LE-LISP, ^ could be defined in the following manner:

```
(dmc |^| ()
      (ascii (logand (readcn) 31)))

to read a mono-character symbol with ascii code 004

    e.g. ^d
```

[[*macro character*]

This macro character is only defined within the arbitrary-precision arithmetic libraries, and describes numbers longer than a line long or using specialized notation.

Warning: this macro character should not be confused with the representation of vectors of S-expressions: #[...]

Backquote macro character

` [*macro character*]

`,` [*macro character*]

`,.` [*macro character*]

`,@` [*macro character*]

This macro character is used to build programs which construct Lisp S-expressions (in general, lists). The S-expressions built by this macro character will use the `quote`, `cons`, `mcons`, `list`, `append`, and `nconc` functions. Everything that follows the backquote character is considered to be the model of the list to construct. The variable elements of this model will be indicated by a comma placed before the expression to calculate. All the other elements of the model are considered as constants. There are three kinds of variable elements:

- simple elements prefixed by a comma
- list segments which will be added to the list by means of the `append` function, prefixed by a comma followed by the ampersand (or at-sign) character "@"
- list segments which will be added to the list by means of the `nconc` function, prefixed by a comma followed by the period character "."

In LE-LISP, `backquote` could be defined in the following manner:

```
(dmc |,| ()
     ; to circumvent a large number of errors
     (error '|,| 'errsxt "outside a `"))

(dmc |`| ()
     (flet ((|,| ()
             (cond
                ((eq (peekcn) #/@)
                  (readcn)
                  (cons '|,@| (read)))
                ((eq (peekcn) #/.)
                  (readcn)
                  (cons '|,.| (read)))
                (t (cons '|,| (read))))))
          (backquotify (read))))

(defun backconstant (x)
   ; test if the object x is a constant
   (or (null x)
       (and (consp x)
            (eq (car x) 'quote)
            (null (cddr x)))))
```

```
(defun backquotify (x)
    ; construction of the backquoted expression
    (cond ((null x) ())
          ((atom x) (list 'quote x))
          ((eq (car x) '|,|) (cdr x))
          ((and (consp (car x)) (eq (caar x) '|,@|))
             ; simplification of the append
             (let ((a (cdar x))
                   (d (backquotify (cdr x))))
                  (cond
                     ; (append x ()) --> x
                     ((null d) a)
                     ; (app x (app . l)) --> (app x . l)
                     ((and (consp d) (eq (car d) 'append))
                         (mcons 'append a (cdr d)))
                     ; nothing to do
                     (t (list 'append a d)))))
          ((and (consp x)
                (consp (car x))
                (eq (caar x) '|,.|))
             (if (cdr x)
                 (list 'nconc
                       (cdar x)
                       (backquotify (cdr x)))
                 (cdar x)))
          (t (let ((a (backquotify (car x)))
                   (d (backquotify (cdr x))))
                  ; simplification of the conses
                  (cond
                    ((null d)
                        ; (cons x ()) --> (list x) or '(x)
                        (if (backconstant a)
                            (list 'quote (list (cadr a)))
                            (list 'list a)))
                    ((and (backconstant a)
                          (backconstant d))
                        ; (cons 'x  'y) --> '(x . y)
                        (list 'quote
                              (cons (cadr a) (cadr d))))
                    ((and (consp d) (eq (car d) 'cons))
                        ; (cons x (cons y z))
                        ;  --> (mcons x y z)
                        (list 'mcons a (cadr d) (caddr d)))
                    ((and (consp d) (eq (car d) 'list))
```

```
                        ; (cons x (list y1 ... yn))
                        ;  --> (list x y1 ... yn)
                        (cons 'list (cons a (cdr d))))
                    ((and (consp d) (eq (car d) 'mcons))
                        ; (cons x (mcons y1 ... yn))
                        ; --> (mcons x y1 ... yn)
                        (mcons 'mcons a (cdr d)))
                    (t  ; the general case
                        (list 'cons a d)))))))

`(a b c)     is read as    '(a b c)
`(a ,b c)        ""         (mcons 'a b ('c))
`(,x . y)        ""         (cons x 'y)
`(a b . ,c)      ""         (mcons 'a 'b c)
`(a b ,@c d e)   ""         (mcons 'a 'b (append c '(d e)))
`(x ,@y ,@z v)   ""         (cons 'x (append y z '(v))
`(x ,.y ,z)      ""         (cons 'x (nconc y (list z)))
```

Sharp macro character

[*macro character*]

This macro character provides a means for converting numbers or for calling the interpreter in the course of a normal read. Its action is very general and can be explained in the following way. When a **#** character occurs in the input stream, the LISP reader reads the next character using the `readch` function. This character, called the **#**-macro selector, must have a function definition in the package whose name is found in the `#:sys-package:sharp` variable. The value returned by a call to this function must be a list (which can be empty), which will be inserted into the object being read by means of the `nconc` function.

Certain macro character selectors accept numeric parameters. In these cases, the function associated with the selector receives this parameter as its argument. The parameters are supplied between the **#** character and the selector. They are always interpreted in base 10.

The following read invokes the selector `r` with the parameter 36.

```
? #36rfoo
= 20328
```

#:sys-package:sharp [*variable*]

This variable contains the name of the package in which the functions associated with the various **#**-macro selectors must reside in order to be found by LE-LISP. By default, its value is `sharp`.

Here is the LISP description of this macro character.

```
(defvar #:sys-package:sharp 'sharp)
```

```
(dms |#| ()
     (let ((c (readcn))
           argnum)
          (when (memq c '#"0123456789")
                (setq argnum (sub c #/0))
                (while (memq (setq c (readcn)) '#"0123456789")
                       (setq argnum (add (sub c #/0)
                                         (mul argnum 10)))))
          (let ((f (getfn #:sys-package:sharp (ascii c) ())))
               (if f
                   (apply f (when argnum (list argnum)))
                   (error '|#| 'errudf c)))))
```

It is possible to define new #-macros with the following function.

(defsharp ch larg s1 ... sN) [*special form*]

defsharp defines a new #-macro. ch is the new selector character. The list larg is the parameter list of the function associated with the selector. This list is empty if the selector does not accept a numerical parameter. If so, it is a list of the usual parameter (the selector), and the function receives the numerical parameter as its argument. defsharp thus has the same syntax as the ordinary definition functions (de, dmd, etc.).

In LE-LISP, defsharp could be defined in the following manner:

```
(df defsharp (ch . f)
    (setfn (symbol #:sys-package:sharp ch) 'expr f)
    ch)
```

There are a number of predefined selectors:

#/ [*#-macro*]

This #-macro returns the internal character code of the next character in the input stream. It is the best way known to enter character codes as data in a program. #/A is a convenient replacement for 65 as the internal character code of the character A.

```
e.g. #/A      is read as      65
     #/a      "   "   "       97
```

#\ [*#-macro*]

#:sharp:value [*property*]

This #-macro returns the value associated with the next symbol in the input stream. This value is found on the P-list of the symbol under the `#:sharp:value` property. This macro is often used to define or retrieve the internal character codes of non-printable characters. If no value is found on the P-list, `#:sharp:value` raises the `errudv` error.

```
#\cr     is read as      13
#\esc     "  "   "       27
#\sp      "  "   "       32

to get the list of current values:

(maploblist (lambda (x) (getprop x '#:sharp:value)))
```

#^ [*#-macro*]

This #-macro returns the internal character code of the next character in the input stream, considered as an `ascii` control character, that is, with bits 6 and 7 set to 0.

```
#^c      is read as    3
#^Z       "  "   "     26
```

#$ [*#-macro*]

This #-macro reads the next number in the input stream in base sixteen: that is, in hexadecimal.

```
#$ffff      is read as    -1
#$1000       "  "   "     4096
```

#% [*#-macro*]

This #-macro reads the next number in the input stream in base two: that is, in binary.

```
#%1100001    is read as    97

#%110         "  "   "     6
```

#<base>r [*#-macro*]

This #-macro reads the next number in the input stream in base `base`, which is a numerical parameter accepted by the #-macro.

```
#8r177       is read as    127
#3r221        "  "   "     25
```

| [*inside #-macro*]

This macro character allows for the construction of 16-bit numeric values from two 8-bit values, with the vertical bar separating the bytes.

This character can only appear after one of the selectors **/**, **** or **^**.

```
#^x|^a     is read as    6145
#^a|/a      "  "   "     353
#\cr|\lf    "  "   "     3338
```

#' [*#-macro*]

Placed before an S-expression, this macro returns the list (**function s-expression**).

```
#'foo    is read as   (function foo)
```

It is therefore an abbreviation for calls to the **function** function.

#" [*#-macro*]

This **#**-macro returns the list of internal character codes of the characters enclosed within quotes. To include the quote character, double it (enter it twice in a row) in the input stream.

```
#"Foo Bar"  is read as the list    (70 111 111 32 66 97 114)
#"F""B"      "  "   "   "    "     (70 34 66)
```

#: [*#-macro*]

This **#**-macro provides a way to refer to the name of a symbol and its package(s). With this selector, one can define or retrieve any kind of package symbol (symbols or lists). It also allows one to describe the type of a vector or a typed character string. This same idiom is used by the printer, when it uses the **#:system:print-package-flag** option.

```
#:pkgc:symb       is read as the symbol <symb>
                  in the package <pkgc>
#:(a . b):foo     ... the symbol foo in the package (a . b)
#:bar:#[e1 e2]    ... the vector [e1 e2] of type bar
```

#. [*#-macro*]

This **#**-macro evaluates the next expression in the input stream. It returns the value of the expression. This **#**-macro provides a way to call the evaluator in the midst of a read.

```
#.(1+ 4)    is read as    5
```

#([*#-macro*]

This **#**-macro reads a list whose first cell is tagged.

```
(tconsp '#(a . b))  ⟹   #(a . b)
```

#[[*#-macro*]

This **#**-macro reads a vector of S-expressions.

```
(vectorp (makevector 2 4))          ⟹   #[4 4]
(equal (makevector 2 'a) #[a a]))   ⟹   t
```

#+ [*#-macro*]

This **#**-macro evaluates the next expression in the input stream. If its value is equal to true, the **#**-macro returns the following expression in the input stream, otherwise it returns **()**, but skips over the following expression. This macro character can thus be used to do conditional reads.

```
#+ gc-on  (defun gc-alarm () )
 ; if the value of variable gc-on is not equal to (),
 ; return the list (defun gc-alarm ()), otherwise return ().
```

#- [*#-macro*]

This **#**-macro is the opposite of the preceding one. If the value of the next expression in the input stream is false (equal to **()**), the expression following it is returned, otherwise it is skipped and the macro returns **()**.

#| [*#-macro*]

This **#**-macro starts a multi-line comment that terminates at the occurrence of the sequence of characters **|#**. There can be multi-line comments in the middle of multi-line comments.

Here are LISP equivalents for the standard #-macros:

```
(defsharp |.| ()
    (list (eval (read))))

(defsharp + ()
    (if (eval (read))
        (list (read))
        (read)
        ()))

(defsharp - ()
    (ifn (eval (read))
         (list (read))
         (read)
```

```
        ()))

(defsharp |'| ()
    (list (list 'function (read))))

(defsharp |/| ()
    (#:sharp:lowbyte (readcn)))

(defsharp |\| ()
    (let ((l (read)))
         (let ((n (getprop l '#:sharp:value)))
              (if n
                  (#:sharp:lowbyte n)
                  (error '#:sharp:value 'errudv l)))))

(mapc
   (lambda (x y) (putprop x y '#:sharp:value))
   '(null bell bs tab lf return cr  esc sp del rubout)
   '(0    7    8  9   10 13     13  27  32 127 127))

(defsharp |^| ()
    (#:sharp:lowbyte (logand 31 (readcn))))

(de #:sharp:lowbyte (n)
    (if (neq (peekcn) 124) ; the vertical bar, or absolute value, character
        (list n)
        (readcn)           ; skip the vertical bar
        (list
          (let ((n1 (logshift n 8)) (n2 (readcn)))
             (selectq n2
              (47 ; i.e., slash
                (logor n1 (readcn)))
              (94 ; i.e., carret
                (logor n1 (logand 31 (readcn))))
              (92 ; i.e., backslash
                (let ((l (read)))
                     (let ((n (getprop l 'sharp:value)))
                          (if n
                              (logor n1 n)
                              (error 'sharp:value 'errudv l)))))
              (t  ; all the others

                  (error '#:sharp:lowbyte 'errsxt n2)))))))

(defsharp |"| ()
```

```
   (let ((l) (i))
        (untilexit eoc
           (if (= (setq i (readcn)) #/")
               (if (= (peekcn) #/")
                   (newl l (readcn))
                   (exit eoc (list (nreverse l))))
               (newl l i)))))))

(defsharp |$| ()
    (with ((ibase 16))
          (let ((r (read)))
               (if (fixp r)
                   (list r)
                   (error '|#$| 'errsxt r)))))

(defsharp |%| ()
    (with ((ibase 2))
          (let ((r (read)))
               (if (fixp r)
                   (list r)
                   (error '|#%| 'errsxt r)))))

(defsharp |r| (base)
    (with ((ibase base))
          (let ((r (read)))
               (if (fixp r)
                   (list r)
                   (error '|#r| 'errsxt r)))))

(defsharp |||| ()
    (skip-sharp-comment)
    ())

(de skip-sharp-comment ()
    (untilexit eoc
       (selectq (readcn)
            (#/# (when (eq (peekcn) #/|)
                       (readcn)
                       (skip-sharp-comment)))
            (#/| (when (eq (peekcn) #/#)
                       (readcn)
                       (exit eoc)))
            (t ()))))

(defsharp |:| ()
```

```
    (with ((typecn #/: 'cpkgc)
           (typecn #/# 'cpname))
          (list (read))))

(defsharp |(| ()
    (reread '(#/())
    (let ((l (read))) (list (tconsmk l))))

(defsharp |[| ()
    (list (apply 'vector (read-delimited-list #/]))))
```

Colon (:) macro character

Even though it is always possible to describe a symbol and its package by using the absolute notation `#:pkgc:symb`, it is often useful to be able to use an abbreviation. This is the purpose of the colon (:) macro character, placed in front a symbol. The symbol will be created in the package whose name is the value stored the `#:sys-package:colon` variable.

#:sys-package:colon [*variable*]

This variable contains the name of the package to use when the macro character colon (:) is used in front of a symbol. By default its value is `user`. This variable also describes the package to use when a symbol beginning with a character of type `cpkgc` is read.

```
(defvar #:sys-package:colon 'local)  ->  #:sys-package:colon
:foo        will be read as          #:local:foo
(defvar #:sys-package:colon
        '#:pk1:pk2)                  ->  #:sys-package:colon
:foo:bar    will be read as          #:pk1:pk2:foo:bar

(typecn #// 'cpkgc)                 / is a package delimter char
/foo        will be read as          #:pk1:pk2:foo
/foo/bar         "  "                #:pk1:pk2:foo:bar
```

Terminal macro characters ! ^L ^A ^E ^F ^P

These macro characters are used to quickly enter commands interactively on the terminal.

^L [*macro character*]

Placed in front of a symbol, this macro character executes the expression `(libloadfile "symbol" t)`.

It therefore allows you to load a LISP file interactively in an extremely concise manner. `^L` returns the name of the file loaded. The suffix `#:system:lelisp-extension` is not added at the end of the filename if it is already present.

In LE-LISP, ^L could be defined in the following manner:

```
(dmc ^L ()
     ; ^L  :  to load a file
     (list 'libloadfile (readstring) t))


    e.g. ^Lfoo     is read as     (libloadfile "foo" t)
```

^A [*macro character*]

CONTROL-A has the same rôle as ^L, but it loads a compiled module with the modular compiler.

In LE-LISP, ^A could be defined in the following manner:

```
(dmc ^A ()
   ; to load a module
   (list 'loadmodule (readstring)))
```

^E [*macro character*]

Provides a way to call the pepe editor to treat the contents of a file, or the result of the evaluation of an expression.

```
^Ebar          is read as     (pepe bar)
^E(pretty foo)  "   "  "      (pepe (pretty foo))
^E^Pfoo bar     "   "  "      (pepe (pretty foo bar))
```

To call pepe on the sorted list of names of global functions

```
^E(sortl (maploblist (lambda (x)
                     (and (null (packagecell x))
                          (typefn x)))))
```

^F [*macro character*]

CONTROL-F lets you call the host system's editor on a file in which a function was defined, and reload the file upon exiting from the editor. The name of the host system's editor is stored in the #:system:editor global variable.

#:system:editor [*variable*]

The #:system:loaded-from-file property, set by the function-definition functions, indicates that files should be edited.

```
if the function bar was loaded from the file foo.ll

      e.g. ^Fbar

; is read as:

(progn (comline (catenate #:system:editor
                          " foo.ll"))
       (load "foo.ll" t))

; but if it was not loaded from a file, a temporary file <file>,
; having the name (gensym), is created and

     ^Fbar

; is read as

(progn (prettyf <file> bar)
       (comline (catenate #:system:editor " <file>")
       (load <file> t))
```

^P [*macro character*]

CONTROL-P calls the pretty-printer on one or several functions whose names are given.

In LE-LISP, ^P could be defined in the following manner:

```
(dmc ^P ()
     (cons 'pretty
           (implode (pname (catenate "("
                                     (readstring)
                                     ")")))))

^Pfoo     is read as   (pretty foo)
^Pfoo bar  "   "  "    (pretty foo bar)
```

! [*macro character*]

Used in front of a line while in interactive mode, this macro character executes the call (`comline` *rest of the line*). It thus provides a very concise way to send a command to the local operating system.

```
!ls -ls     ; is read as     (comline "ls -ls")
```

6.5 Basic output functions

All the following functions print their arguments into the output buffer. The expressions are not actually printed until the buffer is emptied.

This can happen at different times:

1. when the characters reach the right margin of the buffer
2. when the characters reach the end of the buffer
3. upon calls to functions that empty the buffer after placing something into it (`print`, `prinflush`, `terpri`).

In the first of these cases, the progammable interrupt `eol` is triggered, and an end-of-line is inserted into the output stream. In the second case, the `flush` interrupt is triggered. The buffer is then simply emptied. This can only happen if the right margin is beyond the end of the output buffer (see the `rmargin` function).

(prin s_1 ... s_n) [*function with a variable number of arguments*]

`prin` prints the S-expressions `s1` ... `sN` in the output buffer and returns the value of `sN`. This function cannot be simply described in LISP. The expressions will not actually be printed until the buffer is emptied. A call to `prin` without arguments, (`prin`), has absolutely no effect and returns `()`.

(print s_1 ... s_n) [*function with a variable number of arguments*]

This function is similar to `prin`, but it empties the output buffer immediately after printing the S-expression arguments therein, and begins a new line by triggering the `eol` programmable interrupt. A call to `print` without arguments, (`print`), empties the buffer, starts a new line and returns `()`.

In LE-LISP, `print` could be defined in the following manner:

```
(dmd print l
     `(prog1 (prin ,@l) (itsoft 'eol ())))
```

```
  ? (print (1+ 9) (cdr '(a b c))) ; form to evaluate
  10(b c)
  = (b c)                         ; value returned    ? (progn (prin 123) (print 'foo)) ; form
to evaluate
  123foo
  = foo                          ; value returned    ? (progn (repeat 12 (prin 'a))
  ?  (print)) ; form to evaluate
  aaaaaaaaaaaa
  = ()                          ; value returned    ? (print)

  = ()
```

(prinflush s_1 ... s_n) [*function with a variable number of arguments*]

This function is similar to **prin**, but empties the output buffer immediately after having printed the S-expression contained therein (by triggering the **flush** programmable interrupt). **prinflush** does not begin a new line. A call to the function with no arguments, (**prinflush**), empties the output buffer and returns **()**.

In LE-LISP, **prinflush** could be defined in the following manner:

```
(dmd prinflush l
     `(prog1 (prin ,@l) (itsoft 'flush ())))
```

```
? (defun quam (msg)
?  (prinflush msg) ; print the msg into the buffer, empty same
?  (read)) ; read a reply
= quam    ? (quam "number of vega's satellites")
number of vega's satellites ?
xxx ; xxx is the user reply
= xxx
? ; to print several items and then empty the buffer
? (defun prin-and-flush (exp n)
?  (repeat n (prin exp))
?  (prinflush))
= prin-and-flush
```

(terpri n) [*function with an optional argument*]

Skips **n** lines on the current output channel by calling the **print** function **n** times. It is possible to call **terpri** without an argument. The default value of **n** is **1**. Such a call is equivalent to both (**terpri** 1) and to (**print**).

In LE-LISP, **terpri** could be defined in the following manner:

```
(defun terpri n
    (repeat (if n (car n) 1)
            (print)))
```

(princn cn n) [*function with one or two arguments*]

Prints the **cn** character code **n** times into the output stream. If **n** is omitted, the code **cn** is only printed once. **princn** returns the internal character code **cn** as its value.

```
? (let ((cn #/a))
?  (repeat 10 (princn (incr cn))))
bcdefghijk
= t    ? (defun pyr (n1 n2)
?  (when (> n1 0)
```

```
?  (princn #\sp n1)
?  (princn #/* (1+ (* n2 2)))
?  (print)
?  (pyr (1- n1) (1+ n2))))
= pyr    ? (pyr 4 0)
? ; will produce
 *
 ***
 *****
 *******
= ()
```

(princh ch n**)** *[function with one or two arguments]*

This function is similar to the preceding one, except that the character `ch` is specified in the form of a mono-character symbol.

```
e.g. ; print ten '+' characters in the buffer
     (princh '|+| 10)
```

6.6 Controlling the output functions

6.6.1 Limitations on printing

In order to limit the printing of very long structures, and to avoid infinite print loops on circular or shared lists, three functions are available.

(printlength n**)** *[function with an optional argument]*

Provides a means of modifying the maximum number of list elements to be printed by a `print` or `prin` statement. Without an argument, the function returns the current maximum length value. By default, this value is `5000`. If a list contains more elements than the current maximum length value, the remaining elements are not printed, and three suspension points are printed instead. This function can be used to terminate output from functions which are iterating or recursively looping on a list's `cdr`. In order to deactivate the print length limitation, call `printlength` with 0 as its argument. The function returns the current maximum print length value as its result.

```
(printlength 6)       ⇒   6
'(1 2 3 4 5 6 7 8 9)  ⇒   (1 2 3 4 5 6 ...)
```

(printlevel n**)** *[function with an optional argument]*

Provides a means to modify the maximum print depth of a `print` or `prin` statement. (Without an argument, the function returns the current maximum depth value.) By default the maximum print depth is `100`. This value represents the maximum number of unmatched opening parentheses allowed in a list to be printed. Upon overload, the list causing the condition will not be printed,

and the ampersand character `&` is printed in its place. This function allows the results of functions which are iterating or recursively looping on a list's `car` to be printed. In order to deactivate this kind of limitation, call `printlevel` with zero as its argument. The function returns the current maximum print level value as its result.

```
? (printlevel 3)
= 3    ? '(defun foo (l)
?  (if (null (cdr l))
?  l
?  (foo (cdr l))))
= (defun foo (l) (if (null &) ...
```

Here are some examples of printing circular lists.

```
? (printlevel 10)
= 10    ? (printlength 50)
= 50    ? (setq l '(x y z))
= (x y z)    ? (rplacd (cddr l) l)
= (z x y z x y z x y z x y z x y z x y z x y z x y z x
y z x y z x y z x y z x y z x y z x y z x y z x ...    ? (rplaca l l)
= ((((((((((& y z & y z & y z & y z & y z & y z & y z &
y z & y z & y z & y z & y z & y z & y z & y z & y ...
```

(printline n) [*function with an optional argument*]

Provides a way to modify the maximum number of lines which will be printed by a `print` or `prin` statement. By default this value is `2000`. If an expression requires more than this number of lines, the last line will appear with suspension points at the end. In order to deactivate this limit, call `printline` with 0 as its argument. The function returns the current maximum number of print lines allowed as its result.

There is a library which lets LE-LISP functions be printed in a much more readable manner (see chapter 8), as well as a print library for circular or shared lists (see chapter 9).

6.6.2 Standard printing environment

The various LISP objects are printed in the output buffer according to the constraint that the external representation of an atomic object cannot be printed across a line boundary. This is the case for symbols, character strings and numbers.

S-expressions of the form `(quote s)` are printed as `'s` to improve readability.

The *unnameable* number in two's complement, '`-0`', is always written in the following manner:

#$8000 [*unnameable number*]

```
#$8000            ⟹    #$8000
(logshift 1 15)   ⟹    #$8000
```

Finally, all pointers that are not LE-LISP pointers are written in the following manner:

#<> [*non-Lisp pointer*]

```
? (vag '(#$7f . -1))
= #<>
```

On machines with IEEE-compatible floating-point processing, this value may be printed as the value 'negative Not-a-Number': that is, `-NaN.`.

Three system variables and two functions control various print modes:

#:system:print-for-read [*variable*]

This variable contains the print indicator of symbols and character strings.
If `#:system:print-for-read` has value `t`, symbols containing characters of type other than `cpname` are surrounded by vertical bar (`|`) characters when being printed. Character strings with this same property are surrounded by " quote characters. The `|` and " characters are themselves represented by `||` and "" (pairs of `|` or ") in the middle of these objects. Finally, the system prints a space between each argument of the `print`, `prin` or `prinflush` functions.

When this indicator is set, then, the print output can be re-read by the `read` function.

```
? (setq a '(foobar |bar foo| |12| "with ""quote"""))
= (foobar bar foo 12 with "quote")   ? (setq #:system:print-for-read t)
= t   ? a
= (foobar |bar foo| |12| "with ""quote""")
```

#:system:print-case-flag [*variable*]

This variable contains the output case indicator. If `#:system:print-case-flag` has the value `t`, symbols are printed in upper case. If it has the value `()` (the initial, default, value), symbols are printed in the form in which they were entered. Notice that in general the read indicator `#:system:read-case-flag` has the value `()` and that all symbols are therefore read in lower case.

Here are some examples of the distinction between upper and lower case in reading:

```
? (setq #:system:read-case-flag t)
= t   ? (setq a '(foobar FOOBAR FooBar))
= (foobar FOOBAR FooBar)   ? (setq #:system:print-case-flag t)
= T   ? a
= (FOOBAR FOOBAR FOOBAR)   ? (setq #:system:print-case-flag ())
= ()   ? a
= (foobar FOOBAR FooBar)
```

#:system:print-package-flag [*variable*]

This variable contains the package print indicator, which can take three values:

- `()`: Packages are not printed.
- `t` (default): Packages are printed in complete path (full) form: `#:pkg:sym`.

- 0: If the value of the package is equal to the value of the variable #:sys-package:colon, packages are printed in short form, using the macro character :.

```
? (setq #:system:print-package-flag t)
= t    ? (setq #:sys-package:colon 'user)
= user    ? '#:foo:bar
= #:foo:bar    ? ':foo
= #:user:foo    ? (setq #:system:print-package-flag ())
= ()    ? '#:foo:bar
= bar    ? ':foo
= foo    ? (setq #:system:print-package-flag 0)
= 0    ? '#:foo:bar
= #:foo:bar    ? ':foo
= :foo
```

(obase n) *[function with an optional argument]*

Provides a way to modify the numeric output conversion base. If it is called without the argument n, it returns the current output base. As in the case of the corresponding ibase function, the value of obase must be between 2 and 36, inclusive.

```
? 100
= 100    ? (obase 16)    ; obase always returns 10, do you see why?
= 10    ? 100
= 64    ? (obase 17)    ; why not?
= 10    ? (ibase 13)    ; conversion of base 13 to base 17!
= D    ? 23ab
= 107A    ? 34ab
= 18AD
```

(ptype symb n) *[function with one or two arguments]*

Provides a way to modify the value of the p-type of the symbol symb. Called without a second argument n, ptype will simply return the p-type of the symbol symb. The p-type is principally used by the LISP pretty-printer (see chapter 8) to determine the format to use to print the function value of a symbol. ptype returns the current value of the p-type of symb as its value.

6.6.3 Extending the printer

It is possible to redefine the way in which objects are printed, for the standard print functions all search for methods named prin associated with the type of the object to be printed. If such a method exists, it is called in lieu of the standard print function.

Warning: This function invocation takes place in the current printer context. In particular, all printer variables and variable-functions remain in their current state.

(pratom atom) [*function with one argument*]

The standard print function for atoms. `atom` can be of any type other than list. This function must be called from within a specific print method, and provides a way to return to the current print mode.

```
? ; modification of the representation of the symbol || to nil
? (defun #:null:prin (obj)
?  (pratom 'nil))
= #:null:pratom   ? ()
= nil
```

6.7 Input/output for lists

(explode s) [*function with one argument*]

Returns the list of all the internal character codes of the external representation of the expression `s`, which can be of any type whatsoever. `explode` returns the list of codes that would be printed if one asked for `s` to be printed by the `prin` function. (In fact there is not really an `explode` function, properly speaking, in the system; the output of the `prin` function is simply redirected.)

```
(explode -120)          ⟹   (45 49 50 48)
(explode '(car '(a b)))  ⟹   (40 99 97 114 32 39 40 97 32 98 41 41)
```

(explodech s) [*function with one argument*]

This is like the preceding function, except that it returns a list of atoms (mono-character symbols for letters and special characters, numbers for numerals).

In LE-LISP, `explodech` could be defined in the following manner:

```
(defun explodech (s)
    (mapcar 'ascii (explode s)))
```

```
(explodech -120)          ⟹   (- 1 2 0)
(explodech '(car '(a b)))  ⟹   (|(| c a r | | |'| |(| a | | b |)| |)|)
```

(implode ln) [*function with one argument*]

Suppose that `ln` is a list of `ascii` codes. `implode` returns the Lisp object which has this list of internal character codes as its external representation. `implode` is the inverse of `explode`, and provides a way to build new LISP objects from their external representations, analogous to the use of the `read` function. (There is not really a separate `implode` function in the system; the input to the `read` function is simply redirected.)

```
(implode '(45 50 51 55))     ⟹   -237
(implode (explode '(a b)))   ⟹   (a b)
```

(implodech s) [*function with one argument*]

This is similar to the preceding function, except that it uses a list of atoms (mono-character symbols for letters and special characters, numbers for numerals).

In LE-LISP, `implodech` could be defined in the following manner:

```
(defun implodech (s)
    (implode (mapcar 'cascii s)))
```

```
(implodech '(- 1 2 3))          ⟹   -123
(implodech (explodech '(a b)))  ⟹   (a b)
```

6.8 Input/output on character strings

Input/output can be thought of as a problem of transformations between an arbitrary object and a character string. The following functions let you carry out either *reading* operations within a character string that is considered as an input stream, or *writing* operations within a character string that is considered as an output stream. To obtain these functions, load the `stringio` module.

(with-input-from-string string . body) [*macro*]

Redefines the input stream as `string`, and evaluates sequentially the forms contained in `body`. After the evaluation of `body`, each `read` call performs a read operation within `string`. This function returns the value of the final evaluation inside `body`.

```
(with-input-from-string "12 34" (list (read) 55 (read)))  ⟹   (12 55 34)
```

(with-output-to-string string n . body) [*macro*]

Redefines the output stream as `string`, and evaluates sequentially the forms contained in `body`. After the evaluation of `body`, each `print` call performs a write operation within `string`. The first printing occurs at position `n` in `string`. This function returns the number of characters written in `string`.

```
(with-output-to-string (setq s "----") 0 (print 0))             ⟹   1
s                                                               ⟹   "0---"
(with-output-to-string (setq s "----") 1 (print 'a)(print 'b))  ⟹   2
s                                                               ⟹   "-ab-"
```

(read-from-string string) [*function with one argument*]

Uses the `read` function to perform reading operations within `string`, as if the latter were an input buffer. The returned value is that of `read`. If `string` is not in fact a string, the value returned is `string`.

In LE-LISP, `read-from-string` could be defined in the following manner:

```
(defun read-from-string (s)
    (if (stringp s)
        (with-input-from-string s (read))
       s))
```

```
(read-from-string "12")             ⟹   12
(type-of (read-from-string "12"))   ⟹   fix
(read-from-string 1.2)              ⟹   1.2
(read-from-string "as 12")          ⟹   as
```

(print-to-string s) *[function with one argument]*

Creates a string that represents the `s` object printed by means of `print`.

```
(print-to-string 1.2)       ⟹   "1.2"
(print-to-string '(a . b))  ⟹   "(a . b)"
```

6.9 Input/Output buffer management

All the read and print functions described up to this point work using buffers. Read functions read characters in the input buffer, and print functions write characters in the output buffer.

These buffers are filled and emptied by the system using the three programmable interrupts: `bol` (beginning-of-line), `eol` (end-of-line), and `flush`.

There is actually a separate buffer for each input or output channel, so I/O can be done on several channels in parallel. The buffer character string store is allocated automatically by the system when new channels are created.

Its length, 1024 characters, cannot be modified.

6.9.1 Input buffer

Introduction

A character string, `inbuf`, contains the contents of the current input buffer. Two indices into this string, `inmax` and `inpos`, indicate the end of the buffer and the position of the current read. `inpos` is the index of the next character to be read in the string.

If `inpos` becomes equal to `inmax` during a read operation, this means that all the characters available in the buffer have been read. At this point the programmable interrupt `bol` is triggered. The function invoked by this interrupt must refill the input buffer and set `inmax` to indicate how many characters were put into the buffer.

Manipulating the input buffer

All these functions manipulate the buffer of the current input channel, which is selected by the `inchan` function.

(inbuf n cn) *[function with one or two optional arguments]*

Returns the character string representing the current contents of the input buffer. The string-handling functions provide an easy means for manipulating this value. Here, for instance, is a particularly efficient way to implement the `readstring` function:

```
(defun readstring ()
    (when (= (inpos) (inmax))
          (itsoft 'bol ()))
    (prog1 (substring (inbuf) 0 (sub (inmax) 2))
           (inpos (inmax))))
```

`inbuf` accepts two optional arguments, giving access to the current input buffer contents. The following equivalences hold:

```
(inbuf n)          ==       (sref (inbuf) n)
(inbuf n cn)       ==       (sset (inbuf) n cn)
```

(inmax n) *[function with an optional argument]*

Called with the argument `n`, `inmax` provides a way to specify the maximum number of characters available in the input buffer. If `n` is not supplied, `inmax` returns the current number of characters in the input buffer. This function is used mainly inside a `bol` function, to specify the number of characters to be loaded into the buffer.

(inpos n) *[function with an optional argument]*

Called with the argument `n`, `inpos` changes the position of the read pointer (index) into the input buffer. Without an argument, it returns the current value of this index. This function is very rarely used.

Beginning-of-line programmable interrupt

When there are no more characters to read in the input buffer (that is, when `imax` characters have been read from the input buffer), the system triggers an interrupt called `bol`. The function associated with this interrupt fills the input buffer with characters read from the `inchan` channel, adds end-of-line characters `#\cr` and `#\lf` if necessary, and sets the `inmax` index. Of course it is possible to manage this interrupt yourself, by setting the `#:sys-package:itsoft` variable; in this case, you must fill the buffer and set the `inmax` variable appropriately. The system sets `inpos` back to zero after it processes a `bol` interrupt, so it is superfluous to do this, even if you are managing the process.

bol *[programmable interrupt]*

This programmable interrupt is triggered by the `read`, `readcn`, `peekcn`, `readstring` and associated functions, when the input buffer is empty: that is, when `inpos` = `inmax`.

(bol) [*function with no arguments*]

This function reads the next line from the terminal or from a file, puts it into the input buffer, and sets `inmax`. This is the function that permits input processing on lines typed by the user (character erasure with `#\bs` or `#\del`, line erasure with `#^X` or `#^U` and which echoes control characters with a `^` character followed by the printable character.)

The `bol` function uses the virtual terminal functions (`tyi`, `tycn`, etc.) to read characters and, unless otherwise programmed, echo them. It is thus usually sufficient to define a new terminal type in order to effect reads from new input streams (windows, serial ports, etc.).

What follows is a partial description of the `bol` function in LISP. We describe niether reads on files, nor the line editing aspects, nor the echoing of control characters. The interested reader can consult the source of the `edlin` line editor in the standard library.

```
(defun bol ()
    (if (fixp (inchan))
        (...code for reading files...)  ; cannot be described in Lisp
        (let ((max 0))
             (tystring (prompt) (slen (prompt)))
             (cond
                  (#:system:line-mode-flag
                   (setq max (tyinstring (inbuf))))
                  ((not #:system:real-terminal-flag)
                   (until (memq (sset (inbuf) max (tyi)) '(#\cr #\lf))
                          (incr max)))
                  (t
                     (let ((inbuf (inbuf))
                           (char 0))
                          (until (memq (setq char (tyi)) '(#\cr #\lf))
                                 (tycn char)
                                 (sset inbuf max char)
                                 (incr max))
                          (tynewline))))
             (sset (inbuf) max #\cr)
             (incr max)
             (sset (inbuf) max #\lf)
             (incr max)
             (inmax max))))
```

Here is an example `bol` function that counts the number of lines the user types into the terminal. It changes the prompt to display this value.

```
? (defvar #:numbering:line 0)
= #:numbering:line
? (defun #:numbering:bol ()
?     (when (null (inchan))
```

```
?           (prompt (catenate (incr #:numbering:line) "? ")))
?     (super-itsoft 'numbering 'bol ()))
= #:numbering:bol
? (setq #:sys-package:itsoft (cons 'numbering #:sys-package:itsoft))
= (numbering)
1?
2? 123
= 123
3? ()
= ()
4? (defun foo (x)
5?    (+ x x))
= foo
6? (foo 12)
= 24
7?
```

6.9.2 Output buffer

Introduction

A character string, `outbuf`, holds the contents of the current output buffer. An index into the string, `outpos`, marks the current write position (the position in the buffer where the next character should be written). Two indices, `lmargin` and `rmargin`, mark the left and right margins, respectively, of the output buffer.

If `output` becomes equal to `rmargin` during a write, this means that a program is trying to write a character beyond the right margin of the output buffer. The programmable interrupt `eol` is triggered at this point. The function invoked by this interrupt must empty the output buffer onto the current output channel (terminal or file), add an end-of-line mark, fill the output buffer with the *space* character, and set the `outpos` index to the left margin (`lmargin`).

If `outpos` reaches the end of the buffer during a write, this means that a program is trying to write a line longer than the buffer. This is possible when the right margin (`rmargin`) is beyond the end of the buffer. In this case, the `flush` programmable interrupt is triggered. The function invoked by this interrupt must empty the output buffer onto the current output channel, fill the output buffer with *space* characters, and set the `outpos` index to zero.

Manipulating the output buffer

`(lmargin n)` *[function with an optional argument]*

Sets the left print margin to `n`. By default, at system initialization, `lmargin` is `0`. If `n` is omitted, the left margin value is not changed. In any case, `lmargin` returns the value of the current left margin. It is used primarily by the LISP pretty-printer to automatically manage the placement of left braces in the printing of control structures.

`(rmargin n)` [*function with an optional argument*]

Sets the right print margin to `n`. By default, at system initialization, `rmargin` is `78`. If `n` is omitted, the right margin is not changed. In any case, `rmargin` returns the current value of the right margin. It is used primarily to set the line length for the output terminal being used (teletype, screen, printer, etc.).

It is possible to set the right margin outside the output buffer by executing:

```
(rmargin (add1 (slen (outbuf))))
```

In this case, the `eol` programmable interrupt is never called, but at the end of the buffer, the `flush` programmable interrupt will always be called.

`(outpos n)` [*function with an optional argument*]

Sets the current output buffer pointer (index) to `n`. This index is always treated as the first free character position in the buffer. `outpos` returns its current value.

`(outbuf n cn)` [*function with one or two optional arguments*]

Returns the character string representing the current contents of the output buffer. This string can be manipulated by the usual string-handling functions.

`outbuf` can optionally take two arguments, giving access to the contents of the output buffer. The following equivalences hold:

```
(outbuf n)       ==    (sref (outbuf) n)
(outbuf n cn)    ==    (sset (outbuf) n cn)
```

Programmable interrupts `eol` and `flush`

`eol` [*programmable interrupt*]

The `eol` programmable interrupt is triggered by the `prin`, `print`, `prinflush`, `princn` and `princh` functions when the current write position, `outpos`, goes beyond the right buffer margin, `rmargin`. The `print` and `terpri` functions invoke this programmable interrupt more explicitly.

The function invoked by this interrupt must perform the following operations:

- empty the output buffer, from position 0 through position `outpos`, onto the current output channel;
- put an end-of-line onto the channel;
- fill the output buffer with the *space* character;
- set the `outpos` index to the value of the left margin, `lmargin`.

(eol) [*function with no arguments*]

This function performs the standard processing of the `eol` programmable interrupt. What follows is a partial description of it in LISP. We do not describe printing onto files, which cannot be described in LISP.

Notice that this function uses the `tystring` and `tynewline` virtual terminal functions, to print to the terminal. This means that it is usually sufficient to define a new type of virtual terminal in order to redirect writes onto new streams (music synthesizers, windows, etc.).

```
(defun eol ()
    (if (fixp (outchan))
        (...code for writing to file...) ; cannot be described in lisp
        (tystring (outbuf) 0 (outpos)) ; empty the buffer to the terminal
        (tynewline)                    ; print an end-of-line
        (fillstring (outbuf) 0 #\sp (rmargin)) ; clean the buffer
        (outpos (lmargin))))          ; set to left margin
```

flush [*programmable interrupt*]

This programmable interrupt is triggered by the print functions when the current print position, `outpos`, reaches the end of the output buffer. This is possible if the right margin, `rmargin`, is greater than the length of the print buffer. The interrupt is also triggered explicitly by the `prinflush` function.

The function invoked by this by this interrupt must perform the following operations:

- empty the output buffer, from position 0 through position `outpos`, onto the current output channel;
- fill the output buffer with the *space* character;
- set the `outpos` index to the beginning of the output buffer.

(flush) [*function with no arguments*]

This function performs the standard processing of the `eol` programmable interrupt. What follows is a partial description of it in LISP. We do not describe writes to files, which cannot be described in LISP.

```
(defun flush ()
    (if (fixp (outchan))
        (...code for printing onto files...) ; cannot be described in lisp
        (tystring (outbuf) 0 (outpos)) ; empty the buffer to the terminal
        (fillstring (outbuf) 0 #\sp (rmargin)) ; clean up the buffer
        (outpos 0)))                   ; set to beginning of buffer
```

The `eol` and, less frequently, `flush` programmable interrupts can be redefined in order to extend the printing system.

The following function returns the list of lines printed by the evaluation of its body, in the form of a list of character strings:

```
(df stream-output #:system:f
    (let ((#:sys-package:itsoft 'stream-output)
          (#:system:l ()))          ; the resulting list
         (eprogn #:system:f)
         (nreverse #:system:l)))

(defun #:stream-output:eol ()
    (newl #:system:l (substring (outbuf) 0 (outpos)))
    (fillstring (outbuf) 0 #\sp (rmargin))
    (outpos (lmargin)))
```

6.10 Functions on I/O streams

LE-LISP manages I/O *streams* composed of:

- an I/O terminal (3 distinct streams)
- a number (in general 12) input or output files.

A *channel*, `chan`, is a number associated with an open stream. All open streams have channel numbers.

They are used to connect open streams to the current input or output stream. By convention, the "number" of the terminal's input stream is `()`, and the "numbers" of the terminal's two output streams (normal stream and virtual terminal stream) are `()` and `t`.

In LE-LISP, a *file specification*, `file`, is a character string (or an argument that can be converted into a string by using the `string` function) that has the same syntax as the host operating system's file-naming scheme.

```
"llib.virtty.ll"
"/usr/local/lelisp/foo.ll"
"lelisp$disk:[lelisp.llib]vdt.ll"
">udd>vlsi>lelisp>startup.ll"
"\lelisp\llib\pepe.ll"
```

LE-LISP has a standard virtual file management library, which is described in the following section. All arguments of type `file` in the following function descriptions can be virtual file descriptors.

All the functions described in this section will use the file system of the host system, which can return error codes in the case of file I/O problems. In these cases, the `errios` LISP error is triggered, which has the following default screen display:

```
** <fn> : i/o error : <n>
```

where `fn` is the name of the LISP function that was called, and `n` is the error code returned by the host file system.

Errors detected by LE-LISP are:

- `-1` function not implemented
- `-2` no more channels available
- `-3` incorrect channel number
- `-4` non-open channel

Positive-valued error codes are operating-system specific; in general the error number is `1`.

In certain cases it is possible to get the system to print the error message itself. The following variable can be set to choose this option.

#:system:print-msgs [*variable*]

If the value of this variable is different than 0, the host operating system will print (if it can) the error messages it encounters in I/O management.

6.10.1 Default directories and extensions

To permit functions to be written in a file system-independent manner, these variables are always initialized:

#:system:llib-directory [*variable*]

This variable contains the name of the directory where the standard LE-LISP library is to be found.

#:system:llub-directory [*variable*]

This variable contains the name of the directory where the user LE-LISP library is to be found.

#:system:lelisp-extension [*variable*]

This variable contains the default extension of LE-LISP source files.

6.10.2 Selecting I/O streams

(openi file) [*function with one argument*]

Opens the file named `file` *in read mode* and returns the associated channel number: `chan`.

```
(openi "foo.ll")  ⟹  11
```

(openib file) [*function with one argument*]

Opens the file named `file` *in binary read mode* and returns the associated channel number: `chan`. This file can only be read with the `readcn` and `readch` functions.

```
(openib "foo.dir")  ==>  11
```

(openo file) [*function with one argument*]

Opens the file named `file` *in write mode* and returns the associated channel number: `chan`. If this file already exists, it is destroyed and recreated as an empty file.

```
(openo "src.list")  ==>  10
```

(openob file) [*function with one argument*]

Opens the file named `file` *in binary write mode* and returns the associated channel number: `chan`. If this file already exists, it is destroyed and recreated as an empty file. Writes to such a file must be done with the `princn` or `princh` functions.

```
(openob "foo.o")  ==>  10
```

(opena file) [*function with one argument*]

Opens the file named `file` *in append mode* and returns the associated channel number: `chan`. If this file does not already exist, it will be created. If it does exist, its file pointer is positioned at the end of the file, so that all writes to the file will take place at the end of its current contents.

```
(opena "subsys.log")  ==>  9
```

(openab file) [*function with one argument*]

Opens the file named `file` *in binary append mode* and returns the associated channel number: `chan`. If this file does not already exist, it will be created. If it does exist, its file pointer is positioned at the end of the file, so that all writes to the file will take place at the end of its current contents. Writes to such a file must be done with the `princn` or `princh` functions.

```
(openab "subsys.lib")  ==>  9
```

(inchan chan) [*function with an optional argument*]

Ties the current input stream to the channel `chan`. This means that all succeeding reads (until the next call to `inchan`) will be on the channel `chan`. If no argument is given, `inchan` returns the number of the current input channel. An error `-4` occurs when the channel associated with the current input stream has been closed (with the `eof` or `close` function), and a read is requested, but no new input channel has been selected. This error also ties the current input stream to the terminal value, `()`.

(outchan chan) [*function with an optional argument*]

Ties the current output stream to the channel `chan`. This means that all succeeding writes (until the next call to `outchan`) will be on the channel `chan`. If no argument is given, `outchan` returns the number of the current output channel. An error `-4` occurs when the channel associated with the current output stream has been closed (with the `close` function), and a write is requested, but no new output channel has been selected. This error also ties the current output stream to the terminal value, `()`.

(channel chan) *[function with an optional argument]*

Returns a descriptor of channel number `chan`. This function gives control of the state of channels. If the argument `chan` is omitted, a list of all the channel descriptors is returned. A channel descriptor is a list of the form `(state name)`.

`state` is the value of the channel's state:

- `0` non-open channel
- `1` channel open in read mode
- `2` channel open in write mode
- `3` channel open in binary read mode
- `4` channel open in binary write mode

`name` is the filename associated with the channel, if one exists.

In order to get the number of available channels, evaluate:

```
(length (channel))
```

(close chan) *[function with an optional argument]*

Closes channel number `chan`, and always returns `t`. If no argument is given, `close` closes all open channels and then re-opens the terminal.

(input file) *[function with one argument]*

This function closes the current input stream, opens the file `file` in read mode, and ties the current input stream to its channel number.

Warning: This function loses track of the number of the previously open channel, and can only be used in the command loop at the terminal. An end-of-file will nevertheless raise the `errudt` error. (See the following section.)

In LE-LISP, `input` could be defined in the following manner:

```
(defun input (f)
    (when (fixp (inchan))
          (close (inchan)))
```

```
(if f
    (inchan (openi f))
    (inchan ()))
```

Here is a rather unorthodox way to load a file from the terminal, by printing the values of the evaluations:

```
(input <f>)
```

(output file) [*function with one argument*]

This function closes the current output stream, opens the file named `file`, and selects this channel for output.

Warning: This function loses track of the number of the previously open output channel and must be used only from the terminal to store the values of evaluations in a file. To restore output to the terminal, simply evaluate:

```
(output ())
```

In LE-LISP, `output` could be defined in the following manner:

```
(defun output (f)
    (when (fixp (outchan))
          (close (outchan)))
    (if f
        (outchan (openo f))
        (outchan ())))
```

Here is an example of file handling:

```
; copyfile function, which copies the entire contents of the
; input file <filin> to the output file <filout>, compacting
; at the same time the representations of lisp expressions
; in the latter.

   (defun copyfile (filin filout)
       (with ((inchan (openi filin))
              (outchan (openo filout))
              (obase 10)
              (lmargin 0)
              (rmargin 70))
             (let ((#:system:print-for-read t)
                   (#:system:print-package t))
                  (untilexit eof (print (read))))
             (close (outchan))
             filout))
```

```
? (with ((outchan (openo "foo.ll")))
?  (print "(defun foo (x) (+ x x))"))
= (defun foo (x) (+ x x))   ? (copyfile 'foo.ll 'bar.ll)
= bar.ll
```

6.10.3 End-of-file programmable interrupt

eof *[programmable interrupt]*

The end of a file causes the `eof` programmable interrupt, which receives as a parameter the number of the channel on which the end-of-file occurred.

The function invoked by this programmable interrupt must in general close the channel using the `close` function, and cause an escape with name `eof`.

(eof chan) *[function with one argument]*

This function closes channel number `chan` and propagates an escape with name `eof`. If this function is called while an S-expression is being read (for instance in the middle of reading a list), it causes a syntax error: *End of file during read*.

In LE-LISP, `eof` could be defined in the following manner:

```
(defun eof (n)
    (close n)
    (inchan ())
    (if #:system:in-read-flag
        (error 'read 'errsxt "eof during a read")
        (exit eof n)))
```

The `eof` programmable interrupt can be redefined by the user.

The following functions, for example, provide a way to load a sequence of files named `file`.1 .. `file`.n all at once. S-expressions can straddle two files.

```
(defvar #:sys-package:colon 'load-in-sequence)

(defvar :number)  ; the current extension
(defvar :file)    ; the filename

(de load-in-sequence (file)
    (let ((:number 1)
          (:file file)
          (#:sys-package:itsoft
             (cons 'load-in-sequence #:sys-package:itsoft)))
         (loadfile (catenate file "." :number) t)))
```

```
(de :eof (n)
    (if (not (probefile (catenate :file "." (1+ :number))))
        ; no more files:
        ; standard end-of-file processing
        (super-itsoft 'load-in-sequence 'eof (list n))
        ; go on to the next file in the sequence
        (print "closing " n ", opening " (1+ n))
        (close n)
        (inchan (openi (catenate :file "." (incr :number))))))
```

6.10.4 Functions on files

(probefile file) [*function with one argument*]

Returns `t` if `file` exists, and `()` if not.

In certain operating systems (like UNIX), a call to `probefile` with an empty argument—either `""` or `()`—will always return `t`.

In LE-LISP, `probefile` could be defined in the following manner:

```
(defun probefile (f)
    (let ((i (catcherror () (openi f))))
         (when (consp i)
               (close (car i))
               t)))
```

(renamefile ofile nfile) [*function with two arguments*]

Changes the name of file `ofile` to `nfile`, and returns `t`.

(copyfile ofile nfile) [*function with two arguments*]

Copies the contents of file `ofile` in the file `nfile`, and returns `t`.

(deletefile file) [*function with one argument*]

Deletes the file named `file`, and returns `t`.

(create-directory dir) [*function with one argument*]

Creates the directory `dir`, and returns `t`.

(delete-directory dir) [*function with one argument*]

Deletes the directory `dir`, and returns `t`.

6.10.5 `load` function and `autoload` mode

(loadfile `file i`**)** [*function with two arguments*]

Loads the file named `file` and evaluates all the expressions it contains. The indicator `i` is the value of the variable `#:system:redef-flag` (see the description of the definition functions) during the load. `loadfile` always returns `file` as its value.

In LE-LISP, `loadfile` could be defined in the following manner:

```
(defun loadfile (file redef?)
    (ifn (probefile file)
         (error 'loadfile "file unknown" file)
         (let ((#:system:loaded-from-file file)
               (#:system:redef-flag redef?)
               (#:sys-package:colon #:sys-package:colon)
               (#:system:in-read-flag ())
               (inchan (inchan)) )
              (inchan (openi file))
              (protect (untilexit eof (eval (read)))
                       (let ((in (inchan)))
                            (when in (close in)))
                       (inchan inchan) ))
         file ))
```

(load `file i`**)** [*special form*]

This function is the `fsubr` form of the preceding function. Moreover, the `i` argument is optional and equals `()` by default.

In LE-LISP, `load` could be defined in the following manner:

```
(df load (file . redef?)
    (loadfile file (car redef?)) )
```

(autoload `file` $\mathtt{sym}_1 \ldots \mathtt{sym}_n$**)** [*special form*]

Since it is tiresome to load several files of functions by hand at every execution, LE-LISP allows the use of `autoload` functions (which load automatically the first time they are called).

`file` is the name of a file that should be loaded automatically, using the `loadfile` function, if one of the symbols $\mathtt{sym}_1 \ldots \mathtt{sym}_n$ is to be evaluated as a function. File loading follows the call-by-necessity convention.

In LE-LISP, `autoload` could be defined in the following manner:

```
(df autoload (f . l)
    ; (autoload file at1 ... atn)
```

```
(mapc (lambda (at)
          (eval (list 'dm at 'l
                      (list 'remfn (kwote at))
                      (list 'load f t)
                      'l)))
        l)))
```

The call `(autoload pretty pprint)` results in the following definition of the `pprint` function:

```
(dm pprint l
    (remfn 'pprint)
    (load pretty t)
    l)
```

This results in the `macro` being called at the first evaluation of (`pprint` ...). This macro destroys itself (to avoid looping in case the function is not defined in the file), then evaluates (*load pretty t*), which silently loads the file `pretty`.

Since the value returned by the `macro` was the original call (`pretty` ...) itself, `eval` re-evaluates this form, in which `pretty` is now defined.

6.10.6 File access paths

An ordered set of prefixes or libraries can be defined, to avoid always having to explicitly specify complete filenames. This set is found in the variable:

`#:system:path` [*variable*]

This variable contains the directories that will be searched when file access is requested. At the initialization of LE-LISP, it contains the names of the system directories:

```
(defvar #:system:path
        (list ""
              #:system:llib-directory
              #:system:llub-directory
              #:system:llmod-directory
              #:system:llobj-directory
              #:system:lltest-directory
              #:system:virtty-directory
              #:system:virbitmap-directory
              #:system:system-directory))
```

`(search-in-path path file)` [*function with two arguments*]

Returns the complete filename of the file `file` if it is found in one of the directories named in `path`. If no such file is found, the function returns `()`.

In LE-LISP, `search-in-path` could be defined in the following manner:

```
(defun search-in-path (path file)
    (when path
          (let ((real-file (catenate (if (consp path)
                                         (car path)
                                         path)
                                     file)))
               (if (probefile real-file)
                   real-file
                   (when (consp path)
                         (search-in-path (cdr path)
                                         file))))))
```

(probepathf file) [*function with one argument*]

If the file named `file#:system:lelisp-extension` exists in one of the directories named in `#:system:path`, `probepathf` returns its complete name. Otherwise it returns `()`.

In LE-LISP, `probepathf` could be defined in the following manner:

```
(defun probepathf (file)
    (search-in-path
        #:system:path
        (catenate file #:system:lelisp-extension)))
```

6.10.7 Access to libraries

(libloadfile file i) [*function with two arguments*]

This function is similar to the `loadfile` function, except that it searches for the file named `file` in the various directories named in the `#:system:path` variable. It returns the actual name of the file loaded, or else raises an error if the file is not found.

In LE-LISP, `libloadfile` could be defined in the following manner:

```
(defun libloadfile (file redef?)
    (let ((real-file (probepathf file)))
         (ifn real-file
              (error 'libloadfile 'errfile file)
              (loadfile real-file redef?) )))
```

(libload file i) [*special form*]

This is the `fsubr` form of the preceding function. Furthermore, the `i` argument is optional, and has the value `()` by default.

In LE-LISP, `libload` could be defined in the following manner:

```
(df libload (file . redef?)
    (libloadfile file (car redef?)))
```

For an example, try `(libload hanoi)`.

`(libautoload file sym`$_1$ `... sym`$_n$`)` [*special form*]

This function is similar to the `autoload` function, except that files will be loaded with the `libload` function, instead of with `load`.

6.11 Event loop

The LE-LISP **event loop** is a functional interface that enables you to associate processing functions with various kinds of input events. This event loop has no problems with *interrupts*. While awaiting various types of events, the event loop handles all conventional interrupts.

In a LE-LISP program, a single event loop can take care of several different kinds of input:

- Graphic input generated by the mouse and/or the keyboard.
- Files in the UNIX sense: socket or pipe.
- MAILBOX in the VMS sense.

The implementation of the event loop is based upon the so-called *Virtual Terminal* of LE-LISP, described in chapter 15 of the present manual. In UNIX, the `select` device is brought into play. In VMS, the AST device is used.

Today, the functions described here only work in either UNIX or VMS. The event loop uses a virtual terminal of the `#:tty:evloop` type. In fact, the events loop cannot be used with a virtual terminal that is not of the `#:tty:evloop` type. An exception is the virtual terminal used by AÏDA, which includes a standard integration of the event loop.

To use the event loop of LE-LISP, load the `evloop` module.

`event-loop` [*feature*]

This feature indicates that the event-loop functions are loaded. In other words, the `evloop` module is present.

6.11.1 Functions

`(evloop-init)` [*function with no arguments*]

Initializes the event loop and sets the virtual terminal `#:tty:evloop` as the current terminal. This function should only be called once.

(evloop-stop) [*function with no arguments*]

Stops the event loop and sets the default terminal of the `tty` system as the virtual terminal.

(evloop-restart) [*function with no arguments*]

Restarts the event loop and reassigns the `#:tty:evloop` virtual terminal as the current terminal.

(evloop-disallow-tty-input) [*function with no arguments*]

Inhibits the processing by the event loop of standard user input. This function is only effective if the event loop has been initialized and is active.

(evloop-allow-tty-input) [*function with no arguments*]

Enables the processing by the event loop of standard user input. This function is only effective if the event loop has been initialized and is active.

(evloop-add-input `fd manage-fct arg-to-use`) [*function with three arguments*]

Adds the UNIX file descriptor `fd` to the event loop. Imagine that the event loop is active, and that there's something to read in the case of the `fd` file descriptor. In such a situation, the `manage-fct` function is executed with `arg-to-use` as its argument. This function has no effect if the event loop has not been initialized.

(evloop-remove-input `fd`) [*function with one argument*]

Removes the `fd` file descriptor from the list of file descriptors handled by the event loop. Before deleting `fd`, evaluate `(evloop-input-managedp fd)` to make sure that `fd` is processed by the event loop.

(evloop-change-manage-function `fd new-manage-fct new-arg-to-use`) [*function with three arguments*]

If the file descriptor `fd` is already being processed in the event loop, this function replaces the function to be executed for this descriptor by the function `new-manage-fct` with `new-arg-to-use` as its argument.

(evloop-select) [*function with no arguments*]

This function is not required for standard use of the event loop module. It is only needed by users who wish to implement new types of virtual terminals. This function represents the engine of the event loop. It is called regularly by the system. This is the function that blocks the program while waiting for one or several file descriptors to be ready for reading, at which time their associated functions are executed.

(evloop-readp fd) [*function with one argument*]

Returns `t` if an input is ready for reading. Otherwise, it returns `()`. This is a generalized version of the `eventp` function of the Virtual Bitmap.

(evloop-input-managedp fd) [*function with one argument*]

Returns `t` if the `fd` input is being processed by the event loop. The argument `fd` is typically an integer that designates a file descriptor under UNIX.

(evloop-initialized-p) [*function with no arguments*]

Returns `t` if the event loop has already been initialized.

(evloop-wait) [*function with no arguments*]

Causes the process to wait for an input. After the wait, when a descriptor is ready, the function returns an expression of the form `(fd . (function . arg))`. Here, `fd` is the file descriptor, `function` is the associated processing function and `arg` is an argument. If more than one descriptor is ready to be read, the function returns a list of expressions of the form `(fd . (function . arg))`.

(evloop-set-timeout secs millisecs) [*function with two arguments*]

This function allows you to limit the waiting time of the process in the `evloop-select` and `evloop-wait` functions. This waiting time is indicated in `secs` (seconds) and `millisecs`. When the time has run out, the function specified by `evloop-set-timeout-handler` (described below) is executed. The arguments `secs` and `millisecs` must be positive integers. If both of them are given as zero, this represents an infinite waiting time. So, `evloop-set-timeout` lets you periodically trigger processing operations while awaiting an event. Unlike the `clockalarm` function of LE-LISP, the processing is not triggered during the execution of any other processing apart from the calls to `evloop-select` and `evloop-wait`.

(evloop-set-timeout-handler handler) [*function with one argument*]

The `handler` argument must be either `()` or a function with no argument that is to be called when the time specified by the `evloop-set-timeout` function has run out.

Output processing

We now describe functions that let you add output (files, file descriptors and other entities) into the events loop.

(evloop-add-output fd manage-fct arg-to-use) [*function with three arguments*]

Adds the UNIX file descriptor `fd` into the events loop. If the events loop is active as soon as the file descriptor `fd` is ready, the `manage-fct` function is executed with `arg-to-use` as its argument. This function has no effect if the events loop has not been initialized.

(evloop-remove-output fd) *[function with one argument]*

Removes the file descriptor `fd` from the list of file descriptors being processed for output by the event loop.

(evloop-change-output-manage-function fd new-manage-fct new-arg-to-use) *[function with three arguments]*

If the file descriptor `fd` is already being processed in the event loop, this function replaces the function to be executed by the function `new-manage-fct` with `new-arg-to-use` as its argument.

Processing the Virtual Bitmap

Three functions within this module allow you to manipulate graphic input for the VB (virtual bitmap):

(evloop-add-display display manage-fct) *[function with two arguments]*

Lets you record input (graphic events) associated with the `display` device. When there is a graphic event from the `display` device, the `manage-fct` function is executed with `display` as its argument.

(evloop-remove-display display) *[function with one argument]*

Removes the `display` device from the inputs processed by the events loop.

(evloop-display-managed-p display) *[function with one argument]*

Returns `t` when the display device is being processed by the events loop. If not, the function returns `()`.

6.11.2 Technical notes

Using the events loop in Unix

Most of the functionalities of the events loop are implemented in C code under UNIX. This C code is stored in the `evloop.c` file, which is included in the standard LE-LISP distribution. Before using the events loop, this code should be compiled and present in the LE-LISP binary. This code is not present in a standard binary, but it exists by default when the Virtual Bitmap has been implemented by means of `X11`.

Portability of the events loop under VMS

All the functions that we have just described have been implemented under the VMS operating system. The UNIX concept of a file descriptor has been replaced by the notion of an input/output channel. Under VMS, there are several input types, but only the `MAILBOX` and graphic events are handled by LE-LISP. The VMS user should create the `MAILBOX` and record the associated channel in the loop. It is important to understand that the implementations of these two systems are based upon different principles:

- Under UNIX, the implementation is based upon the `select` functionality.
- Under VMS, the implementation is based upon the AST functionality.

Both modules nevertheless behave identically.

6.11.3 Precautions

An important remark needs to be made concerning the use of the event loop in conjunction with the LE-LISP `save-core` functionality. The use of this module means that the event loop must be reinitialized each time you launch LE-LISP. So, the following actions must be carried out:

1. Use `evloop-remove-input` and `evloop-remove-output` to remove all input/output recorded in the loop.
2. Use `evloop-stop` to stop the loop.
3. Call the `save-core` function.
4. Initialize the loop with the `evloop-init` function.
5. Once again, record all input/output.

Example

Let us look at an example of the use of the events loop with the LE-LISP VB device (virtual bitmap) under `X11`. We need the following utility function that reads all events that are waiting in the `display` device and displays them on the screen:

```
(defun display-manage-events (display)
   (current-display display)
   (while (eventp) (print (read-event))))
```

We launch LE-LISP:

```
% lelispX11
; Le-Lisp (by INRIA) version 15.25 (17/nov/91)   [sony]
; Systeme standard sur X11-windows : ven  26 nov 91 14:25:52
= (31bitfloats abbrev callext compiler date debug defstruct edlin loader
mc68881 messages microceyx pathname pepe pretty setf virbitmap virtty)
```

Load the module that defines the events loop:

```
? (loadmodule 'evloop)
= evloop
```

Initialize the display device:

```
? (setq display (bitprologue))
= #<#:display:x11 X11 trocadero:0.0>
```

Get the file descriptor (socket) associated with `display`:

```
? (evloop-init)
= t    ? (setq fd (send 'file-descriptor display))
= 3
```

Add it to the standard events loop:

```
? (evloop-add-file-descriptor fd 'display-manage-events display)
= t
```

Create a window:

```
? (setq w (create-window 'window 0 0 200 200 "lisp window1" 1 1))
= #<#:image:rectangle:window lisp window1>
```

Initialize the display device on a second screen:

```
? (setq display1 (bitprologue '|X11| "host1:0"))
= #<#:display:x11 X11 trocadero:0.0>
```

Get the file descriptor (socket) associated with the second display:

```
? (setq fd1 (send 'file-descriptor display))
= 4
```

Add it to the standard events loop:

```
? (evloop-add-file-descriptor fd1 'display-manage-events display)
= t
```

Create a second window on the screen of the host machine.

```
? (setq w1 (create-window 'window 0 0 200 200 "lisp window2" 1 1))
= #<#:image:rectangle:window lisp window2>
```

All mouse/keyboard events associated with the windows `w` and `w1` will be displayed, as well as anything typed by the user in the LE-LISP startup window:

```
#:event:#[enterwindow-event #<#:image:rectangle:window lisp window1>
() 175 194 167 169 () ()]
#:event:#[down-event #<#:image:rectangle:window lisp window1>
0 143 154 135 129 () ()]
#:event:#[up-event #<#:image:rectangle:window lisp window1>
0 143 154 135 129 () ()]
#:event:#[leavewindow-event #<#:image:rectangle:window lisp window1>
() 279 254 271 229 () ()]
#:event:#[enterwindow-event #<#:image:rectangle:window lisp window2>
() 174 177 166 152 () ()]
#:event:#[ascii-event #<#:image:rectangle:window lisp window2>
97 193 147 185 122 () ()]
```

```
#:event:#[leavewindow-event #<#:image:rectangle:window lisp window2>
() 225 180 217 155 () ()]
```

Terminate the session:

```
? (end)
May Le-Lisp be with you ...
```

6.12 Virtual file system

This section describes the virtual file system implemented in LE-LISP . This facility provides a way to manipulate files without regard to the host operating system. It is easily extensible to new operating systems.

In order to gain access to functions described in this section, the module named `path` must be loaded. Since this module redefines some standard I/O functions, it should not be loaded into the system more than once.

First we describe the new LE-LISP object type `pathname`, and the meaning of its fields. Then we describe the field access and manipulation functions. Finally we describe some higher-level functions, and the means to redefine the access functions.

`pathname` [*feature*]

This flag indicates whether the functions treating pathnames are loaded into the system: that is, whether the *path* module is present.

6.12.1 Pathnames

The pathname object is a LE-LISP structure that represents file names without regard to particular operation system conventions. In some systems, these objects could represent other structures, for example interprocess communications channels, etc.

Programmers wanting to use these primitives to increase the portability of their code will have to restrain the syntax of pathnames to a subset common to and compatible with all the operating system environments in which LE-LISP is used. In the rest of this chapter, any functionality which could be somehow non-portable to any system, will be explicitly described as such.

Pathnames do not necessarily represent a complete filename. Some fields in the specification of a name could be missing, and in such cases it is useful to designate directories or default field values during evaluation.

`pathname` [*structure*]

This structure describes the LE-LISP object representing operating system-independent pathnames.

In LE-LISP, `pathname` could be defined in the following manner:

```
(defstruct pathname host device directory name type version)
```

- The `host` field should hold the name of the machine on which the file physically exists, in a network-based environment. This field can also contain access control information. The exact syntax of this field depends on the OS and is therefore not portable. For files on machines not supporting network-based file systems, the value of this field is `()`; in the other case, this field contains a character string.
- The `device` field contains the name of the disk on which the file is stored, when the operating system explicitly designates the peripheral device (as do `vms` and `ms/dos`, for instance). If the OS does not support the notion of a file tree, this field has the value `()`; otherwise, it should contain a character string.
- The `directory` field contains the list of directories and sub-directories where files can be found. Each directory name is a character string. This list is in the "intuitive" order (i.e., the first element is the root of the hierarchy), and can designate either an absolute tree, or a relative access path. (See the section on access paths.)
- The `name` field contains the name of the file, in the traditional sense of the term. It is stored as a character string.
- The `type` field contains the character string designating the extension of the filename. This extension indicates in a general way the nature of the contents of the file. Thus, in most LE-LISP implementations, this field has the value `ll` for a LE-LISP source file.
- The `version` field contains a version number. The exact version numbering convention depends on the operations system, but generally the higher the file version number, the more recent the vintage of the version of the file.

(pathname string) *[function with one argument]*

Converts the character string `string` corresponding to a filename in the syntax of the operating system into an object of type `pathname`.

```
? ; Under UNIX, with #:system:print-for-read set to t.

? (pathname "/udd/lelisp/llib/path.ll")
= #:pathname:#[ () () ("udd" "lelisp" "llib") "path" "ll" () ]
? ; The same thing on a VAX/VMS system.

? (pathname "lelisp$disk:[lelisp.llib]path.ll;1")
= #:pathname:#[ () "lelisp$disk:" ("lelisp" "llib") "path" "ll" 1]
```

(pathnamep s) *[function with one argument]*

This predicate returns `()` if its argument is not an object of type `pathname`, and `t` otherwise.

```
? ; Example on VAX/VMS..
? (pathnamep (pathname "[lelisp.llib]path.ll"))
= t    ? ; ..same thing on UNIX.
```

```
? (pathnamep (pathname "/usr/local/lelisp/llib/path.ll"))
= t
```

(equal-pathname path1 path2) [*function with two arguments*]

Returns `t` if the two arguments are equivalent pathnames.

Examples under UNIX:

```
? (equal-pathname #u"/a/./b" #u"/a/b")
= t    ? (equal-pathname #u"/a/../b" #u"/b")
= ()
```

This function considers that `/a/../b` and `/b` are different UNIX pathnames because of the possible presence of symbolic links.

(namestring path) [*function with one argument*]

Returns a character string that is the external representation, for the host operating system, of the argument `path`. This external form is obtained by combining the pathname elements according to the operating system-specific syntax rules.

Warning: No syntax validation is carried out by LE-LISP. Obviously, invalid filenames can be generated.

#p [*#-macro*]

This macro character returns the path that follows it, parsing it according to the host operating system syntax.

In LE-LISP, `#p` could be defined in the following manner:

```
(defsharp p ()
          (ncons (pathname (read))))
```

#u [*#-macro*]

This macro character performs a function similar to the one described immediately above, but it parses the `pathname` according to UNIX syntax on *all* operating systems. It offers a portable way to enter pathnames.

```
? ; Example on a VAX/VMS system.
? #u"/usr/lelisp/llib/path.ll"
= #p"[usr.lelisp.llib]path.ll"
```

(#:pathname:prin path) [*function with one argument*]

Defines the print method for objects with type `pathname`. Its behavior depends on the value of the `#:system:print-for-read` indicator. If this latter has the value `()`, pathnames are

printed in the system-specific form of the host operating system. If it is not equal to `()`, the pathnames are printed as LE-LISP structures. This feature allows pathnames to be saved in files in a machine-independent way. The standard form is, however, more convenient to the user who knows the local operating system filename syntax.

```
? ; Example on a UNIX system.
? (setq #:system:print-for-read ())
= ()
? #p"/usr/local/lelisp/llib/test.ll"
= #p"/usr/local/lelisp/llib/test.ll"
? (setq #:system:print-for-read t)
= t
? #p"/usr/local/lelisp/llib/test.ll"
= #:pathname:#[ () () ( "usr" "local" "lelisp" "llib") "test" "ll" () ]
```

6.12.2 Relative access paths

Most operating systems organize files in a hierarchy of directories and sub-directories, and give the user the means to move around in the tree. The definition of a 'current working directory' is fundamental to many systems, and filename syntax is then relative to this current directory.

LE-LISP allows keywords to be substituted for certain pathname fields so that these relative access paths can be formed.

Warning: Relative access paths are not necessarily portable to every operating system on which LE-LISP is used.

`#:pathname:up` [*indicator*]

This keyword can be placed in the `directory` field of a `pathname` structure. It signifies that the filename is to be relative to the next higher level of the file-system hierarchy. In the first position of the `directory` field, it indicates that the `pathname` is relative to the user's current working directory.

On a UNIX system with `#:system:print-for-read` set to `t`:

```
? #p"../foo/bar.ll"
= #:pathname:#[ () () (#:pathname:up "foo") "bar" "ll" () ]
```

`#:pathname:current` [*indicator*]

This keyword can only be placed in the first position of the list in a `pathname`'s `directory` field. It indicates that this `pathname` starts in the user's current directory, instead of beginning at the root of the file system.

On a VAX/VMS system:

```
? #p"[.foo]bar.ll"
= #:pathname:#[ () () (#:pathname:current "foo") "bar" "ll" ()]
```

`#:pathname:wild` [*indicator*]

This keyword can be placed in any field in a `pathname`, and will be replaced by a particular value by the operating system. Pathnames containing this indicator cannot be passed as arguments directly to LE-LISP file access functions. First a `wildcard` function (described below) must be executed to convert the wildcard into a particular value.

In the following example, we select all files in `../foo` that have `toto` as their name, with any extension whatsoever. On a UNIX system with `#:system:print-for-read` set to `t`:

```
? #p"../foo/toto.*"
= #:pathname:#[ () () (#:pathname:up "foo") "toto" #:pathname:wild () ]
```

`(wildcard` `path)` [*function with one argument*]

`(expand-pathname` `path)` [*function with one argument*]

This function takes a pathname argument containing fields with `#:pathname:wild` values, or a character string in the host system file-naming syntax, and returns the list of expanded pathnames of files on the host machine's file system that match the pattern. If no such file is found, it returns `()`.

On a UNIX system:

```
? (expand-pathname #p"../*/*.ll")
= (#p"../lisp/path.ll" #p"../demo/games.ll")
```

On an OS/2 machine, the catalogue cannot be expanded:

```
? (expand-pathname #p"\lelisp\llib\path*")
= (#p"\lelisp\llib\path.ll")
```

6.12.3 Pathname manipulation functions

Arguments to these functions can be either pathnames or character strings representing the external file specifications (i.e., filenames) for the host operating system. In the second case, an intermediate conversion into pathname representation is done. It is thus recommended, for reasons of efficiency, to avoid making frequent calls to these functions with character string (i.e., host filename) arguments.

`(pathname-host` `path)` [*function with one argument*]

This function returns the `host` field of its argument, or `()`. This field has the type `string`.

`(pathname-device` `path)` [*function with one argument*]

This function returns the `device` field of its argument, or `()`. This field has the type `string`.

`(pathname-directory` `path)` [*function with one argument*]

This function returns the `directory` field of its argument, or `()`. This field is a list of character strings, or the empty list `()`.

`(pathname-name path)` [*function with one argument*]

This function returns the `name` field of its argument, or `()`. This field has the type `string`.

`(pathname-type path)` [*function with one argument*]

This function returns the `type` field of its argument, or `()`. This field has the type `string`.

`(pathname-version path)` [*function with one argument*]

This function returns the `version` field of its argument, or `()`. This field has the type `fix`.

All the fields of objects of `pathname` type can be modified by means of the following functions:

`(set-pathname-host path host)` [*function with two arguments*]

Lets you modify the `host` field of the argument `path` of `pathname` type. The function returns the new value of the field. The `host` argument must be either of `string` type or equal to `()`.

`(set-pathname-device path device)` [*function with two arguments*]

Lets you modify the `device` field of the argument `path` of `pathname` type. The function returns the new value of the field. The `device` argument must be either of `string` type or equal to `()`.

`(set-pathname-directory path dir)` [*function with two arguments*]

Lets you modify the `directory` field of the argument `path` of `pathname` type. The function returns the new value of the field. The `dir` argument must be a list whose elements are character strings or one of the symbols `#:pathname:up`, `#:pathname:current` or `#:pathname:wild`.

`(set-pathname-name path name)` [*function with two arguments*]

Lets you modify the `name` field of the argument `path` of `pathname` type. The function returns the new value of the field. The `name` argument must be either of `string` type, equal to `()`, or the `#:pathname:wild` symbol.

`(set-pathname-type path type)` [*function with two arguments*]

Lets you modify the `type` field of the argument `path` of `pathname` type. The function returns the new value of the field. The `type` argument must be either of `string` type or equal to `()`.

`(set-pathname-version path version)` [*function with two arguments*]

Lets you modify the `version` field of the argument `path` of `pathname` type. The function returns the new value of the field.

`*default-pathname-defaults*` [*variable*]

This variable is initialized to the empty pathname (all fields equal to `()`). It is used to indicate the default values of various fileds during pathname creation and merging operations.

`(make-pathname` element$_1$ ... element$_n$`)` [*function with a variable number of arguments*]

This function takes a variable number of arguments, and builds a pathname. The arguments must be supplied in the following order: `host`, `device`, `directory`, `name`, `type` and `version`. If an argument is missing or has the value `()`, the value for the same field in the global variable `*default-pathname-defaults*` is used.

The arguments are as follows:

- `host` is either a character string or `()`.
- `device` is either a character string or `()`.
- `directory` is either a list of character strings or `()`.
- `name` is either a character string or `()`.
- `type` is either a character string or `()`.
- `version` is either an integer or `()`.

On a UNIX system with `#:system:print-for-read` set to `()`:

```
? *default-pathname-defaults*
= #p""
? (make-pathname () () () "toto" "ll" ())
= #p"toto.ll"
? (setq *default-pathname-defaults* #p"/udd/lelisp/")
= #p"/udd/lelisp/"
? (make-pathname () () () "toto" "ll" ())
= #p"/udd/lelisp/toto.ll"
```

`(file-namestring` path`)` [*function with one argument*]

This function returns the character string corresponding to the external specification of the filename, type, and version part of its (pathname) argument for the host file system.

On a UNIX system with `#:system:print-for-read` set to `()`:

```
? (file-namestring #p"/udd/lelisp/test.ll.3")
= test.ll.3
```

`(host-namestring` path`)` [*function with one argument*]

This function returns the character string representing the file system's host machine, or if this field is equal to `()`, the empty character string `""`.

On a VAX/VMS machine:

```
? (host-namestring #p"asterix::[lelisp.llib]startup.ll")
= asterix::
```

(device-namestring path) [*function with one argument*]

This function returns the character string representing the disk on which the file system resides, or if this field is equal to `()`, the empty character string `""`.

On a VAX/VMS machine:

```
? (device-namestring "lelisp$disk:[lelisp.llib]startup.ll")
= lelisp$disk:
```

(directory-namestring path) [*function with one argument*]

This function returns the character string representing the directory tree stored in the `directory` field of its argument, in the syntax of the host operating system.

On a UNIX system with `#:system:print-for-read` set to `()`:

```
? (directory-namestring #p"/udd/lelisp/toto.ll")
= /udd/lelisp/   ? ; Example on a VAX/VMS machine.
? (directory-namestring #p"lelisp$disk:[lelisp.llib]startup.ll;1")
= [lelisp.llib]
```

(merge-pathnames path default-path) [*function with two arguments*]

This function merges two pathnames. It returns a pathname in which the fields missing from the first argument are filled in with values from the corresponding fields from the second argument.

On a UNIX system with `#:system:print-for-read` set to `()`:

```
? (merge-pathnames #p"test.ll" #p"/usr/spool/uucp/toto.ll.3")
= #p"/usr/spool/uucp/test.ll.3"
```

(combine-pathnames path1 path2) [*function with two arguments*]

This function has two arguments: the pathnames `path1` and `path2`. It returns the pathname that results from their composition. The composition of two pathnames is obtained in the following manner:

- Choose the `host` field of `path2`. If it is empty, then choose the `host` field of `path1`.
- Choose the `device` field of `path2`. If it is empty, then choose the `device` field of `path1`.
- If the `directory` field of `path2` describes a relative path, then concatenate the `directory` fields of the two pathnames. If the `directory` field of `path2` is not relative, then take it

as it stands. The resulting `directory` field might be simplified in both cases.

- The `name`, `type` and `version` fields are taken solely from `path2`. Those of `path1` are disregarded.

```
? (combine-pathnames #u"/home/" #u"foo.sh")
= #p"/home/foo.sh"
? (combine-pathnames #u"/usr/lelisp/llib/" #u"../llobj/pretty.lo")
= #p"/usr/lelisp/llobj/pretty.lo"
? (combine-pathnames #u"/usr/lelisp/llib/path.ll" #u"../llmod/path.lm")
= #p"/usr/lelisp/llmod/path.lm"
? (combine-pathnames #u"/usr/lelisp/llib/path.ll"
?                    #u"/usr/lelisp/llobj/path.lo")
= #p"/usr/lelisp/llobj/path.lo"
```

This function is different from `merge-pathnames` in that the latter is used to fill in the missing fields of a pathname, whereas `combine-pathnames` is a composition of two things:

- Certain fields indicate the place where the file is located: `host`, `device` and `directory`.
- Other fields, taken from the second pathname, indicate the nature of what the file contains: `name`, `type` and `version`.

Note that, for any two pathnames `p1` and `p2` whatsoever, the evaluation

```
? (probefile (combine-pathnames p1 p2))
```

produces exactly the same result as

```
? (with ((current-directory p1)) (probefile p2))
```

(enough-namestring path default) [*function with one or two arguments*]

Builds the shortest possible character string that differentiates the pathname `path` from the default value `default`, which is an optional parameter. If the second argument is not supplied, values stored in `*default-pathname-defaults*` are used instead.

On a UNIX system with `#:system:print-for-read` set to `()`:

```
? (enough-namestring #p"/udd/lelisp/toto.ll" #p"/udd/lelisp/foo.bar")
= toto.ll
```

(true-pathname path) [*function with one argument*]

Some operating systems allow users to define special symbols which can appear thereafter in filenames. Examples are UNIX environment variables or VMS or TOPS20 logical names. The `true-pathname` function translates a given pathname (which may well contain such a symbol) into its real ('hard') pathname. The translation scheme that is used depends on the system.

On UNIX systems, shell variables can be used in either the head or tail positions in a filename: that is, either as the first or last elements in the filename. LE-LISP will search the `shell`

environment for variables in these positions that begin with the `$` character. Consequently, the UNIX dereferencing syntax must be respected.

On VMS systems, the `pathname` is examined to see whether any of its fields correspond to logical names. If so, they are translated. This translation is not performed recursively. If an exact (physical or 'hard') filename is required, it might be necessary to apply the `true-pathname` function to the result.

On a UNIX system with `#:system:print-for-read` set to `()`:

```
? (true-pathname #p"$HOME/.lelisp")
= #p"/nfs/home/rodine/.lelisp"   ? (true-pathname #p"/dev/$TERM")
= #p"/dev/xterm"
```

On a VAX/VMS system:

```
? (true-pathname "sys$login:test.ll")
= #p"amon$dra2:[dana]test.ll"
```

If this functionality is not offered by a given operating system, `true-pathname` returns its argument unchanged.

(user-homedir-pathname) [*function with no arguments*]

This function returns the pathname corresponding to the notion of *Home-directory* for operating systems that support this. In other cases, it returns the name of a directory, which must always be valid.

On a UNIX system with `#:system:print-for-read` set to `()`:

```
? (user-homedir-pathname)
= #p"/nfs/home/rodine"
```

On a VAX/VMS system:

```
? (user-homedir-pathname)
= #p"sys$login:"
```

(control-file-pathname name path) [*function with one or two arguments*]

This function takes a character string as its first argument, and returns the associated pathname found in the user's home directory. The aim of this function is to provide a standard means of creating initialization control files for applications written in LE-LISP. The default name of this file depends on the operating system:

- Under UNIX, it is `$HOME/.`<*name*>.
- Under VMS, it is `SYS$LOGIN:.`<*name*>.
- Under OS/2, it is `$HOME`<*name*>`.ini`.

If the `path` argument is supplied, it must be the name of the initialization file, of `string` or

`pathname` type, for the `name` product.

On a UNIX system:

```
? (control-file-pathname 'aida)
= #p"/udd/chatelet/dana/.aida"
```

On a VAX/VMS system:

```
? (control-file-pathname 'aida)
= #p"sys$login:.aida"
```

On an OS/2 system:

```
? (control-file-pathname 'aida)
= #p"\udd\chatelet\dana\aida.ini"
```

An imposed name on a UNIX system:

```
? (control-file-pathname 'csh #p".cshrc")
= #p"/udd/chatelet/dana/.cshrc"   ? (control-file-pathname 'csh)
= #p"/udd/chatelet/dana/.cshrc"
```

(temporary-file-pathname name) [*function with one argument*]

This function provides a standard way to create a file in a temporary storage area provided by the operating system, with the name given as argument. It returns a pathname.

On a UNIX system with `#:system:print-for-read` set to `()`:

```
? (temporary-file-pathname "toto")
= #p"/tmp/toto"
```

On a VAX/VMS system:

```
? (temporary-file-pathname "toto")
= #p"sys$scratch:toto"
```

(current-directory path) [*function with an optional argument*]

If used without an argument, this function returns the user's current working directory. If the `path` argument is supplied, it must be either a pathname whose directory specification—obtained by the call `(directory-namestring path)`—is valid, or a character string in the filename syntax of the host system. The function changes the current working directory of the LE-LISP process to the given argument.

(directoryp path) [*function with one argument*]

This function lets you find out whether or not the `path` pathname is a catalogue. `directoryp` has the same relationship with catalogues as `probefile` with files. The `path` argument can be of `pathname` or `string` type. `directoryp` is a natural complement to `current-directory`. `directoryp` returns either a pathname or `()`. The `name` field of the pathname is always empty, so that the pathname is a perfect description of a catalogue.

6.12.4 Portability

When the path module is loaded, the set of standard functions which work on files designated herein by `file` is redefined (cf. section 6.9) to accept both character strings and also pathnames.

The following variable and function provide a way to test the absolute portability of filenames.

`*portable-pathname*` *[variable]*

When this global variable is set to a value different than `()`, all functions using pathnames verify that their pathname arguments are portable over the entire set of machines supporting LE-LISP implementations, by using the predicate described next. This feature is particularly attractive to developpers building portable applications in LE-LISP.

`(portable-pathname-p path)` *[function with one argument]*

This function takes a pathname as argument. It tests the validity of this pathname for the various operating systems supporting LE-LISP implementations. It returns `()` if the given pathname is not portable, and sends messages describing the possible incompatibilities found.

6.13 Features

LE-LISP can manage the names of libraries loaded into memory.

Here are the names of features in the standard library:

`abbrev`	definition of abbreviations
`callext`	connection module for C
`compiler`	standard or modular compiler
`complex`	complex-number arithmetic
`complice`	modular compiler
`date`	date manipulation library
`debug`	debugging tools
`defstruct`	structure definitions
`edlin`	default line editor
`event-loop`	interface of the event loop
`format`	format library
`gen`	library of generic operators for bignums
`genr`	library of generic operators
`hash-table`	hash tables
`libcir`	circular objects library
`loader`	loader
`messages`	multi-language messages
`microceyx`	MicroCeyx object-oriented language
`module`	module management functions
`n`	natural integers of arbitrary precision

`pathname`	virtual file system
`pepe`	full-page editor
`pretty`	S-expression pretty-printer
`q`	rationals of arbitrary precision
`ratio`	rational-number arithmetic
`setf`	generalized assignment
`sets`	mathematical sets
`stringio`	I/O on strings
`termcap`	`termcap`-to-`virtty` translator
`terminfo`	`terminfo`-to-`virtty` translator
`virbitmap`	virtual bitmap
`virtty`	virtual terminal
`window`	virtual windowing
`z`	relative integers of arbitrary precision

`(list-features)` [*function with no arguments*]

Returns the list of features currently in memory.

`(add-feature` `symb``)` [*function with one argument*]

Adds the name `symb` to the list of loaded features.

`(rem-feature` `symb``)` [*function with one argument*]

Removes the name `symb` from the list of loaded features.

`(featurep` `symb``)` [*function with one argument*]

Returns `symb` if the feature `symb` is found in the list of loaded features.

Table of contents

6 Input and output **6-1**

6.1 Introduction ... 6-1
6.1.1 Characters ... 6-1
6.1.2 Character strings ... 6-1
6.1.3 Lines ... 6-2
6.1.4 Channels ... 6-2
6.1.5 Programmable I/O interrupts ... 6-2
6.2 Basic input functions ... 6-3
6.2.1 Inside `read` ... 6-7
6.3 Use of the terminal for input ... 6-8
6.4 Standard reader ... 6-9
6.4.1 Reading symbols ... 6-9
6.4.2 Reading character strings ... 6-10
6.4.3 Reading integer and rational numbers ... 6-11
6.4.4 Reading floating-point numbers ... 6-12
6.4.5 Reading lists ... 6-12
6.4.6 Reading vectors ... 6-13
6.4.7 Reading comments ... 6-13
6.4.8 Types of characters ... 6-14
6.4.9 Macro characters ... 6-18
6.5 Basic output functions ... 6-33
6.6 Controlling the output functions ... 6-35
6.6.1 Limitations on printing ... 6-35
6.6.2 Standard printing environment ... 6-36
6.6.3 Extending the printer ... 6-38

6.7 Input/output for lists 6-39
6.8 Input/output on character strings 6-40
6.9 Input/Output buffer management 6-41
6.9.1 Input buffer 6-41
6.9.2 Output buffer 6-44
6.10 Functions on I/O streams 6-47
6.10.1 Default directories and extensions 6-48
6.10.2 Selecting I/O streams 6-48
6.10.3 End-of-file programmable interrupt 6-52
6.10.4 Functions on files 6-53
6.10.5 `load` function and `autoload` mode 6-54
6.10.6 File access paths 6-55
6.10.7 Access to libraries 6-56
6.11 Event loop 6-57
6.11.1 Functions 6-57
6.11.2 Technical notes 6-60
6.11.3 Precautions 6-61
6.12 Virtual file system 6-63
6.12.1 Pathnames 6-63
6.12.2 Relative access paths 6-66
6.12.3 Pathname manipulation functions 6-67
6.12.4 Portability 6-74
6.13 Features 6-74

Function Index

`(read)` [*function with no arguments*] 6-3
`(stratom n strg i)` [*function with three arguments*] 6-3
`(readstring)` [*function with no arguments*] 6-4
`(readline)` [*function with no arguments*] 6-4
`(readcn)` [*function with no arguments*] 6-5
`(readch)` [*function with no arguments*] 6-5
`(peekcn)` [*function with no arguments*] 6-5
`(peekch)` [*function with no arguments*] 6-5
`(reread lcn)` [*function with one argument*] 6-5
`(read-delimited-list cn)` [*function with one argument*] 6-6
`(teread)` [*function with no arguments*] 6-7
`#:system:in-read-flag` [*variable*] 6-7
`(curread)` [*function with no arguments*] 6-7
`#:system:real-terminal-flag` [*variable*] 6-9
`#:system:line-mode-flag` [*variable*] 6-9
`(prompt strg)` [*function with an optional argument*] 6-9
`#:system:read-case-flag` [*variable*] 6-10
`(ibase n)` [*function with an optional argument*] 6-12
`cnull` [*character type*] 6-14
`cbcom` [*character type*] 6-14
`cecom` [*character type*] 6-14
`cquote` [*character type*] 6-14
`clpar` [*character type*] 6-14
`crpar` [*character type*] 6-14
`cdot` [*character type*] 6-14
`csep` [*character type*] 6-15

cpkgc [*character type*] 6-15
csplice [*character type*] 6-15
cmacro [*character type*] 6-15
cstring [*character type*] 6-15
cpname [*character type*] 6-15
csymb [*character type*] 6-15
cmsymb [*character type*] 6-16
(typecn cn symb) [*function with one or two arguments*] 6-16
(typech ch symb) [*function with one or two arguments*] 6-16
(dmc ch () s_1 ... s_n) [*special form*] 6-18
(dms ch () s_1 ... s_n) [*special form*] 6-19
' [*macro character*] 6-20
^ [*macro character*] 6-20
[[*macro character*] 6-20
` [*macro character*] 6-20
, [*macro character*] 6-21
,. [*macro character*] 6-21
,@ [*macro character*] 6-21
[*macro character*] 6-23
#:sys-package:sharp [*variable*] 6-23
(defsharp ch larg s1 ... sN) [*special form*] 6-24
#/ [*#-macro*] 6-24
#\ [*#-macro*] 6-24
#:sharp:value [*property*] 6-25
#^ [*#-macro*] 6-25
#$ [*#-macro*] 6-25
#% [*#-macro*] 6-25
#<base>r [*#-macro*] 6-25
| [*inside #-macro*] 6-26
#' [*#-macro*] 6-26
#" [*#-macro*] 6-26
#: [*#-macro*] 6-26
#. [*#-macro*] 6-26
#([*#-macro*] 6-27

`#[` [*#-macro*] 6-27

`#+` [*#-macro*] 6-27

`#-` [*#-macro*] 6-27

`#|` [*#-macro*] 6-27

`#:sys-package:colon` [*variable*] 6-30

`^L` [*macro character*] 6-30

`^A` [*macro character*] 6-31

`^E` [*macro character*] 6-31

`^F` [*macro character*] 6-31

`#:system:editor` [*variable*] 6-31

`^P` [*macro character*] 6-32

`!` [*macro character*] 6-32

`(prin` s_1 ... s_n`)` [*function with a variable number of arguments*] 6-33

`(print` s_1 ... s_n`)` [*function with a variable number of arguments*] 6-33

`(prinflush` s_1 ... s_n`)` [*function with a variable number of arguments*] 6-34

`(terpri n)` [*function with an optional argument*] 6-34

`(princn cn n)` [*function with one or two arguments*] 6-34

`(princh ch n)` [*function with one or two arguments*] 6-35

`(printlength n)` [*function with an optional argument*] 6-35

`(printlevel n)` [*function with an optional argument*] 6-35

`(printline n)` [*function with an optional argument*] 6-36

`#$8000` [*unnameable number*] 6-36

`#<>` [*non-Lisp pointer*] 6-37

`#:system:print-for-read` [*variable*] 6-37

`#:system:print-case-flag` [*variable*] 6-37

`#:system:print-package-flag` [*variable*] 6-37

`(obase n)` [*function with an optional argument*] 6-38

`(ptype symb n)` [*function with one or two arguments*] 6-38

`(pratom atom)` [*function with one argument*] 6-39

`(explode s)` [*function with one argument*] 6-39

`(explodech s)` [*function with one argument*] 6-39

`(implode ln)` [*function with one argument*] 6-39

`(implodech s)` [*function with one argument*] 6-40

`(with-input-from-string string . body)` [*macro*] 6-40

`(with-output-to-string string n . body)` [*macro*] 6-40
`(read-from-string string)` [*function with one argument*] 6-40
`(print-to-string s)` [*function with one argument*] 6-41
`(inbuf n cn)` [*function with one or two optional arguments*] 6-42
`(inmax n)` [*function with an optional argument*] 6-42
`(inpos n)` [*function with an optional argument*] 6-42
`bol` [*programmable interrupt*] 6-42
`(bol)` [*function with no arguments*] 6-43
`(lmargin n)` [*function with an optional argument*] 6-44
`(rmargin n)` [*function with an optional argument*] 6-45
`(outpos n)` [*function with an optional argument*] 6-45
`(outbuf n cn)` [*function with one or two optional arguments*] 6-45
`eol` [*programmable interrupt*] 6-45
`(eol)` [*function with no arguments*] 6-46
`flush` [*programmable interrupt*] 6-46
`(flush)` [*function with no arguments*] 6-46
`#:system:print-msgs` [*variable*] 6-48
`#:system:llib-directory` [*variable*] 6-48
`#:system:llub-directory` [*variable*] 6-48
`#:system:lelisp-extension` [*variable*] 6-48
`(openi file)` [*function with one argument*] 6-48
`(openib file)` [*function with one argument*] 6-49
`(openo file)` [*function with one argument*] 6-49
`(openob file)` [*function with one argument*] 6-49
`(opena file)` [*function with one argument*] 6-49
`(openab file)` [*function with one argument*] 6-49
`(inchan chan)` [*function with an optional argument*] 6-49
`(outchan chan)` [*function with an optional argument*] 6-50
`(channel chan)` [*function with an optional argument*] 6-50
`(close chan)` [*function with an optional argument*] 6-50
`(input file)` [*function with one argument*] 6-50
`(output file)` [*function with one argument*] 6-51
`eof` [*programmable interrupt*] 6-52
`(eof chan)` [*function with one argument*] 6-52

`(probefile file)` [*function with one argument*] 6-53
`(renamefile ofile nfile)` [*function with two arguments*] 6-53
`(copyfile ofile nfile)` [*function with two arguments*] 6-53
`(deletefile file)` [*function with one argument*] 6-53
`(create-directory dir)` [*function with one argument*] 6-53
`(delete-directory dir)` [*function with one argument*] 6-53
`(loadfile file i)` [*function with two arguments*] 6-54
`(load file i)` [*special form*] 6-54
`(autoload file` $\texttt{sym}_1$ ... $\texttt{sym}_n$`)` [*special form*] 6-54
`#:system:path` [*variable*] 6-55
`(search-in-path path file)` [*function with two arguments*] 6-55
`(probepathf file)` [*function with one argument*] 6-56
`(libloadfile file i)` [*function with two arguments*] 6-56
`(libload file i)` [*special form*] 6-56
`(libautoload file` $\texttt{sym}_1$ ... $\texttt{sym}_n$`)` [*special form*] 6-57
`event-loop` [*feature*] 6-57
`(evloop-init)` [*function with no arguments*] 6-57
`(evloop-stop)` [*function with no arguments*] 6-58
`(evloop-restart)` [*function with no arguments*] 6-58
`(evloop-disallow-tty-input)` [*function with no arguments*] 6-58
`(evloop-allow-tty-input)` [*function with no arguments*] 6-58
`(evloop-add-input fd manage-fct arg-to-use)` [*function with three arguments*] 6-58
`(evloop-remove-input fd)` [*function with one argument*] 6-58
`(evloop-change-manage-function fd new-manage-fct new-arg-to-use)` [*function with three arguments*] 6-58
`(evloop-select)` [*function with no arguments*] 6-58
`(evloop-readp fd)` [*function with one argument*] 6-59
`(evloop-input-managedp fd)` [*function with one argument*] 6-59
`(evloop-initialized-p)` [*function with no arguments*] 6-59
`(evloop-wait)` [*function with no arguments*] 6-59
`(evloop-set-timeout secs millisecs)` [*function with two arguments*] 6-59
`(evloop-set-timeout-handler handler)` [*function with one argument*] 6-59
`(evloop-add-output fd manage-fct arg-to-use)` [*function with three arguments*] 6-59
`(evloop-remove-output fd)` [*function with one argument*] 6-60

(evloop-change-output-manage-function fd new-manage-fct new-arg-to-use) [*function with three arguments*] 6-60
(evloop-add-display display manage-fct) [*function with two arguments*] 6-60
(evloop-remove-display display) [*function with one argument*] 6-60
(evloop-display-managed-p display) [*function with one argument*] 6-60
pathname [*feature*] 6-63
pathname [*structure*] 6-63
(pathname string) [*function with one argument*] 6-64
(pathnamep s) [*function with one argument*] 6-64
(equal-pathname path1 path2) [*function with two arguments*] 6-65
(namestring path) [*function with one argument*] 6-65
#p [*#-macro*] 6-65
#u [*#-macro*] 6-65
(#:pathname:prin path) [*function with one argument*] 6-65
#:pathname:up [*indicator*] 6-66
#:pathname:current [*indicator*] 6-66
#:pathname:wild [*indicator*] 6-67
(wildcard path) [*function with one argument*] 6-67
(expand-pathname path) [*function with one argument*] 6-67
(pathname-host path) [*function with one argument*] 6-67
(pathname-device path) [*function with one argument*] 6-67
(pathname-directory path) [*function with one argument*] 6-68
(pathname-name path) [*function with one argument*] 6-68
(pathname-type path) [*function with one argument*] 6-68
(pathname-version path) [*function with one argument*] 6-68
(set-pathname-host path host) [*function with two arguments*] 6-68
(set-pathname-device path device) [*function with two arguments*] 6-68
(set-pathname-directory path dir) [*function with two arguments*] 6-68
(set-pathname-name path name) [*function with two arguments*] 6-68
(set-pathname-type path type) [*function with two arguments*] 6-68
(set-pathname-version path version) [*function with two arguments*] 6-69
default-pathname-defaults [*variable*] 6-69
(make-pathname element$_1$ ... element$_n$) [*function with a variable number of arguments*] 6-69
(file-namestring path) [*function with one argument*] 6-69

(host-namestring path) [*function with one argument*] 6-70
(device-namestring path) [*function with one argument*] 6-70
(directory-namestring path) [*function with one argument*] 6-70
(merge-pathnames path default-path) [*function with two arguments*] 6-70
(combine-pathnames path1 path2) [*function with two arguments*] 6-70
(enough-namestring path default) [*function with one or two arguments*] 6-71
(true-pathname path) [*function with one argument*] 6-71
(user-homedir-pathname) [*function with no arguments*] 6-72
(control-file-pathname name path) [*function with one or two arguments*] 6-72
(temporary-file-pathname name) [*function with one argument*] 6-73
(current-directory path) [*function with an optional argument*] 6-73
(directoryp path) [*function with one argument*] 6-73
portable-pathname [*variable*] 6-74
(portable-pathname-p path) [*function with one argument*] 6-74
(list-features) [*function with no arguments*] 6-75
(add-feature symb) [*function with one argument*] 6-75
(rem-feature symb) [*function with one argument*] 6-75
(featurep symb) [*function with one argument*] 6-75

Chapter 7

System functions

The functions described in this chapter provide the means to create sub-systems written in LE-LISP. By means of programmable interrupts, it is possible to define new interaction loops and new ways to handle errors.

7.1 Programmable interrupts

LE-LISP handles its internal interrupts by calling a specialized function for each of these interrupts. The names of these functions are not fixed, but are dynamically sought when an interrupt occurs. This search follows the search path stored in the `#:sys-package:itsoft` variable. Thus the internal interrupts are programmable.

`#:sys-package:itsoft` [*variable*]

This variable contains the list of packages in which functions associated with internal programmable interrupts must belong. By default its value is `()`, so initially only the global package (named `||`) is searched for these interrupt functions. All the internal programmable interrupts have a function defined in the global package, which is therefore the interrupt function called by default.

The value of this variable can be a list of symbols or a single symbol. In the former case, the function associated with the programmable interrupt named `foo` is the first function named `foo` found in one of the packages named in the list. In the latter case, it is the function named `foo` found in the package having the same name as the symbol, or in one of its parent packages.

The function search is performed in 'horizontal' order in the first case, and in 'vertical' order, by climbing the package hierarchy, in the second case. It is modelled by the following function:

```
(defun get-it-handler (itpath itname)
    (cond ((symbolp itpath)
               (getfn itpath itname))
          ((consp itpath)
               (or (getfn1 (car itpath) itname)
                   (get-it-handler (cdr itpath) itname)))))
```

LE-LISP has twelve programmable internal interrupts:

```
bol               beginning-of-line
eol               end-of-line
flush             end-of-buffer
eof               end-of-file
toplevel          interaction loop
gc-before-alarm   beginning of gc
gcalarm           end of gc
syserror          internal error
stepeval          entrance into eval
user-interrupt    user interrupt
clock             clock machine interrupt
```

but it is entirely possible to define your own programmable interrupts.

(itsoft symb larg) *[function with two arguments]*

Explicitly calls the programmable interrupt named `symb` with `larg` as its argument list. The definition of this function has the following approximate form:

```
(defun itsoft (nom larg)
    (apply (get-it-handler #:sys-package:itsoft nom) larg))
```

but it creates a special scope block and resets the evaluator's trace flag to `()`.

(super-itsoft package symb larg) *[function with three arguments]*

Explicitly invokes the programmable interrupt named `symb` with `larg` as its argument list. It assumes that there is a function `#:package:symb` to be invoked by the `itsoft` function. The function invoked by `super-itsoft` is the function which would be invoked by `itsoft` if `#:package:symb` did not exist.

`super-itsoft` is particularly useful for putting pre- or post-processors on programmable interrupts. For example, to keep a count in a variable of the number of lines printed onto the screen:

```
(defvar #:count:nlines 0)

(defun #:count:eol ()
    (incr #:count:nlines)
    (super-itsoft 'count 'eol ()))

(setq #:sys-package:itsoft (cons 'count #:sys-package:itsoft))
```

The following example implements a utility similar to `more` on UNIX.

```
(defvar #:sys-package:colon 'minimore)

(defvar :count (tyymax))
(defvar :string "--more--")
```

```
(defun more ()
    (setq #:sys-package:itsoft
          (cons 'minimore #:sys-package:itsoft)))

(defun morend ()
    (setq #:sys-package:itsoft
          (delq 'minimore #:sys-package:itsoft)))

(defun :bol ()
    (setq :count (tyymax))
    (super-itsoft '#.#:sys-package:colon 'bol ()))

(defun :eol ()
    (when (= :count 0)
          (tystring :string (slength :string))
          (selectq (tyi)
             ((#^m #^j) (setq :count 1))
             (#^d       (setq :count (div (tyymax) 2)))
             (#\\sp      (setq :count (tyymax)))
             (#/q       (setq :count (tyymax))
                        (for (i (sub1 (slength :string)) -1 0)
                             (tyback (chrnth i :string)))
                        (fillstring (outbuf) 0 #\\sp (outpos))
                        (outpos 0)
                        (exit #:system:toplevel-tag))
             (t         (setq :count 1)))
         (for (i (sub1 (slength :string)) -1 0)
              (tyback (chrnth i :string))))
    (decr :count)
    (super-itsoft '#.#:sys-package:colon 'eol ()))
```

7.2 Machine interrupts

LE-LISP handles three machine interrupts, the user interrupt, the clock interrupt, and the event interrupt.

The management of these interrupts can be activated (authorized) or inhibited (masked). During the execution of the function associated with these programmable machine interrupts, they are masked. It is up to the user to authorize (reset) them if that becomes necessary.

(without-interrupts e_1 ... e_n) [*special form*]

This function masks machine interrupts while it evaluates the expressions e_1 ... e_n, in the same way as `progn`. The value of e_n is returned.

7.2.1 User interrupt

It is possible at any moment to cause a *user interrupt* by sending a special character from the keyboard. The specific character depends on the system, but it is always available.

`user-interrupt` [*programmable interrupt*]

This is the name of the programmable machine interrupt triggered by the user's keyboard action.

`(user-interrupt)` [*function with no arguments*]

This is the default function executed by the `user-interrupt` programmable interrupt.

In LE-LISP, `user-interrupt` could be defined in the following manner:

```
(defun user-interrupt ()
    (with-interrupts
        (itsoft 'syserror '(break break ()))))
```

7.2.2 Real-time clock

`(clockalarm n)` [*function with one argument*]

`n` is a floating-point number indicating the number of seconds the system should wait before emitting a single `clock` programmable machine interrupt. If `n` equals 0.0, the real-time clock is deactivated and all previous interrupt requests are lost. Note that very small values (< 0.001 seconds) should not be used, for the time it takes to process the `clock` programmable interrupt itself is considerable (especially if a context switch is thereby generated, see the `suspend` function).

`clock` [*programmable interrupt*]

This is the name of the programmable machine interrupt triggerred by the real-time clock.

`(clock)` [*function with no arguments*]

This is also the name of the default function executed by the `clock` programmable interrupt.

In LE-LISP, `clock` could be defined in the following manner:

```
(defun clock ())
```

7.3 Multiple tasks

LE-LISP has an integrated multi-task system. It has only three basic functions.

(schedule `fnt` e_1 ... e_n**)** [*special form*]

(suspend) [*function with no arguments*]

(resume `env`**)** [*function with one argument*]

The `schedule` function lets you define a *scheduling function*. The `suspend` function provides a way of stopping a task from running, and `resume` lets you start it running again.

In `schedule`, `fnt` is a one-argument function. This function evaluates (*à la* `progn`) the expressions $e_1 \ldots e_n$. In general the value returned by `schedule` is e_n, but if in the course of the evaluations an explicit or implicit (by way of the programmable interrupts) call to the `suspend` function occurs, the current calculation is stopped, the dynamic execution environment is saved, and the function `fnt` associated with the most recently encountered `schedule` function is called with the saved environment as its argument. Finally, `resume` is a function which takes an environment built by the `suspend` function as its argument, which resumes an interrupted calculation where it left off.

7.3.1 Basic sequencers

Using the three functions just described, it is possible to define the following basic sequencers. Time sharing is done using the real-time clock. These function are found in the file named `schedule` in the standard library.

(parallel e_1 ... e_n**)** [*special form*]

Launches the evaluations of the expressions $e_1 \ldots e_n$ in parallel. It always returns `()`.

(parallelvalues e_1 ... e_n**)** [*special form*]

Launches the evaluations of the expressions $e_1 \ldots e_n$ in parallel, and returns the list of the results.

(tryinparallel e_1 ... e_n**)** [*special form*]

Launches the evaluations of the expressions $e_1 \ldots e_n$ in parallel. When one of these evaluations terminates, `tryinparallel` returns its value and stops the rest of the evaluations.

Example: using the real-time clock to do time-sharing among tasks.

```
(defvar #:system:clock-tick 0.05)

(defun clock ()
    ; it's the clock that causes the suspension of the tasks.
```

```
    (when (debug) (princn #/.))
    (suspend))

(defun clockstart ()
     ; start the clock
     (clockalarm #:system:clock-tick))

(defun clockstop ()
     ; stop the clock
     (clockalarm 0.))

(df parallel :ll
    (without-interrupts
        (let ((:l (append :ll ())))
            (while :l
                (schedule (lambda (:v)
                              (newr :l (list 'resume (kwote :v))))
                     (let ((:e (nextl :l)))
                          (clockstart)
                          (with-interrupts (eval :e))
                          (clockstop)))))))

(df parallelvalues :ll
    (without-interrupts
        (let ((:ltask ())
              (:l :ll)
              (:r (makelist (length :ll) ()))
              (:i -1))
            (while (or :l :ltask)
                (schedule (lambda (:v) (newr :ltask :v))
                    (clockstart)
                    (ifn :l
                         (resume (nextl :ltask))
                         (let ((:e (nextl :l)))
                              (incr :i)
                              (rplaca (nthcdr :i :r)
                                      (with-interrupts (eval :e)))))
                    (clockstop)))
            :r)))

(df tryinparallel :ll
    (without-interrupts
        (let ((:ltask ())
              (:l :ll))
            (tag :return-value
                 (while (or :l :ltask)
                        (schedule (lambda (:v) (newr :ltask :v))
                            (clockstart)
                            (let ((:e (nextl :l)))
                                 (if :e
                                     (exit :return-value
                                           (prog1
                                                (with-interrupts
                                                    (eval :e))
                                                (clockstop))))
                                (resume (nextl :ltask)))))))))
```

Here are examples of things that use the parallel sequencers.

```
? (defun fib (n)
?     (cond ((= n 1) 1)
?           ((= n 2) 1)
?           (t (apply '+ (parallelvalues (fib (1- n))
?                                        (fib (- n 2)))))))
= fib

? (defun neg (n)
?    (if (= n 0)
?        'negative
?        (neg (1+ n))))
= neg

? (defun pos (n)
?    (if (= n 0)
?        'positive
?        (pos (1- n))))
= pos

? (defun sign (n)
? ; a very fast way to check the sign of a number:
?    (if (= n 0)
?        'zero
?        (tryinparallel (neg n) (pos n))))
= sign
```

7.4 Errors provoked by the Le-Lisp system

`syserror` [*programmable interrupt*]

When an error is raised during an evaluation, the programmable interrupt named `syserror` is triggered. It invokes a function (whose name is computed according to the explanation given in §7.1) that takes three arguments:

- the name of the function that raised the error;
- the error name;
- the faulty argument.

The standard behavior is that a message describing the error condition is printed on the current output stream.

This message always contains the name of the function which raised the error, an explanatory message and the faulty argument.

LE-LISP also provides a way to explicitly raise an error with the `error` function, and to test whether an evaluation would raise an error with the `catcherror` and `errset` functions.

7.4.1 Standard error processing

(syserror `symb s1 s2`**)** [*function with three arguments*]

This is the default function invoked by the `syserror` programmable interrupt. `symb` is the name of the function that raised the error. `s1` is the error name. `s2` is the faulty argument or argument list. `syserror` will print a complete error message onto the current output stream using the `printerror` function, then unbinds all variables and returns to the Lisp top-level. In `debug` mode, it is also possible to enter an inspection loop before returning to the top-level in order to examine the state of various local variables or to run certain evaluations (see Chapter 11).

(printerror `symb s1 s2`**)** [*function with three arguments*]

Prints a complete error message of the following form:

```
** <symb> : <s1> : <s2>
```

`symb` is the name of the function that raised the error, `s1` is the error name, and `s2` is generally the faulty argument. There are about thirty predefined error types available in the system.

7.4.2 Explicit call and error test

(error `s1 s2 s3`**)** [*function with three arguments*]

Lets you call an error explicitly. `s1` is the name of the function that causes the error, `s2` is the error type, and `s3` is the faulty argument.

In LE-LISP, `error` could be defined in the following manner:

```
(dmd error l
     `(itsoft 'syserror ,l))
```

(catcherror `i` $s_1 \ldots s_n$**)** [*special form*]

`i` is a flag that is evaluated and then set in the system variable `#:system:error-flag`. The expressions $s_1 \ldots s_n$ are evaluated sequentially. If an error is raised during one of these evaluations, `catcherror` immediately returns `()` and, depending upon the state of the `#:system:error-flag` variable, prints an error message using the `printerror` function. If the evaluation of the expressions raises no error, `catcherror` returns a one-element list which contains the value of the last expression evaluated, that is, of s_n to distinguish this value from `()` which is returned upon error.

`catcherror` never catchs the `errudt` undefined escape error. It has to be caught by using the special form `lock`.

(errset `e i`**)** [*macro*]

This is a predecessor of the previous function, and is included in LE-LISP for historical and compatibility reasons only.

In LE-LISP, `errset` could be defined in the following manner:

```
(dmd errset (e i) `(catcherror ,i ,e))
```

(err s_1 ... s_n) [*special form*]

Evaluates the expressions s_1 ... s_n. The value of s_n is the value returned by the dynamically-englobing `catcherror`, or by `toplevel` if no `catcherror` was dynamically defined.

Warning: This value must be an atom if you want to place the `catcherror` in an error context: that is, when `err` returns a simulated error. The value must be a list if `err` returns a value without error. In the latter case, the result—which is also that of `catcherror`—will be interpreted as a list whose first element is the expected result.

7.4.3 Examples of exception handling

Let us look at some examples of error exception handling.

Normal behavior of the `errduv` error (undefined variable):

```
? (gensym)
= g101   ? (eval (gensym))
** eval : undefined variable : g102
```

Redefinition of the `errduv` error—the undefined variable is set to `()`:

```
? (defun #:pierre:syserror (f m a)
?  (if (eq m 'errudv)
?  (set a ())
?  (syserror f m a)))
= #:pierre:syserror

? (setq #:sys-package:itsoft 'pierre)
= pierre

? (setq x (gensym))
= g103   ? (eval x)
= ()   ? x
= g103   ? (eval 'g103)
= ()
```

Redefinition of the same error—the undefined variables will be changed into constants:

```
? (defun #:christian:syserror (f m a)
?  (if (eq m 'errudv)
?  (set a a)
?  (syserror f m a)))
= #:christian:syserror
```

```
? (setq #:sys-package:itsoft 'christian)
= christian

? (setq x (gensym))
= g104   ? (eval x)
= g104   ? (eval 'g104)
= g104
```

Return to the normal error handling:

```
? (setq #:sys-package:itsoft ())
= ()

? (eval (gensym))
** eval : undefined variable : g105
```

Redefinition of the `errudf` error:

```
? (defun danger (f l)
?  (f l))
= danger   ? (danger 'car '(a b c))
** eval : undefined function : f


? (defun #:odd:syserror (#:system:f #:system:m #:system:a)
?  (if (and (eq #:system:f 'eval)
?  (eq #:system:m 'errudf))
?  (eval #:system:a)
?  (syserror #:system:f #:system:m #:system:a)))
= #:odd:syserror   ? (defvar #:sys-package:itsoft 'odd)
= odd   ? (danger 'car '(a b c))
? a     ; if there is no function f in the system
```

7.5 Access to the evaluator

LE-LISP has three internal evaluator control functions. The are used to implement the interpreter and some debugging tools (*trace, incremental execution and dynamic control*) which will be described in Chapter 11.

The evaluator has an internal flag which for reasons of efficiency is not a LISP variable. When this flag which is set, every call to the evaluator, that is to the `eval` function, causes the `stepeval` programmable interrupt.

`stepeval` [*programmable interrupt*]

This programmable interrupt will call a function with the value to be evaluated as its argument.

(stepeval e env) *[function with two arguments]*

This is the default function called by the `stepeval` programmable interrupt. `e` is the form to be evaluated and `env` is the lexical environment in which evaluation is to be done.

By default, `stepeval` prints onto the current output stream:

```
-->   the form that was to have been evaluated
<--   the result of the evaluation of this form
```

In LE-LISP, `stepeval` could be defined in the following manner:

```
(defun stepeval (e env)
    (print "-->" e)
    (print "<--" (traceval e env)))
```

(traceval e env) *[function with one or two arguments]*

Evaluates the expression `e` in trace mode, that is with the internal trace flag set. If `env` is supplied, it is used as the lexical environment for the evaluation. (`env` is produced by the `stepval` function.) `traceval` returns the value of `e` evaluated in trace mode.

(cstack) *[function with no arguments]*

Each time it enters a function, an escape, a lock, the evaluator builds an activation block on the dynamic execution stack. `cstack` returns the activation blocks on the Lisp stack, represented as a list of lists, the elements of which have the coded type of the block in their `car` and the block arguments in their `cdr`. Here is the list of block types currently built by the system.

```
 1      lambda
 2      flet
 3      tag
 4      itsoft
 5      lock
 6      protect
 7      sysprot
 8      schedule
 9      tagbody
10      block
```

7.6 Core-image files

It is possible to save the entire system in a core-image file at any time and restore it at a later time. This feature is particularly useful for saving compiled sub-systems and calling them up very rapidly.

Warning: When a core-image file is restored, execution starts up exactly where it was interrupted, although the I/O channels and other host operating system parameters, like timers, cannot be restored.

#:system:core-directory [*variable*]

This variable contains the name of the directory that holds the standard system core-image files.

#:system:core-extension [*variable*]

This variable contains the default extension of core-image filenames.

(save-core file) [*function with one argument*]

Saves the current memory image in the core-image file named `file` and returns the value `t`. This function is a basic primitive, and it is suggested that `save-std` be used to create standard core-image files. You can save a screen by combining `save-core` and `bitmap-save`.

(restore-core file) [*function with one argument*]

Restores an core-image into memory from the core-image file named `file`.

`restore-core` returns the value `()`, in order to be able to distinguish it from returns from `savecore`, but continues evaluation at the same place at which it was stopped by the corresponding `save-core`.

See `bitmap-restore` in chapter 18.

Here are some examples of how to use the preceding functions:

```
? (save-core 'image)
= t    ? (restore-core 'image)
= ()

? (progn (print 1)
?  (save-core 'qwer)
?  (print 2))
1
2
= 2    ? (restore-core 'qwer)
2
= 2
```

Here is a good way to save a core-image so that you get a sub-system.

```
? (progn
?  (save-core 'bar)
?  (core-init-std "You are in the system: bar"))
; Le-Lisp (by INRIA) version 15.24 ( 2/Jan/91) [sun4]
You are in the system: bar : Tue Jan 8 19 11:50:55
```

```
   = (31bitfloats abbrev callext compiler date debug defstruct display edlin
loader messages microceyx pathname pepe pretty setf virbitmap virtty)
```

To restart it, type this expression.

```
   ? (restore-core 'bar)
   ; Le-Lisp (by INRIA) version 15.24 ( 2/Jan/91) [sun4]
   You are in the system: bar : Tue Jan 8 19 11:50:55
   = (31bitfloats abbrev callext compiler date debug defstruct display edlin
loader messages microceyx pathname pepe pretty setf virbitmap virtty)
```

Here is another example, resulting in the temporary suspension of the interpreter.

```
   ? (defun foo (n)
   ?  (if (> n 0)
   ?  (prog1 (* n (foo (1- n)))
   ?  (print n))
   ?  (save-core 'foo)
   ?  (print 'at-bottom)
   ?  1))
   = foo

   ? (foo 4)
   at-bottom
   1
   2
   3
   4
   = 24
```

Here is something to avoid, in order not to send the system looping forever.

```
   ? (progn
   ?  (print "I start")
   ?  (save-core 'qaz)
   ?  (print "I continue")
   ?  (restore-core 'qaz)
   ?  (print "I finish"))
   I start
   I continue
   I continue
   I continue
.....
```

7.7 Installation functions

The functions described in this section facilitate the loading and compiling of the standard or the modular LE-LISP system environments. They are used only in the construction of core-images files, that is, when the system is being installed at a new site.

These functions are defined in the `startup` file in the standard library, which is loaded automatically at system startup if no core-image file is being restored.

(load-std `im min ed env ld cmp`) [*function with one to six arguments*]

`load-std` loads all or part of the interpreter's standard environment, and optionally builds a core-image file. This function accepts from one to six arguments, which are flags. Arguments that are not supplied are set to `()`.

If the `im` flag is not `()`, it is passed to the function `save-std` after the environment is loaded, so that a core-image file named `im` is created.

The `min` flag indicates that the minimum required system environment is to be loaded, including only toplevel, virtual terminal and virtual bitmap functions.

The `ed` flag indicates that the full-page `pepe` editor is to be loaded.

The `env` flag indicates that the LE-LISP development environment, including structures, sorting, tables, external procedures, stepping, formatted printing, generic arithmetic and debugging inspection loop, is to be loaded.

The `ld` flag indicates that the loader, which provides a way to dynamically load compiled `lisp` modules and `lap` programs, is to be loaded.

The `cmp` flag indicates that the standard LE-LISP compiler is to be loaded. Loading the compiler causes the *compiled-code loader* to be loaded if it is not already present in the LISP environment.

Here is how to load the whole environment, without creating a core-image file.

```
? (load-std () t t t t t)
```

(llcp-std `name`) [*function with one argument*]

Compiles the interpreter environment that was stored by the `load-std` function. If the argument `name` is not equal to `()`, it is passed to the `save-std` function after compilation so that a core-image file of the compiled system will be built.

It is absolutely necessary to use this function in order to compile the Lisp environment, since this compilation poses some difficult problems (most notably the compilation of the loader) which the `compile-all-in-core` function cannot resolve.

Here is how to load the environment, compile it, and save it in a compiled core-image.

```
? (load-std () t t t t t)
? (llcp-std 'compile)
```

(load-stm `im min ed env ld cmp`) [*function with one to six arguments*]

This function is similar to the `load-std` function, except that it uses modules that were pre-compiled by Complice. It loads all or part of the environment and optionally builds a core-image file. This function also accepts from one to six flag arguments; arguments not

explicitly provided are set to `()`.

(load-cpl `im min ed env ld cmp`) [*function with one to six arguments*]

This function is similar to the `load-std` function, except that it loads compiled modules instead of interpreted ones. Moreover the `cmp` flag indicates in this case that the Complice compiler, and not the standard compiler, should be loaded.

Since the environment loaded by this function has already been compiled, the `llcp-std` function need not be used after a call to `load-cpl`.

Here is how to load the compiled development environment and save it in the core-image file called `mylisp`.

```
? (load-cpl "mylisp" t () t)
```

(core-init-std `msg`) [*function with one argument*]

Performs the standard initializations after a core-image file has been restored. The virtual terminal and the virtual bitmap are loaded (qualified by the two variables described below), the banner is printed (by the `herald` function), and a start-up file, whose name is system-dependent, is optionally loaded. The system then prints the welcome message `msg` as well as the date the core-image file was built. `core-init-std` returns `features-list`, the list of features loaded into memory.

In LE-LISP, `core-init-std` could be defined in the following manner:

```
(defun core-init-std (msg)
    (when #:system:initty-after-restore-flag
          (initty))
    (when #:system:inibitmap-after-restore-flag
          (inibitmap))
    (herald)
    (when #:system:unixp
          (let ((f (catenate (getenv "home") "/.lelisp")))
               (when (probefile f)
                     (loadfile f t))))
    (print "; " msg " : " (date))
    (sortl (list-features)))
```

#:system:initty-after-restore-flag [*variable*]

If this flag is true, the `core-init-std` function will initialize the terminal after each `restore-core` using the `initty` function. Its default value is `t`.

#:system:inibitmap-after-restore-flag [*variable*]

If this flag is true, the `core-init-std` function will initialize the virtual bitmap after each `restore-core` using the `initbitmap` function. Its default value is `t`.

(save-std name msg fnt-sav fnt-rest) *[function of two, three or four arguments]*

`name` is a character string or a LISP symbol. This function constructs a core-image file called `name` with the suffix `#:system:core-extension`. This core-image file is created in the directory named `#:system:core-directory`.

Just after the creation of the core-image file by `save-std`, a function is launched automatically. This function is either `fnt-sav`, if supplied, or `core-init-std`.

When the core-image file is restored using the `restore-core` function, the `fnt-rest` function is automatically run. If this function is not supplied, `save-std` uses the same function as for saving: `fnt-save`, if supplied, or else the `core-init-std` function. Finally the main interaction loop, the programmable interrupt called `toplevel`, is launched.

One of two functions is run immediately after a core-image file has been created by the `save-std` function. If the `fn-std}` function is provided, then it will be run; otherwise the same function that is in core-image file restoration, `fnt-rest`, is run.

The `save-std` function is used by the `load-std`, `load-cpl` and `llcp-std` functions when their `im` parameter is not equal to `()`.

Various errors can be produced during the restoration of a core-image file created by the `save-std` function. They often occur during initialization of the virtual terminal or the virtual bitmap. See the documentation concerning the `initty` and `inibitmap` functions.

Here is typical use of the `save-std` function:

```
? (save-std 'application ``Welcome to the Application''
?  'end 'core-init-std)
Wait, I am saving : Welcome to the Application
```

This results in a core-image file named `application` being saved which will execute the `core-init-std` function when it is restored. After the creation of the core-image file, the application will exit from the LE-LISPsystem.

7.8 Calling and leaving the system

(herald) *[function with no arguments]*

Prints the system banner on the entry terminal. Before reporting any errors found in your system, please run this function to identify it exactly.

(system) *[function with no arguments]*

Returns the name of the LE-LISP system that runs it. This name depends on the system you are using.

#:system:unixp [*variable*]

This variable has the value `t` if the host operating system is UNIX and `()` otherwise.

(version) [*function with no arguments*]

Returns the version number of the LE-LISP system in the form of a number. Today, this number would be `15.25`.

```
(version)           ==>   15.25
(floatp (version))  ==>   15.25
(> (version) 15)    ==>   15.25
```

(subversion) [*function with no arguments*]

Returns the sub-version number of the LE-LISP system in the form of a number. Today, this number would be sub-version `1` within version `15.25`.

```
(subversion)           ==>   1
(numberp (subversion)) ==>   1
```

at-end [*programmable interrupt*]

At the call of the `end` function, the programmable interrupt `at-end` is activated. It is going to invoke a function (whose name is explained in Section 7.1) that uses (as an argument) that argument that was passed to the `end` function.

(at-end return-code) [*function with an optional argument*]

This is the default function that is automatically started by the programmable interrupt `at-end`. By default, this function doesn't do anything.

Note: The use of this function will virtually always require you to quit the application at the time of the call to the `end` function.

(end return-code) [*function with an optional argument*]

Starts the programmable interrupt `at-end`, stops the evaluation currently underway, closes all open files, leaves the LE-LISP system for good, and returns control to the host operating system. `end` is the only predefined function that does not return a value.

If no argument is supplied, LE-LISP automatically prints the exit banner. If an argument is furnished, the banner is not printed. This argument can be used by the host operating system to determine whether or not the system exited normally or not. If the value is other than `()`, an error has occurred.

A normal return to the host system, with banner:

```
? (end)
Que Le-Lisp soit avec vous.
```

A normal return to the host system, without banner:

```
? (end ())
```

An abnormal return to the host system:

```
? (end 1)
```

7.9 Top level

The principal loop, or `top-level`, of the LE-LISP system consists of the infinite evaluation of the form:

```
(while t (itsoft 'toplevel ()))
```

It is thus the programmable interrupt `toplevel` which determines the operating mode of the LE-LISP system. This interrupt is quite obviously redefinable by any user who wants to build their own system.

`toplevel` [*programmable interrupt*]

This is the name of the programmable interrupt run by the LE-LISP top level.

`(toplevel)` [*function with no arguments*]

This function will, in the standard environment:

- read an S-expression from the current input stream;
- evaluate this S-expression;
- print the result of this evaluation onto the current output stream.

`#:toplevel:status` [*variable*]

This variable holds the top level flag. If this flag is false, the `toplevel` function does not print the results of evaluations, and in any other case, it prints the values returned by each evaluation.

`#:toplevel:read` [*variable*]

This variable contains the last form read by the top level.

`#:toplevel:eval` [*variable*]

This variable contains the last value computed by the top level.

(#:system:toplevel-tag) [*escape*]

This is the name of the escape that is used to return directly to the top level.

At system initialization the `toplevel` function is equivalent to:

```
(defvar #:toplevel:status ())
(defvar #:toplevel:read   ())
(defvar #:toplevel:cread  ())
(defvar #:toplevel:eval   ())

(defun toplevel ()
    (tag #:system:toplevel-tag
         (setq #:toplevel:read  #:toplevel:cread
               #:toplevel:cread (read)
               #:toplevel:eval  (eval #:toplevel:cread))
         (when #:toplevel:status
               (print "= " #:toplevel:eval)))
```

Warning: A redefinition of this function that fails to evaluate the forms read renders the entire LE-LISP system totally unusable.

Example of the kind of definition to avoid:

```
(defun toplevel () (read) (print 'swamp))
```

7.10 Garbage collector

Memory zones that contain LISP objects are allocated dynamically. When one of the zones is saturated, the *garbage collector* is automatically called to recover unused objects.

If this effort at recovery comes to naught, a fatal error is raised. The names of this kind of fatal error all begin with `errf` and have the form `errf`xxx.

`errfcns` : the list zone is full. The screen display is:

```
***** fatal error : no room for lists.
```

`errfvec`: the vector zone is full. The screen display is:

```
***** fatal error : no room for vectors.
```

`errfstr`: the character string zone is full. The screen display is:

```
***** fatal error : no room for strings.
```

`errfsym`: the symbol zone is full. The screen display is:

```
***** fatal error : no room for symbols.
```

`errfflt`: the floating-point zone is full. The screen display is:

```
***** fatal error : no room for floats.
```

`errffix`: the integer (fixed-point) zone is full. The screen display is:

```
***** fatal error : no room for fixs.
```

LE-LISP also manages two other memory spaces, which do not contain LISP objects:

- The *code space*, which contains code generated by the compiler. This space is managed by the loaders. If space runs out in it, the `errfcod` fatal error occurs, with the screen display:

  ```
  ***** fatal error : no room for code.
  ```

- The *heap space*, which contains values of character strings, values of S-expression vectors, etc. The heap space is dynamically managed and is compacted when space runs out. If one of these garbage collections fails to render space available, the `errfhep` fatal error is raised, with the screen display:

  ```
  ***** fatal error : no room for heap.
  ```

LE-LISP uses a single dynamic execution stack for both control and data. Code resulting in stack overflow or underflow raises the `errfstk` fatal error, which has the screen display:

```
***** fatal error : stack overflow.
```

This error is not fatal. A stack tolerance scheme lets the execution stack be inspected and manipulated with the standard debugging tools even after a "full stack" error has been raised.

(gc i) [*function with an optional argument*]

`gc` calls the *garbage-collector* explicitly. If no argument is provided, `gc` returns `t`. If an argument is given, `gc` returns a list just like that which the `gcinfo` function returns (see the explanation immediately following this one.)

(gcinfo i) [*function with an optional argument*]

If called with no argument, `gcinfo` returns a list containing information on the last memory recovery done. If an argument is supplied, `gcinfo` returns a list describing the initial memory state of the system at start-up, before any allocation was done.

The list, in either case, is in the following form:

```
(gc <n1> <n2> <n3> <n4> <n5> <n6> <n7> <n8>
    cons   <ncons>
    symbol <nsymb>
    string <nstrg>
    vector <nvect>
    float  <nfloat>
    fix    <nfix>
    heap   <nheap>
    code   <ncode>)
```

where

- `n1` is the total number of gcs done (since the session began) due to a shortage of list cells.
- `n1` is the total number of gcs done (since the session began) due to a shortage of symbols.
- `n3` is the total number of gcs done (since the session began) due to a shortage of character strings.
- `n4` is the total number of gcs done (since the session began) due to a shortage of vectors.
- `n5` is the total number of gcs done (since the session began) due to a shortage of floating-point store.
- `n6` is the total number of gcs done (since the session began) due to a shortage of integer store.
- `n7` is the total number of gcs done due to a shortage of heap store.
- `n8` is the total number of gcs done due to explicit calls to the `gc` function.

The other values indicate the remaining size of each memory zone. These values are expressed in the *number of objects* if this is sufficiently small, or else in a list of which the `car` is in `k objets` and the `cdr` in `number of objects` remaining.

In this case the number is in a list.

- `ncons` is the number of list cells remaining in the list zone.
- `nsymb` is the number of symbols remaining in the symbol zone.
- `nstrg` is the number of strings remaining in the string zone.
- `nvect` is the number of vectors remaining in the vector zone.
- `nfloat` is the number of floats remaining in the floating-point zone.
- `nfix` is the number of fixed-point numbers remaining in the fixed-point number zone, if this latter exists.
- `nheap` is the size of the remaining heap store in the heap zone.
- `ncode` is the size of the remaining code store in the code zone.

gcalarm [*programmable interrupt*]

This is a programmable interrupt that is automatically run by the system after each garbage collection. It provides a way to dynamically control memory recovery.

(gcalarm) [*function with no arguments*]

The gcalarm function is called by the gcalarm programmable interrupt. The standard gcalarm function supplied by default does nothing.

The following redefinition of gcalarm provokes a 'soft' error when too many list cells—that is, more than remained free in the list zone—are consumed by the user program:

```
(defun gcalarm ()
    (let ((nbcons (cadr (memq 'cons (gcinfo)))))
         (when (and (fixp nbcons) (< nbcons 1000))
               (print "stopping the run on the list cell bank.")
               (error 'gcalarm "number of list cells remaining" nbcons)))
```

gc-before-alarm [*programmable interrupt*]

This is a programmable interrupt that is automatically run by the system before each memory recovery.

Warning: During its operation, this programmable interrupt *must not* allocate memory, or it will throw the gc into an infinite loop. This interrupt only indicates to user programs that a gc cycle has begun. It could be used, for instance, to change the appearance or position of the cursor, to account for gc elapsed time, etc.

(gc-before-alarm) [*function with no arguments*]

This function is called by the gc-before-alarm programmable interrupt. The standard gcalarm function supplied by default does nothing.

The following redefinition of the gc-related programmable interrupts prints the time used by each gc cycle on the console.

```
(defvar gc-runtime ())

(defun gc-before-alarm ()
    (setq gc-runtime (runtime)))

(defun gcalarm ()
    (let ((rt (- (runtime) gc-runtime)))
         (with ((outchan ()))
               (print "gc runtime: " gc-runtime " sec."))))
```

(freecons cons) [*function with one argument*]

Enables you to put the cons onto the cons freelist. This function yields performance gains in memory management. In certain well-understood cases, obsolete objects can be incrementally freed by using

`freecons` as soon as they are no longer needed. `freecons` is not interruptible and always returns `()`.

Warning: This function can permanently and definitively confuse the memory management if it is called on a `cons` to which active pointers still exist.

`freecons` cannot be easily described in LISP.

```
(defun number-of-cons ()
    ; return the number of conses on the cons freelist
    (let ((x (cons () ())) (n 0))
         (freecons x)
         (while x (incr n) (nextl x))
         n))
```

`(freetree tree)` [*function with one argument*]

Enables you to put the tree `tree` onto the cons freelist. `tree` cannot be a circular or shared list. Calling `freetree` on such an object runs the risk of destroying the whole system. Like the preceding function, `freetree` facilitates, in certain well-understood cases, memory management gains via incremental liberation of obsolete objects. `freetree` is not interruptible and always returns `()`.

Warning: This function can permanently and definitively confuse the memory management if it is called on a `cons` to which active pointers still exist.

In LE-LISP, `freetree` could be defined in the following manner:

```
(defun freetree (s)
    (when (consp s)
          (freecons s)
          (freetree (car s))
          (freetree (cdr s))))
```

7.11 Date processing

7.12 Other access to the system

LE-LISP offers access to other system characteristics. Considering the significant differences among systems, it would be wise, before using the following functions, to first test their effects in simple cases.

`(runtime)` [*function with no arguments*]

Returns the `cpu` time used since this LE-LISP session began. This time is given in either integer or floating-point seconds, depending upon the system being used.

`(time e)` [*function with one argument*]

Returns the time used by the `cpu` to evaluate the expression `e`. As in the last function, the time is given in either integer or floating-point seconds, depending upon the system being used.

In LE-LISP, `time` could be defined in the following manner:

```
(defun time (e)
    (let ((runt (runtime)))
         (eval e)
         (- (runtime) runt)))
```

```
(time '(fib 20))  ⟹   0.42
```

(sleep n) [*function with one argument*]

Requests the host system to put LE-LISP into an inactive state for `n` seconds. This number `n` is given in integer or floating-point units, depending on the system being used.

(comline strg) [*function with one argument*]

Sends the command line `strg` to the host operating system. This function is not available on every implementation of LE-LISP.

A macro character at the top level—the *exclamation point*—makes it simple to perform a `comline`:

```
? (comline "pwd")
/nfs/home2/rodine/itex
= t    ? (comline 'nil)
sh: nil: not found
= t    ? !ls -ls ; a listing of the current directory is printed....
= t
```

(getenv strg) [*function with one argument*]

Sends the string `strg`, which is presumed to be the name of a system variable, to the host operating system. `getenv` returns a number or system atom which is either a non-null value associated with `strg` or else `()` if this variable name has no value in the host operating system context. This function is not available on all LE-LISP systems.

```
(getenv "TERM")      ⟹   xterm
(getenv "DISPLAY")   ⟹   unix:0.0
(getenv "MYPORT")    ⟹   ()
```

Table of contents

7 System functions **7-1**

7.1 Programmable interrupts 7-1
7.2 Machine interrupts 7-3
7.2.1 User interrupt 7-3
7.2.2 Real-time clock 7-4
7.3 Multiple tasks 7-5
7.3.1 Basic sequencers 7-5
7.4 Errors provoked by the Le-Lisp system 7-7
7.4.1 Standard error processing 7-8
7.4.2 Explicit call and error test 7-8
7.4.3 Examples of exception handling 7-9
7.5 Access to the evaluator 7-10
7.6 Core-image files 7-11
7.7 Installation functions 7-13
7.8 Calling and leaving the system 7-16
7.9 Top level 7-18
7.10 Garbage collector 7-19
7.11 Date processing 7-23
7.12 Other access to the system 7-23

Function Index

`#:sys-package:itsoft` [*variable*] 7-1
`(itsoft symb larg)` [*function with two arguments*] 7-2
`(super-itsoft package symb larg)` [*function with three arguments*] 7-2
`(without-interrupts` e_1 ... e_n`)` [*special form*] 7-3
`user-interrupt` [*programmable interrupt*] 7-4
`(user-interrupt)` [*function with no arguments*] 7-4
`(clockalarm n)` [*function with one argument*] 7-4
`clock` [*programmable interrupt*] 7-4
`(clock)` [*function with no arguments*] 7-4
`(schedule fnt` e_1 ... e_n`)` [*special form*] 7-5
`(suspend)` [*function with no arguments*] 7-5
`(resume env)` [*function with one argument*] 7-5
`(parallel` e_1 ... e_n`)` [*special form*] 7-5
`(parallelvalues` e_1 ... e_n`)` [*special form*] 7-5
`(tryinparallel` e_1 ... e_n`)` [*special form*] 7-5
`syserror` [*programmable interrupt*] 7-7
`(syserror symb s1 s2)` [*function with three arguments*] 7-8
`(printerror symb s1 s2)` [*function with three arguments*] 7-8
`(error s1 s2 s3)` [*function with three arguments*] 7-8
`(catcherror i` s_1 ... s_n`)` [*special form*] 7-8
`(errset e i)` [*macro*] 7-9
`(err` s_1 ... s_n`)` [*special form*] 7-9
`stepeval` [*programmable interrupt*] 7-10
`(stepeval e env)` [*function with two arguments*] 7-11
`(traceval e env)` [*function with one or two arguments*] 7-11
`(cstack)` [*function with no arguments*] 7-11

#:system:core-directory [*variable*] 7-12
#:system:core-extension [*variable*] 7-12
(save-core file) [*function with one argument*] 7-12
(restore-core file) [*function with one argument*] 7-12
(load-std im min ed env ld cmp) [*function with one to six arguments*] 7-14
(llcp-std name) [*function with one argument*] 7-14
(load-stm im min ed env ld cmp) [*function with one to six arguments*] 7-14
(load-cpl im min ed env ld cmp) [*function with one to six arguments*] 7-15
(core-init-std msg) [*function with one argument*] 7-15
#:system:initty-after-restore-flag [*variable*] 7-15
#:system:inibitmap-after-restore-flag [*variable*] 7-16
(save-std name msg fnt-sav fnt-rest) [*function of two, three or four arguments*] 7-16
(herald) [*function with no arguments*] 7-16
(system) [*function with no arguments*] 7-16
#:system:unixp [*variable*] 7-17
(version) [*function with no arguments*] 7-17
(subversion) [*function with no arguments*] 7-17
at-end [*programmable interrupt*] 7-17
(at-end return-code) [*function with an optional argument*] 7-17
(end return-code) [*function with an optional argument*] 7-17
toplevel [*programmable interrupt*] 7-18
(toplevel) [*function with no arguments*] 7-18
#:toplevel:status [*variable*] 7-18
#:toplevel:read [*variable*] 7-18
#:toplevel:eval [*variable*] 7-18
(#:system:toplevel-tag) [*escape*] 7-19
(gc i) [*function with an optional argument*] 7-20
(gcinfo i) [*function with an optional argument*] 7-20
gcalarm [*programmable interrupt*] 7-22
(gcalarm) [*function with no arguments*] 7-22
gc-before-alarm [*programmable interrupt*] 7-22
(gc-before-alarm) [*function with no arguments*] 7-22
(freecons cons) [*function with one argument*] 7-22
(freetree tree) [*function with one argument*] 7-23

`(runtime)` [*function with no arguments*] 7-23
`(time e)` [*function with one argument*] 7-24
`(sleep n)` [*function with one argument*] 7-24
`(comline strg)` [*function with one argument*] 7-24
`(getenv strg)` [*function with one argument*] 7-24

Chapter 8

S-expression pretty-printer

The predefined functions `print` and `prin` are normally used to print LE-LISP S-expressions. The only things done to render them readable are :

- a space is inserted between atoms ;
- atoms (symbols and numbers) are not broken up by end-of-lines.

These measures are clearly insufficient for program display. The pretty-print functions, using variable indentation and line feeds where necessary, bring out the control structure of code and so render it much more readable.

To further increase readability, calls to the `quote` function are represented by the macro `'` , and the macro forms of `lets` are printed in full.

To demonstrate the utility of the pretty-printer, we first present a function in the manner it is printed by the normal print functions (for a given terminal or window line width) :

```
? (getdef '#:pretty:cond)
= (defun #:pretty:cond () (with ((lmargin) (add (lmargin 3))) (while (consp l)
(terpri) (if (#:pretty:inlinep (car l) (nextl l) (let ((l (nextl l) (f t))
(princn 40) (when (consp l (#:pretty:p (next l))) (when l (#:pretty:progn))
(#:pretty:pdot))))))
```

Here is how the pretty-printer presents it :

```
? (pretty #:pretty:cond)
(de #:pretty:cond ()
    (with ((lmargin (add (lmargin) 3)))
          (while (consp l)
                 (terpri)
                 (if (#:pretty:inlinep (car l))
                     (nextl l)
                     (let ((l (nextl l)) (f t))
                          (princn 40)
                          (when (consp l) (#:pretty:p (nextl l)))
                          (when l (#:pretty:progn))
                          (#:pretty:pdot))))))

= ()
```

8.0.1 Pretty-print functions

The following functions are found in the file named `pretty` in the standard library, and they are autoloaded (automatically loaded) the first time one of them is called.

`pretty` [*feature*]

This feature indicates whether the pretty-printer is loaded into memory.

`(pprint s)` [*function with one argument*]

Pretty-prints the expresssion `s` onto the current output stream and goes to the next output line (a la `print`). It returns `s` as its value.

`(pprin s)` [*function with one argument*]

Pretty-prints the expression `s` onto the current output stream and stays on the same output line (a la `prin`). It returns `s` as its value.

`(pretty` $\texttt{sym}_1$ ... $\texttt{sym}_n$`)` [*special form*]

The arguments $\texttt{sym}_1$ ...$\texttt{sym}_n$ are symbols that have function definitions. pretty prints these functions onto the current ouptut stream and returns `()` as its value. This function also knows about redefined or traced functions, so it will use the true definitions of these functions. There is a macro character that effects this function: `^P`.

`(prettyf file` $\texttt{sym}_1$ ... $\texttt{sym}_n$`)` [*special form*]

This function is similar to the previous one, except that it prints the functions $\texttt{sym}_1$ ...$\texttt{sym}_n$ into the output file named `file`. It returns `file` as its value.

For example, to create a file named `foo.ll` containing the text of the definitions of the functions `foo` and `bar`, evaluate `(prettyf "foo.ll" foo bar)`.

`(prettyend)` [*function with no arguments*]

Reclaims the space occupied by the pretty-print functions (about 700 list cells). It can be called when the pretty-printer is no longer being used. The functions return to `autoload` status and the `pretty` feature disappears.

8.0.2 Control of pretty-printer functions

The pretty-printer uses the `lmargin` and `rmargin` functions to calculate the size of line it can use. Two variables control the printing of constants in program:

#:pretty:quotelevel [*variable*]

This variable contains the value to give to the `printlevel` function when a constant is to be printed. By default it is equal to zero.

#:pretty:quotelength [*variable*]

This variable contains the value to give to the `printlength` function when a constant is to be printed. By default it is equal to zero.

8.0.3 Standard pretty-printer format

By default the pretty-printer prints S-expressions on a single line. In the case of overflow, it starts over at the left margin using the following seven standard formats :

```
Format 1 : progns

          (progn
            <e1>
            ...
            <en>)

Format 2 : ifs

           (if <e1>
               <e2>
               ...
               <en>)

Format 3 : defuns

           (defun <e1> <e2>
               <e3>
               ....
               <en>)

Format 4 : conds

           (cond
                 (<e11>
                    <e12> ... <eix>)
                 ....
                 (<en1>
                    <en2> ... <enz>))

Format 5 : selectqs

           (selectqs <e1>
                (<e21>
                    <e22> ... <e2x>)
                ....
                 (<en1>
                    <en2> ... <enz>))
```

```
Format 6 : setqs

           (setq <e1> <e2>
                    ....
                 <en-1> <en>)

Format 7 : tagbodys

           (tagbody
                  <e1>
             <label>
                  ...
                  <en>)
```

These formats are stored in the symbols' P-types. These P-types are accessed with the predefined `ptype` function, so any function can be assigned a default pretty-print format type.

8.0.4 Extending the pretty-printer

It is also possible to define special print formats for extended LE-LISP types. These are functions named `pretty` in the extended type package.

```
? (defstruct foo a b c)
= foo
? (setq v (#:foo:make))
= #:foo:[() () ()
? (#:foo:b v '( 1 2 3 4 5))
= (1 2 3 4 5)
? (#:foo:c v 100)
= 100
? v
= #:foo:[() (1 2 3 4 5) 100]
? (defun #:foo:pretty (x)
?     (princh "<")
?     (pprin (#:foo:b x))
?     (princh "/")
?     (pprin (#:foo:c x))
?     (princh "/")
?     (pprin (#:foo:a x))
?     (princh ">"))
= #:foo:pretty
? (pprint (list v v v))
(<(1 2 3 4 5)/100/()>
 <(1 2 3 4 5)/100/()>
 <(1 2 3 4 5)/100/()>)
= (#:foo:[() (1 2 3 4 5) 100] #:foo:[() (1 2 3 4 5) 100]
#:foo:[() (1 2 3 4 5) 100])
```

Table of contents

8 S-expression pretty-printer **8-1**

8.0.1 Pretty-print functions 8-2

8.0.2 Control of pretty-printer functions 8-2

8.0.3 Standard pretty-printer format 8-3

8.0.4 Extending the pretty-printer 8-4

Function Index

`pretty` [*feature*] 8-2

`(pprint s)` [*function with one argument*] 8-2

`(pprin s)` [*function with one argument*] 8-2

`(pretty sym`$_1$ `...` `sym`$_n$`)` [*special form*] 8-2

`(prettyf file sym`$_1$ `...` `sym`$_n$`)` [*special form*] 8-2

`(prettyend)` [*function with no arguments*] 8-2

`#:pretty:quotelevel` [*variable*] 8-3

`#:pretty:quotelength` [*variable*] 8-3

Chapter 9

Specialized output

This chapter presents three subjects:

- The `format` printing function.
- A set of functions that work on circular or shared lists.
- The multi-language message facility.

9.1 Formatted output

`format` [*feature*]

This feature indicates whether the `format` function is loaded into memory.

9.1.1 Print-formatting function

(`format` `dest cntrl` e_1 ... e_n) [*function with two or more arguments*]

Prints the `cntrl` character string onto the `dest` output channel. This string can contain *print directives* made up of tilde characters (`~`) followed by a format-specification character. When `cntrl` is printed, these directives cause the evaluation of the LISP objects $e_1 \ldots e_n$, which can be of any defined type. The first directive in the control string `cntrl` describes the print format of the e_1 argument, the second that of e_2, etc.

The argument `dest` specifies the output channel used for printing. It can take various values:

- an input/output channel number opened by the either the `openo` or `opena` function,
- the symbol `t`, in which case printing is done on the terminal channel (value `()`),
- the symbol `()`, in which case the output is directed to a character string, which is returned by the `format` function as its value, and no other printing is actually done.

(prinf <cntrl> <e1> ... <eN>) [*SUBR à 1 or N arguments*]

is identical to the preceding function except that output takes place in the current output channel.

(printf <cntrl> <e1> ... <eN>) [*SUBR à 1 or N arguments*]

is identical to the `prinf` function except that it prints a line skip using the `terpri` function at the end of the printing.

The directive most often used is `~A`, which effects the same print format as the `prin` command. Here is an example of printing with the `~A` format:

```
? (defun loves (x y)
?     (prinf "~A loves ~A." x y)
?     (terpri))
= loves   ? (loves 'romeo 'juliet)
romeo loves juliet.
= t
```

Some print formats accept modifiers and numerical parameters.

A *modifier* is one of the two characters : or `@`, occurring immediately before the format specification character. Modifiers are used to change the action of the format directive that they modify. Their specific actions depend, of course, on what they modify.

Here is an example of the `~C` directive, which handles characters:

```
? (format () "the character ~C" #/a)
= "the character a"
```

The `~C` directive can be modified by a colon sign, indicating that the name of the character is required:

```
? (format () "the character named ~:C" #^a)
= "the character named control-A"
```

The `~C` directive can be modified by a `@`, indicating that we want to obtain a character that is readable by a `#`-macro:

```
? (format () "(setq char ~@C)" #^a)
= "(setq char #^A)"
```

Numeric parameters appear between the tilde character and the modifiers, if there are any. They specify various characteristics of the format to which they are applied: the number of columns to be used, the fill character, tab positions and the numeric base for printing.

When several numeric parameters are required, the must be separated by commas. Certain parameters are optional, and take on default values if they are not supplied.

Numeric parameters can be entered in various forms:

- An integer number, read in the current input base.
- A quoted character whose value becomes the internal character code (number) of the

character.

- The '`v`' or '`V`' character. The value of the parameter is the value of the argument $\texttt{e}_i$ that is next in order. The argument printed will then be the following one: namely $\texttt{e}_{i+1}$.
- The `#` character. The parameter's value is the number of $\texttt{e}_i$ arguments remaining to be formatted.

The `~D` directive is used for base-10 numbers. It accepts two optional parameters: the number of columns and the fill character to be used.

First, we use the `~D` directive without parameters:

```
? (format () "the number ~D" 314)
= "the number 314"
```

Here is an example of the `~D` directive with the number of columns to be used, namely `12`:

```
? (format () "the number ~12D" 314)
= "the number          314"
```

The following example shows the `~D` directive with both the number of columns, `12`, and the fill character, namely `'0`:

```
? (format () "the number ~12,'0D" 314)
= "the number 000000000314"
```

The number of columns can be given in the argument list:

```
? (format () "the number ~vD" 10 314)
= "the number        314"
```

Both the number of columns and the fill character can be given in the argument list:

```
? (format () "the number ~v,vD" 10 #/. 314)
= "the number .......314"
```

9.1.2 Format directives

Throughout this section, optional parameters in the syntactic forms of the various format directives are enclosed in square brackets.

~A [*print format*]

This is the **ASCII** print format, which prints an argument in the same way that `prin` would. It has the following syntactic forms:

`~A`

`~`*mincol*[`,`*pad*]`A`

`~<`*mincol*`,`*colinc*`,`*minpad*[`,`*pad*]`A`

The `@` sign can be used as a modifier.

In the case of the syntax involving one or two parameters, *mincol* is the minimum width of the printed string, and the result is padded with either `#\sp`, by default, or the *pad* character. Normally, the string is left-padded. It is right-padded when the `@` modifier is used.

In the case of the syntax involving three or four parameters, the result is padded by the number *minpad* of *pad* characters, then adjusted to a minimum width of *mincol* by increments of *colinc* characters. The respective default values of the four parameters are zero, one, zero and `#\sp`.

```
(format () "~A" 'abcd)              ⟹   "abcd"
(format () "~6A" 'abcd)             ⟹   "  abcd"
(format () "~6@A" 'abcd)            ⟹   "abcd  "
(format () "~6,'*A" 'abcd)          ⟹   "**abcd"
(format () "~1,,2A" 'abcd)          ⟹   "  abcd"
(format () "~8,,2A" 'abcd)          ⟹   "    abcd"
(format () "~8,10,2,v@A" #/u 'abcd) ⟹   "abcduuuuuuuuuuuu"
```

~S [*print format*]

This **S-expression** directive is similar to the previous one, except that it sets the `#:system:print-for-read` indicator. The printed result can then be re-read by the LISP reader. This directive accepts the same parameters and modifiers as the `~A` directive.

```
(format () "~S ~A" "abcd" "efgh")  ⟹   """abcd"" efgh"
```

~R [*print format*]

This **radix** directive has the following syntactic forms:

`~`*base*`R`

`~`*base*`,`*mincol*`[,`*padchar*`]R`

The `@` sign can be used as a modifier.

If the next argument is an integer or rational number, it is printed in the output base referred to as *base*. (See the `obase` function.) If the argument is not numeric, the `~A` format is used. The *mincol* and *pad* parameters, as well as the `@` modifier, play the same rôle as for the `~A` directive.

```
(format () "~2R" 15)        ⟹   "1111"
(format () "~3,5,'*@R" 13)  ⟹   "111**"
```

~D [*print format*]

This **decimal** directive is equivalent to the `~10R` directive, specifying a base-10 print format. It has the following syntactic forms:

`~D`

`~`*mincol*[`,`*padchar*]`D`

The `@` sign can be used as a modifier.

```
(format () "~D" 5)        ⟹   "5"
(format () "~3,'0D" 5)    ⟹   "005"
(format () "~3,'*@D" 5)   ⟹   "5**"
(format () "~7D" 'abcd)   ⟹   "   abcd"
```

~B [*print format*]

This **binary** directive is equivalent to the `~2R` directive, specifying a base-2 print format. `~B` accepts the same parameters and modifier as the `~D` directive.

```
(format () "~5,vB" #/0 15)   ⟹   "01111"
```

~O [*print format*]

This **octal** directive is equivalent to the `~8R` directive, specifying a base-8 print format. `~O` accepts the same parameters and modifier as the `~D` directive.

```
(format () "~#,'0O" 63 7 8)   ⟹   "077"
```

~X [*print format*]

This **hexadecimal** directive is equivalent to the `~16R` directive, specifying a base-16 print format. `~X` accepts the same parameters and modifier as the `~D` directive.

```
(format () "~5@X" 17)   ⟹   "11   "
```

~P [*print format*]

This **English-plural** directive prints the character `s` if the next argument is not equal to `1`. Otherwise, it prints nothing.

It can be used with the signs `@` and `:` as modifiers. The `@` modifier produces either the character `y`, when the next argument is equal to `1`, or the characters `ies` when the next argument is not equal to `1`. If the `:` modifier is present, the argument tested is the one that was just processed by the format function. This allows an argument and its plural suffix to be printed from a single occurrence of the argument in the argument list. The `:` modifier, in this case, inhibits argument pointer incrementation.

```
(format () "~P" 1)                    ⟹   ""
(format () "~P" 'abcd)                ⟹   "s"
(format () "~D tr~:@P/~D win~:P" 7 1)  ⟹   "7 tries/1 win"
(format () "~D tr~:@P/~D win~:P" 1 0)  ⟹   "1 try/0 wins"
(format () "~D tr~:@P/~D win~:P" 1 3)  ⟹   "1 try/3 wins"
```

~C

[*print format*]

This **character** directive prints the character whose internal character code is equal to the next argument.

It can be used with the signs `@` and `:` as modifiers. The `@` modifier assures that the printed character will be re-readable by one of the `#`-macros: `#/`, `#^` or `#\`. The `:` modifier prints the name of the character.

```
(format () "~C" #/+)     ⟹   "+"
(format () "~@C" #/+)    ⟹   "#/+"
(format () "~@C" #\sp)   ⟹   "#\sp"
(format () "~@C" #^A)    ⟹   "#^A"
(format () "~:C" #/+)    ⟹   "+"
(format () "~:C" #^A)    ⟹   "control-A"
(format () "~:C" #^I)    ⟹   "tab"
```

~E

[*print format*]

This **exponential floating-point** directive has the following syntactic form:

~[*width*],[*digits*],[*exp*],[*factor*],[*oflow*],[*pad*],[*letter*]E

The `@` sign can be used as a modifier.

This format prints the next argument, which must be numeric. The *width* parameter specifies the width of the resulting string. If the result is wider than *width* characters, then a string of *oflow* characters, of length *width*, is printed instead of a mangled numeric string.

The *digits* parameter specifies the number of significant digits, and *exp* is the number of digits in the exponent. Character fill with *pad*, if any, is to the left. The *factor* parameter—whose default value is one—indicates the number of digits to the left of the decimal point. The modifier `@` indicates that a plus sign `+` should be printed if the argument is not negative. The exponent is always printed with a sign. By default, the *letter* parameter is `e`: the character that separates the exponent from the mantissa.

If the argument is not a floating-point value, the ~*width*`D` format is used. If *width*, *digits* and *exp* are omitted, free-print format of the `prin` style is used.

```
(format () "~9,2E" 3.14159)              ⟹   "  3.14e+0"
(format () "~13,6,2,vE" -7 3.14159)      ⟹   ".00000003e+08"
(format () "~13,6,2,vE" -6 3.14159)      ⟹   "0.0000003e+07"
(format () "~13,6,2,vE" -5 3.14159)      ⟹   " 0.000003e+06"
(format () "~13,6,2,vE" -4 3.14159)      ⟹   " 0.000031e+05"
(format () "~13,6,2,vE" -3 3.14159)      ⟹   " 0.000314e+04"
(format () "~13,6,2,vE" -2 3.14159)      ⟹   " 0.003142e+03"
(format () "~13,6,2,vE" -1 3.14159)      ⟹   " 0.031416e+02"
(format () "~13,6,2,vE" 0 3.14159)       ⟹   " 0.314159e+01"
(format () "~9,2,1,,'*E" 3.14159)        ⟹   "  3.14e+0"
```

```
(format () "~9,2,1,,'*E" -3.14159)        ⟹   " -3.14e+0"
(format () "~9,2,1,,'*E" 1100.)           ⟹   "  1.10e+3"
(format () "~9,2,1,,'*E" 1.1e+13)         ⟹   "*********"
(format () "~9,3,2,-2,'%@E" 3.14159)      ⟹   "+.003e+03"
(format () "~9,3,2,-2,'%@E" -3.14159)     ⟹   "-.003e+03"
(format () "~10,3,2,2,'?,,'$E" 3.14159)   ⟹   " 31.42$-01"
(format () "~10,3,2,2,'?,,'$E" -3.14159)  ⟹   "-31.42$-01"
(format () "~10,3,2,2,'?,,'$E" 1100.)     ⟹   " 11.00$+02"
(format () "~10,3,2,2,'?,,'$E" 1.1e+13)   ⟹   " 11.00$+12"
```

~F

[*print format*]

This **fixed-format floating-point** directive has the following syntactic form:

~[*width*],[*digits*],[*factor*],[*oflow*],[*pad*]F

The @ sign can be used as a modifier.

This format prints the next argument, which must be numeric. The *width* parameter specifies the width of the resulting character string, with optional padding by the *pad* character to the left. By default, *pad* is the space character: #\sp. The *digits* parameter indicates the number of digits to the right of the decimal point. The *factor* parameter is a scaling factor. The number actually printed is ten to the power *factor* times the argument. If the result string is wider than *width* characters, then a string of *oflow* characters, of length *width*, is printed. The @ modifier indicates that a plus sign + should be printed when the argument is non-negative.

```
(format () "~6F" 3.14159)          ⟹   "3.1416"
(format () "~6F" -3.14159)         ⟹   "-3.142"
(format () "~6F" 100)              ⟹   "100.00"
(format () "~6F" 1234.0)           ⟹   "1234.0"
(format () "~6F" 0.006)            ⟹   "0.0060"
(format () "~,2F" 3.14159)         ⟹   "3.14"
(format () "~6,2F" 3.14159)        ⟹   "  3.14"
(format () "~6,2@F" 3.14159)       ⟹   " +3.14"
(format () "~6,2,1,'*F" 3.14159)   ⟹   " 31.42"
(format () "~6,2,1,'*F" 100)       ⟹   "******"
(format () "~6,2,1,'*F" 0.006)     ⟹   "  0.06"
```

~G [*print format*]

This **general floating-point** directive has the same syntactic structure as the ~E directive:

~[*width*],[*digits*],[*exp*],[*factor*],[*oflow*],[*pad*],[*letter*]G

The @ sign can be used as a modifier.

As in the case of the ~F and ~E directives, this format prints the next argument, which must be numeric. The actual printing is carried out as if either the ~F or the ~E directive were used:

- If the given argument can be handled successfully by means of the fixed-format floating-point ~F directive, then that is how the ~G directive in fact behaves. If a value for the *exp* parameter is supplied, then *exp* is incremented by two to obtain the number of blanks on the right. Otherwise, four blanks are added.
- If, however, the value to be printed overflows when the ~F directive is used, then the ~E directive is applied, with all its default arguments.

~~ [*print format*]

This **tilde** directive has the following syntactic structure:

~[*number*]~

The ~~ directive prints a single tilde character. When the *number* parameter is supplied, a string of tilde characters, of length *number*, is produced.

```
(format () "~~")      ⟹   "~"
(format () "~v~" 3)   ⟹   "~~~"
```

~% [*print format*]

This **newline** directive has the following syntactic structure:

~[*number*]%

The ~% format directive prints a single #\lf character. When the *number* parameter is supplied, a string of #\lf characters, of length *number*, is produced.

This directive is included only to ensure compatibility with COMMON LISP. It is preferable to call the terpri function.

```
(pname (format () "~% a"))    ⟹   (13 10 32 97)
(pname (format () "~2% a"))   ⟹   (13 10 13 10 32 97)
```

~#\lf

[*print format*]

This **skip** directive has the following syntactic structure:

~#\lf *character*

Either a : or a @ character can be used as modifier.

This directive is made up of the tilde character ~ followed immediately by an end-of-line mark.

Without modifiers, it skips over the end-of-line character and any white-space characters that follow. With @, it prints the end-of-line, but skips any white-space that follows. With :, it skips the end-of-line but prints all characters that follow, including white-space, according to the rest of the format statement. This directive is included solely for reasons of COMMON LISP compatibility.

```
(format () (string '(#/~ #\lf #\sp #\tab #/b)))              ⟹   "b"
(pname (format () (string '(#/~ #/: #\lf #\sp #\tab #/b))))  ⟹   (32 9 98)
(pname (format () (string '(#/~ #/@ #\lf #\sp #\tab #/b))))  ⟹   (10 98)
```

~T

[*print format*]

This **tabulation** directive has the following syntactic structure:

~T

~*colnum*,*colinc*T

The character @ can be used as a modifier.

The ~T directive moves the cursor to the next tab stop, printing spaces if necessary to produce white-space. Both parameters have default values of one.

Without @, tabulation is absolute. Spaces are printed up to the column *colnum*. Column-numbering starts at zero. The cursor is displaced to the column following the one that is specified. If the cursor is in column *colnum* or beyond, it is repositioned at the column whose number is *colnum* + k * *colinc*, where k takes on the minimum possible non-negative value.

Used with @, ~T performs relative tabulation. The cursor is moved *colnum* positions forward, and to the column whose number is k * *colinc*, where k takes on the minimum possible value.

```
(format () "~T")              ⟹   " "
(format () "~8T")             ⟹   "        "
(format () "ab~8,2T")         ⟹   "ab      "
(format () "0123456789~8T")   ⟹   "0123456789"
(format () "0123456789~8,5T") ⟹   "0123456789   "
(format () "ab~2,5@T")        ⟹   "ab   "
```

~*

[*print format*]

This **ignore** directive has the following syntactic structure:

~[*number*]%

Either a : or a @ character can be used as modifier.

The ~* directive ignores the next argument. The default value of *number* is one, except when the modifier @ appears, in which case its default value is zero. Without modifiers, the next *number* arguments are skipped. With the : modifier, the argument pointer is moved back by *number* arguments. With the @ modifier, the $number^{th}$ argument of the original argument list, counting from zero, will be read next. In an indirection or an iteration, access is relative to the sublists involved.

```
(format () "~2*~A" 1 2 3 4)         ⟹   "3"
(format () "~D ~:* ~D" 1 2)         ⟹   "1  1"
(format () "~D~2* ~:2*~D" 12 3 4)   ⟹   "12 3"
(format () "~D ~D ~@*~D" 1 2 3)     ⟹   "1 2 1"
```

~?

[*print format*]

This **indirection** directive has the following syntactic structure:

~?

The @ character can be used as modifier.

The ~? format directive performs an indirection. The next argument, which must be a character string, is taken to be the format specification. If no @ modifier is given, the argument to this format specification is the argument that follows it, which should therefore be a list. With the @ modifier present, the arguments following the format specification are used directly.

```
(format () "~? ~D" "<~A ~D>" '("foo" 5) 7)   ⟹   "<foo 5> 7"
(format () "~@? ~D" "<~A ~D>" "foo" 5 7)     ⟹   "<foo 5> 7"
```

~[;]

[*print format*]

This **conditional** directive has the following syntactic structures:

~[*number*][$strg_0$~;$strg_1$ ... ~;$strg_n$~]

~[*number*][$strg_0$~;$strg_1$ ... ~:;*default*~]

~:[*false*~;*true*~]

~@[*true*~]

Warning: In the first two syntactic structures shown above, the square brackets around the *number* parameter indicate, as usual, that it is optional. But all other square brackets included in these four lines—and printed in a different typeface—are actual terminal characters that must appear as such in your LISP code.

This directive performs a conditional print format. The $strg_i$ format is used, where i is equal

to *number* if it is supplied, or to zero if *number* is not present. If there are not enough *strg* parameters, then *default* is used. If this last parameter is not given, then nothing is printed. The colon modifier : serves as a boolean argument. If the @ modifier is used, then either the argument is (), in which case nothing is printed, and the next argument will be used as the next format, or the argument does not equal (), and it is then printed according to the format *true*. A ~^ in a conditional returns to the embedding ~? or ~{, if this exists.

```
(format () "~[siamese~;manx~;persian~] cat" 0)                    ==>  "siamese cat"
(format () "~[siamese~;manx~;persian~] cat" 2)                    ==>  "persian cat"
(format () "~[siamese~;manx~;persian~] cat" 8)                    ==>  " cat"
(format () "~[siamese~;manx~:;persian~] cat" 8)                   ==>  "persian cat"
(format () "~:[true~;false~]" t)                                  ==>  "false"
(format () "~:[true~;false~]" ())                                 ==>  "true"
(format () "~@[ print level = ~D~]~@[ print length = ~D~]" () 5)  ==>  " print length = 5"
```

~{}

[*print format*]

This **iteration** directive has the following syntactic structure:

~[*number*]{*strg*~[:]}

The modifiers : and @ can be used, possibly combined.

This format directive performs an iteration. *strg* is applied to as many arguments as are given.

If *number* is supplied, at most *number* loops through *strg* will be performed. If a ~:} is present, at least one loop is performed, unless *nb* is equal to zero. If *strg* is empty, the next argument—which should be a character string—is used. Without modifiers, the iteration is executed using the argument that immediately follows, which must in this case be a list. With @ all the remaining arguments are used. If a : is present, each iteration is performed on a different constituent sub-list each time around. Note that the ~{ and ~[format directives can only be embedded if they concern formats of different type.

```
(format () "The winners are:~ ~S~." '(fred harry))       ==>  "The winners are: fred harry."
(format () "Pairs:~ <~S,~S>~." '(a 1 b 2))               ==>  "Pairs: <a,1> <b,2>."
(format () "Pairs:~ <~S,~S>~." '(a 1))                   ==>  "Pairs: <a,1>."
(format () "Pairs:~@ <~S,~S>~." 'a 1 'b 2)               ==>  "Pairs: <a,1> <b,2>."
(format () "Pairs:~:@ <~S,~S>~." '(a 1) '(b 2) '(c 3))  ==>  "Pairs: <a,1> <b,2> <c,3>."
(format () "~2~D ~" '(1 2 3 4))                          ==>  "1 2 "
```

~^

[*print format*]

This **escape** directive has the following syntactic structure:

~[*n1*],[*n2*],[*n3*][:]^

This format directive performs an escape. Formatting is terminated from within indirections or iteration loops, or if no format directives remain.

If no parameters are given, the number of remaining arguments is considered to be the sole parameter. If there is only one parameter, its value is examined and, if it is zero, the exit is performed. If there are two parameters and they are equal, the exit occurs. If three parameters are given, then the exit occurs if *n1* <= *n2* or if *n2* <= *n3*, or both. In an indirection with a :, the escape occurs at the sub-list level, except when the : modifier is present.

```
(format () "Done. ~^^D warning~:P.~^^D error~:P.")       ==>  "Done. "
(format () "Done. ~^^D warning~:P.~^^D error~:P." 3)     ==>  "Done. 3 warnings."
(format () "Done. ~^^D warning~:P. ~^^D error~:P." 1 5)  ==>  "Done. 1 warning. 5 errors."
```

9.2 Handling circular or shared objects

Although the print limiting functions described in chapter 6 truncate the display of circular or shared structures, and thus give an abbreviated view of them, they still do not show clearly which list cells or vectors are actually being shared.

Special functions for printing and reading circular objects (lists, vectors, circular symbols on packages) are furnished in the standard library. They can be loaded with the following command:

```
(libload libcir)
```

libcir [*feature*]

This flag indicates whether the library of special functions for circular objects has been loaded.

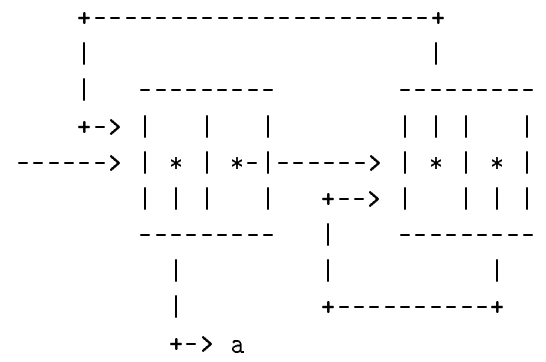
(#n=) [*#-macro*]

Enumerates a shared or circular object that can then be referenced by the **#n#** macro.

(#n#) [*#-macro*]

References the n^{th} shared or circular object, numbered by **#n=**.

For example, **#1=(a . #2=(#1# . #2#))** has the following structure:



(cirprin e) [*function with one argument*]

(cirprinflush e) [*function with one argument*]

(cirprint e) [*function with one argument*]

These functions print their arguments using the **#**-macros **#n=** and **#n#** on circular or shared constructions that are passed to them, if they cannot be printed normally.

```
(printlevel 8)                          ==>  8
(progn (setq e '(1)) (rplaca e e) e)    ==>  (((((((&)))))))
(cirprint e)                            ==>  #1=(#1#)
(setq s (cirlist 1))                    ==>  (1 1 1 1 1 1 1 1 ...
(cirprint s)                            ==>  #1=(1 . #1#)
(cirprint '(a a))                       ==>  (a a)
(progn (setq v #[0]) (vset v 0 v) v)    ==>  #[#[#[#[#[#[#[&]]]]]]]
(cirprint v)                            ==>  #1=#[#1# ]
(progn (setq x '(p q) y (list x 'foo x))
       (rplacd (last y) (cdr y)) y)     ==>  ((p q) foo (p q) foo (p q) ...
(cirprint y)                            ==>  (#2=(p q) . #1=(foo #2# . #1#))
```

(cirequal e1 e2) [*function with two arguments*]

(cirnequal e1 e2) [*function with two arguments*]

These functions compare their two arguments, even if they involve circular or shared constructions. In these cases, equality is understood as structure isomorphism.

```
(setq x '(a) y (list x x) z (list '(a) '(a)))  ==>  ((a) (a))
(cirprint y)                                   ==>  (#1=(a) . #1#)
(equal y z)                                    ==>  t
(cirequal x y)                                 ==>  ()
```

(circopy e) [*function with one argument*]

Copies its argument, reproducing circular or shared structures.

```
(progn (setq x (circopy y)) (cirprint x))  ==>  (#1=(a) . #1#)
(cirequal x y)                             ==>  t
```

#:libcir:package-parano [*variable*]

If this variable is equal to **t**, it indicates that packages of symbols can contain circular structures. Use of such structures is normally strictly forbidden. Functions operating on

circular or shared objects are seriously slowed down, but in this way some nasty errors can be chased away.

```
(progn (setq x 'foo) (packagecell x x) x)  ==>  ...
(cirprint x)                               ==>  #1=#:#1#:foo
```

9.3 Multi-language messages

Products based upon LE-LISP are being marketed in many different countries. This means that LE-LISP messages have to be localized into various foreign languages. The goal of the present section is to describe an easy and efficient methodology for handling this question.

The initial LE-LISP file called `startup.ll` offered a rudimentary solution to this problem. The global variable `#:system:foreign-language` had the value `t` when messages were to appear in English, and `()` when they were to appear in French. This variable was used in LE-LISP libraries in the following manner:

```
(defvar :errnht
   #+ #:system:foreign-language  "not a Hash Table"
   #- #:system:foreign-language  "L'argument n'est pas une table de hachage"
)

(defvar :errbht
   #+ #:system:foreign-language  "Bad type for a Hash Table"
   #- #:system:foreign-language  "Le type de la table de hachage est inconnu"
)
   ...
   (error 'get-hash ':errnht foo)
   ...
```

This solution is too limited. The choice of a language was made when the file was loaded and about to be interpreted, or when the file was compiled, and it could not be changed afterwards. Only two languages could be handled, and you had to create a separate compiled file for each language.

The new solution offers the following advantages:

- You can handle several different languages.
- You can change your choice of a language, dynamically, during program execution.
- If you prefer, you can limit your choice to a single language.
- For reasons of compatibility, the new solution does not interfere with the old one.

The basic idea is that languages and messages are handled within the context of a global multi-language database that can be shared by all applications.

`messages` [*feature*]

This feature indicates whether the multi-language database is loaded into memory.

9.3.1 Languages

From the user's viewpoint, a language is simply a symbol. There are two predefined languages: `english` and `french`. Any mention a language which is not a symbol raises the error: `error-wrong-language`. Any use of a language that has not been recorded in the database raises another error: `error-not-recorded-language`. Here are the standard definitions of these errors:

```
(defmessage error-wrong-language
   (english "not a language name")
   (french  "l'argument n'est pas un nom de langue"))

(defmessage error-not-recorded-language
   (english "not a recorded language")
   (french  "l'argument n'est pas une langue enregistre'e"))
```

(record-language language) [*function with one argument*]

The `language` argument is the name of a new language which is recorded in the multi-language database. If this language existed already, the database is not modified. The function returns `language` in the first case, and `()` in the second. This function can raise the error: `error-wrong-language`.

```
(record-language 'polish)  ⟹  polish
```

(current-language language) [*function with an optional argument*]

Without an argument, this function returns the symbolic name of the current language. Its argument, if it has one, is a language that was previously recorded by the `record-language` function. In this case, `language` becomes the new current language. This function can raise the errors `error-wrong-language` and `error-not-recorded-language`.

(default-language language) [*function with an optional argument*]

Without an argument, this function returns the symbolic name of the default language. Its argument, if it has one, is a language that was previously recorded by the `record-language` function. In this case, `language` becomes the new default language. The default language is `english`.
This function can raise the errors `error-wrong-language` and `error-not-recorded-language`.

(remove-language language) [*function with one argument*]

Removes `language` and all messages associated with this language in the database. If `language` did in fact exist in the database, then this symbol is returned. If not, the function returns `()`. The two predefinied languages `english` and `french` can't be removed. This function can raise the two errors: `error-wrong-language` and `error-not-recorded-language`.

(message-languages) [*function with no arguments*]

Returns the list of all the language known to the database. This list does *not* represent all or even part of the actual data structures recorded in the multi-language database. So, modifications to this list have no effect whatsoever on the database itself.

```
(message-languages)  ⟹  (polish esperanto english french)
```

9.3.2 Messages

A message name is a symbol of the variable kind. The functions described in this section raise the error, `error-wrong-message`, if the `message` argument is not a symbol of this kind. The `string` value that is associated with a message name must be a character string. If not, the error `error-not-string-message` is raised. Here are the standard definitions of these errors:

```
(defmessage error-wrong-message
   (english "not a message name")
   (french  "l'argument n'est pas un nom de message"))

(defmessage error-not-string-message
   (english "not a message string")
   (french  "l'argument n'est pas une chaîne de message"))
```

(get-message `message`**)** [*function with one argument*]

Returns the character string associated with the message name `message` in the current language or, if this message is not defined, in the default language. The current language—if such a language still exists (see the `remove-language` function)—is the one returned by the `current-language` function. The default language is the one returned by the `default-language` function. If no such message exists, the string representing the `message` name itself is returned. This function can raise the error `error-wrong-message`.

#M [*sharp macro*]

#m [*sharp macro*]

The `#M` macro character enables you to access the value of a message:

```
#Mfoo     is read as     (get-message (quote foo))
#M:gee    is read as     (get-message (quote :gee))
```

(get-message-p `message`**)** [*function with one argument*]

Identical to `get-message`, except that `get-message-p` never raises an error. It is generally used as a predicate.

(put-message `message language string`**)** [*function with three arguments*]

Within the multi-language message database, this function associates `string` with `message` in the language corresponding to `language`. The function returns the value `message`. It can raise the following errors: `error-wrong-language`, `error-not-recorded-language`, `error-wrong-message` and `error-not-string-message`.

(remove-message message language) [*function with two arguments*]

Removes `message`, in the language corresponding to `language`, from the multi-language message database. This function returns the value `message`, and can raise the following errors: `error-wrong-language`, `error-not-recorded-language` and `error-wrong-message`.

(defmessage message def_1 ... def_n) [*macro*]

This macro provides a convenient way of associating a list of values in different languages to a certain message name. In other words, it lets you group together, in one place, all the linguistic variants of a particular message. The function returns the value `message`, and can trigger the following errors: `error-wrong-language`, `error-not-recorded-language`, `error-wrong-message` and `error-not-string-message`.

In LE-LISP, `defmessage` could be defined in the following manner:

```
(defmacro defmessage (msg . ldef)
   `(progn ,@(mapcar (lambda (def) `(put-message ,msg ,@def))
                     ,ldef)
           ',msg))
```

Here is an example:

```
(defmessage smeci-error-not-a-prototype
  (english    "Not a prototype")
  (french     "L'argument n'est pas un prototype")
  (polish     "Argumentyws nie jest prototypuxzy")
  (esperanto  "La argumento ne estas prototipo"))
```

(get-all-messages language) [*function with one argument*]

Returns the list of all messages recorded in the language `language`. `get-all-messages` might raise the errors `error-wrong-language` or `error-not-recorded-language`.

To find out how many messages have been pre-recorded in LE-LISP:

```
(length (get-all-messages 'english))  ==>  167
(length (get-all-messages 'french))   ==>  167
```

9.3.3 Advice

Since the names of messages are global, it is important to use different message names in each application. You can use LE-LISP packages, or you might prefer to write names in full: for example, `#:smeci:error22` or `smeci-error-not-a-prototype`.

Let us look at some examples of the handling of multi-language messages. You can temporarily change the language of a message in the following manner:

```
(with ((current-language 'polish))
      ....)
```

The following example shows how to obtain the list of all the messages defined in a particular language:

```
(defun get-all-messages (language)
   (with ((current-language language)
          (default-language language))
      (mapcoblist (lambda (msg)
                    (when (get-message-p msg)
                          (list msg))))))
```

To destroy all messages in all languages except English, act as follows:

```
(mapc (lambda (lang) (when (neq lang 'english)
                           (remove-language lang)))
      (message-languages))
```

Finally, you might wish to verify that all the messages in English, say, have French equivalents. The following example does this, and it returns you the list of English messages that do not yet have French equivalents:

```
(defun verifyfrench ()
   (with ((current-language 'english)
          (default-language 'english))
      (mapcoblist (lambda (msg)
                    (when (get-message-p msg)
                          (with ((current-language 'french)
                                 (default-language 'french))
                             (if (get-message msg)
                                 ()
                                 (list msg))))))))
```

Table of contents

9 Specialized output **9-1**

9.1 Formatted output 9-1

9.1.1 Print-formatting function 9-1

9.1.2 Format directives 9-3

9.2 Handling circular or shared objects 9-12

9.3 Multi-language messages 9-14

9.3.1 Languages 9-15

9.3.2 Messages 9-16

9.3.3 Advice 9-17

Function Index

`format` [*feature*] 9-1

`(format dest cntrl e`$_1$ `... e`$_n$`)` [*function with two or more arguments*] 9-1

`(prinf <cntrl> <e1> ... <eN>)` [*SUBR à 1 or N arguments*] 9-2

`(printf <cntrl> <e1> ... <eN>)` [*SUBR à 1 or N arguments*] 9-2

`~A` [*print format*] 9-3

`~S` [*print format*] 9-4

`~R` [*print format*] 9-4

`~D` [*print format*] 9-4

`~B` [*print format*] 9-5

`~O` [*print format*] 9-5

`~X` [*print format*] 9-5

`~P` [*print format*] 9-5

`~C` [*print format*] 9-6

`~E` [*print format*] 9-6

`~F` [*print format*] 9-7

`~G` [*print format*] 9-8

`~~` [*print format*] 9-8

`~%` [*print format*] 9-8

`~#\lf` [*print format*] 9-9

`~T` [*print format*] 9-9

`~*` [*print format*] 9-9

`~?` [*print format*] 9-10

`~[;]` [*print format*] 9-10

`~{}` [*print format*] 9-11

`~^` [*print format*] 9-11

`libcir` [*feature*] 9-12

(#n=) [*#-macro*] 9-12
(#n#) [*#-macro*] 9-12
(cirprin e) [*function with one argument*] 9-13
(cirprinflush e) [*function with one argument*] 9-13
(cirprint e) [*function with one argument*] 9-13
(cirequal e1 e2) [*function with two arguments*] 9-13
(cirnequal e1 e2) [*function with two arguments*] 9-13
(circopy e) [*function with one argument*] 9-13
#:libcir:package-parano [*variable*] 9-13
messages [*feature*] 9-14
(record-language language) [*function with one argument*] 9-15
(current-language language) [*function with an optional argument*] 9-15
(default-language language) [*function with an optional argument*] 9-15
(remove-language language) [*function with one argument*] 9-15
(message-languages) [*function with no arguments*] 9-16
(get-message message) [*function with one argument*] 9-16
#M [*sharp macro*] 9-16
#m [*sharp macro*] 9-16
(get-message-p message) [*function with one argument*] 9-16
(put-message message language string) [*function with three arguments*] 9-17
(remove-message message language) [*function with two arguments*] 9-17
(defmessage message def$_1$ **...** **def**$_n$**)** [*macro*] 9-17
(get-all-messages language) [*function with one argument*] 9-17

Chapter 10

Rational and complex arithmetic

Le-Lisp has libraries which extend the basic generic arithmetic to operations on relative, rational, and complex numbers.

Since the version 15.22, Le-Lisp has a new, very fast implementation of relative and rational numbers: the result of a joint development effort at DEC PRL [Serpette&al. 1989] and INRIA [mvs 89].

In order to load the old rational number library, named `ratio`, type the following:

```
(loadmodule 'ratio)   or
^Aratio
```

To load the new, and preferred, rational number library, type:

```
(loadmodule 'bnq)   or
^Abnq
```

To load the complex number library, named `complex`, type:

```
(loadmodule 'complex)   or
^Acomplex
```

`ratio` [*feature*]

`q` [*feature*]

`complex` [*feature*]

These features indicate whether the various libraries mentioned above are present.

10.1 Rationals (file Q)

10.1.1 Rational number I/O

Rationals are entered in the form z/n, where the numerator z is a relative number, and the denominator n is a positive whole number or zero.

```
12345678910  ⟹  12345678910
00000123     ⟹  123
-1234567     ⟹  -1234567
2/3          ⟹  2/3
4/6          ⟹  2/3
5/1          ⟹  5
-10/2        ⟹  -5
1/0          ⟹  1/0
-2/0         ⟹  1/0
```

The system prints out the rationals in a compact form. Insignificant zeros are suppressed, values are simplified (numerator and denominator are each divided by their greatest common denominator, or gcd), and values with denominator equal to 1 are transformed into elements of Z (the relative integers).

The evaluation of a rational number returns the number itself, so it need not be *quoted.*

Notice the presence in the rationals of the values for positive and negative infinity, `1/0` and `-1/0` (respectively), as well as a value representing undefined results, `0/0`. Any arithmetic operation containing an expression whose value is `0/0` returns `0/0`, which can be considered to be an error value. The undefined value `0/0` actually represents the closed interval [`-1/0,1/0`] which contains all the rationals including 'plus infinity' and 'minus infinity.' Due to the presence of these elements, the Q of LE-LISP does not qualify as a field.

Decimal output – approximating values in Q

In addition to the standard LE-LISP print routines, which display rational values in their exact reduced forms, a variable-precision approximate decimal form is also available.

The function which sets the output precision is:

`(precision n)` *[function with an optional argument]*

Called with no argument, `precision` returns the current print precision. If an argument is supplied, the print precision is determined according to the following scheme:

If the argument is any non-numeric constant, including `t`, the print mode is set to produce exact reduced representation of rationals.

If the argument is a floating-point value, its integer part indicates the number of digits that should be printed after the decimal point in the current output base (`obase`).

When the denominator is small, the decimal form (which is always periodic) is exact, so LE-LISP

prints the repeating part between braces (`{` and `}`). Otherwise the number of digits requested is printed after the decimal point, followed by an ellipse (...).

If the argument is a positive integer of type `fixp`, it indicates the number of terms to be printed in continuous normal fraction notation.

Rationals printed in any form other than `(precision t)` are not re-readable by the LE-LISP reader.

```
(precision 10.)   ⟹   10.
22/7              ⟹   3.{142857}       ; exact representation
333/106           ⟹   3.1{415094339...}   ; 10 digits to the right of .
103993/33102      ⟹   3.1{415926530...}
(precision 10)    ⟹   10
22/7              ⟹   /3 7/            ; 22/7 = (+ 3 (/ 7))
333/106           ⟹   /3 7 15/
103993/33102      ⟹   /3 7 15 1 292/
```

10.1.2 Tests for type

Objects can be tested for membership in the set of integers Z (= ...,-1,0,1,2,...) or in the rationals Q by using the predicates:

`(rationalp q)` [*function with one argument*]

Returns the value of `q` if it is rational, or `()` if not.

```
(precision t)        ⟹   t
(rationalp 234567)   ⟹   234567
(rationalp -4/6)     ⟹   -2/3
(rationalp 2.34567)  ⟹   ()   ; a float is not a rational
```

`(integerp z)` [*function with one argument*]

Returns the value of `z` if it is integral, or `()` if not.

```
(integerp -4/2)   ⟹   -2
(integerp 2)      ⟹   2
(integerp 4/3)    ⟹   ()
(integerp 2.)     ⟹   ()    ; a float is not an integer.
```

These two functions are generic. To extend them to type `foo`, it is sufficient to define `#:foo:integerp` and `#:foo:rationalp`.

10.1.3 Generic rational arithmetic

All of the generic arithmetic functions of LE-LISP (see chapter 4) work on Q, and try when possible to perform exact calculations. The syntax of these calls is the same as those with LE-LISP small integer arguments.

```
(+ 3 (/ (+ 7 (/ 15))))  ⟹  333/106
(quotient 333/106 1)    ⟹  3        ; modulo is in #:ex:mod
(modulo 333/106 1)      ⟹  15/106   ; here it is now!
(* 22/7 7/11 -1/3)      ⟹  -2/3
(< 1/3 2 7/2)           ⟹  1/3
(* 1/0 0)               ⟹  0/0      ; q*0=0 is false in q
(- 1/0)                 ⟹  1/0
(+ 1/0 -1/0)            ⟹  0/0
```

Integer division (`quotient` n d), given n and d in Q, returns a quotient d in Z and a remainder r (stored in the variable `#:ex:mod`) in Q such that:

```
n = d*q + r, with 0 <= r < abs(d).
```

10.1.4 Functions limited to Z

(gcd z_1 ... z_n) *[function with a variable number of arguments]*

Calculates the greatest common divisor of its relative integer arguments $z_1 \ldots z_n$.

(pgcd z0 z1) *[function with two arguments]*

Similar to the last function, but limited to two arguments and without verification of their type(s).

(even? z) *[function with one argument]*

Returns the value of `z` if this is an even relative integer, and `()` if not.

(fact n) *[function with one argument]*

Calculates the factorial `1*2* ... *n = n!`, for positive integer arguments. *NB*: By convention, `0!` is equal to `1`.

(fib n) *[function with one argument]*

Computes the n^{th} Fibonacci number, where F(0)=0, F(1)=1, F(i+2) = F(i) + F(i+1). (Of course, `n` must be a positive integer value.)

```
(gcd 864164 11578 -168 252)  ⟹  14
(fact 30)                    ⟹  265252859812191058636308480000000
(fib 200)                    ⟹  280571172992510140037611932413038677189525
```

10.1.5 Functions limited to rational arguments (limited to Q)

The `numerator` and `denominator` functions give access to the numerator and the denominator, respectively, of rationals, without reducing them by their common factors. Computation on Q is

done directly on the representation of rationals as entered; reduction is explicitly done only when `prin` or `integerp` functions are called.

`(numerator f)` *[function with one argument]*

`(denominator f)` *[function with one argument]*

```
(numerator 6/10)          ⟹   6      ; no reduction done
(denominator 6/10)        ⟹   10
(setq a (prin 6/10))      ⟹   3/5    ; reduction done by print
(numerator a)             ⟹   3      ; a has been reduced
(numerator 12)            ⟹   12
(denominator (fib 100)))  ⟹   1
```

The power function (**) tries in more cases than the `power` function to do exact arithmetic.

`(** n m)` *[function with two arguments]*

Computes the product `n * n * ... * n` (`m` times), for `m` a positive integer.

```
(** 2 128)   ⟹   340282366920938463463374607431768211456
(** 10 10)   ⟹   10000000000
(** 2/3 10)  ⟹   1024/59049
```

10.1.6 Two examples

The zeta function (`zeta` n e) computes the sum of the terms 1/i**e, for i from 1 to n.

```
(defun zeta (n e)
    (let ((z 0))
         (for (i 1 1 n) (setq z (+ z (/ (** i e)))))
         z))
```

For e=1, one obtains the harmonic numbers, a divergent series.

For e=2, the series converges slowly toward the square of pi divided by six.

For e=3, the series converges toward zeta(3), which Apéry showed to be transcendant.

Let's compute (qe n), the sum of the terms $1/i!$, for i from 1 to n. This expression converges rapidly toward the number e.

```
(defun qe (n)
    (let ((e 0))
         (for (i 0 1 n) (setq e (+ e (/ (fact i)))))
         e))

(zeta 10 1)      ⟹   7381/2520
(zeta 10 2)      ⟹   1968329/1270080
```

```
(zeta 10 3)       ⟹   19164113947/16003008000
(qe 10)           ⟹   9864101/3628800
(precision 20.)   ⟹   20.
(zeta 10 1)       ⟹   2.92896825396825396825...
(zeta 10 2)       ⟹   1.54976773116654069035...
(zeta 10 3)       ⟹   1.19753198567419325166...
(qe 10)           ⟹   2.71828180114638447971...
```

10.2 Complex numbers (the field C)

After charging `complex`, the complex number manipulation package, the symbols `i` and `pi` have the values [i] and 3.141593.... It would not be wise to change their values.

10.2.1 Complex number I/O

`[` *[macro character]*

Complex numbers are printed between square brackets (`[,]`) in the form `[xi+y]`, where x is the imaginary part and y is the real part. Some simplifications are used: `[xi]` when y=0, `[xi-y]` when y<0, `[i+y]` when x=1, and `[-i+y]` when x=-1. Thus the pure imaginary number is known as `[i]`, and its square is equal to -1.

`#C` *[#-macro]*

Complex numbers can be also represented in the form #C(y x), where x is the imaginary part and y is the real part. In order to get this printed representation, evaluate (`precision` 'cl).

```
(precision 'cl)   ⟹   cl
[i]               ⟹   #C(0 1)
[2i+1]            ⟹   #C(1 2)
[-i+1]            ⟹   #C(1 -1)
```

10.2.2 Tests for type

`(complexp c)` *[function with one argument]*

Returns the value of `c` if it is complex, or `()` if not.

`(realp r)` *[function with one argument]*

Returns the value of `r` if it is numeric and not complex, or `()` if not. This is a generic function which can be extended to the type `foo` by defining `foo:realp`.

```
(realp 2)         ⟹   2
```

```
(realp 2.3)          ⇒   2.3
(realp [2/3])        ⇒   2/3
(realp [i])          ⇒   ()
(complexp [-1234])   ⇒   ()
(complexp [-i])      ⇒   [-i]
```

10.2.3 Complex generic arithmetic

The arithmetic functions known as $+, -, *$, and $\div$ have been extended to the complex numbers.

```
(+ [i] 3)            ⇒   [i+3]
(* [i] [i])          ⇒   -1
(* [i] (+ [i] [i]))  ⇒   -2
(* (* 1/3 [i]) [i])  ⇒   -1/3
(+ [i] (/ [i]))      ⇒   0
(/ [2i+3])           ⇒   [-2/13i+3/13]
(<?> [i] 1)          ⇒   [i-1]
(<?> [-2i+3] [i+1])  ⇒   [-i+1]
(float [2/3i+5])     ⇒   [.6666666i+5.]
```

Whole division using `quotient` makes no sense in C, and raises an error. The same is true of `truncate` and `floor`.

The `float` function converts real and imaginary parts of complex numbers into floating-point numbers. The comparison function applied to two complex numbers, `(<?> [ai+b] [ci+d])`, returns the value `[(<?> a c)i + (<?> b c)]` which only makes mathematical sense when the result is 0, that is when `[ai+b] = [ci+d]`.

The domain of the `sqrt` and `log` functions extends to C following the formulas:

`sqrt(-r)=rsqrt(r)  log(-r)=pi*i+log(r)`.

The `asin` and `acos` functions are defined on C by:

acos(z)=pi/2-asin(z) and asin(z)=-i*log(iz+sqrt(1-z*z)).

```
(sqrt -2)   ⇒   [1.414213i]
(log -2)    ⇒   [3.141593i+.6931472]
(log -1)    ⇒   [3.141593i]
(log 0)     ⇒   1/0
(sqrt -1)   ⇒   [i]
(asin 2)    ⇒   [-1.316958i+1.570796]
(acos 2)    ⇒   1.570796
```

10.2.4 Functions limited to the complex numbers (limited to C)

(makecomplex c i) *[function with two arguments]*

This function returns [ii + c], in which the arguments `i` and `c` must be reals.

(realpart c) *[function with one argument]*

If c = [xi+y], realpart returns y.

(imagpart c) *[function with one argument]*

If c = [xi+y], imagpart returns x.

(conjugate c) *[function with one argument]*

If c = [xi+y], conjugate returns [-xi+y].

```
(conjugate [2/5i+3/5])  ⟹   [-2/5i+3/5]
(realpart [2i-3])       ⟹   -3
(imagpart [2i-3])       ⟹   2
```

10.2.5 Polar coordinates

We write [rc:tc] for the polar representation of c = [yi+x]. The value of the angle tc=arctan(y/x) modulo pi is determined by the location of [yi+x] in the complex plane. The following relationships hold:

```
if y>0 and x>0, 0<tc<pi/2.          if y>0 and x<0, pi/2<tc<pi.
if y<0 and x>0, -pi/2<tc<0.         if y<0 and x<0, -pi<tc<-pi/2.

                  if y>0 and x=0, tc=pi/2.
                  if y<0 and x=0, tc=-pi/2.
```

The modulus rc is given by (abs c), while the angle tc can be obtained with:

(phase c) *[function with one argument]*

If c = [xi+y] = [rc:tc], phase returns tc.

(signum c) *[function with one argument]*

If c = [xi+y] = [rc:tc], signum returns [1:tc] = c / (abs c).

```
(phase [i+1])       ⟹   .7853982
(phase [-i+1])      ⟹   -.7853982
(phase [i-1])       ⟹   2.356194
(phase [-i-1])      ⟹   -2.356194
(abs [i+1])         ⟹   1.414214
(signum (/ pi 2))   ⟹   1
```

The complex logarithm and exponential functions are defined in relation to the polar representation as follows:

log[r:t] = [ti+log(r)] and exp[r:t] = [exp(r)sin(t)i+exp(r)cos(t)] .

(cis r) *[function with one argument]*

Computes [sin(r)i+cos(r)].

The `sqrt` function is defined for the complex numbers by:

```
sqrt(z) = exp(log(z)/2)
```

The trigonometric functions are defined by:

```
sin(z)  = (exp[ci]-exp[-ci])/2i
cos(z)  = (exp[ci]+exp[-ci])/2
atan(z) = -i*log([zi+1]/sqrt(1+z*z))
tan(z)  = sin(z)/cos(z)
```

The results of the following calculations depend on the floating-point precision used.

```
(log (exp [3i+2]))   ==>   [3.i+2.]
(exp (log [3i+2]))   ==>   [3.i+2.]
(exp (* i pi))       ==>   [0.i-1.]
(setq a (sqrt [i]))  ==>   [.7071067i+.7071067]
(* a a)              ==>   [i]
(asin (sin i))       ==>   [i]
(cos (acos [i]))     ==>   [i]
(atan [i])           ==>   1/0
(atan (tan [i]))     ==>   [i]
```

10.2.6 Hyperbolic functions

(sinh z) *[function with one argument]*

Computes `(exp(z)-exp(-z))/2`.

(cosh z) *[function with one argument]*

Computes `(exp(z)+exp(-z))/2`.

(tanh z) *[function with one argument]*

Computes `sinh(z)/cosh(z)`.

(asinh z) *[function with one argument]*

Computes `log(z+sqrt(1+z*z))`.

`(acosh z)` *[function with one argument]*

Computes `log(z+(z+1)sqrt((z-1)/z+1))`.

`(atanh z)` *[function with one argument]*

Computes `log((1+z)sqrt(1-1/x*x))`.

10.3 A complex mini-extension of generic arithmetic

This example, the construction of a simplified version of the module `complex`, illustrates the idea of an extension to the generic arithmetic package. While it is incomplete, does no error checking, and does not exactly agree with the preceding description of the complex numbers (C), we present it for its pedagogical utility.

First we put ourselves in the `C` package:

```
(setq #:sys-package:colon 'C)
```

10.3.1 Representation

A complex C is represented by its real part (:r c) and its complex part (:i c).

```
(defstruct C r i)
```

To test the type C:

```
(defun complexp (c) (eq 'C (type-of c)))
```

To construct a+ib = [a:b] :

```
(defun makecomplex (a b)
    (if (= 0 b)
        a
        (let ((c (:make)))
             (:i c b)
             (:r c a)
             c)))
```

The pure complex number i*i=-1:

```
(defvar i (makecomplex 0 1))
```

10.3.2 I/O for C

By convention, a+ib is written [bi+a]:

```
(defun :prin (c)
    (prin "[" (:i c) "i+" (:r c) "]"))
```

To be able to reread these complex numbers:

```
(dmc  |[| ()
      (let ((j) (r))
           (setq j          ; read the imaginary part
                (with ((typecn #/i 'csep)) (read)))
           (readcn)         ; read i.
           (readcn)         ; read the +.
           (setq r (read))  ; read the real part.
           (readcn)         ; read the ].
           (makecomplex r j)))
```

10.3.3 Complex arithmetic

The comparison is only defined in the case of equality:

```
(defun :<?> (a b)
    (if (and (complexp b) (= (:i a) (:i b)) (= (:r a) (:r b)))
        0
        (error '<?>
               "complex comparison undefined"
               (list a b))))
```

Binary addition:

```
(defun :+ (c r)
    (if (complexp r)
        (makecomplex (+ (:r c) (:r r)) (+ (:i c) (:i r)))
        (makecomplex (+ r (:r c)) (:i c))))
```

Extension of + to fixed- and floating-point values:

```
(defun #:fix:+ (r c) (+ c r))   ; r+c = c+r

(defun #:float:+ (r c) (+ c r))
```

Binary subtraction:

```
(defun :- (c r) (:+ c (0- r)))  ;  c-r = c+(-r)
```

Extension of − to fixed-point and floating-point values:

```
(defun #:fix:- (r c) (+ r (0- c)))

(defun #:float:- (r c) (+ r (0- c)))
```

Unary negation:

```
(defun :0- (c) (makecomplex (- (:r c)) (- (:i c))))
```

Binary multiplication:

```
(defun :* (c r)
    (if (complexp r)
        (makecomplex (- (* (:r c) (:r r)) (* (:i c) (:i r)))
                     (+ (* (:r c) (:i r)) (* (:i c) (:r r))))
        (makecomplex (* r (:r c)) (* r (:i c)))))
```

Extension of * to fixed- and floating-point values:

```
(defun #:fix:* (r c) (* c r))   ; r*c = c*r

(defun #:float:* (r c) (* c r))
```

Binary division:

```
(defun :/ (a b) (:* a (1/ b)))
```

Extension of / to fixed- and floating-point values:

```
(defun #:fix:/ (r c) (+ r (1/ c)))

(defun #:float:/ (r c) (+ r (1/ c)))
```

Unary multiplicative inverse `1/`: 1/a+ib = a/m-ib/m with m=a*a+b*b:

```
(defun :1/ (c)
    (:* (makecomplex (:r c) (- (:i c))) (/ (:module2 c))))

(defun #:fix:1/ (r) (/ (float r)))
```

Compute the square of the modulus of c:

```
(defun :module2 (c)
    (+ (* (:i c) (:i c)) (* (:r c) (:r c))))
```

Compute the modulus of c:

```
(defun :abs (c) (if (zerop (:r c)) (abs (:i c)) (sqrt (:module2 c))))
```

10.3.4 exp, log and sqrt functions

`exp[ai+b] = exp(b)*[sina*i+cosa]`:

```
(defun :exp (c)
    (:*  (makecomplex (cos (:i c)) (sin (:i c))) (exp (:r c))))
```

The number pi:

```
(defvar pi (* 4 (atan 1)))
```

`log(r*e**i*theta) = log(r)+i*theta`

```
; The definition that follows is mathematically false,
; to simplify things!

(defun :log (c)
    (makecomplex (log (:abs c))
                 (if (= 0 (:r c))
                     (/ pi 2)
                     (atan (/ (:i c) (:r c))))))
```

To extend the `log` function to negative fixed-point values:

```
(defun #:fix:log (r)
    (selectq (<?> r 0)
      (1 (log (float r)))
      (0 (- (/ 0)))
      (-1 (makecomplex (log (- r)) pi))))
```

Extension to floats:

```
(defun #:float:log (r) (:log r))
```

(sqrt c) = c**1/2 = e**(log(c)/2)

```
(defun :sqrt (c) (:exp (/ (:log c) 2)))
```

To extend the `sqrt` function to negative fixed-point and floating-point values:

```
(defun #:fix:sqrt (r)
    (selectq (<?> r 0)
      (1 (sqrt (float r)))
      (0 0)
      (-1 (makecomplex 0 (sqrt (- r))))))

(defun #:float:sqrt (r) (#:fix:sqrt r))
```

Table of contents

10 Rational and complex arithmetic **10-1**

10.1 Rationals (file Q) 10-2

10.1.1 Rational number I/O 10-2

10.1.2 Tests for type 10-3

10.1.3 Generic rational arithmetic 10-3

10.1.4 Functions limited to Z 10-4

10.1.5 Functions limited to rational arguments (limited to Q) 10-4

10.1.6 Two examples 10-5

10.2 Complex numbers (the field C) 10-6

10.2.1 Complex number I/O 10-6

10.2.2 Tests for type 10-6

10.2.3 Complex generic arithmetic 10-7

10.2.4 Functions limited to the complex numbers (limited to C) 10-7

10.2.5 Polar coordinates 10-8

10.2.6 Hyperbolic functions 10-9

10.3 A complex mini-extension of generic arithmetic 10-10

10.3.1 Representation 10-10

10.3.2 I/O for C 10-10

10.3.3 Complex arithmetic 10-11

10.3.4 exp, log and sqrt functions 10-12

Function Index

`ratio` [*feature*] 10-1
`q` [*feature*] 10-1
`complex` [*feature*] 10-1
`(precision n)` [*function with an optional argument*] 10-2
`(rationalp q)` [*function with one argument*] 10-3
`(integerp z)` [*function with one argument*] 10-3
`(gcd` z_1 ... z_n`)` [*function with a variable number of arguments*] 10-4
`(pgcd z0 z1)` [*function with two arguments*] 10-4
`(even? z)` [*function with one argument*] 10-4
`(fact n)` [*function with one argument*] 10-4
`(fib n)` [*function with one argument*] 10-4
`(numerator f)` [*function with one argument*] 10-5
`(denominator f)` [*function with one argument*] 10-5
`(** n m)` [*function with two arguments*] 10-5
`[` [*macro character*] 10-6
`#C` [*#-macro*] 10-6
`(complexp c)` [*function with one argument*] 10-6
`(realp r)` [*function with one argument*] 10-6
`(makecomplex c i)` [*function with two arguments*] 10-7
`(realpart c)` [*function with one argument*] 10-8
`(imagpart c)` [*function with one argument*] 10-8
`(conjugate c)` [*function with one argument*] 10-8
`(phase c)` [*function with one argument*] 10-8
`(signum c)` [*function with one argument*] 10-8
`(cis r)` [*function with one argument*] 10-9
`(sinh z)` [*function with one argument*] 10-9

`(cosh z)` [*function with one argument*] 10-9
`(tanh z)` [*function with one argument*] 10-9
`(asinh z)` [*function with one argument*] 10-9
`(acosh z)` [*function with one argument*] 10-10
`(atanh z)` [*function with one argument*] 10-10

Chapter 11

Debugging tools

This chapter describes several utilities, written in LISP, that make it easier to debug your LISP programs. They are automatically loaded the first time they are called. They include tools supporting step-by-step execution of code, program tracing, and an inspection loop which allows the environment to be examined on the occurrence of errors. All these tools use the virtual multiple windows if they are available.

`debug` [*feature*]

This feature indicates whether the debugging tools are loaded into memory.

`(debugend)` [*function with no arguments*]

Recovers memory that was being used by the debugging tools. It also removes the `debug` feature from the environment. Each debugging function will be loaded anew the next time it is called.

11.1 Tracing

Recall that it is possible to trace all the internal calls to the `eval` function during the execution of a program by using the `traceval` function. (See the description of this function.) This is useful for understanding the *internal* behavior of the evaluator, but not very handy for debugging larger programs because it provides too much information for that purpose.

Therefore, it is also possible to trace calls to any function. A trace prints certain information the user has selected at the entry and the exit of a function. The printing can be controlled by a LISP expression which is evaluated when the evaluator enters and exits the function.

It is also possible to conditionally enter an inspection loop at the entry of a function. As a final possibility, you can redirect the output of the trace by assigning a channel to the `*trace-output*` global variable.

11.1.1 Stepping functions

`(trace` $\texttt{trace}_1$ ... $\texttt{trace}_n$`)` [*special form*]

Traces the functions described by arguments $\texttt{trace}_1 \ldots \texttt{trace}_n$. These arguments are not evaluated. They are either function names or lists which have function names in their `car`s. The former kind of argument activates a standard trace on the given function. The latter form lets the caller give trace parameters for the function involved, *forms* to be evaluated on function entry and exit, conditional trace points, and conditional breakpoints. These trace parameters are descibed in section 11.1.2. `trace` returns a list of all the functions affected.

`(untrace` $\texttt{sym}_1$ ... $\texttt{sym}_n$`)` [*special form*]

Disables tracing on functions named by the symbols $\texttt{sym}_1 \ldots \texttt{sym}_n$. These names are not evaluated. If no argument is given, that is, if the call was `(untrace)`, all current traces will be disabled. `untrace` returns a list of the functions affected.

11.1.2 Trace parameters

The standard trace on a function prints the following information:

- Upon entering the function, its name and the name(s) and value(s) of its parameter(s).
- Just before exiting the function, its name followed by the value it returns.

List arguments to `trace` must have the following form:

`(fn` $\texttt{par}_1 \ldots \texttt{par}_n$`)`

where `fn` is the name of the function to trace and $\texttt{par}_1 \ldots \texttt{par}_n$ are the parameters.

The parameters recognized by the `trace` function are:

- `(when e)` : conditional trace.
- `(break e)` : conditional breakpoint
- `(step e)` : conditional switch into stepwise execution mode.
- `(entry` $\texttt{e}_1 \ldots \texttt{e}_m$`)` : sets LISP forms to evaluate upon function entry.
- `(exit` $\texttt{e}_1 \ldots \texttt{e}_m$`)` : sets LISP forms to evaluate upon function exit.

The `when` parameter is a LISP form which is evaluated at function entry and exit. If the evaluation of this form results in `()` the other trace conditions `break`, `step`, `entry` and `exit` are ignored.

The `break` parameter is a LISP form which is evaluated at function entry. If the evaluation does not result in `()` an *inspection loop* is activated at the function entry point. (See the `break` function described in section 11.2.) Note that the `break` parameter is evaluated after the `when` parameter. The breakpoint is created only when both the `when` and `break` parameters are true.

The `step` parameter is a LISP form which is evaluated after the `break` parameter but before the body of the function being traced. If its value is different than `()`, the body of the function is evaluated in stepwise execution mode. (See the `step` function described in section 11.3.) Otherwise

the body of the function is evaluated normally. The `step` parameter is only evaluated when the `when` parameter is true.

All the operations which are activated *upon entry into a function* are done after variable bindings have been done. All the operations which happen *upon exit from a function* are done after the evaluation of the last form in the function, in the environment of the function. In addition, as the evaluator exits the function, the variable `#:trace:value` will contain the value that the function will return, and it can be used for examining or printing this value.

If some values are not furnished in a call to `trace`, they take on the values they would have in a standard trace. They are defined as follows for a two-argument function called `foo`:

```
(trace (foo (when t)      ; non-conditional trace - always active
            (break ())    ; no breakpoints created
            (step ())     ; execute normally, not in step mode
            (entry (print "foo --> x=" x ", y=" y))
            (exit  (print "foo <-- " #:trace:value))))
```

11.1.3 Trace global variables

#:trace:arg1 [*variable*]

#:trace:arg2 [*variable*]

#:trace:arg3 [*variable*]

The values of these variables are only meaningful during the evaluation of `when`, `break`, `entry`, and `exit` parameters of a compiled traced function, which can be a predefined compiled function or a function compiled by the user.

Their values depend upon the type of the traced function, as follows:

- For `subr0` functions, none of these values are meaningful.
- For `subr1` functions, `#:trace:arg1` contains the value of the function's first argument.
- For `subr2` functions, `#:trace:arg1` and `#:trace:arg2` contain the values of the function's first and second arguments, respectively.
- For `subr3` functions, `#:trace:arg1`, `#:trace:arg2`, and `#:trace:arg3` contain the values of the function's first, second, and third arguments, respectively.
- For `subrn` functions, `#:trace:arg1` contains the list of values of all the function's arguments.
- For `fsubr`, `msubr` and `dmsubr` functions, `#:trace:arg1` contains the list of the function's non-evaluated arguments.

The use of these variables in trace parameters can result in a very fine-grained trace of compiled functions.

Here are some examples of their use:

- Breakpoint on `car` when the argument's value is a number:
  `(trace (car (when (numberp #:trace:arg1)) (break t)))`
- Trace `putprop` when the value of the third argument is `()`:
  `(trace (putprop (when (null #:trace:arg3))))`
- Trace calls to `list` with seven arguments:
  `(trace (list (when (= (length #:trace:arg1) 7))))`
- Trace all the `setq`s on the variable `foo`:
  `(trace (setq (when (eq (car #:trace:arg1) 'foo))))`

Perverse effects can result from using the trace functions. The following call reverses the argument order of all calls to the `cons` function.

```
(trace (cons
         (entry (psetq #:trace:arg1 #:trace:arg2
                       #:trace:arg2 #:trace:arg1))))
```

#:trace:value [*variable*]

This variable contains the last value returned by a traced function. This value is only meaningful inside forms associated with trace `exit` parameters.

#:trace:not-in-trace-flag [*variable*]

This variable has the value `()` during the evaluation of trace parameters, and `t` at other times. It is used by the tracer as a supplementary condition on function tracing. Functions are not actually traced unless this variable has the value `()`. This enables traced functions to be used in the evaluation of trace parameters without sending the evaluator into an infinite loop. For example the `print` function can be traced even though it is being used to print trace messages.

Here is an example of a trace of the `print` function. Follow it carefully—we are also tracing `toplevel`'s calls to `print`, `(print (eval (read)))` :

```
? (trace print)
= print ---> ((print))
(print)
print <--- (print)
? (print 1 2 3)
print ---> (1 2 3)
123
print <--- 3
= print ---> (3)
3
print <--- 3
```

```
? (untrace)
= (print)
```

#:trace:trace [*variable*]

This variable contains the list of all the functions being traced at a given moment.

11.1.4 Example of trace use

We reproduce here a LE-LISP session illustrating the use of the tracer:

Switch into debug mode.

```
? (debug t)
= t
```

```
? (defun rv (s res)
?  (if (null s)
?  res
?  (rv (cdr s) (cons (car s) res))))
= rv
```

A normal try, without tracing.

```
? (rv '(1 2 3) ())
= (3 2 1)
```

Standard tracing.

```
? (trace rv)
= (rv)
```

```
? (rv '(1 2 3) ())
rv ---> s=(1 2 3) res=()
rv ---> s=(2 3) res=(1)
rv ---> s=(3) res=(2 1)
rv ---> s=() res=(3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
= (3 2 1)
```

Pick some information to be printed upon function entry and exit.

```
? (trace (rv (entry (print "rv: res=" res))
?  (exit (print "rv returns:" #:trace:value))))
= ((rv (entry (print rv: res= res)) (exit (print rv returns: #:trace:value))))
```

```
? (rv '(1 2 3) ())
rv: res=()
rv: res=(1)
rv: res=(2 1)
rv: res=(3 2 1)
rv returns:(3 2 1)
rv returns:(3 2 1)
rv returns:(3 2 1)
rv returns:(3 2 1)
```

```
  = (3 2 1)
```

Trace conditionally.

```
? (trace (rv (when (< (length s) 2))))
  = ((rv (when (< (length s) 2))))   ? (rv '(1 2 3) ())
  rv ---> s=(3) res=(2 1)
  rv ---> s=() res=(3 2 1)
  rv <--- (3 2 1)
  rv <--- (3 2 1)
  = (3 2 1)
```

Invoke an inspection loop on a condition.

```
? (trace (rv (break (null s))))
  = ((rv (break (null s))))   ? (rv '(1 2 3) ())
  rv ---> s=(1 2 3) res=()
  rv ---> s=(2 3) res=(1)
  rv ---> s=(3) res=(2 1)
  rv ---> s=() res=(3 2 1)
  ** rv : break : tracebreak
  (defun rv (s res)
   (if (null s) res (rv (cdr s) (cons (car s) res))))


>? v
 s=()
 res=(3 2 1)
>? r
rv <--- (3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
= (3 2 1)
```

Use stepwise execution mode.

```
? (trace (rv (step (= 1 (length s)))))
  = ((rv (step (= 1 (length s)))))   ? (rv '(1 2 3) ())
  rv ---> s=(1 2 3) res=()
  rv ---> s=(2 3) res=(1)
  rv ---> s=(3) res=(2 1)
   1 -> (if (null s) res (rv (cdr s) (cons & res))) step>? ; return
   2 -> (null s) step>? ; return
   3 -> s step>? ; return
   3 <- (3)
   2 <- ()
   2 -> (rv (cdr s) (cons (car s) res)) step>? ; return
   3 -> (cdr s) step>? ; return
   4 -> s step>? ; return
   4 <- (3)
   3 <- ()
   3 -> (cons (car s) res) step>? <
   3 <- (3 2 1)
```

```
rv ---> s=() res=(3 2 1)
 4 -> (if (null s) res (rv (cdr s) (cons & res))) step>? <
 4 <- (3 2 1)
rv <--- (3 2 1)
 2 <- (3 2 1)
 1 <- (3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
rv <--- (3 2 1)
= (3 2 1)
```

Suppress tracing. `? (untrace rv)`

```
= (rv)
```

Redirect the trace output. `? (de foo (x) (car '(1 2 3)))`

```
= foo   ? (trace foo)
= (foo)   ? (let ((*trace-output* (openo "see")))
?  (protect (foo '(1 2 3))
?  (close *trace-output*)))
= 1   ? !cat see
foo ---> x=(1 2 3)
foo <--- 1
= t   ? (foo '(1 2 3))
foo ---> x=(1 2 3)
foo <--- 1
= 1
```

11.2 Break and debug mode

When an error is raised, a message is printed and the environment is restored, which results in the reinitialization of all the dynamic objects. Sometimes, however, it is preferable to stay in the environment which raised the error to examine the variables or the stack as they were *when the error was raised.* This is accomplished by switching to debug mode.

`(debug s)` [*special form*]

If the argument `s`, which is not evaluated, is equal to `t`, debug mode is globally activated. If the argument is `()`, debug mode is globally deactivated. If the argument `s` is an expression, the debug mode will be activated only during the evaluation of this expression. `debug` returns the value of the evaluation of `s` as its value. This incidentally allows any Lisp expression to be submitted to the `debug` statement as an argument for debugging.

`(break)` [*function with no arguments*]

Outside debug mode, this function just causes a return to the LISP `toplevel` by calling the `err` function. (See the explanation of the `err` function.) In debug mode it activates an *inspection loop* in its dynamic call environment. The inspection loop is recognizable by its `>?` prompt. One can

return to the LISP `toplevel` from the inspection loop by evaluating (`exit break`) or using the shorthand `t`. `break` is systematically called by the standard error-handling function `syserror`. In debug mode, therefore, all user errors send the evaluator into an inspection loop.

If a new error is raised in the inspection loop, `break` will only be called recursively if the form (`debug t`) was evaluated from within the first inspection loop. In this case, the prompt will indicate the number of active embedded calls to `break`.

11.2.1 Inspection (or debug) loop

An inspection (or debug) loop is a `read-eval-print` loop which runs in a function's local environment. It is created in debug mode by a call to the `break` function; that is, when an error, a breakpoint (see the `trace` function), or an explicit user call to the function occurs. An inspection loop can be recognized by its prompt ">?".

An inspection loop can be terminated by returning to the Lisp `toplevel` or by continuing the interrupted computation.

When a debug loop starts executing, the name of the function being called it is printed on the screen and, if possible, the section of the function currently being evaluated is highlighted. (This operation uses the `tyattrib` function.)

In a debug loop any LISP function whatsoever can be evaluated, just like at the system `toplevel`. The expression evaluation takes place, however, in the local environment of the function which called the loop. It is therefore possible to examine the values of the function's parameters. In addition, a number of *commands* let the environments of calls waiting on the stack be pushed or popped, so parameters and local variables of all the waiting functions can be examined.

The stack inspection commands take the form of characters to be typed at the beginning of a line. If a LISP form which begins with a command character is to be evaluated, it must be preceded by the *space* character.

The debug, or inspection, loop commands are:

- **v**: Print the current function's local variables.
- **e**: Print the full error message of the error that raised the debug loop.
- **.** (dot or period): Print the current function.
- **+**: Pop a function call off the stack.
- **–**: Push the current function (back) onto the stack.
- **h**(istory): Print the first few function calls beginning with the current one. The number of levels printed is the value of the `#:system:stack-depth` global variable.
- **H**: Print all suspended function calls.
- **q**(uit): Leave the debug loop.
- **t**(op): Return immmediately to the LISP top level.

- **r**: Continue the program which caused the debug loop to be activated. This function provides a way to continue evaluation after a breakpoint and to correct the *undefined variable (*`errudv`*)* and *undefined function (*`errudf`*)* errors.
- **z**: Continue a function stopped at a breakpoint in stepwise execution mode.
- **?**: Print this list of debug loop commands.

Those commands which terminate the debug loop also restore the stack context by pushing function contexts which may have been popped before continuing execution.

Switch to global debug mode.
```
? (debug t)
= t
```
Define an incorrect function.
```
? (defun foo (x)
?  (if (= x 0)
?  (lisp 1)
?  (cons (1+ x) x (foo (1- x)))))
= foo
```
Try the function.
```
? (foo 3)
** cons : wrong number of arguments : 2
(ds cons subr2)
 -----
(defun foo (x)
 (if (= x 0) (lisp 1) (cons (1+ x) x (foo (1- x)))))
 ----------------------------
```
(The function was printed with the faulty part (cons...) underlined. `cons` *requires two arguments! Return to the top level.)*

```
>? t
```

Correct the function.
```
? (defun foo (x)
?  (if (= x 0)
?  (lisp 1)
?  (mcons (1+ x) x (foo (1- x)))))
** de : function redefined : foo
= foo
? (foo 3)
** eval : undefined function : lisp
```

```
(defun foo (x)
    (if (= x 0) (lisp 1)  (mcons (1+ x) x (foo (1- x)))))
                --------
>? ; The part that caused the error (lisp 1) is highlighted.
>? ; Now examine the local variables.
>? v
 x=0
>? ; Evaluate some lisp forms...
>? x
= 0
>? (mcons (1+ x) x '(1))
= (1 0 1)
```

```
>? ; Print a history trace of the function calls on the stack.
>? h
   [stack   4]   (foo ...)
   [stack   3]   (foo ...)
   [stack   2]   (foo ...)
   [stack   1]   (foo ...)
>? ; List of the available debug comands...
>? ?
; v:   show variables
; h:   print top of stack
; h:   print complete stack
; e:   show error message
; .:   show current stack frame
; +:   down stack
; -:   up stack
; t:   back to toplevel
; q:   exit inspection loop
; r:   resume
; z:   step traced functions
; ?:   list commands

>? ; Pop a function call...
>? +
(defun foo (x)
    (if (= x 0) (lisp 1) (mcons (1+ x) x (foo (1- x)))))
>? ; x equals 1 in this environment.
>? v
 x=1
>? ; Push the call back on the stack.
>? -
(defun foo (x)
    (if (= x 0) (lisp 1) (mcons (1+ x) x (foo (1- x)))))
                --------
>? ; Continue the computation by correcting the "undefined function".
>? r
function >? list
= (4 3 3 2 2 1 1)
```

The function `foo` *must be corrected.* `? (defun foo (x)`

```
  ?  (if (= x 0)
  ?  (list 1)
  ?  (mcons (1+ x) x (foo (1- x)))))
  ** de : function redefined : foo
  = foo   ? (foo 3)
```

`= (4 3 3 2 2 1 1)` *Set a conditional break point.* `? (trace (foo (break (= x 0))))`

```
  = ((foo (break (= x 0))))   ? (foo 5)
```

```
  foo ---> x=5
  foo ---> x=4
  foo ---> x=3
  foo ---> x=2
  foo ---> x=1
  foo ---> x=0
  ** foo : break : tracebreak
  (defun foo (x)
   (if (= x 0) (list 1) (mcons (1+ x) x (foo (1- x)) )))
   ------------


>? v
 x=0
>? ; Let's modify foo's parameter.
>? (setq x 4)
= 4
>? ; Continue the computation...
>? r
foo ---> x=3
foo ---> x=2
foo ---> x=1
foo ---> x=0
** foo : break : tracebreak
(defun foo (x)
    (if (= x 0) (list 1) (mcons (1+ x) x (foo (1- x)) )))
                                          ------------
>? ; We are at the breakpoint again...
>? ; finish the computation.
>? r
foo <--- (1)
foo <--- (2 1 1)
foo <--- (3 2 2 1 1)
foo <--- (4 3 3 2 2 1 1)
foo <--- (5 4 4 3 3 2 2 1 1)
foo <--- (2 1 5 4 4 3 3 2 2 1 1)
foo <--- (3 2 2 1 5 4 4 3 3 2 2 1 1)
foo <--- (4 3 3 2 2 1 5 4 4 3 3 2 2 1 1)
foo <--- (5 4 4 3 3 2 2 1 5 4 4 3 3 2 2 1 1)
foo <--- (6 5 5 4 4 3 3 2 2 1 5 4 4 3 3 2 2 1 1)
= (6 5 5 4 4 3 3 2 2 1 5 4 4 3 3 2 2 1 1)
```

11.2.2 Debug mode functions

The functions described in this section may be useful to users who want to extend the debug mode functionality.

(printstack n s) [*function with one or two optional arguments*]

Prints a visual representation of the last `n` levels of the Lisp dynamic execution stack or the whole stack if `n` is not furnished. If the `s` argument is given, it must be a list returned by the `cstack` function. Such a list represents a LISP stack state, and it is the contents of this list, and not the current state of the stack, that would be printed if `s` were present in the call.

(debug-command cn) [*function with one argument*]

The `cn` argument is an internal character code. This function executes the debug loop command associated with this character.

#:system:stack-depth [*variable*]

The value of this variable is a number that indicates the number of stack levels that the **h** debug loop command will print. Its initial value is 5. This variable is ignored by the **H** command, which prints the entire stack contents.

#:system:debug-line [*variable*]

This variable provides a way to limit the number of lines printed by the debug loop information messages. Its value is the maximum number of lines to be printed, and its initial value is 5. This variable is particularly useful for limiting the printing of functions which have caused errors.

11.3 Stepwise execution

Single-step mode lets the user stop at each internal evaluator call. During these halts, dynamic objects can be examined and the dynamic behavior of a program can thus be followed very closely.

As in the case of tracing, you can redirect the display of stepwise execution by means of the `*trace-output*` variable. It is also possible to read the stepwise commands by assigning a channel, opened as input, to the `*trace-input*` global variable.

Warning: The current implementation of this feature does not allow the evaluator to be stopped during the evaluation of `&nobind` functions.

In order to implement this incremental execution feature, the system invokes the `stepeval` programmable interrupt, with the form to evaluate as its argument, at each internal call to the evaluator.

Here are the various actions possible at each pause:

- **return**: Continue the stepwise evaluation one step at a time.
- <: Evaluate the current expression without stopping at each step and then continue in single-step mode after the evaluation is done.
- **q**: Quit single-step mode and return the value of the expression that launched it.

- **=**: Activate a debug (inspection) loop, in which all the debug loop commands are available. Return to single-step mode by typing the **I** command in the debug loop.
- **h**: Print a history of the suspended evaluations.
- **?**: Print an abbreviated command list.

(step s) [*special form*]

Evaluate the expression `s` in single-step mode. `step` returns the value of the evaluation of `s`.

(unstep s_1 ... s_n) [*special form*]

Suspend single-step mode during the evaluation of the expressions s_1 ... s_n and return the value of s_n.

Change globally into debug mode.

```
? (debug t)
= t
? (defun f (n)
?  (if (<= n 1)
?  1
?  (* (f (1- n)) n)))
= f

? (step (f 5))
 1 -> 5 step> 1 <- 5
 1 -> (if (<= n 1) 1 (* (f \&) n)) step>?  ; return
  2 -> (<= n 1) step>?  ; return
   3 -> n step>?  ; return
   3 <- 5
   3 -> 1 step>?  ; return
   3 <- 1
  2 <- ()
  2 -> (* (f (1- n)) n) step>?  ; return
   3 -> (f (1- n)) step>?  ; return
    4 -> (1- n) step>?  ; return
     5 -> n step>?  ; return
     5 <- 5
    4 <- 4
    4 -> (if (<= n 1) 1 (* (f \&) n)) step>?  ; return
     5 -> (<= n 1) step>?  ; return
      6 -> n step>?  ; return
      6 <- 4
      6 -> 1 step>?  ; return
      6 <- 1
     5 <- ()
     5 -> (* (f (1- n)) n) step>? ?

(The single-step mode commands are:)
```

```
;   CR  go to next expression
;   .   see current expression
;   <   evaluate without step and come back
;   q   return to toplevel
;   h   view history
;   ?   this message ...
     5 -> (* (f (1- n)) n) step>? h
1 (if (<= n 1) 1 (* (f (1- n)) n))
 2 (* (f (1- n)) n)
  3 (f (1- n))
   4 (if (<= n 1) 1 (* (f (1- n)) n))
    5 (* (f (1- n)) n)
     5 -> (* (f (1- n)) n) step>? =
** step : break : (* (f (1- n)) n)
(defun f (n)
    (if (<= n 1) 1 (* (f (1- n)) n) ))
                  ----------------
>? v
 n=4
>? h
   [stack   2]   (f ...)
   [stack   1]   (f ...)
>? r
     5 -> (* (f (1- n)) n) step>?  ; return
      6 -> (f (1- n)) step>? <
      6 <- 6
      6 -> n step>?  ; return
      6 <- 4
     5 <- 24
    4 <- 24
   3 <- 24
   3 -> n step>?  ; return
   3 <- 5
  2 <- 120
 1 <- 120
= 120
```

Table of contents

11 Debugging tools **11-1**

11.1 Tracing 11-1

11.1.1 Stepping functions 11-1

11.1.2 Trace parameters 11-2

11.1.3 Trace global variables 11-3

11.1.4 Example of trace use 11-5

11.2 Break and debug mode 11-7

11.2.1 Inspection (or debug) loop 11-8

11.2.2 Debug mode functions 11-11

11.3 Stepwise execution 11-12

Function Index

`debug` [*feature*] 11-1

`(debugend)` [*function with no arguments*] 11-1

`(trace` $\texttt{trace}_1$ ... $\texttt{trace}_n$`)` [*special form*] 11-2

`(untrace` $\texttt{sym}_1$ ... $\texttt{sym}_n$`)` [*special form*] 11-2

`#:trace:arg1` [*variable*] 11-3

`#:trace:arg2` [*variable*] 11-3

`#:trace:arg3` [*variable*] 11-3

`#:trace:value` [*variable*] 11-4

`#:trace:not-in-trace-flag` [*variable*] 11-4

`#:trace:trace` [*variable*] 11-5

`(debug s)` [*special form*] 11-7

`(break)` [*function with no arguments*] 11-7

`(printstack n s)` [*function with one or two optional arguments*] 11-12

`(debug-command cn)` [*function with one argument*] 11-12

`#:system:stack-depth` [*variable*] 11-12

`#:system:debug-line` [*variable*] 11-12

`(step s)` [*special form*] 11-13

`(unstep` $\texttt{s}_1$ ... $\texttt{s}_n$`)` [*special form*] 11-13

Chapter 12

Loader/assembler LLM3

The heart of the LE-LISP system is written in the LLM3 virtual machine language [Chailloux 85b]. This language is available in LISP. It is used by the various LE-LISP compilers but can also be used to write new standard functions, or to write compilers for languages other than LISP. This chapter describes the functions that provide access to internal machine resources and the syntax used by the LLM3 loader/assembler, often called LAP, for *Lisp Assembly Program*.

12.1 Access to memory and the CPU

Functions provided in this section allow you to directly access memory and the CPU. The LLM3 loader/assembler uses a special memory zone, called the *code zone*, to load and store instructions. The following functions are obviously machine-dependent, and are used to build the LLM3 loader/assembler.

These functions are designed to operate on all possible memory addresses, for all machine architectures. Since LE-LISP integers are 16 bits long, they cannot always represent memory addresses. A memory address—referred to as `addr` throughout this chapter—is therefore represented in LISP in one of two possible ways:

- By a 16-bit number with an extended sign, to represent a complete 32-bit address.
- By a `cons` of two numbers of the form `(high . low)`. The `car`, `high`, contains the 16 high-order bits of the address. The `cdr`, `low`, contains the 16 low-order bits.

The address	`#$001f3c00`	is stored as	`(#$1f . #$3c00)`
" "	`#$00008000`	" "	`(0 . #$8000)`
" "	`#$ffff0000`	" "	`(#$ffff . 0)`
" "	`#$0000ffff`	" "	`(0 . #$ffff)`
" "	`#$ffff8100`	" "	`#$8100`
" "	`#$00001000`	" "	`#$1000`

The functions that follow test the validity of memory address-type arguments. They can raise the `errdna` error, which has the following default screen display:

```
** <fn> : bad address : <s>
```

Here, `fn` is the name of the function called, and `s` is the argument that is not of type memory address.

(#:system:ccode addr)

[function with an optional argument]

This function reads—or writes, if the `addr` argument is given—the current memory load address in the code zone.

(#:system:ecode)

[function with no arguments]

Reads the last memory address in the code zone.

(addadr addr1 addr2)

[function with two arguments]

Returns the sum of the two memory addresses `addr1` and `addr2`. It allocates a new `cons` if its result must be represented in the `cons` form.

```
(addadr '(3 . 657) '(4 . 4567))       ⟹   (7 . 5224)
(addadr '(1 . #$8000) '(1 . #$8000))  ⟹   (3 . 0)
(addadr -1 -1)                        ⟹   -2
(addadr -1 1)                         ⟹   0
(addadr #$7fff 1)                     ⟹   (0 . #$8000)
(addadr 1 '(1 . 234))                 ⟹   (1 . 235)
(addadr 1 '(1 . #$ffff))              ⟹   (2 . 0)
(addadr '(1 . #$ffff) 1)              ⟹   (2 . 0)
(addadr '(#$ffff . #$8001) 1)         ⟹   -32766
```

(subadr addr1 addr2)

[function with two arguments]

Returns the difference of the two addresses `addr1` and `addr2`. It allocates a new `cons` if its result must be represented in the `cons` form.

```
(subadr '(56 . 7899) '(45 . 3333))    ⟹   (11 . 4566)
(subadr '(1 . #$8000) '(1 . #$8001))  ⟹   -1
(subadr 0 '(3 . 3))                   ⟹   (-4 . -3)
(subadr 1 '(1 . 0))                   ⟹   (-1 . 1)
(subadr 1 '(1 . #$ffff))              ⟹   (-2 . 2)
```

(incradr addr n)

[function with two arguments]

Adds `n` to the memory address `addr`. If this address is in the form of a `cons`, `incradr` physically modifies the `cons` cell.

```
(setq addr '(3 . #$fffe))  ⟹   (3 . -2)
(incradr addr 1)           ⟹   (3 . -1)
```

```
(incradr addr 1)          ⟹   (4 . 0)
(incradr addr 1)          ⟹   (4 . 1)
(incradr addr 100)        ⟹   (4 . 101)
addr                      ⟹   (4 . 101)
```

(gtadr addr1 addr2) [*function with two arguments*]

Returns `t` if the memory address `addr1` is greater than the address `addr2` and `()` otherwise.

```
(setq addr '(4 . 101))    ⟹   (4 . 101)
(gtadr addr '(4 . -10))   ⟹   ()
(gtadr addr '(4 . 100))   ⟹   t
(gtadr addr '(4 . 101))   ⟹   ()
```

(loc s) [*function with one argument*]

Provides a way to get the absolute memory address of the LISP object `s`. The memory address of a LISP object is the pointer to it. This function provides a means of locating in memory objects that are created dynamically. `loc` cannot be described in LISP.

```
(loc '(a b c))   ⟹   (15 . 288)
(loc '(a b c))   ⟹   (13 . 1200)
```

(vag addr) [*function with one argument*]

This is the inverse of the `loc` function. `vag` returns the LISP object whose absolute memory address is supplied as the argument. So, `(vag (loc s))` is equivalent to `s` itself. The `vag` function cannot be described in LISP.

> *Warning*: `vag` does not test whether or not `addr` is the address of a real LISP object, and an erroneous call to this function could raise an error on the host system. In particular it is possible to refer to objects that have disappeared:

```
(let ((x (loc (cons 'a 'b))))
     (gc)
     (vag x))       ; the value of x no longer exists!
                    ; vag returns a pointer to the
                    ; free list of conses!
```

(memory addr n) [*function with one or two arguments*]

`memory` samples, or if the second numeric argument `n` is provided changes, the word of memory whose address `addr` is given as its first argument. This function does no perform validity checking and must be used with the utmost care. It returns a number representing the value of the memory word (after modification, if the two-argument form was used). This function is limited by the nature of the host CPU: 8-bit or 16-bit words, word alignment considerations, etc.

(call addr a1 a2 a3) [*function with four arguments*]

The first argument `addr` must be the memory address of a memory-resident subroutine. `call` will call this subroutine after having loaded the three values `a1`, `a2` and `a3` into the LLM3 accumulators A1, A2, and A3, respectively. This call format is used by `subr0`s, `subr1`s, `subr2`s, `subr3`s, `fsubr`s, `msubr`s and `dmsubr`s. (See the section on function calls 12.7.) This function must be used with care, since it performs no tests for the validity of the executed code.

`call` returns as its value the contents of the A1 accumulator after the code beginning at `addr` has been executed.

Warning: The code executed by means of a `call` statement must end with an LLM3 `return` instruction, and load the A1 register with a LISP value to be returned by the `call` function.

(calln addr l) [*function with two arguments*]

This function is similar to `call`, but pushes the list of arguments contained in `l` onto the stack before calling the subroutine stored at memory address `addr`. The number of arguments pushed will be available in the A4 accumulator. This is the form that `nsubr`s take. (See the section on function calls 12.7.)

Warning: The code invoked by means of `calln` must remove its arguments from the stack (using the number of arguments value stored in A4 and, for example, the LLM3 `adjstk` instruction), terminate with the LLM3 `return` instruction, and load the A1 register with a LISP value to be returned by the `calln` function.

12.2 LLM3 memory loader

loader [*feature*]

This feature indicates whether the LLM3 loader/assembler is currently in memory.

(loader l i) [*function with two arguments*]

The `loader` function loads the list of LLM3 instructions `l` into memory. The format of these instructions is described in the next section. If the indicator `i` is true, the loader will print, in the host machine language, the assembly listing of these instructions on the current output stream. `loader` returns `()` as its value.

#:ld:special-case-loader [*variable*]

The pseudo-instruction `fentry` changes the functional value of functions during loading. In certain difficult boot-strapping cases (loading the loader itself, for example), functional values must be resolved at the end of the load (when the pseudo-instruction `end` is encountered). The `#:ld:special-case-loader` variable is `t` in these special cases, but `()` by default.

#:ld:shared-strings [*variable*]

This variable indicates whether character string constants should be shared during the loading of LLM3 instructions.

12.3 LLM3 instruction format

A list of LLM3 instructions contains atoms representing the labels and lists representing the instructions. An instruction is itself a list of the form:

(codop $op_1 \ldots op_n$)

where codop is the instruction mnemonic, and $op_1 \ldots op_n$ are the operands. Only the codop field is obligatory.

12.4 Modules and labels

A module is a set of lists of instructions beginning with the title pseudo-instruction and ending with the end pseudo-instruction.

There are three types of labels:

1. Labels local to a list of instructions.
2. Labels local to a module.
3. Global labels.

Labels local to a list of instructions are represented by integers or by symbols declared using the local pseudo-instruction. A symbolic label which is local to a list of instructions must be declared before it is used. These labels must be resolved before the end of the list of instructions, or when the endl pseudo-instruction is encountered. All references to this kind of label generate jumps relative to the program counter.

Labels local to a module are always symbolic. They are defined using the entry pseudo-instruction. They can be undefined at incremental calls to the loader but must be resolved when the end pseudo-instruction is encountered. All references to this kind of label generate jumps relative to the program counter.

Global labels are also always symbolic. They are defined using the fentry pseudo-instruction. All references to this type of label generate an access to the functional value of the symbol. All the standard functions are of this type.

12.5 LLM3 instruction operands

LLM3 has four types of operands:

1. Registers (accumulators).
2. LISP object immediate values.
3. LISP object fields.
4. Labels.

`a1` [*LLM3 operand*]

`a2` [*LLM3 operand*]

`a3` [*LLM3 operand*]

`a4` [*LLM3 operand*]

These four operands represent the registers (accumulators) of the LLM3 machine.

`nil` [*LLM3 operand*]

This operand represents the special null symbol `||`, which is used as the end-of-list marker and as the boolean *false* value.

`quote exp` [*LLM3 operand*]

This operand represents a LE-LISP constant of any type.

`car accu` [*LLM3 operand*]

`cdr accu` [*LLM3 operand*]

These operands represent access to the constituents of a list element.

`cval accu` [*LLM3 operand*]

`plist accu` [*LLM3 operand*]

`fval accu` [*LLM3 operand*]

pkgc `accu` [*LLM3 operand*]

oval `accu` [*LLM3 operand*]

alink `accu` [*LLM3 operand*]

pname `accu` [*LLM3 operand*]

These operands represent access to the various constituents of a symbol.

cvalq `symb` [*LLM3 operand*]

This operand gives direct access to the value of the symbol named `symb`, considered as a variable: that is, its `cval`.

fvalq `symb` [*LLM3 operand*]

This operand gives direct access to the value of the symbol named `symb`, considered as a function: that is, its `fval`.

typ `accu` [*LLM3 operand*]

val `accu` [*LLM3 operand*]

These operands represent access to constituents of objects of type character string or vector of S-expressions.

& `n` [*LLM3 operand*]

This operand represents the $\texttt{n}^{th}$ element on the stack. The top element of the stack is represented by `(& 0)`, the next-to-top element by `(& 1)`, and so on.

lab [*LLM3 operand*]

This operand refers to a label which is local to a list of instructions or to a module.

@ `lab` [*LLM3 operand*]

This operand represents the address of an LLM3 label. This label is either local to a list of instructions or to a module.

eval `expr` [*LLM3 operand*]

This operand dynamically calls the LE-LISP evaluator on the expression `expr`. The value returned by the evaluation of an `eval` operand must be a legal LLM3 operand.

12.6 Pseudo-instructions

`title` `symb` [*LLM3 pseudo-instruction*]

Declares the name of a module. Inside a module, `title` appears as the first instruction.

`local` `symb` [*LLM3 pseudo-instruction*]

`local` provides a way to declare a label local to a list of instructions. By default, numeric labels are always local, and symbolic labels are always global.

`fentry` `symb ftype` [*LLM3 pseudo-instruction*]

The `symb` argument is the name of the LISP symbol that will have a global function (of type `subr`) written in LLM3 as its function value. `ftype` must be one of the following LE-LISP functional types: `subr0`, `subr1`, `subr2`, `subr3`, `nsubr`, `fsubr`, `msubr` or `dmsubr`.

`entry` `symb ftype` [*LLM3 pseudo-instruction*]

The `symb` argument is the name of the LISP symbol that will have a local function (of type `subr`) written in LLM3 as its function value. `ftype` is one of the following LE-LISP functional types: `subr0`, `subr1`, `subr2`, `subr3`, `nsubr`, `fsubr`, `msubr` ou `dmsubr`. After loading, this function is no longer accessible.

`endl` [*LLM3 pseudo-instruction*]

Terminates the loading of a local function. All local labels must be resolved at this point and are then discarded.

`end` [*LLM3 pseudo-instruction*]

Terminates the loading of a module.

`eval` `e` [*LLM3 pseudo-instruction*]

Evaluates the expression `e` within the context of the loader. It loads nothing into memory unless explicitly instructed to do so.

12.7 Basic instructions

12.7.1 Moving pointers

Along with the `mov` instruction, which provides a means to move any pointer to a LISP object, there are other specialized instructions for moving pointers or bytes in the heap storage or on the stack.

`mov op1 op2` *[LLM3 instruction]*

Moves the pointer contained in the source operand `op1` to the destination operand `op2`. The destination operand cannot be an immediate value.

12.7.2 Pointer comparisons

`cabeq op1 op2 lab` *[LLM3 instruction]*

Performs a pointer comparison. If the operand `op1` is equal to the operand `op2`, a branch will occur to the local label `lab`.

`cabne op1 op2 lab` *[LLM3 instruction]*

Like `cabeq`, `cabne` does a pointer comparison. If the operand `op1` is not equal to the operand `op2`, a branch will occur to the local label `lab`.

12.7.3 Control

`bra lab` *[LLM3 instruction]*

Branches to the local label `lab`, which must be within the LLM3 module. The branch is done relative to the program counter.

`jmp symb` *[LLM3 instruction]*

Jumps to the address of the functional value associated with the symbol `symb`. The branch is an indirection through the `fval` of the symbol `symb`. Jumps to the code of standard functions are `jmp`s and not `bra`s.

`bri op` *[LLM3 instruction]*

Branches indirectly to the address contained in the operand `op`.

`brx llab op` *[LLM3 instruction]*

Performs an indexed branch through the list of labels `llab`. `op` is the zero-relative index used. The list of labels is a list of operands of label-reference type: that is, `(@ lab)`.

sobgez `op lab` [*LLM3 instruction*]

Decrements the operand `op`. If this operand is greater than or equal to zero, a branch is done to the label `lab`.

nop [*LLM3 instruction*]

Takes up a little space, wastes a little time, and finally does nothing visible.

12.8 Stack

LLM3 has a single stack for both control and data. It is used as an implicit operand in some special stack-oriented instructions. These instructions—which use the stack pointer register, referred to as `SP`—do not require the stack to grow in a specific direction. Also, they do not require the stack to have implicit boundary-overflow tests built in. The stack normally occupies the space of 6K objects.

12.8.1 Management of the stack pointer

Two instructions provide for the explicit manipulation of the stack pointer `SP`.

stack `op` [*LLM3 instruction*]

Moves the current contents of the stack pointer into the operand `op`.

sstack `op` [*LLM3 instruction*]

Moves the operand `op` into the stack pointer.

12.8.2 As a control stack

Three instructions provide for the manipulation of return addresses stored on the stack.

call `lab` [*LLM3 instruction*]

Pushes the current program counter onto the top of the stack and branches to the local label `lab`, which must be in the current LLM3 module. The branch is direct and relative.

calli `op` [*LLM3 instruction*]

Pushes the current program counter onto the top of the stack and branches to the address contained in the operand `op`.

jcall `symb` [*LLM3 instruction*]

This function is similar to `calli`, but the destination of the branch can be in another LLM3 module. The branch is an indirection through the `fval` of the symbol `symb`. All the standard function calls must use `jcall` and not `call`.

`return` [*LLM3 instruction*]

Pops the top address off the stack, and it becomes the new value of the program counter.

12.8.3 As a data stack

`push op` [*LLM3 instruction*]

Pushes the operand `op` onto the stack. The stack pointer is modified as required.

`pop op` [*LLM3 instruction*]

Pops the value on the top of the stack into the operand `op`. The stack pointer is modified as required.

`adjstk op` [*LLM3 instruction*]

Adjusts the stack pointer so that the `op` last objects on the stack are popped off. If `op` is negative, then -`op` places are created at the top of the stack.

For the next two instructions, assume that the index for the top of the stack is zero, that the index for the second element is one, that the index for the third element is two, and so on.

`movxsp op1 op2` [*LLM3 instruction*]

Moves the operand `op1` into the `op2`th position on the stack.

`xspmov op1 op2` [*LLM3 instruction*]

Moves the object contained in the `op1`th stack position into the operand `op2`.

12.9 List cell `cons` operations

12.9.1 Test for list cell type

`btcons op lab` [*LLM3 instruction*]

If the operand `op` has type list cell, then `btcons` branches to the address `lab`.

`bfcons op lab` [*LLM3 instruction*]

If the operand `op` does not have type list cell, then `bfcons` branches to the address `lab`.

12.9.2 Access to list cell fields

These operands are only valid for pointers to list cells. Code written in LLM3 has to check that `accu` always contains a pointer to a list cell before using these operands.

`car` `accu` [*LLM3 operand*]

`cdr` `accu` [*LLM3 operand*]

12.9.3 Creation of list cells

There is no special instruction for this task. List cell creation is accomplished by explicitly calling one of the `cons` family LE-LISP functions such as `cons`, `ncons` or `xcons`.

12.10 `nil`

The `nil` operand contains the value of the end-of-list indicator. This is equivalent to the boolean *false* value.

`btnil` `op lab` [*LLM3 instruction*]

If the operand `op` is equal to `nil`, `btnil` branches to the address `lab`.

`bfnil` `op lab` [*LLM3 instruction*]

If the operand `op` is not equal to `nil`, `bfnil` branches to the address `lab`.

12.11 Symbols

12.11.1 Test for symbol type

`btsymb` `op lab` [*LLM3 instruction*]

If the operand `op` is a symbol, `btsymb` branches to the address `lab`.

`bfsymb` `op lab` [*LLM3 instruction*]

If the operand `op` is not a symbol, `bfsymb` branches to the address `lab`.

12.11.2 Access to the fields of a symbol

These operands are only valid for pointers to symbols. Code written in LLM3 has to check that `accu` always contains a pointer to a symbol before using these operands.

`cval accu` [*LLM3 operand*]

`cvalq symb` [*LLM3 operand*]

`plist accu` [*LLM3 operand*]

`fval accu` [*LLM3 operand*]

`fvalq symb` [*LLM3 operand*]

`oval accu` [*LLM3 operand*]

`alink accu` [*LLM3 operand*]

`pkgc accu` [*LLM3 operand*]

`pname accu` [*LLM3 operand*]

12.11.3 Variables

A variable is a symbol whose value can be changed. (See the function `variablep`.)

`btvar op lab` [*LLM3 instruction*]

If the operand `op` is a variable, `btvar` branches to the address `lab`.

`bfvar op lab` [*LLM3 instruction*]

If the operand `op` is not a variable, `bfvar` branches to the address `lab`.

12.12 Numbers

`llm3` uses two types of numbers:

- Integer numbers of 16 bits.
- Floating-point numbers of 31, 32, 48 or 64 bits, depending on the implementation.

12.12.1 16-bit integer numbers

Tests for integer type

`btfix op lab` [*LLM3 instruction*]

If the operand `op` is an integer, `btfix` branches to the address `lab`.

`bffix op lab` [*LLM3 instruction*]

If the operand `op` is not an integer, `bffix` branches to the address `lab`.

Instructions for numeric calculations

`incr op` [*LLM3 instruction*]

Performs the integer calculation `op + 1` → `op`.

`decr op` [*LLM3 instruction*]

Performs the integer calculation `op - 1` → `op`.

`plus op1 op2` [*LLM3 instruction*]

Performs the integer calculation `op2 + op1` → `op2`.

`diff op1 op2` [*LLM3 instruction*]

Performs the integer calculation `op2 - op1` → `op2`.

`negate op` [*LLM3 instruction*]

Performs the integer calculation `0 - op` → `op`.

`times op1 op2` [*LLM3 instruction*]

Performs the integer calculation `op2` × `op1` → `op2`.

`quo op1 op2` [*LLM3 instruction*]

Performs the integer calculation `op2` ÷ `op1` → `op2`.

rem op1 op2 [*LLM3 instruction*]

Performs the integer calculation op2 *rem* op1 → op2.

Instructions for integer numeric comparisons

cnbeq op1 op2 lab [*LLM3 instruction*]

Performs an integer numeric comparison. If the operand op1 is equal to the operand op2, a branch to the label lab occurs.

cnbne op1 op2 lab [*LLM3 instruction*]

Performs an integer numeric comparison. If the operand op1 is not equal to the operand op2, a branch to the label lab occurs.

cnble op1 op2 lab [*LLM3 instruction*]

Performs an integer numeric comparison. If the operand op1 is less than or equal to the operand op2, a branch to the label lab occurs.

cnblt op1 op2 lab [*LLM3 instruction*]

Performs an integer numeric comparison. If the operand op1 is less than the operand op2, a branch to the label lab occurs.

cnbge op1 op2 lab [*LLM3 instruction*]

Performs an integer numeric comparison. If the operand op1 is greater than or equal to the operand op2, a branch to the label lab occurs.

cnbgt op1 op2 lab [*LLM3 instruction*]

Performs an integer numeric comparison. If the operand op1 is greater than the operand op2, a branch to the label lab occurs.

Instructions for performing logical operations

Operands of these instructions must always be 16-bit integer values. The results of these instructions are always 16-bit values.

land op1 op2 [*LLM3 instruction*]

Performs the boolean calculation op2 *and* op1 → op2.

lor op1 op2 [*LLM3 instruction*]

Performs the boolean calculation op2 *or* op1 → op2.

lxor op1 op2 [*LLM3 instruction*]

Performs the boolean calculation op2 *xor* op1 → op2.

lshift op1 op2 [*LLM3 instruction*]

The operand op2 is shifted op1 bit positions, and the result is stored in op2. If op1 is positive, a left shift (multiplication) is performed. If it is negative, a right shift (division) is performed.

12.12.2 Floating-point numbers

Tests for floating-point type

btfloat op lab [*LLM3 instruction*]

If the operand op is a floating-point number, btfloat branches to the label lab.

bffloat op lab [*LLM3 instruction*]

If the operand op is not a floating-point number, bffloat branches to the label lab.

Floating-point calculations

fplus op1 op2 [*LLM3 instruction*]

Performs the floating-point calculation op2 + op1 → op2.

fdiff op1 op2 [*LLM3 instruction*]

Performs the floating-point calculation op2 - op1 → op2.

ftimes op1 op2 [*LLM3 instruction*]

Performs the floating-point calculation op2 × op1 → op2.

fquo op1 op2 [*LLM3 instruction*]

Performs the floating-point calculation op2 ÷ op1 → op2.

Floating-point comparisons

`cfbeq op1 op2 lab` [*LLM3 instruction*]

Performs a floating-point numeric comparison. If the operand `op1` is equal to the operand `op2`, a branch to the label `lab` occurs.

`cfbne op1 op2 lab` [*LLM3 instruction*]

Performs a floating-point numeric comparison. If the operand `op1` is not equal to the operand `op2`, a branch to the label `lab` occurs.

`cfblt op1 op2 lab` [*LLM3 instruction*]

Performs a floating-point numeric comparison. If the operand `op1` is less than the operand `op2`, a branch to the label `lab` occurs.

`cfble op1 op2 lab` [*LLM3 instruction*]

Performs a floating-point numeric comparison. If the operand `op1` is less than or equal to the operand `op2`, a branch to the label `lab` occurs.

`cfbgt op1 op2 lab` [*LLM3 instruction*]

Performs a floating-point numeric comparison. If the operand `op1` is greater than the operand `op2`, a branch to the label `lab` occurs.

`cfbge op1 op2 lab` [*LLM3 instruction*]

Performs a floating-point numeric comparison. If the operand `op1` is greater than or equal to the operand `op2`, a branch to the label `lab` occurs.

12.13 Vectors of Lisp pointers

12.13.1 Test for vector of pointers

`btvect op lab` [*LLM3 instruction*]

If `op` is a vector of pointers, then `btvect` branches to the label `lab`.

`bfvect op lab` [*LLM3 instruction*]

If `op` is not a vector of pointers, then `bfvect` branches to the label `lab`.

12.13.2 Access to internal fields of a vector of pointers

`val accu` [*LLM3 operand*]

This operand gives access to the heap pointer of a pointer.
Warning: This value cannot reside in a register, since the GC could become perturbed.

`typ accu` [*LLM3 operand*]

This operand gives access to the type of a vector.

12.13.3 Access to elements of a vector of pointers

hpxmov `vect n op` [*LLM3 instruction*]

Moves the n^{th} element of the vector of pointers `vect` into the operand `op`.

hpmovx `op vect n` [*LLM3 instruction*]

Moves the operand `op` into the n^{th} element of the vector of pointers `vect`.

12.13.4 Creation

There is no special insruction for the creation of vectors of pointers. Call the LE-LISP function `makevector`.

12.14 Character strings

12.14.1 Test for character string

btstrg `op lab` [*LLM3 instruction*]

If `op` is a character string, `btstrg` branches to the label `lab`.

bfstrg `op lab` [*LLM3 instruction*]

If `op` is not a character string, `bfstrg` branches to the label `lab`.

12.14.2 Access to the internal fields of a character string

val `accu` [*LLM3 operand*]

This operand returns a string's heap pointer. *Warning*: This value cannot reside in a register, since the GC might be perturbed.

typ `accu` [*LLM3 operand*]

This operand returns the type of a string.

12.14.3 Access to characters

hbxmov `strg index op` [*LLM3 instruction*]

Moves the $index^{th}$ character of the LISP string `strg` into the operand `op`.

hbmovx `op strg index` [*LLM3 instruction*]

Moves the operand `op` into the `index`th character position of the LISP string `strg`.

12.14.4 Creation

There is no special instruction to create strings. Call the LE-LISP function `makestring`.

12.15 Heap zone

hgsize `op1 op2` [*LLM3 instruction*]

Puts the size of the character string or vector pointer object `op1` in operand `op2`.

12.16 Extending the Loader/Assembler

It is possible to extend the syntax of the LLM3 loader/assembler by defining functions in special packages. These extensions are, by definition, machine-specific.

ld-codop [*constant*]

This is the name of the package in which functions associated with extended op codes must reside.

ld-dir [*constant*]

This is the name of the package in which functions associated with extended direct operands must reside.

ld-ind [*constant*]

This is the name of the package in which functions associated with extended indirect operands must reside.

12.17 Functions

12.17.1 Types of functions

Functions written in LLM3 have one of four types:

1. `subr`s, which evaluate their arguments.
2. `fsubr`s, which do not evaluate their arguments.

3. `msubr`s, which do not evaluate their arguments.
4. `dmsubr`s, which do not evaluate their arguments.

12.17.2 Function calling rules

- In the case of a `subr` with 0, 1, 2, or 3 arguments (referred to as a `subr0`, a `subr1`, a `subr2` or a `subr3`), arguments are passed in the registers A1, A2, and A3 respectively.
- In the case of a `subr` with more than three arguments, or with a variable number of arguments (referred to as an `nsubr`), the arguments are all pushed onto the stack and the number of arguments is placed in A4.
- In the case of a `fsubr`, the list of arguments, not evaluated (that is, the `cdr` of the call), is placed in A1.
- In the case of a `msubr`, the list of arguments, not evaluated (that is, the call form itself), is placed in A1.
- In the case of a `dmsubr`, the list of arguments, not evaluated (that is, the `cdr` of the call), is placed in A1.

All functions return a value in A1.

12.18 Examples

This section gives short examples of functions written in LLM3. There is a test file in the test directory named `testlap` which contains all the loader tests. It is also possible to examine the LLM3 code generated by the compilers. (See the next chapter.)

```
? ; These examples were produced on a Vax.
? ;
? ; Manual translation, into lap, of the celebrated fib function:
?
? ; (defun fiblap (n)
? ;     (if (le n 2)
? ;         1
? ;         (add (fiblap (sub n 1))
? ;              (fiblap (sub n 2)))))
?
? (defvar fiblap '(
?           (fentry fiblap subr1)
?           (cnbgt a1 '2 100)
?           (mov '1 a1)
?           (return)
? 100
?           (diff '1 a1)
?           (push a1)
?           (call fiblap)
?           (mov a1 a2)
?           (pop a1)
```

```
?           (push a2)
?           (diff '1 a1)
?           (call fiblap)
?           (pop a2)
?           (plus a2 a1)
?           (return)))
?
? (loader fiblap t)
    (fentry fiblap subr1)     000454e0
    (cnbgt a1 '2 100)         000454e0   b1 52 02 14 00
    (mov '1 a1)               000454e5   d0 01 52
    (return)                  000454e8   05
100                           000454e9
    (diff '1 a1)              000454e9   b7 52
    (push a1)                 000454eb   dd 52
    (call fiblap)             000454ed   10 f1
    (mov a1 a2)               000454ef   d0 52 53
    (pop a1)                  000454f2   d0 8e 52
    (push a2)                 000454f5   dd 53
    (diff '1 a1)              000454f7   b7 52
    (call fiblap)             000454f9   10 e5
    (pop a2)                  000454fb   d0 8e 53
    (plus a2 a1)              000454fe   a0 53 52
    (return)                  00045501   05
    (endl)                    00045502
= ()
? (fiblap 20)
= 6765
?
? ; Manual translation, into lap, of the dlq function:
? ;
? ; (defun dlq (a l)
? ;     (cond ((not (consp l)) ())
? ;           ((eq a (car l)) (dlq a (cdr l)))
? ;           (t (cons (car l) (dlq a (cdr l))))))
?
? (defvar llap1 '(
?        (fentry dlq subr2)
?        (btcons a2 1001)
?        (mov nil a1)
?        (return)
?     1001
?        (cabne a1 (car a2) 1003)
?        (mov (cdr a2) a2)
?        (bra dlq)
?     1003
?        (push (car a2))
?        (mov (cdr a2) a2)
?        (call dlq)
?        (mov a1 a2)
?        (pop a1)
?        (jmp cons))
= llap1
?
? (loader llap1 t)
    (fentry dlq subr2)        00043e2c
```

```
    (btcons a2 1001)          00043e2c   d1 5a 53 15 00
    (mov nil a1)              00043e31   d0 58 52
    (return)                  00043e34   05
1001                          00043e35
    (cabne a1 (car a2) 1003)  00043e35   d1 52 63 12 00
    (mov (cdr a2) a2)         00043e3a   d0 a3 04 53
    (bra dlq)                 00043e3e   11 ec
1003                          00043e40
    (push (car a2))           00043e40   dd 63
    (mov (cdr a2) a2)         00043e42   d0 a3 04 53
    (call dlq)                00043e46   10 e4
    (mov a1 a2)               00043e48   d0 52 53
    (pop a1)                  00043e4b   d0 8e 52
    (jmp cons)                00043e4e   17 d8 c8 3a
    (endl)                    00043e52
= ()
?
? (dlq 'a '(b a c a b))
= (b c b)
?
? ; This example demonstrates Lisp character string manipulation.
? ; The function definition is given below in Le-Lisp:
?
? ; (defun screat (x y)
? ;     (let ((s (makestring 6 #/x)))
? ;          (sset s x #/a)
? ;          (sset s y #/b)
? ;          s))
?
? (defvar llap2 '(
?       (fentry screat subr2)
?       (push a1)
?       (push a2)
?       (mov '6 a1)
?       (mov '#/x a2)
?       (jcall makestring)
?       (hbmovx '#/a a1 (& 1))
?       (hbmovx '#/b a1 (& 0))
?       (adjstk '2)
?       (return)))
= llap2
?
? (loader llap2 t)
    (fentry screat subr2)     00045e94
    (push a1)                 00045e94   dd 52
    (push a2)                 00045e96   dd 53
    (mov '6 a1)               00045e98   d0 06 52
    (mov '88 a2)              00045e9b   d0 8f 58 00 00 00 53
    (jcall makestring)        00045ea2   16 d8 88 51
    (hbmovx '97 a1 (& 1))     00045ea6   d0 62 51 d0 ae 04 50 90 8f 61 40
                              00045eb1   a1 08
    (hbmovx '98 a1 (& 0))     00045eb3   d0 62 51 d0 6e 50 90 8f 62 40 a1
                              00045ebe   08
    (adjstk '2)               00045ebf   c0 08 5e
    (return)                  00045ec2   05
    (endl)                    00045ec3
```

```
= ()
?
? (screat 2 4)
= "xxaxbx"
?
? ; This last example demonstrates vector manipulation.
? ; invector is similar to the vector function, but produces
? ; an inverted list. This example also shows the manipulation
? ; of n-ary functions in llm3.
?
? (defvar llap3 '(
?       (fentry invector nsubr)
?       (push a4)
?       (mov a4 a1)
?       (mov nil a2)
?       (jcall makevector)
?       (pop a4)
?       (mov '0 a3)
?       (bra 1005)
?     1002
?       (pop a2)
?       (hpmovx a2 a1 a3)
?       (plus '1 a3)
?     1005
?       (sobgez a4 1002)
?       (return)))
= llap3
?
? (loader llap3 t)
    (fentry invector nsubr)   00045ec4
    (push a4)                 00045ec4   dd 55
    (mov a4 a1)               00045ec6   d0 55 52
    (mov nil a2)              00045ec9   d0 58 53
    (jcall makevector)        00045ecc   16 d8 e8 53
    (pop a4)                  00045ed0   d0 8e 55
    (mov '0 a3)               00045ed3   d0 00 54
    (bra 1005)                00045ed6   11 00
1002                          00045ed8
    (pop a2)                  00045ed8   d0 8e 53
    (hpmovx a2 a1 a3)         00045edb   d0 62 51 d0 53 44 a1 08
    (plus '1 a3)              00045ee3   b6 54
1005                          00045ee5
    (sobgez a4 1002)          00045ee5   b7 55 18 ef
    (return)                  00045ee9   05
    (endl)                    00045eea
= ()
?
? (invector 1 2 3 4 5)
= #[5 4 3 2 1]
```

Table of contents

12 Loader/assembler LLM3 **12-1**

12.1 Access to memory and the CPU 12-1
12.2 LLM3 memory loader 12-4
12.3 LLM3 instruction format 12-5
12.4 Modules and labels 12-5
12.5 LLM3 instruction operands 12-6
12.6 Pseudo-instructions 12-8
12.7 Basic instructions 12-9
12.7.1 Moving pointers 12-9
12.7.2 Pointer comparisons 12-9
12.7.3 Control 12-9
12.8 Stack 12-10
12.8.1 Management of the stack pointer 12-10
12.8.2 As a control stack 12-10
12.8.3 As a data stack 12-12
12.9 List cell `cons` operations 12-12
12.9.1 Test for list cell type 12-12
12.9.2 Access to list cell fields 12-13
12.9.3 Creation of list cells 12-13
12.10`nil` 12-13
12.11Symbols 12-13
12.11.1Test for symbol type 12-13
12.11.2Access to the fields of a symbol 12-14
12.11.3Variables 12-14
12.12Numbers 12-14

12.12.1 16-bit integer numbers 12-15
12.12.2 Floating-point numbers 12-17
12.13 Vectors of Lisp pointers 12-19
12.13.1 Test for vector of pointers 12-19
12.13.2 Access to internal fields of a vector of pointers 12-19
12.13.3 Access to elements of a vector of pointers 12-20
12.13.4 Creation 12-20
12.14 Character strings 12-20
12.14.1 Test for character string 12-20
12.14.2 Access to the internal fields of a character string 12-20
12.14.3 Access to characters 12-20
12.14.4 Creation 12-21
12.15 Heap zone 12-21
12.16 Extending the Loader/Assembler 12-21
12.17 Functions 12-21
12.17.1 Types of functions 12-21
12.17.2 Function calling rules 12-22
12.18 Examples 12-22

Function Index

`(#:system:ccode addr)` [*function with an optional argument*] 12-2
`(#:system:ecode)` [*function with no arguments*] 12-2
`(addadr addr1 addr2)` [*function with two arguments*] 12-2
`(subadr addr1 addr2)` [*function with two arguments*] 12-2
`(incradr addr n)` [*function with two arguments*] 12-2
`(gtadr addr1 addr2)` [*function with two arguments*] 12-3
`(loc s)` [*function with one argument*] 12-3
`(vag addr)` [*function with one argument*] 12-3
`(memory addr n)` [*function with one or two arguments*] 12-3
`(call addr a1 a2 a3)` [*function with four arguments*] 12-4
`(calln addr l)` [*function with two arguments*] 12-4
`loader` [*feature*] 12-4
`(loader l i)` [*function with two arguments*] 12-4
`#:ld:special-case-loader` [*variable*] 12-4
`#:ld:shared-strings` [*variable*] 12-5
`a1` [*LLM3 operand*] 12-6
`a2` [*LLM3 operand*] 12-6
`a3` [*LLM3 operand*] 12-6
`a4` [*LLM3 operand*] 12-6
`nil` [*LLM3 operand*] 12-6
`quote exp` [*LLM3 operand*] 12-6
`car accu` [*LLM3 operand*] 12-6
`cdr accu` [*LLM3 operand*] 12-6
`cval accu` [*LLM3 operand*] 12-6
`plist accu` [*LLM3 operand*] 12-6
`fval accu` [*LLM3 operand*] 12-6

`pkgc accu` [*LLM3 operand*] 12-7
`oval accu` [*LLM3 operand*] 12-7
`alink accu` [*LLM3 operand*] 12-7
`pname accu` [*LLM3 operand*] 12-7
`cvalq symb` [*LLM3 operand*] 12-7
`fvalq symb` [*LLM3 operand*] 12-7
`typ accu` [*LLM3 operand*] 12-7
`val accu` [*LLM3 operand*] 12-7
`& n` [*LLM3 operand*] 12-7
`lab` [*LLM3 operand*] 12-7
`@ lab` [*LLM3 operand*] 12-7
`eval expr` [*LLM3 operand*] 12-8
`title symb` [*LLM3 pseudo-instruction*] 12-8
`local symb` [*LLM3 pseudo-instruction*] 12-8
`fentry symb ftype` [*LLM3 pseudo-instruction*] 12-8
`entry symb ftype` [*LLM3 pseudo-instruction*] 12-8
`endl` [*LLM3 pseudo-instruction*] 12-8
`end` [*LLM3 pseudo-instruction*] 12-8
`eval e` [*LLM3 pseudo-instruction*] 12-8
`mov op1 op2` [*LLM3 instruction*] 12-9
`cabeq op1 op2 lab` [*LLM3 instruction*] 12-9
`cabne op1 op2 lab` [*LLM3 instruction*] 12-9
`bra lab` [*LLM3 instruction*] 12-9
`jmp symb` [*LLM3 instruction*] 12-9
`bri op` [*LLM3 instruction*] 12-9
`brx llab op` [*LLM3 instruction*] 12-9
`sobgez op lab` [*LLM3 instruction*] 12-10
`nop` [*LLM3 instruction*] 12-10
`stack op` [*LLM3 instruction*] 12-10
`sstack op` [*LLM3 instruction*] 12-10
`call lab` [*LLM3 instruction*] 12-10
`calli op` [*LLM3 instruction*] 12-10
`jcall symb` [*LLM3 instruction*] 12-11
`return` [*LLM3 instruction*] 12-12

`push op` [*LLM3 instruction*] 12-12
`pop op` [*LLM3 instruction*] 12-12
`adjstk op` [*LLM3 instruction*] 12-12
`movxsp op1 op2` [*LLM3 instruction*] 12-12
`xspmov op1 op2` [*LLM3 instruction*] 12-12
`btcons op lab` [*LLM3 instruction*] 12-12
`bfcons op lab` [*LLM3 instruction*] 12-12
`car accu` [*LLM3 operand*] 12-13
`cdr accu` [*LLM3 operand*] 12-13
`btnil op lab` [*LLM3 instruction*] 12-13
`bfnil op lab` [*LLM3 instruction*] 12-13
`btsymb op lab` [*LLM3 instruction*] 12-13
`bfsymb op lab` [*LLM3 instruction*] 12-13
`cval accu` [*LLM3 operand*] 12-14
`cvalq symb` [*LLM3 operand*] 12-14
`plist accu` [*LLM3 operand*] 12-14
`fval accu` [*LLM3 operand*] 12-14
`fvalq symb` [*LLM3 operand*] 12-14
`oval accu` [*LLM3 operand*] 12-14
`alink accu` [*LLM3 operand*] 12-14
`pkgc accu` [*LLM3 operand*] 12-14
`pname accu` [*LLM3 operand*] 12-14
`btvar op lab` [*LLM3 instruction*] 12-14
`bfvar op lab` [*LLM3 instruction*] 12-14
`btfix op lab` [*LLM3 instruction*] 12-15
`bffix op lab` [*LLM3 instruction*] 12-15
`incr op` [*LLM3 instruction*] 12-15
`decr op` [*LLM3 instruction*] 12-15
`plus op1 op2` [*LLM3 instruction*] 12-15
`diff op1 op2` [*LLM3 instruction*] 12-15
`negate op` [*LLM3 instruction*] 12-15
`times op1 op2` [*LLM3 instruction*] 12-15
`quo op1 op2` [*LLM3 instruction*] 12-15
`rem op1 op2` [*LLM3 instruction*] 12-16

`cnbeq op1 op2 lab` [*LLM3 instruction*] 12-16
`cnbne op1 op2 lab` [*LLM3 instruction*] 12-16
`cnble op1 op2 lab` [*LLM3 instruction*] 12-16
`cnblt op1 op2 lab` [*LLM3 instruction*] 12-16
`cnbge op1 op2 lab` [*LLM3 instruction*] 12-16
`cnbgt op1 op2 lab` [*LLM3 instruction*] 12-16
`land op1 op2` [*LLM3 instruction*] 12-16
`lor op1 op2` [*LLM3 instruction*] 12-17
`lxor op1 op2` [*LLM3 instruction*] 12-17
`lshift op1 op2` [*LLM3 instruction*] 12-17
`btfloat op lab` [*LLM3 instruction*] 12-17
`bffloat op lab` [*LLM3 instruction*] 12-17
`fplus op1 op2` [*LLM3 instruction*] 12-17
`fdiff op1 op2` [*LLM3 instruction*] 12-17
`ftimes op1 op2` [*LLM3 instruction*] 12-17
`fquo op1 op2` [*LLM3 instruction*] 12-17
`cfbeq op1 op2 lab` [*LLM3 instruction*] 12-18
`cfbne op1 op2 lab` [*LLM3 instruction*] 12-18
`cfblt op1 op2 lab` [*LLM3 instruction*] 12-19
`cfble op1 op2 lab` [*LLM3 instruction*] 12-19
`cfbgt op1 op2 lab` [*LLM3 instruction*] 12-19
`cfbge op1 op2 lab` [*LLM3 instruction*] 12-19
`btvect op lab` [*LLM3 instruction*] 12-19
`bfvect op lab` [*LLM3 instruction*] 12-19
`val accu` [*LLM3 operand*] 12-19
`typ accu` [*LLM3 operand*] 12-19
`hpxmov vect n op` [*LLM3 instruction*] 12-20
`hpmovx op vect n` [*LLM3 instruction*] 12-20
`btstrg op lab` [*LLM3 instruction*] 12-20
`bfstrg op lab` [*LLM3 instruction*] 12-20
`val accu` [*LLM3 operand*] 12-20
`typ accu` [*LLM3 operand*] 12-20
`hbxmov strg index op` [*LLM3 instruction*] 12-20
`hbmovx op strg index` [*LLM3 instruction*] 12-21

`hgsize op1 op2` [*LLM3 instruction*] 12-21
`ld-codop` [*constant*] 12-21
`ld-dir` [*constant*] 12-21
`ld-ind` [*constant*] 12-21

Chapter 12

Loader/assembler LLM3

The heart of the LE-LISP system is written in the LLM3 virtual machine language [Chailloux 85b]. This language is available in LISP. It is used by the various LE-LISP compilers but can also be used to write new standard functions, or to write compilers for languages other than LISP. This chapter describes the functions that provide access to internal machine resources and the syntax used by the LLM3 loader/assembler, often called LAP, for *Lisp Assembly Program*.

12.1 Access to memory and the CPU

Functions provided in this section allow you to directly access memory and the CPU. The LLM3 loader/assembler uses a special memory zone, called the *code zone*, to load and store instructions. The following functions are obviously machine-dependent, and are used to build the LLM3 loader/assembler.

These functions are designed to operate on all possible memory addresses, for all machine architectures. Since LE-LISP integers are 16 bits long, they cannot always represent memory addresses. A memory address—referred to as `addr` throughout this chapter—is therefore represented in LISP in one of two possible ways:

- By a 16-bit number with an extended sign, to represent a complete 32-bit address.
- By a `cons` of two numbers of the form `(high . low)`. The `car`, `high`, contains the 16 high-order bits of the address. The `cdr`, `low`, contains the 16 low-order bits.

The address	`#$001f3c00`	is stored as	`(#$1f . #$3c00)`
" "	`#$00008000`	" "	`(0 . #$8000)`
" "	`#$ffff0000`	" "	`(#$ffff . 0)`
" "	`#$0000ffff`	" "	`(0 . #$ffff)`
" "	`#$ffff8100`	" "	`#$8100`
" "	`#$00001000`	" "	`#$1000`

The functions that follow test the validity of memory address-type arguments. They can raise the `errdna` error, which has the following default screen display:

```
** <fn> : bad address : <s>
```

Here, `fn` is the name of the function called, and `s` is the argument that is not of type memory address.

(#:system:ccode addr)

[function with an optional argument]

This function reads—or writes, if the `addr` argument is given—the current memory load address in the code zone.

(#:system:ecode)

[function with no arguments]

Reads the last memory address in the code zone.

(addadr addr1 addr2)

[function with two arguments]

Returns the sum of the two memory addresses `addr1` and `addr2`. It allocates a new `cons` if its result must be represented in the `cons` form.

```
(addadr '(3 . 657) '(4 . 4567))       ⟹   (7 . 5224)
(addadr '(1 . #$8000) '(1 . #$8000))  ⟹   (3 . 0)
(addadr -1 -1)                        ⟹   -2
(addadr -1 1)                         ⟹   0
(addadr #$7fff 1)                     ⟹   (0 . #$8000)
(addadr 1 '(1 . 234))                 ⟹   (1 . 235)
(addadr 1 '(1 . #$ffff))              ⟹   (2 . 0)
(addadr '(1 . #$ffff) 1)              ⟹   (2 . 0)
(addadr '(#$ffff . #$8001) 1)         ⟹   -32766
```

(subadr addr1 addr2)

[function with two arguments]

Returns the difference of the two addresses `addr1` and `addr2`. It allocates a new `cons` if its result must be represented in the `cons` form.

```
(subadr '(56 . 7899) '(45 . 3333))    ⟹   (11 . 4566)
(subadr '(1 . #$8000) '(1 . #$8001))  ⟹   -1
(subadr 0 '(3 . 3))                   ⟹   (-4 . -3)
(subadr 1 '(1 . 0))                   ⟹   (-1 . 1)
(subadr 1 '(1 . #$ffff))              ⟹   (-2 . 2)
```

(incradr addr n)

[function with two arguments]

Adds `n` to the memory address `addr`. If this address is in the form of a `cons`, `incradr` physically modifies the `cons` cell.

```
(setq addr '(3 . #$fffe))  ⟹   (3 . -2)
(incradr addr 1)           ⟹   (3 . -1)
```

```
(incradr addr 1)          ⟹   (4 . 0)
(incradr addr 1)          ⟹   (4 . 1)
(incradr addr 100)        ⟹   (4 . 101)
addr                      ⟹   (4 . 101)
```

(gtadr addr1 addr2) [*function with two arguments*]

Returns `t` if the memory address `addr1` is greater than the address `addr2` and `()` otherwise.

```
(setq addr '(4 . 101))    ⟹   (4 . 101)
(gtadr addr '(4 . -10))   ⟹   ()
(gtadr addr '(4 . 100))   ⟹   t
(gtadr addr '(4 . 101))   ⟹   ()
```

(loc s) [*function with one argument*]

Provides a way to get the absolute memory address of the LISP object `s`. The memory address of a LISP object is the pointer to it. This function provides a means of locating in memory objects that are created dynamically. `loc` cannot be described in LISP.

```
(loc '(a b c))   ⟹   (15 . 288)
(loc '(a b c))   ⟹   (13 . 1200)
```

(vag addr) [*function with one argument*]

This is the inverse of the `loc` function. `vag` returns the LISP object whose absolute memory address is supplied as the argument. So, `(vag (loc s))` is equivalent to `s` itself. The `vag` function cannot be described in LISP.

> *Warning*: `vag` does not test whether or not `addr` is the address of a real LISP object, and an erroneous call to this function could raise an error on the host system. In particular it is possible to refer to objects that have disappeared:

```
(let ((x (loc (cons 'a 'b))))
     (gc)
     (vag x))       ; the value of x no longer exists!
                    ; vag returns a pointer to the
                    ; free list of conses!
```

(memory addr n) [*function with one or two arguments*]

`memory` samples, or if the second numeric argument `n` is provided changes, the word of memory whose address `addr` is given as its first argument. This function does no perform validity checking and must be used with the utmost care. It returns a number representing the value of the memory word (after modification, if the two-argument form was used). This function is limited by the nature of the host CPU: 8-bit or 16-bit words, word alignment considerations, etc.

(call addr a1 a2 a3) [*function with four arguments*]

The first argument `addr` must be the memory address of a memory-resident subroutine. `call` will call this subroutine after having loaded the three values `a1`, `a2` and `a3` into the LLM3 accumulators A1, A2, and A3, respectively. This call format is used by `subr0`s, `subr1`s, `subr2`s, `subr3`s, `fsubr`s, `msubr`s and `dmsubr`s. (See the section on function calls 12.7.) This function must be used with care, since it performs no tests for the validity of the executed code.

`call` returns as its value the contents of the A1 accumulator after the code beginning at `addr` has been executed.

Warning: The code executed by means of a `call` statement must end with an LLM3 `return` instruction, and load the A1 register with a LISP value to be returned by the `call` function.

(calln addr l) [*function with two arguments*]

This function is similar to `call`, but pushes the list of arguments contained in `l` onto the stack before calling the subroutine stored at memory address `addr`. The number of arguments pushed will be available in the A4 accumulator. This is the form that `nsubr`s take. (See the section on function calls 12.7.)

Warning: The code invoked by means of `calln` must remove its arguments from the stack (using the number of arguments value stored in A4 and, for example, the LLM3 `adjstk` instruction), terminate with the LLM3 `return` instruction, and load the A1 register with a LISP value to be returned by the `calln` function.

12.2 LLM3 memory loader

loader [*feature*]

This feature indicates whether the LLM3 loader/assembler is currently in memory.

(loader l i) [*function with two arguments*]

The `loader` function loads the list of LLM3 instructions `l` into memory. The format of these instructions is described in the next section. If the indicator `i` is true, the loader will print, in the host machine language, the assembly listing of these instructions on the current output stream. `loader` returns `()` as its value.

#:ld:special-case-loader [*variable*]

The pseudo-instruction `fentry` changes the functional value of functions during loading. In certain difficult boot-strapping cases (loading the loader itself, for example), functional values must be resolved at the end of the load (when the pseudo-instruction `end` is encountered). The `#:ld:special-case-loader` variable is `t` in these special cases, but `()` by default.

#:ld:shared-strings [*variable*]

This variable indicates whether character string constants should be shared during the loading of LLM3 instructions.

12.3 LLM3 instruction format

A list of LLM3 instructions contains atoms representing the labels and lists representing the instructions. An instruction is itself a list of the form:

(codop $op_1 \ldots op_n$)

where codop is the instruction mnemonic, and $op_1 \ldots op_n$ are the operands. Only the codop field is obligatory.

12.4 Modules and labels

A module is a set of lists of instructions beginning with the title pseudo-instruction and ending with the end pseudo-instruction.

There are three types of labels:

1. Labels local to a list of instructions.
2. Labels local to a module.
3. Global labels.

Labels local to a list of instructions are represented by integers or by symbols declared using the local pseudo-instruction. A symbolic label which is local to a list of instructions must be declared before it is used. These labels must be resolved before the end of the list of instructions, or when the endl pseudo-instruction is encountered. All references to this kind of label generate jumps relative to the program counter.

Labels local to a module are always symbolic. They are defined using the entry pseudo-instruction. They can be undefined at incremental calls to the loader but must be resolved when the end pseudo-instruction is encountered. All references to this kind of label generate jumps relative to the program counter.

Global labels are also always symbolic. They are defined using the fentry pseudo-instruction. All references to this type of label generate an access to the functional value of the symbol. All the standard functions are of this type.

12.5 LLM3 instruction operands

LLM3 has four types of operands:

1. Registers (accumulators).
2. LISP object immediate values.
3. LISP object fields.
4. Labels.

`a1` [*LLM3 operand*]

`a2` [*LLM3 operand*]

`a3` [*LLM3 operand*]

`a4` [*LLM3 operand*]

These four operands represent the registers (accumulators) of the LLM3 machine.

`nil` [*LLM3 operand*]

This operand represents the special null symbol `||`, which is used as the end-of-list marker and as the boolean *false* value.

`quote exp` [*LLM3 operand*]

This operand represents a LE-LISP constant of any type.

`car accu` [*LLM3 operand*]

`cdr accu` [*LLM3 operand*]

These operands represent access to the constituents of a list element.

`cval accu` [*LLM3 operand*]

`plist accu` [*LLM3 operand*]

`fval accu` [*LLM3 operand*]

pkgc `accu` [*LLM3 operand*]

oval `accu` [*LLM3 operand*]

alink `accu` [*LLM3 operand*]

pname `accu` [*LLM3 operand*]

These operands represent access to the various constituents of a symbol.

cvalq `symb` [*LLM3 operand*]

This operand gives direct access to the value of the symbol named `symb`, considered as a variable: that is, its `cval`.

fvalq `symb` [*LLM3 operand*]

This operand gives direct access to the value of the symbol named `symb`, considered as a function: that is, its `fval`.

typ `accu` [*LLM3 operand*]

val `accu` [*LLM3 operand*]

These operands represent access to constituents of objects of type character string or vector of S-expressions.

& `n` [*LLM3 operand*]

This operand represents the $\texttt{n}^{th}$ element on the stack. The top element of the stack is represented by `(& 0)`, the next-to-top element by `(& 1)`, and so on.

lab [*LLM3 operand*]

This operand refers to a label which is local to a list of instructions or to a module.

@ `lab` [*LLM3 operand*]

This operand represents the address of an LLM3 label. This label is either local to a list of instructions or to a module.

eval `expr` [*LLM3 operand*]

This operand dynamically calls the LE-LISP evaluator on the expression `expr`. The value returned by the evaluation of an `eval` operand must be a legal LLM3 operand.

12.6 Pseudo-instructions

`title` `symb` [*LLM3 pseudo-instruction*]

Declares the name of a module. Inside a module, `title` appears as the first instruction.

`local` `symb` [*LLM3 pseudo-instruction*]

`local` provides a way to declare a label local to a list of instructions. By default, numeric labels are always local, and symbolic labels are always global.

`fentry` `symb ftype` [*LLM3 pseudo-instruction*]

The `symb` argument is the name of the LISP symbol that will have a global function (of type `subr`) written in LLM3 as its function value. `ftype` must be one of the following LE-LISP functional types: `subr0`, `subr1`, `subr2`, `subr3`, `nsubr`, `fsubr`, `msubr` or `dmsubr`.

`entry` `symb ftype` [*LLM3 pseudo-instruction*]

The `symb` argument is the name of the LISP symbol that will have a local function (of type `subr`) written in LLM3 as its function value. `ftype` is one of the following LE-LISP functional types: `subr0`, `subr1`, `subr2`, `subr3`, `nsubr`, `fsubr`, `msubr` ou `dmsubr`. After loading, this function is no longer accessible.

`endl` [*LLM3 pseudo-instruction*]

Terminates the loading of a local function. All local labels must be resolved at this point and are then discarded.

`end` [*LLM3 pseudo-instruction*]

Terminates the loading of a module.

`eval` `e` [*LLM3 pseudo-instruction*]

Evaluates the expression `e` within the context of the loader. It loads nothing into memory unless explicitly instructed to do so.

12.7 Basic instructions

12.7.1 Moving pointers

Along with the `mov` instruction, which provides a means to move any pointer to a LISP object, there are other specialized instructions for moving pointers or bytes in the heap storage or on the stack.

`mov op1 op2` *[LLM3 instruction]*

Moves the pointer contained in the source operand `op1` to the destination operand `op2`. The destination operand cannot be an immediate value.

12.7.2 Pointer comparisons

`cabeq op1 op2 lab` *[LLM3 instruction]*

Performs a pointer comparison. If the operand `op1` is equal to the operand `op2`, a branch will occur to the local label `lab`.

`cabne op1 op2 lab` *[LLM3 instruction]*

Like `cabeq`, `cabne` does a pointer comparison. If the operand `op1` is not equal to the operand `op2`, a branch will occur to the local label `lab`.

12.7.3 Control

`bra lab` *[LLM3 instruction]*

Branches to the local label `lab`, which must be within the LLM3 module. The branch is done relative to the program counter.

`jmp symb` *[LLM3 instruction]*

Jumps to the address of the functional value associated with the symbol `symb`. The branch is an indirection through the `fval` of the symbol `symb`. Jumps to the code of standard functions are `jmp`s and not `bra`s.

`bri op` *[LLM3 instruction]*

Branches indirectly to the address contained in the operand `op`.

`brx llab op` *[LLM3 instruction]*

Performs an indexed branch through the list of labels `llab`. `op` is the zero-relative index used. The list of labels is a list of operands of label-reference type: that is, `(@ lab)`.

sobgez `op lab` [*LLM3 instruction*]

Decrements the operand `op`. If this operand is greater than or equal to zero, a branch is done to the label `lab`.

nop [*LLM3 instruction*]

Takes up a little space, wastes a little time, and finally does nothing visible.

12.8 Stack

LLM3 has a single stack for both control and data. It is used as an implicit operand in some special stack-oriented instructions. These instructions—which use the stack pointer register, referred to as `SP`—do not require the stack to grow in a specific direction. Also, they do not require the stack to have implicit boundary-overflow tests built in. The stack normally occupies the space of 6K objects.

12.8.1 Management of the stack pointer

Two instructions provide for the explicit manipulation of the stack pointer `SP`.

stack `op` [*LLM3 instruction*]

Moves the current contents of the stack pointer into the operand `op`.

sstack `op` [*LLM3 instruction*]

Moves the operand `op` into the stack pointer.

12.8.2 As a control stack

Three instructions provide for the manipulation of return addresses stored on the stack.

call `lab` [*LLM3 instruction*]

Pushes the current program counter onto the top of the stack and branches to the local label `lab`, which must be in the current LLM3 module. The branch is direct and relative.

calli `op` [*LLM3 instruction*]

Pushes the current program counter onto the top of the stack and branches to the address contained in the operand `op`.

jcall `symb` [*LLM3 instruction*]

This function is similar to `calli`, but the destination of the branch can be in another LLM3 module. The branch is an indirection through the `fval` of the symbol `symb`. All the standard function calls must use `jcall` and not `call`.

`return` [*LLM3 instruction*]

Pops the top address off the stack, and it becomes the new value of the program counter.

12.8.3 As a data stack

`push` `op` [*LLM3 instruction*]

Pushes the operand `op` onto the stack. The stack pointer is modified as required.

`pop` `op` [*LLM3 instruction*]

Pops the value on the top of the stack into the operand `op`. The stack pointer is modified as required.

`adjstk` `op` [*LLM3 instruction*]

Adjusts the stack pointer so that the `op` last objects on the stack are popped off. If `op` is negative, then -`op` places are created at the top of the stack.

For the next two instructions, assume that the index for the top of the stack is zero, that the index for the second element is one, that the index for the third element is two, and so on.

`movxsp` `op1 op2` [*LLM3 instruction*]

Moves the operand `op1` into the `op2`th position on the stack.

`xspmov` `op1 op2` [*LLM3 instruction*]

Moves the object contained in the `op1`th stack position into the operand `op2`.

12.9 List cell `cons` operations

12.9.1 Test for list cell type

`btcons` `op lab` [*LLM3 instruction*]

If the operand `op` has type list cell, then `btcons` branches to the address `lab`.

bfcons `op lab` [*LLM3 instruction*]

If the operand `op` does not have type list cell, then `bfcons` branches to the address `lab`.

12.9.2 Access to list cell fields

These operands are only valid for pointers to list cells. Code written in LLM3 has to check that `accu` always contains a pointer to a list cell before using these operands.

car `accu` [*LLM3 operand*]

cdr `accu` [*LLM3 operand*]

12.9.3 Creation of list cells

There is no special instruction for this task. List cell creation is accomplished by explicitly calling one of the `cons` family LE-LISP functions such as `cons`, `ncons` or `xcons`.

12.10 nil

The `nil` operand contains the value of the end-of-list indicator. This is equivalent to the boolean *false* value.

btnil `op lab` [*LLM3 instruction*]

If the operand `op` is equal to `nil`, `btnil` branches to the address `lab`.

bfnil `op lab` [*LLM3 instruction*]

If the operand `op` is not equal to `nil`, `bfnil` branches to the address `lab`.

12.11 Symbols

12.11.1 Test for symbol type

btsymb `op lab` [*LLM3 instruction*]

If the operand `op` is a symbol, `btsymb` branches to the address `lab`.

bfsymb `op lab` [*LLM3 instruction*]

If the operand `op` is not a symbol, `bfsymb` branches to the address `lab`.

12.11.2 Access to the fields of a symbol

These operands are only valid for pointers to symbols. Code written in LLM3 has to check that `accu` always contains a pointer to a symbol before using these operands.

`cval accu` [*LLM3 operand*]

`cvalq symb` [*LLM3 operand*]

`plist accu` [*LLM3 operand*]

`fval accu` [*LLM3 operand*]

`fvalq symb` [*LLM3 operand*]

`oval accu` [*LLM3 operand*]

`alink accu` [*LLM3 operand*]

`pkgc accu` [*LLM3 operand*]

`pname accu` [*LLM3 operand*]

12.11.3 Variables

A variable is a symbol whose value can be changed. (See the function `variablep`.)

`btvar op lab` [*LLM3 instruction*]

If the operand `op` is a variable, `btvar` branches to the address `lab`.

`bfvar op lab` [*LLM3 instruction*]

If the operand `op` is not a variable, `bfvar` branches to the address `lab`.

12.12 Numbers

`llm3` uses two types of numbers:

- Integer numbers of 16 bits.
- Floating-point numbers of 31, 32, 48 or 64 bits, depending on the implementation.

12.12.1 16-bit integer numbers

Tests for integer type

`btfix op lab` [*LLM3 instruction*]

If the operand `op` is an integer, `btfix` branches to the address `lab`.

`bffix op lab` [*LLM3 instruction*]

If the operand `op` is not an integer, `bffix` branches to the address `lab`.

Instructions for numeric calculations

`incr op` [*LLM3 instruction*]

Performs the integer calculation `op + 1` → `op`.

`decr op` [*LLM3 instruction*]

Performs the integer calculation `op - 1` → `op`.

`plus op1 op2` [*LLM3 instruction*]

Performs the integer calculation `op2 + op1` → `op2`.

`diff op1 op2` [*LLM3 instruction*]

Performs the integer calculation `op2 - op1` → `op2`.

`negate op` [*LLM3 instruction*]

Performs the integer calculation `0 - op` → `op`.

`times op1 op2` [*LLM3 instruction*]

Performs the integer calculation `op2` × `op1` → `op2`.

`quo op1 op2` [*LLM3 instruction*]

Performs the integer calculation `op2` ÷ `op1` → `op2`.

rem `op1 op2` [*LLM3 instruction*]

Performs the integer calculation `op2` *rem* `op1` → `op2`.

Instructions for integer numeric comparisons

cnbeq `op1 op2 lab` [*LLM3 instruction*]

Performs an integer numeric comparison. If the operand `op1` is equal to the operand `op2`, a branch to the label `lab` occurs.

cnbne `op1 op2 lab` [*LLM3 instruction*]

Performs an integer numeric comparison. If the operand `op1` is not equal to the operand `op2`, a branch to the label `lab` occurs.

cnble `op1 op2 lab` [*LLM3 instruction*]

Performs an integer numeric comparison. If the operand `op1` is less than or equal to the operand `op2`, a branch to the label `lab` occurs.

cnblt `op1 op2 lab` [*LLM3 instruction*]

Performs an integer numeric comparison. If the operand `op1` is less than the operand `op2`, a branch to the label `lab` occurs.

cnbge `op1 op2 lab` [*LLM3 instruction*]

Performs an integer numeric comparison. If the operand `op1` is greater than or equal to the operand `op2`, a branch to the label `lab` occurs.

cnbgt `op1 op2 lab` [*LLM3 instruction*]

Performs an integer numeric comparison. If the operand `op1` is greater than the operand `op2`, a branch to the label `lab` occurs.

Instructions for performing logical operations

Operands of these instructions must always be 16-bit integer values. The results of these instructions are always 16-bit values.

land `op1 op2` [*LLM3 instruction*]

Performs the boolean calculation `op2` *and* `op1` → `op2`.

lor op1 op2 [*LLM3 instruction*]

Performs the boolean calculation op2 *or* op1 → op2.

lxor op1 op2 [*LLM3 instruction*]

Performs the boolean calculation op2 *xor* op1 → op2.

lshift op1 op2 [*LLM3 instruction*]

The operand op2 is shifted op1 bit positions, and the result is stored in op2. If op1 is positive, a left shift (multiplication) is performed. If it is negative, a right shift (division) is performed.

12.12.2 Floating-point numbers

Tests for floating-point type

btfloat op lab [*LLM3 instruction*]

If the operand op is a floating-point number, btfloat branches to the label lab.

bffloat op lab [*LLM3 instruction*]

If the operand op is not a floating-point number, bffloat branches to the label lab.

Floating-point calculations

fplus op1 op2 [*LLM3 instruction*]

Performs the floating-point calculation op2 + op1 → op2.

fdiff op1 op2 [*LLM3 instruction*]

Performs the floating-point calculation op2 - op1 → op2.

ftimes op1 op2 [*LLM3 instruction*]

Performs the floating-point calculation op2 × op1 → op2.

fquo op1 op2 [*LLM3 instruction*]

Performs the floating-point calculation op2 ÷ op1 → op2.

Floating-point comparisons

`cfbeq op1 op2 lab` [*LLM3 instruction*]

Performs a floating-point numeric comparison. If the operand `op1` is equal to the operand `op2`, a branch to the label `lab` occurs.

`cfbne op1 op2 lab` [*LLM3 instruction*]

Performs a floating-point numeric comparison. If the operand `op1` is not equal to the operand `op2`, a branch to the label `lab` occurs.

`cfblt op1 op2 lab` [*LLM3 instruction*]

Performs a floating-point numeric comparison. If the operand `op1` is less than the operand `op2`, a branch to the label `lab` occurs.

`cfble op1 op2 lab` [*LLM3 instruction*]

Performs a floating-point numeric comparison. If the operand `op1` is less than or equal to the operand `op2`, a branch to the label `lab` occurs.

`cfbgt op1 op2 lab` [*LLM3 instruction*]

Performs a floating-point numeric comparison. If the operand `op1` is greater than the operand `op2`, a branch to the label `lab` occurs.

`cfbge op1 op2 lab` [*LLM3 instruction*]

Performs a floating-point numeric comparison. If the operand `op1` is greater than or equal to the operand `op2`, a branch to the label `lab` occurs.

12.13 Vectors of Lisp pointers

12.13.1 Test for vector of pointers

`btvect op lab` [*LLM3 instruction*]

If `op` is a vector of pointers, then `btvect` branches to the label `lab`.

`bfvect op lab` [*LLM3 instruction*]

If `op` is not a vector of pointers, then `bfvect` branches to the label `lab`.

12.13.2 Access to internal fields of a vector of pointers

val `accu` [*LLM3 operand*]

This operand gives access to the heap pointer of a pointer.
Warning: This value cannot reside in a register, since the GC could become perturbed.

typ `accu` [*LLM3 operand*]

This operand gives access to the type of a vector.

12.13.3 Access to elements of a vector of pointers

hpxmov `vect n op` [*LLM3 instruction*]

Moves the n^{th} element of the vector of pointers `vect` into the operand `op`.

hpmovx `op vect n` [*LLM3 instruction*]

Moves the operand `op` into the n^{th} element of the vector of pointers `vect`.

12.13.4 Creation

There is no special insruction for the creation of vectors of pointers. Call the LE-LISP function `makevector`.

12.14 Character strings

12.14.1 Test for character string

btstrg `op lab` [*LLM3 instruction*]

If `op` is a character string, `btstrg` branches to the label `lab`.

bfstrg `op lab` [*LLM3 instruction*]

If `op` is not a character string, `bfstrg` branches to the label `lab`.

12.14.2 Access to the internal fields of a character string

val `accu` [*LLM3 operand*]

This operand returns a string's heap pointer. *Warning*: This value cannot reside in a register, since the GC might be perturbed.

typ `accu` [*LLM3 operand*]

This operand returns the type of a string.

12.14.3 Access to characters

hbxmov `strg index op` [*LLM3 instruction*]

Moves the `index`th character of the LISP string `strg` into the operand `op`.

hbmovx `op strg index` [*LLM3 instruction*]

Moves the operand `op` into the `index`th character position of the LISP string `strg`.

12.14.4 Creation

There is no special instruction to create strings. Call the LE-LISP function `makestring`.

12.15 Heap zone

hgsize `op1 op2` [*LLM3 instruction*]

Puts the size of the character string or vector pointer object `op1` in operand `op2`.

12.16 Extending the Loader/Assembler

It is possible to extend the syntax of the LLM3 loader/assembler by defining functions in special packages. These extensions are, by definition, machine-specific.

ld-codop [*constant*]

This is the name of the package in which functions associated with extended op codes must reside.

ld-dir [*constant*]

This is the name of the package in which functions associated with extended direct operands must reside.

ld-ind [*constant*]

This is the name of the package in which functions associated with extended indirect operands must reside.

12.17 Functions

12.17.1 Types of functions

Functions written in LLM3 have one of four types:

1. `subr`s, which evaluate their arguments.
2. `fsubr`s, which do not evaluate their arguments.
3. `msubr`s, which do not evaluate their arguments.
4. `dmsubr`s, which do not evaluate their arguments.

12.17.2 Function calling rules

- In the case of a `subr` with 0, 1, 2, or 3 arguments (referred to as a `subr0`, a `subr1`, a `subr2` or a `subr3`), arguments are passed in the registers A1, A2, and A3 respectively.
- In the case of a `subr` with more than three arguments, or with a variable number of arguments (referred to as an `nsubr`), the arguments are all pushed onto the stack and the number of arguments is placed in A4.
- In the case of a `fsubr`, the list of arguments, not evaluated (that is, the `cdr` of the call), is placed in A1.
- In the case of a `msubr`, the list of arguments, not evaluated (that is, the call form itself), is placed in A1.
- In the case of a `dmsubr`, the list of arguments, not evaluated (that is, the `cdr` of the call), is placed in A1.

All functions return a value in A1.

12.18 Examples

This section gives short examples of functions written in LLM3. There is a test file in the test directory named `testlap` which contains all the loader tests. It is also possible to examine the LLM3 code generated by the compilers. (See the next chapter.)

```
? ; These examples were produced on a Vax.
? ;
? ; Manual translation, into lap, of the celebrated fib function:
?
? ; (defun fiblap (n)
? ;     (if (le n 2)
```

```
? ;         1
? ;         (add (fiblap (sub n 1))
? ;              (fiblap (sub n 2)))))
?
? (defvar fiblap '(
?           (fentry fiblap subr1)
?           (cnbgt a1 '2 100)
?           (mov '1 a1)
?           (return)
? 100
?           (diff '1 a1)
?           (push a1)
?           (call fiblap)
?           (mov a1 a2)
?           (pop a1)
?           (push a2)
?           (diff '1 a1)
?           (call fiblap)
?           (pop a2)
?           (plus a2 a1)
?           (return)))
?
? (loader fiblap t)
    (fentry fiblap subr1)     000454e0
    (cnbgt a1 '2 100)         000454e0   b1 52 02 14 00
    (mov '1 a1)               000454e5   d0 01 52
    (return)                  000454e8   05
100                           000454e9
    (diff '1 a1)              000454e9   b7 52
    (push a1)                 000454eb   dd 52
    (call fiblap)             000454ed   10 f1
    (mov a1 a2)               000454ef   d0 52 53
    (pop a1)                  000454f2   d0 8e 52
    (push a2)                 000454f5   dd 53
    (diff '1 a1)              000454f7   b7 52
    (call fiblap)             000454f9   10 e5
    (pop a2)                  000454fb   d0 8e 53
    (plus a2 a1)              000454fe   a0 53 52
    (return)                  00045501   05
    (endl)                    00045502
= ()
? (fiblap 20)
= 6765
?
? ; Manual translation, into lap, of the dlq function:
? ;
? ; (defun dlq (a l)
? ;     (cond ((not (consp l)) ())
? ;           ((eq a (car l)) (dlq a (cdr l)))
? ;           (t (cons (car l) (dlq a (cdr l))))))
?
? (defvar llap1 '(
?        (fentry dlq subr2)
?        (btcons a2 1001)
?        (mov nil a1)
?        (return)
```

```
?     1001
?        (cabne a1 (car a2) 1003)
?        (mov (cdr a2) a2)
?        (bra dlq)
?     1003
?        (push (car a2))
?        (mov (cdr a2) a2)
?        (call dlq)
?        (mov a1 a2)
?        (pop a1)
?        (jmp cons))
= llap1
?
? (loader llap1 t)
    (fentry dlq subr2)        00043e2c
    (btcons a2 1001)          00043e2c   d1 5a 53 15 00
    (mov nil a1)              00043e31   d0 58 52
    (return)                  00043e34   05
1001                          00043e35
    (cabne a1 (car a2) 1003)  00043e35   d1 52 63 12 00
    (mov (cdr a2) a2)         00043e3a   d0 a3 04 53
    (bra dlq)                 00043e3e   11 ec
1003                          00043e40
    (push (car a2))           00043e40   dd 63
    (mov (cdr a2) a2)         00043e42   d0 a3 04 53
    (call dlq)                00043e46   10 e4
    (mov a1 a2)               00043e48   d0 52 53
    (pop a1)                  00043e4b   d0 8e 52
    (jmp cons)                00043e4e   17 d8 c8 3a
    (endl)                    00043e52
= ()
?
? (dlq 'a '(b a c a b))
= (b c b)
?
? ; This example demonstrates Lisp character string manipulation.
? ; The function definition is given below in Le-Lisp:
?
? ; (defun screat (x y)
? ;     (let ((s (makestring 6 #/x)))
? ;          (sset s x #/a)
? ;          (sset s y #/b)
? ;          s))
?
? (defvar llap2 '(
?       (fentry screat subr2)
?       (push a1)
?       (push a2)
?       (mov '6 a1)
?       (mov '#/x a2)
?       (jcall makestring)
?       (hbmovx '#/a a1 (& 1))
?       (hbmovx '#/b a1 (& 0))
?       (adjstk '2)
?       (return)))
= llap2
```

```
?
? (loader llap2 t)
    (fentry screat subr2)     00045e94
    (push a1)                 00045e94   dd 52
    (push a2)                 00045e96   dd 53
    (mov '6 a1)               00045e98   d0 06 52
    (mov '88 a2)              00045e9b   d0 8f 58 00 00 00 53
    (jcall makestring)        00045ea2   16 d8 88 51
    (hbmovx '97 a1 (& 1))     00045ea6   d0 62 51 d0 ae 04 50 90 8f 61 40
                              00045eb1   a1 08
    (hbmovx '98 a1 (& 0))     00045eb3   d0 62 51 d0 6e 50 90 8f 62 40 a1
                              00045ebe   08
    (adjstk '2)               00045ebf   c0 08 5e
    (return)                  00045ec2   05
    (endl)                    00045ec3
= ()
?
? (screat 2 4)
= "xxaxbx"
?
? ; This last example demonstrates vector manipulation.
? ; invector is similar to the vector function, but produces
? ; an inverted list. This example also shows the manipulation
? ; of n-ary functions in llm3.
?
? (defvar llap3 '(
?       (fentry invector nsubr)
?       (push a4)
?       (mov a4 a1)
?       (mov nil a2)
?       (jcall makevector)
?       (pop a4)
?       (mov '0 a3)
?       (bra 1005)
?     1002
?       (pop a2)
?       (hpmovx a2 a1 a3)
?       (plus '1 a3)
?     1005
?       (sobgez a4 1002)
?       (return)))
= llap3
?
? (loader llap3 t)
    (fentry invector nsubr)   00045ec4
    (push a4)                 00045ec4   dd 55
    (mov a4 a1)               00045ec6   d0 55 52
    (mov nil a2)              00045ec9   d0 58 53
    (jcall makevector)        00045ecc   16 d8 e8 53
    (pop a4)                  00045ed0   d0 8e 55
    (mov '0 a3)               00045ed3   d0 00 54
    (bra 1005)                00045ed6   11 00
1002                          00045ed8
    (pop a2)                  00045ed8   d0 8e 53
    (hpmovx a2 a1 a3)         00045edb   d0 62 51 d0 53 44 a1 08
    (plus '1 a3)              00045ee3   b6 54
```

```
1005                            00045ee5
    (sobgez a4 1002)            00045ee5   b7 55 18 ef
    (return)                    00045ee9   05
    (endl)                      00045eea
= ()
?
? (invector 1 2 3 4 5)
= #[5 4 3 2 1]
```

Table of contents

12 Loader/assembler LLM3 **12-1**

12.1 Access to memory and the CPU 12-1
12.2 LLM3 memory loader 12-4
12.3 LLM3 instruction format 12-5
12.4 Modules and labels 12-5
12.5 LLM3 instruction operands 12-6
12.6 Pseudo-instructions 12-8
12.7 Basic instructions 12-9
12.7.1 Moving pointers 12-9
12.7.2 Pointer comparisons 12-9
12.7.3 Control 12-9
12.8 Stack 12-10
12.8.1 Management of the stack pointer 12-10
12.8.2 As a control stack 12-10
12.8.3 As a data stack 12-12
12.9 List cell `cons` operations 12-12
12.9.1 Test for list cell type 12-12
12.9.2 Access to list cell fields 12-13
12.9.3 Creation of list cells 12-13
12.10`nil` 12-13
12.11Symbols 12-13
12.11.1Test for symbol type 12-13
12.11.2Access to the fields of a symbol 12-14
12.11.3Variables 12-14
12.12Numbers 12-14

12.12.1 16-bit integer numbers 12-15
12.12.2 Floating-point numbers 12-17
12.13 Vectors of Lisp pointers 12-19
12.13.1 Test for vector of pointers 12-19
12.13.2 Access to internal fields of a vector of pointers 12-19
12.13.3 Access to elements of a vector of pointers 12-20
12.13.4 Creation 12-20
12.14 Character strings 12-20
12.14.1 Test for character string 12-20
12.14.2 Access to the internal fields of a character string 12-20
12.14.3 Access to characters 12-20
12.14.4 Creation 12-21
12.15 Heap zone 12-21
12.16 Extending the Loader/Assembler 12-21
12.17 Functions 12-21
12.17.1 Types of functions 12-21
12.17.2 Function calling rules 12-22
12.18 Examples 12-22

Function Index

`(#:system:ccode addr)` [*function with an optional argument*] 12-2
`(#:system:ecode)` [*function with no arguments*] 12-2
`(addadr addr1 addr2)` [*function with two arguments*] 12-2
`(subadr addr1 addr2)` [*function with two arguments*] 12-2
`(incradr addr n)` [*function with two arguments*] 12-2
`(gtadr addr1 addr2)` [*function with two arguments*] 12-3
`(loc s)` [*function with one argument*] 12-3
`(vag addr)` [*function with one argument*] 12-3
`(memory addr n)` [*function with one or two arguments*] 12-3
`(call addr a1 a2 a3)` [*function with four arguments*] 12-4
`(calln addr l)` [*function with two arguments*] 12-4
`loader` [*feature*] 12-4
`(loader l i)` [*function with two arguments*] 12-4
`#:ld:special-case-loader` [*variable*] 12-4
`#:ld:shared-strings` [*variable*] 12-5
`a1` [*LLM3 operand*] 12-6
`a2` [*LLM3 operand*] 12-6
`a3` [*LLM3 operand*] 12-6
`a4` [*LLM3 operand*] 12-6
`nil` [*LLM3 operand*] 12-6
`quote exp` [*LLM3 operand*] 12-6
`car accu` [*LLM3 operand*] 12-6
`cdr accu` [*LLM3 operand*] 12-6
`cval accu` [*LLM3 operand*] 12-6
`plist accu` [*LLM3 operand*] 12-6
`fval accu` [*LLM3 operand*] 12-6

`pkgc accu` [*LLM3 operand*] 12-7
`oval accu` [*LLM3 operand*] 12-7
`alink accu` [*LLM3 operand*] 12-7
`pname accu` [*LLM3 operand*] 12-7
`cvalq symb` [*LLM3 operand*] 12-7
`fvalq symb` [*LLM3 operand*] 12-7
`typ accu` [*LLM3 operand*] 12-7
`val accu` [*LLM3 operand*] 12-7
`& n` [*LLM3 operand*] 12-7
`lab` [*LLM3 operand*] 12-7
`@ lab` [*LLM3 operand*] 12-7
`eval expr` [*LLM3 operand*] 12-8
`title symb` [*LLM3 pseudo-instruction*] 12-8
`local symb` [*LLM3 pseudo-instruction*] 12-8
`fentry symb ftype` [*LLM3 pseudo-instruction*] 12-8
`entry symb ftype` [*LLM3 pseudo-instruction*] 12-8
`endl` [*LLM3 pseudo-instruction*] 12-8
`end` [*LLM3 pseudo-instruction*] 12-8
`eval e` [*LLM3 pseudo-instruction*] 12-8
`mov op1 op2` [*LLM3 instruction*] 12-9
`cabeq op1 op2 lab` [*LLM3 instruction*] 12-9
`cabne op1 op2 lab` [*LLM3 instruction*] 12-9
`bra lab` [*LLM3 instruction*] 12-9
`jmp symb` [*LLM3 instruction*] 12-9
`bri op` [*LLM3 instruction*] 12-9
`brx llab op` [*LLM3 instruction*] 12-9
`sobgez op lab` [*LLM3 instruction*] 12-10
`nop` [*LLM3 instruction*] 12-10
`stack op` [*LLM3 instruction*] 12-10
`sstack op` [*LLM3 instruction*] 12-10
`call lab` [*LLM3 instruction*] 12-10
`calli op` [*LLM3 instruction*] 12-10
`jcall symb` [*LLM3 instruction*] 12-11
`return` [*LLM3 instruction*] 12-12

`push op` [*LLM3 instruction*] 12-12
`pop op` [*LLM3 instruction*] 12-12
`adjstk op` [*LLM3 instruction*] 12-12
`movxsp op1 op2` [*LLM3 instruction*] 12-12
`xspmov op1 op2` [*LLM3 instruction*] 12-12
`btcons op lab` [*LLM3 instruction*] 12-12
`bfcons op lab` [*LLM3 instruction*] 12-12
`car accu` [*LLM3 operand*] 12-13
`cdr accu` [*LLM3 operand*] 12-13
`btnil op lab` [*LLM3 instruction*] 12-13
`bfnil op lab` [*LLM3 instruction*] 12-13
`btsymb op lab` [*LLM3 instruction*] 12-13
`bfsymb op lab` [*LLM3 instruction*] 12-13
`cval accu` [*LLM3 operand*] 12-14
`cvalq symb` [*LLM3 operand*] 12-14
`plist accu` [*LLM3 operand*] 12-14
`fval accu` [*LLM3 operand*] 12-14
`fvalq symb` [*LLM3 operand*] 12-14
`oval accu` [*LLM3 operand*] 12-14
`alink accu` [*LLM3 operand*] 12-14
`pkgc accu` [*LLM3 operand*] 12-14
`pname accu` [*LLM3 operand*] 12-14
`btvar op lab` [*LLM3 instruction*] 12-14
`bfvar op lab` [*LLM3 instruction*] 12-14
`btfix op lab` [*LLM3 instruction*] 12-15
`bffix op lab` [*LLM3 instruction*] 12-15
`incr op` [*LLM3 instruction*] 12-15
`decr op` [*LLM3 instruction*] 12-15
`plus op1 op2` [*LLM3 instruction*] 12-15
`diff op1 op2` [*LLM3 instruction*] 12-15
`negate op` [*LLM3 instruction*] 12-15
`times op1 op2` [*LLM3 instruction*] 12-15
`quo op1 op2` [*LLM3 instruction*] 12-15
`rem op1 op2` [*LLM3 instruction*] 12-16

`cnbeq op1 op2 lab` [*LLM3 instruction*] 12-16
`cnbne op1 op2 lab` [*LLM3 instruction*] 12-16
`cnble op1 op2 lab` [*LLM3 instruction*] 12-16
`cnblt op1 op2 lab` [*LLM3 instruction*] 12-16
`cnbge op1 op2 lab` [*LLM3 instruction*] 12-16
`cnbgt op1 op2 lab` [*LLM3 instruction*] 12-16
`land op1 op2` [*LLM3 instruction*] 12-16
`lor op1 op2` [*LLM3 instruction*] 12-17
`lxor op1 op2` [*LLM3 instruction*] 12-17
`lshift op1 op2` [*LLM3 instruction*] 12-17
`btfloat op lab` [*LLM3 instruction*] 12-17
`bffloat op lab` [*LLM3 instruction*] 12-17
`fplus op1 op2` [*LLM3 instruction*] 12-17
`fdiff op1 op2` [*LLM3 instruction*] 12-17
`ftimes op1 op2` [*LLM3 instruction*] 12-17
`fquo op1 op2` [*LLM3 instruction*] 12-17
`cfbeq op1 op2 lab` [*LLM3 instruction*] 12-18
`cfbne op1 op2 lab` [*LLM3 instruction*] 12-18
`cfblt op1 op2 lab` [*LLM3 instruction*] 12-19
`cfble op1 op2 lab` [*LLM3 instruction*] 12-19
`cfbgt op1 op2 lab` [*LLM3 instruction*] 12-19
`cfbge op1 op2 lab` [*LLM3 instruction*] 12-19
`btvect op lab` [*LLM3 instruction*] 12-19
`bfvect op lab` [*LLM3 instruction*] 12-19
`val accu` [*LLM3 operand*] 12-19
`typ accu` [*LLM3 operand*] 12-19
`hpxmov vect n op` [*LLM3 instruction*] 12-20
`hpmovx op vect n` [*LLM3 instruction*] 12-20
`btstrg op lab` [*LLM3 instruction*] 12-20
`bfstrg op lab` [*LLM3 instruction*] 12-20
`val accu` [*LLM3 operand*] 12-20
`typ accu` [*LLM3 operand*] 12-20
`hbxmov strg index op` [*LLM3 instruction*] 12-20
`hbmovx op strg index` [*LLM3 instruction*] 12-21

`hgsize op1 op2` [*LLM3 instruction*] 12-21
`ld-codop` [*constant*] 12-21
`ld-dir` [*constant*] 12-21
`ld-ind` [*constant*] 12-21

Chapter 13

Compilation

This chapter describes the two compilers available in LE-LISP version 15.26: the standard compiler and the modular compiler called COMPLICE. They perform the same function, so there is no point in having both of them in your LISP system.

- The `standard compiler` is totally compatible with the interpreter. In particular , it consistently builds the same dynamic and lexical activation blocks as the interpreter. Its only limitations are its inability to process self-modifying programs, its failure to raise the `errudv` error in the case of undefined variables, and the fact that it does not fully implement the dynamic aspect of the `flet` form. It is small, it has fair performance (between three and six times faster than the interpreter), and it permits function definitions to be moved from the list zone into the code zone. This can improve the performance of garbage collection and, depending on the host architecture, can also achieve space savings.
- The `modular compiler`, COMPLICE, produces much better code than the standard compiler. It also supports separate compilation of modules. The code it produces is between two and five times faster than the equivalent standard compiled code (and therefore six to thirty times faster than interpreted code), but certain precautions have to be taken. These are described below. COMPLICE uses a lexical scheme to bind function variables.

The two compilers use the same memory loader, described in the previous chapter.

13.0.1 Calling the compilers

`compiler` [*feature*]

This feature indicates whether one of the compilers is loaded into memory.

`complice` [*feature*]

This feature indicates whether the COMPLICE compiler is loaded into memory.

`(compiler source status print loader)` [*function with four arguments*]

Compiles the function or list of functions referred to as `source`. If `status` is not a list, then all the functions inside functions in `source` are compiled. In calls to the standard compiler, a value of `()` for `status` indicates that none of the functions called by functions in `source` should be compiled. The `print` argument is a flag that indicates, when set, that the list of LLM3 instructions generated by the compiler should be printed on the current output stream. `loader` is a flag that is passed to the loader. `status` indicates the status of functions used inside `source`. If `status` is a list, it is taken to be the list of functions that should *not* be compiled during the compilation of the functions in `source`.

This is not possible using the COMPLICE compiler. If one the functions in `status` is a `subr`, the COMPLICE compiler generates a call to the evaluator when this `subr` function is called. The standard compiler ignores the occurrence of `subr` functions in `status`.

`(compile source status print loader)` [*special form with one, two, three or four arguments*]

This is the `fsubr` form of the preceding function. So, none of its arguments is evaluated. The `status`, `print` and `loader` arguments are optional, with default values of `()`.

`(compile-all-in-core print loader)` [*function with zero, one or two optional arguments*]

Compiles all the functions in the `oblist` of type `subr`, `fsurb`, `macro` or `dmacro`. The `print` and `loader` flags perform the same functions here as they do in the `compiler` function.

In LE-LISP, `compile-all-in-core` could be defined in the following manner:

```
(defun compile-all-in-core flags
    (let ((print-flag (car flags)) (loader-flag (cadr flags)))
         ;First deal with subrs/fsubrs (if the macros use them)
         (compiler (maploblist (lambda (x)
                     (and (null (getprop x 'dont-compile))
                          (memq (or (car (getprop x 'resetfn))
                                    (typefn x))
                                '(subr fsubr)))))
                   t
                   print-flag
                   loader-flag)
         ; then all the macros
         (compiler (maploblist (lambda (x)
                     (and (null (getprop x 'dont-compile))
                     (memq (or (car getprop x 'resetfn))
                           (typefn x))
                             '(macro dmacro)))))
                   t
                   print-flag
                   loader-flag)))
```

(compilefiles source object) [*function with two arguments*]

Compiles all the functions in the file or list of files `source` and stores the result of the compilation in the file named `object`. The extension `#:system:lelisp-extension` is automatically added to the filenames in source that do not already have it. The output file object can be loaded into LISP by using the classic `load`, `loadfile` and `^L` functions.

The `object` argument is optiomal in calls to the COMPLICE modular compiler. By default, it is set to the first element in the list source. In addition, COMPLICE adds the `#:system:obj-extension` extension to the `object` filename, if it does not already have it. In any case, the file should then be loaded with the `loadobjectfile` function.

(precompile exp1 result exp2 operand) [*special form*]

When interpreted, `precompile` returns the value of the evaluation of `exp1` and ignores the other three arguments. The compilers assume that result is the result of the compilation of the expression `exp1`, and they load the LLM3 instructions that it contains. After loading `result`, the compiler executes the expression `exp2` in the compiler environment. The `operand` argument is the LLM3 operand indicating the place where the result can be found. Its default value is `A1`. This function allows precompilers to match the interface that the current compilers present.

(dont-compile exp_1 ... exp_n) [*special form*]

Informs the compiler that the functions exp_1 ... exp_n should never be compiled. To force one of them to be compiled, use the function compiler explicitly.

13.0.2 Compiler macros

Before compilation, LISP forms are subjected to macro-expansion, which recognizes not only user macros (functions of `macro`, `dmacro`, `msubr` or `dmsubr` type), but also two new compiler-specific kinds of macro functions : *open macros* and *closed macros*.

Some system functions have open or closed macro definitions. Functions that have closed macro definitions must not be redefined. Compiler macro definitions are included in the `cpmac` file in the standard library.

13.0.3 Closed macros

Closed macros are always expanded during the compilation phase. In the case of certain functions that are implemented in the interpreter in a `subr` or `fsubr` form, this expansion gives the compiler an opportunity of generating highly efficient code. The `mapc` function, for instance, which generatyes a simple loop, is a `subrn` in the interpreter and a closed macro in the compiler.

Here is a list of the standard closed macros of the compiler:

```
add1        any         atomp      catcherror  cond
decr        defprop     desetq     err         errset
eval-when   every       ifn        incr        letvq
```

```
lognot      loop        map         mapc        mapcan
mapcar      mapcon      mapl        maplist     mapvector
neq         nequal      newl        newr        nextl
null        prog2       psetq       return      setqq
sub1         time       unless    untilexit     when
```

13.0.4 Open macros

For open macros to be expanded during the compilation phase, the `#:compiler:openp` flag must be set. In the standard implementation, the `car` and `cdr` primitives are open compiler macros. The user can use open definitions for any function at all, except those with closed macro definitions. The user who makes such a replacement must verify that the open macro definition is consistent with the regular function definition, so that compatibility between compiled and interpreted code is maintained.

`#:compiler:open-p` [*variable*]

This variable indicates whether open compiler macros should be expanded. When a compiler is loaded, this variable is set by default to `t`. This flag also specifies whether access functions to LE-LISP object fields and the low-level primitives shoud be compiled without tests for argument types. This a less secure but more efficient executable image.

The following functions are affected by this feature :

```
add         car         cdr         comment     div
fadd        fdiv        fmul        fsub        logand
logor       logshift    logxor      mul         objval
packagecell plist       rem         rplaca      rpacd
set         slen        sref        sset        sub
symeval     vlength     vref        vset
```

The generated code is obviously denser and faster, but does not perform tests that the interpreter does. For instance, the `car` function no longer verifies that its argument is in fact a list. Machine errors can therefore be brought about by incorrect programs.

```
? (defun test1 (l)
?     (cons (car l) (cdr l)))
= test1
? (defun test2 (l)
?     (cons (car l) (cdr l)))
= test2
? (defvar #:compiler:open-p t)
= #:compiler:open-p
```

Here is the direct LLM3 expansion for `car cdr` field access:

```
? (compile test1 t t)
```

```
          fentry   test1,subr1
          entry    test1, subr1
          mov      cdr(a1),a2
          mov      car(a1),a1
          jmp      cons
 = (test1)
```

Here are calls to `car` and `cdr` functions :

```
? (defvar #:compiler:open-p ())
= #:compiler:open-p
? (compile test2 t t)
          fentry    test2,subr1
          entry     test2, subr1
          push      a1
          jcall     car
          push      a1
          mov       &1, a1
          jcall     cdr
          mov       a1,a2
          pop       a1
          adjstk    '1
          jmp       cons
= (test2)
```

(defmacro-open name ... fval) [*special form*]

Gives the `name` function an open macro definition. The arguments to this function follow exactly the same description as for `defmacro`.

(make-macro-open name fval) [*function with two arguments*]

This is the `subr` version fo `defmacro-open`. It is particularly useful for generating open macro from programs.

(macro-openp name) [*function with one argument*]

Returns a function that has the open macro definition associated with the `name` function. If `name` does not have an open macro definition, this function returns `()`. The open macro associated with a function can be expanded in the following manner:

```
? (defun expand-open (call)
?     (let ((open-macro (macro-openp (car call))))
?          (when open-macro
?               (apply open-macro)))))
= expand-open
? (defun second (v)
?     (vref v 2))
= second
? (defmacro-open second (v)
```

```
?    `(vref ,v 2))
= second
? (expand-open '(second #[1 2 3]))
= (vref #[1 2 3] 2)
```

(remove-macro-open name) *[function with one argument]*

Removes the open macro definition from the `name` function.

13.0.5 Modules

From this point on, all we have to say concerns COMPLICE exclusively, and not the standard compiler.

COMPLICE is an optimizing compiler that enables large and complex LISP programs to be compiled in a fragmented fashion described as *modular*.

The set of all the files involved in a compilation is referred to as a `module`. The minimum number of files in a module is three. They can be described as follows :

- A `source file` (by default, with the extension `.ll`) contains definitions of LISP functions and methods.
- A `description file` (with the extension `.lm`) describes the contents of the module.
- An `object file` (with the extension `.lo`) is generated by COMPLICE using the two files that were just mentioned.

A typical module contains no more than these three files, but there may be cases in which a module incorporates more than a single source file.

Description file

Let us create module called `mymodule`. When we say that the name of this future module is `mymodule`, this means that the description file of this module will be called `mymodule.lm`. In other words, the name of a module can be defined as the name of its description without the `.lm` extension.

The first item of information in the `mymodule.lm` file starts with the keyword `defmodule`, followed by a symbol that we can choose. You might think of this `defmodule` statement as a declaration of the existence of the module that we are about to create. This first line of the `mymodule.lm` file might read as follows :

```
defmodule mypackage
```

Here, we have chosen the word `mypackage` because this information relates, as we shall see in a moment, to the general theme, in LE-LISP, of *packages*.

We now indicate the name of the source file containing the LISP code that we wish to compile. This is done using the keyword `files`. Suppose that our file is called `mysource`. In that case, the second line of the file `mymodule.lm` would be

```
files (mysource)
```

Suppose that there are functions in `mysource` that need some of functions contained in an existing module, which we call `anothermodule`. In other words, there is another module description file in the system, called `anothermodule.lm` which contains compiled LISP code that we need within `mymodule`. We indicate this fact by using the keyword `import`, followed by the name of this outside module, enclosed in parentheses. So, our `mymodule.lm` description file now reads as follows :

```
defmodule mypackage
files (mysource)
import (anothermodule)
```

In the modular context of compilation this use of the `import` keyword brings us to the all-important concepts of *import* and *export*, which can be thought of as complementary. When we state, as we have just done, that `mymodule` needs to import `anothermodule`, we are implying that, within this 'foreign' module called `anothermodule`, there are various functions that we can use in `mymodule`. In other words, `mymodule` must be able to refer to these functions that are located within the imported module called `anothermodule`. Now, if it is true that there are functions within `anothermodule` that can be referred to in our own module, these functions are described, from the viewpoint of the 'foreign' module called anothermodule, as exported. This means that the creators of `anothermodule` said to themselves, as it were, when they were developing this module : 'Maybe other people might like to use our function called `alpha`, say. So let's export it'.

Just as the people who created `anothermodule` went to the trouble of making some of their functions exportable, we too should think about this question in the case of our construction of `mymodule`. To indicate the various functions in `mymodule` that we are prepared to make available to further modules, we use the keyword `export`. Suppose that our functions `beta` and `gamma` fall into this category. In that case, we would add a fourth line to the `mymodule.lm` description file, which would now read as follows :

```
desmodule mypackage
files (mysource)
import (anothermodule)
export (beta : gamma)
```

We shall explain in a moment why there is a colon before the name of the `gamma` function.

Let us step back to the previous line, with `import` keyword. We said a moment ago that the creators of `anothermodule` exported their alpha function, which we now intend to use in `mymodule`. This means, then, that the file `anothermodule.lm` must contain the following line:

```
export (alpha)
```

To summarize, then : The creators of a module *export* functions that might be required in other modules. Inversely, when you *import* a 'foreign' module into the one that you are creating, you suppose that the creators of this outside module have *exported* the functions that you need for your own module.

Natually, within our file `mysource`, at the base fo `mymodule`, there are no doubt many other functions, besides the exported ones called `beta` and `gamma`, that we do not wish to make available to outside modules. We say that these functions are *local* to our own module.

Keys and values

In the terminology commonly used in the domain of modules, the various keywords `defmodule`, `files`, `import` and `export` are referred to as `keys`, and the indications that accompany them are referred to as the corresponding `values` of these keys. So the general syntactic structure of a module description file could be presented in the following manner:

```
key1      value1
...       ...
...       ...
...       ...
keyn      valuen
```

The four lines of the `mymodule.lm` description that we have just written might be thought of merely as a 'preface', in the sense that COMPLICE, during the compilation of `mymodule`, adds further key/value pairs to the file. But the user does not have to be concerned with theses additonal keys and values. In fact COMPLICE inserts a line just after your preface, warning you not to touch anything beyond that point.

As for the preface, written by the user, it can be in any layout whatsoever, with comments preceded by a semi-colon sign as in LISP. The `files`, `import` and `export` keys can be repeated, so long as the attached value always appears in the form of a list... which could be empty in certain cases. If keys are repeated, only the last instance is used. COMPLICE merely reads this preface, but does not modify it in any way whatsoever.

In the first line that we wrote for the `mymodule.lm` description file, we used the word `mypackage`:

```
defmodule mypackage
```

What this really means is that is a package called `#:mypackage`, and that the exported function referred to as `:gamma`, with a colon, is to be found in this `#:mypackage` package. It is as if we had written the `mymodule.lm` file in the following manner:

```
defmodule anything
files (mysource)
import (anothermodule)
export (beta #:mypackage:gamma)
```

In other words, this title following the keyword `defmodule` is a convenient means of avoiding to have to include these explicit package references in the names of exported files.

In actual fact, the solution adopted by most programmers would consist of using one single symbol, namely `mymodule`, and writing the `mymodule.lm` file in the following manner:

```
defmodule mymodule
```

```
files (mymodule)
import (anothermodule)
export (Beta :gamma)
```

Here, the symbol `mymodule` is used for the name of the module, the name of the package, and even the name of the source file.

The use of the keys and their values in a module description file can be summarized as follows:

- The first key is `defmodule`. The associated value must be a LISP symbol. The sole purpose of this name is to be used as an input package for the remainder of the description file.
- The next key, `files`, is followed by a list indicating the set of LISP source files containing the functions of the module. Both COMPLICE and the `loadmodule` function (described later on) make use of this `files` key and its value.
- The next key, `import`, is followed by a list containing the names of other modules that contain compiled LISP code required for the execution of the present module. The exported functions contained in an imported module are used at execution time by the functions contained in the present module. Both COMPLICE and the `loadmodule` function (described later on) make use of this `import` key and its value.
- The next key is `export`. The associated list indicates the exported functions in the present module. Above all, the names of these exported functions will be known outside the present module, so that other modules could import the present module and call upon these exported functions.

  Structures can be used to define several functions serving as 'lexical sugar' enabling you to avoid having to explicitly indicate the name of each function to be exported. To use this facility, use the special `(structure <structure-name>)` notation in the exported function list. This form concerns structures defined with `defstruct`, `defrecord` and `deftclass`. In the latter case, don't forget to make the name of the structure internal inside the `tclass` package. Only COMPLICE makes use of this `export` key and its value. Here's an example of source code in a file named `mymodule.ll`:

  ```
  (defun myfunct (x) x)
  (defstruct mystruct a b)
  (defrecord myrecord cd)
  (deftclass mytclass e f)
  ```

  Here is the module description file named `mymodule.lm`:

  ```
  defmodule mymodule
  files (numodule)
  export (myfunct
          (structure mystruct)
          (structure myrecord)
          (structure #:tclass:mytclass))
  ```

- There is also an `include` key, which exists solely for compatibility with older versions of COMPLICE. It can now be replaced by the use of the `eval-when` function in the source file. This function is described later on in this chapter.
- COMPLICE inserts into a module description file, for its own needs, various *internal keys* that do not normally concern the programmer. In particular, there is a `cpenv` key that describes the exact compilation environnement that would be transmitted to any other module that happened to import the present one. This key and its associated value would play a role if the creator of the present module were to use the above-mentioned `eval-when` function to modify the compilation environment.

  As we mentioned earlier on, when a module is compiled, only these internal keys are modified by COMPLICE. The so-called 'preface' of the description file, concerning the `defmodule`, `files`, `import` and `export` keys, emains unchanged by the compilation.

Let's look at an example. Consider a file `f1.ll` containing

```
(defun example1 (1) (tmp1 l))
(defun tmp1 (l) (cons (car l) (example1  l)))
```

and a file `f2.ll` containing

```
(defun example2 (l) (tmp2 l))
(defun tmp2 (l) (cons (car l) (example1 l)))
```

The module description file for `f1.ll` is the file `m1.lm` containing

```
defmodule user
files (fl)
export (example1)
```

and the module description file for `f2.ll` si the file `m2.lm` containing

```
defmodule user
files (f2)
export (Example2)
import (m1)
```

Let us look at an example of the use of modules in interpreted form, and the compilation of modules. We start out by loading the interpreted module `m2`:

```
? (loadmodule 'm2 t)
** loadmodule : load interpreted module : m1
** loadmodule : load interpreted module : m2
= m2
? (pretty example1)
(defun example1 (l)
 (tmp1 l))
= ()
? (compilemodule 'm2 t)
= ()
? #:module:interpreted-list
= (m1 m2)
```

Here is an example of the use of a compiled module:

```
? (loadmodule 'm2 t)
= m2
? (pretty example1)
(ds example1 subr1 (19 . 2152))
= ()
? (pretty tmp2)
()
= ()
? (example2 '(1 2 3))
= (1 1 2 3)
```

13.0.6 Source files

The main components of the module under construction come from one or several files. The name of a source file has, by default, the extension `.ll`. A source file must contain all the data that is necessary for the successful construction of the desired module. Here are several points that should be kept in mind:

- Naturally, the source file must contain the definitions of all the functions, macros and other `defstruct` data structures that are to be *exported*. These entities will be made available to users of the present module, and to any other module that imports the present module. As we pointed out earlier on, the names of all these entities must appear in the list that follows the `export` keyword in the description file of the present module.
- The source file also includes other entities, described as *local*, which are used by the exported entities, but without being made available to the outside world. These internal entities can usually be compiled more efficiently than for exported entities, since their calling protocols are relatively simplified.
- Other important elements in a source file are related to the possibilities of an interesting function called `eval-when`, which we examine in the next section, under the general theme of evaluation. This function enables you to control the compilation environment.

COMPLICE generates compiled code for all the functions defined in the files listed under the `files` key, and only for these files. Functions defined in the `import` or `include` files never give rise to code in the compiled module.

The code produced by COMPLICE contains compiled definitions of the exported functions of the module. In other words, the compiled code is associated with the LISP symbol that names each function. The other entities appearing in the source files mentioned after the `files` keyword give rise to code in the compiled module, but no symbols are associated with them.

When you load a compiled module by means of the `loadmodule` function, the imported modules are loaded first of all (except, of course, if they have already been loaded), then the compiled code of the module is loaded into the zone reserved for compiled code in the LE-LISP system. The `include` files are never loaded by the `loadmodule` function. If you want these files to be present in the extension environment of the module, which is not generally the case, you have to load them explicitly.

During compilation, COMPLICE does not automatically import modules imported by imported modules. The `import` key is not transitive during the compilation. In the same way, the compiler never loads the `include` files of imported modules. If you want to use the `include` or `import` files of imported modules, you have to list them explicitly under the corresponding keys of the module, Note that `include` files are loaded in the order in which they are listed. This might be important when these files contain successive the definitions of abbreviations or LISP structures.

13.0.7 Object files

The COMPLICE compiler, which is used to compile modules, builds object files, which contain LLM3 code in `lap` form. These files always have the value of `#:system:obj-extension` as their extension.

`#:system:obj-extension` [*variable*]

This variable contains the default extension of LE-LISP object files.

`(probepatho file)` [*function with one argument*]

Returns the complete filename of the object file `file` if it is found in the `#:system:path` directory list. If the module is not found there `probepatho` returns `()`.

In LE-LISP, `probepatho` could be defined in the following manner:

```
(defun probepatho (mod)
    (search-in-path
        #:system:path
        (catenate mod #:system:obj-extension)))
```

```
? (probepatho 'pretty)
= /nfs/current/lelisp/llobj/pretty.lo
```

`(loadobjectfile file)` [*function with one argument*]

Loads the object file `file` if it is found in the `#:system:path` directory list. In LE-LISP, `loadobjectfile` could be defined in the following manner:

```
(defun loadobjectfile (file)
    (let ((real-file (probepatho file)))
         (ifn real-file
              (error 'loadobjectfile "unknown file" file)
              (loadfile real-file t))))
```

13.0.8 Controlling evaluation

Using modules means that you have to be able to precisely control the moment at which expressions within a module are evaluated. When a module is compiled, the default behavior is that all the

forms in the module are evaluated before the actual moment of compilation. The following special form lets you specify, for each form in a module, whether it should be evaluted when the module is loaded in interpreted mode, when the module is loaded in compiled mode, or during the compilation process.

(eval-when moments . expressions) [*function with a variable number of arguments*]

The `moments` argument is a non-evaluated list of symbols from the set of `eval`, `load`, `local-compile` or `compile`. The `expressions` argument is a sequence of arbitrary legal expressions that be evaluated according to the presence of one of the symbols of the `moments` argument. This evaluation takes place according to the following criteria:

- `eval`: The expressions are evaluated when the source files of the module are loaded with the `libloadfile` function, or from the main interpreter interactive loop.
- `load`: The expressions are evaluated when the module is loaded with the `loadobjectfile` function.
- `local-compile`: The expressions are evaluated when the module is compiled, but they will not be exported.
- `compile`: The expressions are evaluated when the module is compiled, or when any other module that imports this module is compiled.

By default, all the expressions in a module are surrounded by the form

```
(eval-when (eval load local-compile)...)
```

Having seen this description of the `eval-when` function, we can now return to the question (that we started to examine earlier on) of the possible contents of a source file:

- A source file might contain certain definitions that are exploited solely during the compilation of the present module. This possibility is made available by the `eval-when` function with the `local-compile` keyword.
- Other definitions in the source file might be used both when compiling the present module and when compiling another module that imports the present one. This is the case of the `eval-when` function with the `compile` keyword.
- There might be definitions that are useful both at execution time and in the case of a local compilation. This involves using `eval-when` with the `local-compile` and `load` keywords, together with export.
- Finally, there might be definitions that are useful both at execution time and in the case of local and external compilations. This involves using `eval-when` with the `compile` and `load` keywords, together with `export`. The `eval` keyword would be added if you wanted to use such a definition in interpreted mode. In fact, any of the combinations of the keywords `compile`, `local-compile`, `load` and `eval` can be envisaged.

Let us look at examples of the use of evaluation control. The source file called `when` contains the following forms:

```
(eval-when (eval)
          (print "(eval)"))

(eval-when (load)
          (print "(load)"))

(eval-when (local-compile)
          (print "(local-compile)"))

(eval-when (compile)
          (print "(compile)"))

(eval-when (eval load local-compile compile)
          (print "(eval load local-compile compile)"))

(defun first ()
          'first)
```

This has the following module definition:

```
defmodule when
          files (when)
          export (first)
```

It behaves in the following manner:

```
? ^Lwhen
(eval)
(eval load local-compile compile)
= when.ll
? (compilemodule 'when)
(local-compile)
(compile)
(eval load local-compile compile)
(eval load local-compile compile)
(eval load local-compile compile)
= ()
? ^Awhen
(load)
(eval load local-compile compile)
= when
```

The source file named `when2` contains

```
(eval-when (eval)
          (print "**2**(eval)"))

(eval-when (load)
          (print "**2**(load)"))
```

```
(eval-when (local-compile)
          (print "**2**(compile)"))

(eval-when (compile)
          (print "**2**(compile)"))

(eval-when (eval load local-compile compile)
          (pirnt "**2**(eval load local-compile compile)"))

(defun second ()
          'second)
```

It has the following module definition:

```
defmodule (when2)
          files (when2)
          export (second)
          import (when)
```

It behaves in the following manner:

```
?^Lwhen2
**2**(eval)
**2**(eval load local-compile compile)
= when2.ll
? (compilemodule 'when2)
(compile)
(eval load local-compile compile)
**2**(local-compile)
**2**(compile)
**2**(eval load local-compile compile)
**2**(eval load local-compile compile)
**2**(eval load local-compile compile)
=()
? ^Pfirst
()
=()
?^Awhen2
(eval)
(eval load local-compile compile)
**2**(eval)
**2**(eval load local-compile compile)
= when2
? ^Pfirst
(ds first subr0 (8 . 21196))
= ()
```

13.1 Use and manipulation of modules

This section presents seven functions and two variables that are used by COMPLICE.

13.1.1 Functions on modules

(probepathm module) *[function with one argument]*

Returns the complete filename of the module description file of `module`, if it is found in the `#:system:path` directory tree. If the module is not found there, `probepathm` returns `()`.

In LE-LISP, `probepathm`, could be defined in the following manner:

```
(defun probepathm (mod)
    (search-in-path
        #:system:path
        (catenate mod #:system:mod-extension)))
```

```
? (probepathm 'pretty)
= /nfs/current/lelisp/llmod/pretty.lm
```

(readdefmodule module) *[function with one argument]*

Returns a LISP structure containing the information in the module description file of `module`. This struture will be referred to, from now on, as a *module definition*, and will be denoted by the argument `defmod`.

(getdefmodule defmod key) *[function with two arguments]*

Returns the value associated with `key` in the module definition `defmod`.

(setdemodule defmod key value) *[function with three arguments]*

Adds the pair `value/key` to the module definition `defmod`. If the key already exists in the definition, its existing value is replaced by the `value` argument.

(printdefmodule defmod module) *[function with two arguments]*

Updates the module description file that corresponds to the module `module`, using the module definition `defmod`.

13.1.2 Loading and compiling modules

(loadmodule module reload-p interpreted-p) *[function with one, two or three arguments]*

Loads `module` and all the modules that it imports. Imported modules are loaded in the first instance, using a single `loadmodule` function. Loading is recursive. Consequently, modules imported by imported modules are likewise imported, and so on. Only modules that have not already been

loaded into the system are loaded. Nevertheless, the `reload-p` argument provides a means of forcing all the imported modules to be loaded or reloaded.

If the module was compiled, `loadmodule` loads it using the `loadobjectfile` function. If the module was not compiled, `loadmodule` loads the uncompiled source files defined in the `files` key using the `libloadfile` function. The `interpreted-p` flag, when set, forces all the source files of a module to be loaded regardless of whether compiled versions of the module are already loaded.

When a module is loaded from source files, a warning with the following standard screen display is printed:

```
;; load interpreted module : 'module'
```

Here, `module` is the name of the module loaded and interpreted.

The following two variables are maintained by the `loadmodule` function:

`#:module:interpreted-list` [*variable*]

This variable contains the set of names of modules whose files have been loaded by the `libloadfile` function.

`#:module:compiled-list` [*variable*]

This variable contains the set of names of modules whose files have been loaded by the `loadobjectfile` function.

`(compilemodule module ind)` [*function with one or two arguments*]

Compiles `module` with the COMPLICE modular compiler. The result of the compilation is stored in the file named `module` with the extension `#:system:obj-extension`, which is created if it does not already exist. If the `ind` flag is set, the modules imported by `module` are recursively compiled using the same `compilemodule` function.

Files produced by the compilation can be loaded using the `loadobjectfile` function. The compiled module and the modules it imports can be loaded all at once using the `loadmodule` function.

When the flag is not supplied, the definition of `compilemodule` is almost equivalent to the following code:

```
(defun compilemodule (mod)
    (compilefiles
        (getdefmodule (readefmodule mod) 'files)
        mod))
```

The module description is updated, though, to reflect the fact that the module was compiled.

Moreover, functions not exported by the module will no longer exist after the compilation. Write access to the module description file is needed in order to carry out a module compilation.

13.2 Complice

To compile a module, you use the COMPLICE compiler. This section describes the features specific to COMPLICE, as well as its error and warning messages.

13.2.1 Compatibility messages

Whenever possible, COMPLICE compiles variable bindings as lexical bindings. This means that variables do not exist, properly speaking, outside the lexical block in which they are declared. Programs normally executed in intrepreted mode might therefore be rendered incompatible or unworkable after compilation by COMPLICE. In order to alleviate these problems, COMPLICE has a flag called `#:complice:parano-flag`. When this flag is set, explicit calls to the evaluator result in dynamic binding of the variables involved, so that compiled and interpreted versions have the same behavior.

`#:complice:parano-flag` *[variable]*

This variable indicates whether complete compatibility with the interpreter is required. When COMPLICE is first loaded, the value of this flag is `t`, meaning that total compatibility is required. Let us look at an example:

```
? (defun test1 (word) (funcall 'test2))
= test1
? (defun test2 () word)
= test2

(test1 'local)  ⟹   local
```

Now compile `test1` by entering `(compile test1)`. Since the `#:complice:parano-flag` is set to `t`, the `test1` function still works normally:

```
(test1 'local)  ⟹   local
```

Let us now redefine the same `test1` function. Before recompiling it, we shall set the flag to `()` with `(defvar #:complice:parano-flag ())`. Then we recompile the function.

```
(defun test1 (word) (funcall 'test2))
= test1
? (defvar #:complice:parano-flag ())
= #:complice:parano-flag
? (compile test1 t t)
          fentry   test1, subr1
          entry    test1, subr1
          push     @101
          push     'test2
          mov      '1,a4
          jmp      funcall
101       eval     ()
          return
= (test1)
```

This time, the evaluation produces a different result:

```
? (let ((word 'global))
?    (test1 'local))
= global
```

Here is a list of functions that modifity the behavior of the compiler according to the setting of the `#:complice:parano-flag` variable:

```
apply         deset      eval       evlis      flambda
flet          funcall    lambda     lock       set
symeval       time       unexit     unwind
```

13.2.2 Errors and warnings

COMPLICE errors and warning are printed onto the current output stream in one of two formats:

- For an error, the format is

  `E.`*n*`.`*entity*`..`*message*`:`
  argument

- For a warning, the format is

  `W.`*n*`.`*entity*`..`*message*`:`
  argument

The items in italics have the following meanings:

- *n* is the number of the error or warning.
- *entity* is the entity (such as a function or a variable) that was being compiled.
- *message* is a complete explanatory message.
- *argument* is the faulty argument.

When an error is raised, *entity* is not compiled. If a warning occurs, the code produced might be incompatible with the interpreter. See, however, the `#:complice:parano-flag` flag.

`#:complice:warning-flag` [*flag*]

This flag allows you to suppress warning messages. By default, its value is `t`, and warnings are printed. To turn them off, set `#:complice:warning-flag` to `()`.

`#:complice:no-warning` [*variable*]

This variable contains the list of warning message numbers that correspond to messages that you do wish to see printed. When COMPLICE is initially loaded, `#:complice:no-warning` contains the list `(7 8)`.

E.0

[*Complice error message*]

This error message is displayed as follows:

```
E.0.entity..Internal error:
     (f m b)
```

Cause: Sometimes, for efficiency reasons, or in unforeseen cases, COMPLICE does not systematically verify that the code it produces is valid. For example, it might produce the instruction `(add1 . 2)`, which is not legal. In these cases, a machine error is usually raised, and the `m` argument will be equal to `errmac`.

Arguments: `f`, `m` and `b` are standard error handling arguments. (See the explanation of the `syserror` function).

Remedy: Verify that *entity* works in interpreted mode, and verify that the LISP code of *entity* is coherent and correct.

E.1

[*Complice error message*]

This error message is displayed as follows:

```
E.1.entity..Function computed:
     LISP-expression
```

Cause: This error is raised when the first element of a form is neither a symbol nor a lambda-expression, as in the case of `((if x '1+ '1-) 8)`.

Argument: For the above example, the LISP expression is `(if x '1+ '1-)`.

Remedy: Replace this expression with `(funcall (if x '1+ '1-) 8)`.

E.2

[*Complice error message*]

This error message is displayed as follows:

```
E.2.entity..Application of an mlambda:
     lambda-expression
```

Cause: COMPLICE cannot compile a form such as `((mlambda (1 x) '(ncons ,x)) x)`.

Argument: For the above example, the lambda-expression is `(mlambda (1 x) '(ncons ,x))`.

Remedy: If the macro generation can be performed during the compilation, use an intermediate macro function.

E.3

[*Complice error message*]

This error message is displayed as follows:

```
E.3.entity..Error during macro expansion:
     macro-function
```

Cause: All macro expansions, both `macro` and `dmacro`, are performed in the compiler environment. So, the macro functions that use the dynamic environment of the user's program raise this error.

Here is an example:

```
(defun f (a)
    (bclos a (+ a a)))

(dmd bclos (v . pg)
    `(let ((,v ',(symeval v)))
    ,@pg))
```

Argument: For the above example, macro-function is `bclos`.

Remedy: If the macro generation can only be performed a single time in the user-program environment, do it before the compilation in a test case.

E.4

[*Complice error message*]

This error message is displayed as follows:

`E.4.`*entity*`..Function not defined:`
function

Cause: The function designated as *entity* called a function that was not defined at the time the compilation took place. Here is an example:

```
(defun f () (g))
```

Argument: For the above example, the function is `g`.

Remedy : If the problem was not simply the result of a typing error, perform the call by means of a `funcall` such as `(defun f () (funcall g))`.

E.5

[*Complice error message*]

This error message is displayed as follows:

`E.5.`*entity*`..Cannot compile an flet:`
LISP *expression*

Cause: When the `#:complice:parano-flag` flag is not set, COMPLICE refuses to compile a form beginning with `flet`.

Remedy: If you want to temporarily modify the behavior of functions such as `syserror`, `bol` or `eol`, you might use the `#:system:itsoft` programmable-interrupt approach. The other obvious solution is to set `#:complice:parano-flag` to `t`.

E.6

[*Complice error message*]

This error message is displayed as follows:

`E.6.`*entity*`..Cannot compile a letv:`
LISP *expression*

Cause: COMPLICE can only compile dynamic-environment constructions when the variable tree is supplied explicitly. Here is an example:

```
(defun goodf (vals) (letv '(a b) vals (+ a b)))
(defun badf (env vals) (letv env vals (+ a b)))
```

Argument: For the above example, the LISP expression is `env`.

Remedy: None.

W.0 [*Complice warning message*]

This warning message is displayed as follows:

`W.0.`*entity*`..Undeclared global variable:`
variable

Cause: In the transitive closure of `entity`, `variable` can be used without having been instantiated. Here is an example:

```
(defun f () xyz)

      or

(defun last (1) (ifn 1 xyz (let ((xyz 1)) (last (cdr 1)))))
```

Argument: In the above example, the variable is `xyz`.

Remedy: If the problem stems from neither a typing error nor a logical error in the program, give the variable in question a global value using the `defvar` function. The call to `defvar` must be in a file.

W.2 [*Complice warning message*]

This warning message is displayed as follows:

`W.2.`*module*`..Unused function:`
function

Cause: This warning only occurs with the `compilemodule` function. It appears if `function` is created in `module`, but is not used by any of the exported functions.

Remedy: If the function is needed, it should be added to the module description of `module` under the `export` key.

W.3 [*Complice warning message*]

This warning message is displayed as follows:

`W.3.`*entity*`..Wrong function type:`
function type

Cause: If *entity* is the same symbol as *function*, and *type* is not `()`, an attempt was made to compile a non-compilable function: for example, `(compile car)`. Otherwise, *function* is undefined, type is `()`, and the evaluator will be called explicitly.

Remedy: In the first case, the `funcall` function should be used to compile the arguments of `function`. In the second case, define `function` in your source code. (At times, you have to be careful of `synonymq`).

W.4 [*Complice warning message*]

This warning message is displayed as follows:

`W.4.` *module*`..External function:`
function

W.5 [*Complice warning message*]

This warning message is displayed as follows :

`W.5.` *entity*`..Wrong number of arguments:`
function

Cause: The call sequence of `function` inside *entity* does not macth its definition. Here is an example:

```
(defun f () (cons 1 2 3))
```

or

```
(defun f() (g 1 2 3))
(defun g (x y) (cons x y))
```

W.6 [*Complice warning message*]

This warning message is displayed as follows:

`W.6.` *module*`..Function external to module:`
function

Cause: This warning is only produced with the `compilemodule` function. It appears if *function* is used in module, but is not created by one of the imported modules.

Remedy: If it is known which module contains *function*, it should be added to the module description of *module* under the `import` key. Otherwise, arrange things in such a way that `function` is already compiled when `compilemodule` is called, and that it is also there at execution time.

W.7 [*Complice warning message*]

This warning message is displayed as follows:

`W.7.` *entity*`..Calculated function:`
LISP-*expression*

Cause: The *entity* function uses a function such as `funcall` or `apply`. The LISP `expression` is a functional value. Here is an example:

```
(defun tw (f a) (funcall f (funcall f a)))
```

Argument: In the above example, the LISP expression is `f`.

Remedy: If there is no interaction between the called functional value and the calling function, high-performance code is generated when the `#:complice:parano-flag` flag is set to `()`.

W.8 [*Complice warning message*]

This warning message is displayed as follows:

`W.8.`*entity*`..Explicit call to evaluator:`
LISP-*expression*

Cause: The *entity* function uses a function such as `eval` or `set`. Here is an example:

```
(defun symev (v env) (or (cassq v env) (symeval v)))
```

Argument: In the above example, the LISP expression is `v`.

Remedy: If there is no interaction between the called functional value and the calling function, high-performance code is generated when the `#:complice:parano-flag` flag is set to `()`.

W.9 [*Complice warning message*]

This warning message is displayed as follows:

`W.9.compilefiles..Function external to file:`
function

Cause: This warning only occurs with the `compilefiles` function. It appears if *function* is used in one of the files appearing in the arguments to a call to `compilefiles`, but was not created by one of these files.

Remedy: Arrange things is such a way that `function` has already been compiled when `compilefiles` is executed, and that it is present at execution time as well.

W.10 [*Complice warning message*]

This warning message is displayed as follows:

`W.10.makemodule..Module interdependence:`
module-list

Cause: This warning only occurs with the `compilemodule` function. It appears if the compiler detects a circularity in the set of modules being compiled, according to the links defined by the `import` key. In this case, all the files involved in modules in *module-list* will be recompiled once, independently of all links to other members of *module-list*, and then a second time, to resolve dynamic variable bindings. The *module-list* argument does not describe the smallest circularity of inter-module reference. Given the prohibitive cost of the double compilation, you are advised to recompile one of the modules involved in the circularity by hand before trying to compile the modules together a second time.

`W.11` [*Complice warning message*]

This warning message is displayed as follows:

`W.11.`*entity*`..Function redefined in the module:`
module

Cause: This warning indicates that *entity*, which appears in the module being compiled, is also a function exported by *module*, and that the latter is imported by the module being compiled. In this case, it is the local definition that is used during the compilation of the current module.

13.2.3 General remarks and examples

Using defvar

In some particular cases, COMPLICE does not find variables that should be dynamically linked, even when the `#:complice:parano-flag` is set to `t`. Here is an example:

```
(defun line-count ()
    (let ((#:sys-package-itsoft 'count) (#:count:n 0))
         (untilexit eof (read))
         #:count:n))

(defun #:count:bol ()
    (setq #:count:n (add1 #:count:n))
    (bol))
```

In this case, the variable `#:count:n` is used in a dynamic manner, but it will be compiled 'lexically'. To alleviate these problems, a global variable declared by `defvar` can be used for it, and will always be compiled by COMPLICE according to dynamic binding rules.

The above program would be rewritten as follows:

```
(defar #:count:n)

(defun line-count ()
    (let ((#:sys-package-itsoft 'count) (#:count:n 0))
         (untilexit eof (read))
         #:count:n))

(defun #:count:bol ()
    (setq #:count:n (add1 #:count:n))
    (bol))
```

It is therefore suggested that `defvars` be used for variables such as `l`, `n` and `at`.

Using macros

A macro function instantiated in a module, but not exported, will no longer exist after compilation.

For example, the module file `test` might contain

```
defmodule user
   files (test)
   export (test)
```

and the LISP source file `test` might contain

```
(defun test (n) (test-macro n))

(dmd test-macro (n) `(add1 ,n))
```

Once the module is compiled, the `test-macro` function will no longer exist.

Function generators

In processing the `compilefiles` and `compilemodule` functions, COMPLICE determines which functions to compile by isolating the occurrence of the `de`, `df`, `defun`, `setfn`, `dm`, `dmd`, and `defmacro` primitives. There are two compiler features that provide the means to define new function generators.

The first follows from the fact that forms read from files are macro-expanded. It is possible to define macro functions that define calls to the standard primitives introduced above. Here is an example:

```
(dmd defsys (name . fval)
     `(defun ,(symbol 'system name) ,@fval) )

(defsys ob () (oblist 'system))
```

On the other hand, the module description files must specify the complete names of functions to export, as shown here:

```
defmodule system
   files (foo)
   export (#:system:ob)
```

The second phenomenon is due to the fact that forms beginning with `progn` are themselves scanned to determine possible function definitions.

```
(dmd newtype (type make pred)
     `(progn
         (defun ,make (l)
             (let ((v (apply 'vector l)))
                  (typevector v ',type)))
         (newl list-type ',type)
         (defun ,pred (v) (eq (typevector v) ',type))
         (defmacro-open ,pred (v) `(eq (typevector ,,v) ',,type))))

(defvar list-type () )
(newtype cell makecell cellp)
```

Using the precompile function

Among other things, the `precompile` function allows data definition to be delayed until load time. Consider the following example:

```
(defun static-stream ()
(let ((stream '#.(cons -1 (cirlist 0 1))))
     (prog1 (cadr stream)
            (rplacd stream (cddr stream)))))
```

This function is correctly compiled by the `compile` and `compile-all-in-core` functions, but the corresponding `lap` code will not be able to be loaded into a file because of the circularity of the data. In such cases, the `precompile` function can be used in the following manner:

```
(dmd reconstruct (data)
     `(precompile ',(eval data) () () (eval kwote ,data))))

(defun static-stream ()
    (let ((stream (reconstruct (cons -1 (cirlist 0 1)))))
         (prog1 (cadr stream)
                (rplacd stream (cddr stream)))))
```

Undesirable aspects of Complice

The `compilemodule` function can create the object file only in the current working directory.

Lexical control functions such as `tagbody`, `block`, `go` and `return` systematically call specialized evaluation routines.

In imported modules, be wary of any macro definitions that are not inside an `(eval-when (compile...` construction. Such definitions might cause the total loading of the imported module during the compilation of the module that imports it.

Table of contents

13 Compilation **13-1**

13.0.1 Calling the compilers . . . 13-1
13.0.2 Compiler macros . . . 13-3
13.0.3 Closed macros . . . 13-3
13.0.4 Open macros . . . 13-4
13.0.5 Modules . . . 13-6
13.0.6 Source files . . . 13-11
13.0.7 Object files . . . 13-12
13.0.8 Controlling evaluation . . . 13-12
13.1 Use and manipulation of modules . . . 13-15
13.1.1 Functions on modules . . . 13-16
13.1.2 Loading and compiling modules . . . 13-16
13.2 Complice . . . 13-18
13.2.1 Compatibility messages . . . 13-18
13.2.2 Errors and warnings . . . 13-19
13.2.3 General remarks and examples . . . 13-25

Function Index

`compiler` [*feature*] 13-1

`complice` [*feature*] 13-1

`(compiler source status print loader)` [*function with four arguments*] 13-2

`(compile source status print loader)` [*special form with one, two, three or four arguments*] 13-2

`(compile-all-in-core print loader)` [*function with zero, one or two optional arguments*] 13-2

`(compilefiles source object)` [*function with two arguments*] 13-3

`(precompile exp1 result exp2 operand)` [*special form*] 13-3

`(dont-compile` $\texttt{exp}_1$ `...` $\texttt{exp}_n$`)` [*special form*] 13-3

`#:compiler:open-p` [*variable*] 13-4

`(defmacro-open name ... fval)` [*special form*] 13-5

`(make-macro-open name fval)` [*function with two arguments*] 13-5

`(macro-openp name)` [*function with one argument*] 13-5

`(remove-macro-open name)` [*function with one argument*] 13-6

`#:system:obj-extension` [*variable*] 13-12

`(probepatho file)` [*function with one argument*] 13-12

`(loadobjectfile file)` [*function with one argument*] 13-12

`(eval-when moments . expressions)` [*function with a variable number of arguments*] 13-13

`(probepathm module)` [*function with one argument*] 13-16

`(readdefmodule module)` [*function with one argument*] 13-16

`(getdefmodule defmod key)` [*function with two arguments*] 13-16

`(setdemodule defmod key value)` [*function with three arguments*] 13-16

`(printdefmodule defmod module)` [*function with two arguments*] 13-16

`(loadmodule module reload-p interpreted-p)` [*function with one, two or three arguments*] 13-16

`#:module:interpreted-list` [*variable*] 13-17

`#:module:compiled-list` [*variable*] 13-17

`(compilemodule module ind)` [*function with one or two arguments*] 13-17
`#:complice:parano-flag` [*variable*] 13-18
`#:complice:warning-flag` [*flag*] 13-19
`#:complice:no-warning` [*variable*] 13-19
`E.0` [*Complice error message*] 13-20
`E.1` [*Complice error message*] 13-20
`E.2` [*Complice error message*] 13-20
`E.3` [*Complice error message*] 13-20
`E.4` [*Complice error message*] 13-21
`E.5` [*Complice error message*] 13-21
`E.6` [*Complice error message*] 13-21
`W.0` [*Complice warning message*] 13-22
`W.2` [*Complice warning message*] 13-22
`W.3` [*Complice warning message*] 13-22
`W.4` [*Complice warning message*] 13-23
`W.5` [*Complice warning message*] 13-23
`W.6` [*Complice warning message*] 13-23
`W.7` [*Complice warning message*] 13-23
`W.8` [*Complice warning message*] 13-24
`W.9` [*Complice warning message*] 13-24
`W.10` [*Complice warning message*] 13-24
`W.11` [*Complice warning message*] 13-25

Chapter 14

External interfaces

The first section of this chapter describes functions that interface LE-LISP system with external procedures written in other programming languages. The following sections describe the details of the process of interfacing LE-LISP with different languages on different systems. Only the section covering the UNIX system is included in this manual. For the VMS interface, see [Dana86].

14.1 Interface functions

LE-LISP provides ways to call external procedures written in another programming language. These external procedures must be integrated into LE-LISP with the host system's link editor.

The following functions, which provide the interface of external procedures, are not standard on all LE-LISP systems. Check to see if they exist on your system before trying to use them.

(getglobal strg) [*function with one argument*]

Returns the starting address (entry point) of the external procedure name `strg`. This name must exist in the symbol table generated by the link editor when LE-LISP is created. The result of this function can be used as the first argument to `callextern`.

The notion of a *symbol* depends of course upon the the host system. On UNIX systems, for example, C-language names are usually preceded by the underscore character, as in `_getenv`, whereas FORTRAN names are surrounded by this same character, as in `_getenv_`. Depending on the machine on which it is running, LE-LISP takes account of the possible presence of such prefixes. For example, you find the underscore `_` prefix on a Sun, whereas it is a dot on the RS6000.

In no case does the user have to be aware of the convention in use. He or she only needs to use the name of the symbol as it is defined in C.

Of course, for compatibility reasons, the user can indicate a prefix when performing a `getglobal`, but this technique is slow, and it might cause portability problems when the code is taken onto another machine. In any case, the presence of a prefix in the character string sent to `getglobal` is a means of avoiding possible name conflicts between LE-LISP functions and C procedures.

On a Sun4, with no prefix:

```
(getglobal "getenv")          ⟹    (4 . 22192)
```

```
(time '(getglobal "getenv"))  ==>   0.01
```

On a Sun4, with a prefix:

```
(getglobal "_getenv")          ==>   (4 . 22192)
(time '(getglobal "_getenv"))  ==>   0.02
```

With the prefix of an RS6000:

```
(getglobal ".getenv")  ==>   0
```

(callextern address type v_1 t_1 ... v_n t_n)

[*function with a variable number of arguments*]

Calls an external procedure starting at `address`. This call must return a value whose type is `type`. The v_i designate values transmitted to the external procedure, and their corresponding types are indicated by the t_i. These t_i are coded in the following manner:

- `0` —→ pointer
- `1` —→ integer number
- `2` —→ floating-point number
- `3` —→ character string
- `4` —→ vector of S-expressions
- `5` —→ integer number, passed by reference (FORTRAN)
- `6` —→ floating-point number, passed by reference (FORTRAN)
- `7` —→ vector of integer numbers
- `8` —→ vector of floating-point numbers.

The type of the result returned by the external procedure call, referred to as `type`, can only correspond to one of the first four items in this list: either pointer, integer number, floating-point number or character string.

(defextern symb ltype type)

[*special form*]

Dynamically associates a LE-LISP function to an external procedure. `symb` is the name of an external module that is going to become a new LE-LISP function. `ltype` is the list of types of input arguments. The version 15.26 introduces a dynamic verification of the types `<ltype>` at the time of the call to the function `<symb>`.

```
ex : ? (defextern ma-fonction (string) fix)
     ? = ma-function
     ? (ma-function 5)
     ** ma-fonction : the argument is not a string : 5
```

`type` is the type of the return value. The possible types for the `ltype` argument are members of the following set of symbols:

- `t`: a LE-LISP pointer.
- `external`: an external pointer.
- `fix`: an integer number.
- `float`: a floating-point number.
- `string`: a character string.
- `vector`: a vector of S-expressions.
- `rfix`: an integer passed by reference.
- `rfloat`: a float passed by reference.
- `fixvector`: a vector of integers.
- `floatvector`: a vector of floats.

The returned `type` is one of the following symbols:

- `t`: a LE-LISP pointer.
- `external`: an external pointer.
- `fix`: an integer number.
- `float`: a floating-point number.
- `string`: a character string.

Let us examine these symbols more closely:

- The `t` type corresponds to any LE-LISP pointer that is passed without being altered.
- The `external` type corresponds to a pointer external to the LISP memory space. It is represented by a LE-LISP address. A detailed description of this kind of address is provided in section 12.1. Generally, such a pointer is the result of a previous call to an external procedure, and it is sufficient to store and pass it to other external procedures when it is called for.
- The `fix` type corresponds to an integer numeric value.
- The `float` type corresponds to a floating-point numeric value. When the port allows it, this type works equally well in both the 31-bit and 64-bit LE-LISP floating-point modes.
- The `string` type corresponds to the address of the first character of a character string. It is impossible to pass a character string by value.
- The `rfix` type corresponds to a reference to an integer numeric value. This is the type of argument passing that is used in FORTRAN.
- The `rfloat` type corresponds to a reference to a floating-point numeric value. This is the type of argument passing that is used in FORTRAN. When the port allows it, this type works equally well in both 31-bit and 64-bit LE-LISP floating-point mode.

- The `fixvector` type corresponds to the address of the first element of a vector of integer numeric values. Vectors of integers are therefore always passed by reference. Because of the way integers are represented in LE-LISP, this type is different to the type of a vector of S-expressions containing only integer numeric values.
- The `floatvector` type corresponds to the address of the first element of a vector of floating-point numeric values. Vectors of floats are therefore always passed by reference. Because of the way that floats are represented in LE-LISP, this type is different to the type of a vector of S-expression containing only floating-point numeric values. Note that this `floatvector` type works only in 31-bit floating-point mode. You cannot use it in 64-bit floating-point mode.

Example: the following external procedure is defined in C:

```
double c_test (strg,nf,ni,vect)
       char *strg; double nf; int ni; int *vect;
{
  int i;
  printf("the string is %s\n\r", strg);
  printf("the float is %e\n\r", nf);
  printf("the integer is %d\n\r", ni);
  printf("the vector contains vect[0]=%8x, vect[1]=%8x\n\r", vect[0], vect[1]);
  i = vect[0]; vect[0] = vect[1]; vect[1] = i;
  return(nf*ni);
}
```

Here is the corresponding LE-LISP interface:

```
? (defextern c_test (string float fix vector) float)
= c_test   ? (defvar vec #[(a b) #[1 2 3]])
= vec   ? (c_test "fou barr" 123.45 12 vec)
the string is fou barr
the float is 1.234500e+02
the integer is 12
the vector contains vect[0]=  9ed30, vect[1]=  63ae4
= 1481.4   ? vec
= #[#[1 2 3] (a b)]
```

The following external procedure is defined in FORTRAN:

```
      subroutine tabflt (lg, tfloat)
      integer lg,i
      real*4 tfloat(lg)
      do 33 i=1,lg
         tfloat(i)=tfloat(i) * 2.
   33 continue
      return
      end
```

Here is the corresponding LE-LISP interface:

```
? (defextern _tabflt_ (rfix floatvector))
= _tabflt_   ? (setq v #[1.2 4.0 9.9 .25])
= #[1.2 4. 9.9 .25]   ? (_tabflt_ (vlength v) v)
= 1020   ? v
= #[2.4 8. 19.8 .5]
```

`(defextern-cache flag)` [*special form*]

If `flag` is true, the following `defextern`s will be cached. When the flag will be set to false, all the cached `defextern`s will be resolved.

14.2 Links with external procedures under Unix

We explain here how to interface LISP with external procedures written in C, using `defextern`, and how to call LISP from external procedures, using `lispcall`.

These features—the least portable in the system—are described here only for UNIX systems.

14.2.1 Principles

The LE-LISP language is not perfectly adapted to every possible computing task. It is notably ineffective for long numeric calculations such resolutions of differential equations and finite element methods, or bit-based operations on high-resolution screens such as general raster-op procedures. For these tasks, programmers generally prefer languages that perform these special tasks well, such as FORTRAN, C or an assembly language.

LE-LISP provides the means to interface procedures written in other languages. These procedures become regular LISP functions.

It is possible to pass arguments of all types—such as integer and floating-point numeric values, character strings, vectors of LE-LISP objects, and any LISP object—to these procedures, and to receive their typed results.

LE-LISP can therefore be used to drive very fast procedures written in other languages, and can be considered as a tool for interfacing and carrying on dialogues with these procedures.

14.2.2 Calling functions written in C

We describe here the way to interface LISP with C programs. Everything described here is also valid for FORTRAN, which repects the C calling conventions under UNIX, and for those assembly languages that do likewise.

There could be problems with a PASCAL interface, due to file handling procedures. In PASCAL, files must be declared in the `program` declaration. It is therefore difficult to link PASCAL procedures outside a PASCAL context. The solution in this case is to write a PASCAL program that calls the LE-LISP system as an external procedure.

To call a C function from the LISP system, you have to link the compiled function object and the system kernel into the same executable program, and then associate the C entry point to a LISP function. The LISP function call then lets the C program be run.

Linking external procedures

The link operation consists of binding the code of several external procedures into a single executable program. In this case, the LISP system and the C modules containing the C procedures that will be called are linked together.

This link can be created statically, when the system is created, or dynamically at any time, under the control of the system.

The static link operation consists of creating a new executable binary containing the kernel of the LE-LISP system and the C modules. The new binary can then be used to generate a new LISP system. It can be installed on any and all UNIX machines. This is a major undertaking, though, since it requires the recreation of the system every time that one of the C modules is changed.

The dynamic link operation can only be carried out on machines that have linkers—also known as link editors—that make it possible to carry out incremental linking. In the UNIX system, the linker is called using the `ld` command with the `-A` option. The dynamic linking operation happily replaces the static link described above, and eliminates the need for recreation after each change in the C modules. Moreover, the dynamically linked modules are maintained in memory images files. So, the dynamic link actually ends up being more durable than the static link.

Dynamic links

(cload string) [*function with one argument*]

The argument to this function is the name of a compiled module that is to be linked to the system. `cload` calls the UNIX linker `ld` to link the C module with the system being used. The compiled code of the C module is loaded into the LISP `code` memory zone, and can thereby be saved into a LISP memory image. Here is an example:

```
$ lelisp
; Le-Lisp (by INRIA) version 15.24 ( 2/Jan/91) [sun4]
; Standard modular system: Sat 29 Dec 90 19:34:39
= (31bitfloats date microceyx pathname debug setf pepe virbitmap
defstruct virtty compiler pretty abbrev loader messages callext)
```

We compile a C module named `foo.c`, thereby creating the binary object named `foo.o`:

```
? !cc -c foo.c
= t
```

We then link the `foo.o` module to the system:

```
? (cload "foo.o")
= (19 . 2345)
```

The character string argument is passed to the `ld` link editor. This means that options to load libraries, to get a trace of the link edit, or to link several modules at once can be exercised with the `cload` function. For more details, see the UNIX documentation for `ld` on your machine.

Let us look at other examples of the use of the `cload` function. The next call loads the `ash.o` module and the `raster.a` library:

```
(cload "ash.o -lraster")
```

The next example loads the hyperbolic sine entry point from the mathematical function library:

```
(cload "-u _sinh -lm")
```

The next call loads three modules named `foo.o`, `gee.o` and `bar.o`:

```
(cload "foo.o  gee.o  bar.o")
```

The character string argument is the means by which the call to the `ld` link editor is constructed. For more precision, see the UNIX documentation concerning `ld(1)`.

Consider the call `(cload` *string*`)`.

On a Sun, this generates the call

`ld -A` *lelispbin* `-Bstatic -N -x -T` *end* `-o` *temporary string* `-lc`

On an SPS9, this generates the call

`ld -C -A` *lelispbin* `-N -x -T` *end* `-o` *temporary string* `-lc`

Here, *lelispbin* is a path that contains the directory where the currently-running LE-LISP binary is located, *end* is the first available address in the LISP zone reserved for compiled code, and *temporary* is a unique temporary file in the directory `/tmp`.

After the module is linked, the *temporary* file contains the code of the C module. It is loaded into the LISP `code` zone. It is this *temporary* file that will be used in place of *lelispbin* for subsequent calls to `cload`. So, numerous calls to `cload` can be made to sequentially build and modify an executable image.

Static links

The static link is described in the system installation documentaton, in the beginning of the reference manual. It consists of a permanent link between the LE-LISP system `lelispbin` and the relevant C modules, to make a new binary. After this, the system must be configured. This involves loading an environment and creating a memory image, as described in the installation documentation. Generally, an entry point is made in the `makefile` so that the new LISP system can be linked and configured.

Let us look at an example. Here are the `makefile` declarations that allow the `ash.o` module and the `raster.a` library to the `lelispbin` system and to configure a system named `ashlelisp`. See the documentation on the installation of LE-LISP on UNIX for more details.

```
ashlelisp :    ashlelispbin ash.ll
       config ashlelisp ashlelispbin ash.ll
```

```
ashlelispbin : ash.o
       cc -x -n $(cflags) lelispbin.o ../common/lelisp.c \
                          ash.o -lraster -lm -lc -o ashlelispbin
```

An inconvenience of the static link is the need to recreate the complete system each time that the C files are modified. This operation can often be quite time-consuming.

C function declarations

Once the modules are linked to the system, the C functions must be linked to LISP functions. The LISP function `defextern` (see above) performs this association.

LISP type	C type	Argument	Result
`fix`	`long`	Sign extended to 32 bits.	Converted to 16 bits.
`float`	`double`	Value passed.	Lisp float created.
`string`	`char *`	Pointer to `char` string passed.	Lisp string allocated. C string copied into it.
`vector`	`any *`	Vector of Lisp pointers passed.	No change: Lisp receives C pointer 'as is'.
`rfix`	`long *`	Integer passed by reference. Used above all to exchange integers with Fortran.	Not implemented.
`rfloat`	`double *`	Float passed by reference. Single or double precision float changed to double in external language. Used above all to exchange floats with Fortran.	Not implemented.
`fixvector`	`long *`	Array of integers passed. Sign of each integer extended to 32 bits.	Not implemented, but changes made to C array are recovered by Lisp.
`floatvector`	`float *`	Array of floats passed. Single-precision Lisp floats changed to C floats.	Not implemented, but changes made to C array are recovered by Lisp.
`t`	`any *`	No change: C receives the Lisp pointers 'as is'.	No change: Lisp receives C pointers 'as is'. *Warning*: Le-Lisp might do strange things if random pointer values are passed to it.
`external`	`any *`	Lisp object must be a `cons` of two integers coding an address. (See `vag` function in Lisp.) Passed to C.	Lisp `cons` is allocated and initialized with most/least significant parts of the address of the C object. (See `loc` function in Lisp.) The function returns the `cons` as its result.

The above table describes the conversions that take place when arguments are passed to C, and when results are returned. In this table, the so-called 'LISP type' is the type defined at the time of the `defextern`, whereas the so-called 'C type' is the type of the arguments received by the C function connected to LISP.

LISP pointers are either pointers into the LISP data zone or 16-bit short integers. The `lelisp.h` file contains the declarations of the type of C structure to which LISP pointers point. More details are revealed below. The `external` type provides a way of manipulating C structures in LISP. It is not true that any C pointer at all can be manipulated within the system. It must first be converted into a `cons` of two integers that represent the high-order and low-order parts of the address. The `external` type permits values to be converted between C pointer and LISP-doublet representations at each call to, or return from, a C function.

Examples

Let's look at examples of the use of the `defextern` function. For each example, we first show a C program to be called. Next, there is the `defextern` declaration that allows the corresponding LISP function to be built. Finally, there are samples showing the function being used.

The first function takes three arguments—an integer, a string and a float—and returns a float:

```
double foo (x, s, f)
int x; char *s; double f;
{
  printf("integer : %d, string : %s, float : %g\n", x, s, f);
  return(x * strlen(s) * f);
}
```

```
? (defextern foo (fix string float) float)
= foo

? (setq str "hello")
= "hello"   ? (foo 10 str 1.2e+10)
integer : 10, string : hello, float : 1.2e+10
= 6.e+11
```

The next function accepts a string and returns a string.

```
char *getenv();

char *envar (s);
char *s;
{
  return(getenv (s));
}
```

```
? (defextern envar (string) string)
= envar

? (envar "HOME")
= /udd/halles/rodine  ; Now it is a Lisp string
```

We next look at a function that takes a vector. Its argument `v` is declared as having the type

`int v[]`. It will be able to be used on vectors of 16-bit LISP integers, which will appear to it like C vectors of 32-bit integers that happen to have values between zero and 65536, inclusive. It is generally handy, too, to pass the length of the vector as an argument.

```
whatyousay (v, l)
int v[], l;     /* l is the length of the vector */
{
  int i;
  for (i = 0; i < l; i++)
    printf"%d ", v[i]);
  printf("\r\n");
}

? (defextern whatyousay (vector fix))
= whatyousay

? (whatyousay #[1 2 3 4 -4 -3 -2 -1] 8)
1 2 3 4 65532 65533 65534 65535
= 0
```

The next example involves the use of the `external` type. It shows how to manipulate a pointer of any type received from C. In this example, it is meant to be a pointer to a window allocated in the C context.

```
window *
create_a_window (x, y, l, h)
int x, y, l, h;
{
    ...    /* create a structured window object and
              return a pointer to it. */
}

select_a_window (w)
window *w;
{
    ....   /* receive a pointer to a structured window object */
}

? (defextern create_a_window (fix fix fix fix) external)
= create_a_window
? (defextern select_a_window (external))
= select_a_window
?
? (setq a (create_a_window 10 10 400 400))
= (12 . 2348)   ; the pointer that is manipulable in Lisp
?
? (select_a_window a) ; which can be sent back to C when necessary.
= 0
```

Arguments whose type is `t` are passed straight through to C functions 'as is', without modification. These functions therefore receive LISP objects: generally pointers to structures. The integer and

floating-point cases are slightly different. See the remainder of this chapter. The `lelisp.h` file included in the standard distribution contains C structures corresponding to LISP objects.

For example, here is the C structure declaration for `cons`, with two fields, `ll_car` and `ll_cdr`, each of which contains a LISP object:

```
struct LL_CONS {
     LL_OBJECT  ll_car;
     LL_OBJECT  ll_cdr;
};
```

The `LL_OBJECT` type is a polymorphic type that covers all LISP object types such as `LL_SYMBOL` and `LL_CONS`. Since C has no notion of polymorphic type, `LL_OBJECT` is declared as follows:

```
typedef char *LL_OBJECT;
```

Let us look at an example. The following function extracts the `car` of a LISP pair received as its argument:

```
#include "lelisp.h"

LL_OBJECT car_in_c (o)
struct LL_CONS *o;
{
  return (o->ll_car);
}
```

```
? (defextern car_in_c (t) t)
= car_in_c
? (car_in_c '((a) (b) (c)))
= (a)    ? ; warning: it can be dangerous to take the car of just anything!
? (car_in_c 1)
I quit on signal 3
$
```

Symbols are treated in the same way as `cons`. The `ll_alink` and `ll_pname` fields are used by LE-LISP to link symbols in the object, and their contents should never be touched.

```
struct LL_SYMBOL {
     LL_OBJECT  ll_cval;
     LL_OBJECT  ll_plist;
     LL_OBJECT  ll_fval;
     LL_OBJECT  ll_alink;
     LL_OBJECT  ll_pkgc;
     LL_OBJECT  ll_oval;
     char       ll_ftype;
     char       ll_ptype;
     short      ll_pad;
```

```
        LL_OBJECT  ll_pname;
};
```

Character strings and vectors are pointers to pointers to variable-sized structures. This double indirection is needed for memory deallocation. Since variable-sized structures do not exist in C, the following declarations correspond to one-byte objects, such as a single-character constant or a vector with one element. The `ll_xxxsiz` field indicates the real size of the object: that is, the number of characters in a string or the number of elements in a vector. The first element in the object is the `ll_xxxfil` field, and the other elements follow it contiguously in memory. The `ll_xxxtyp` fields contain the types of strings and vectors.

```
struct LL_STRING {
     struct {
        struct LL_STRING *ll_strarr;
               int       ll_strsiz;
               char      ll_strfil;
        } *ll_strobj;
        LL_OBJECT ll_strtyp;
};

struct LL_VECTOR {
     struct {
        struct LL_VECTOR *ll_vecarr;
        int              ll_vecsiz;
        LL_OBJECT        ll_vecfil;
     } *ll_vecobj;
     LL_OBJECT ll_vectyp;
};
```

Here is an example function that prints the characters in a LISP string one by one:

```
one_by_one (string)
struct LL_STRING *string;
{
  int i;
  int size;
  char *s;

  size = (string->ll_strobj)->ll_strsiz; /* the size of the string */

  s = &((string->ll_strobj)->ll_strfil); /* the address of the first
                                            character */

  for (i = 0; i < size; i++)
     printf("character %d :%c\r\n", i, *(s+i));
}
```

LISP objects of the `fix` type are not pointers, but rather immediate values. Integers are 16-bit numbers. A LISP object of type `integer` is therefore a 32-bit pointer whose first 16 bits are always zero.

Floats can be implemented in various different ways according to host system architecture. The

next example shows how to send to C a vector of LISP floats. C raises each float to the second power, and LISP recovers the vector with these side effects.

C_LL_FIX [*Fonction C*]

The function C, C_LL_FIX allows you to transform a C integer into a Lisp object of the type fix. This function does not carry out overflow control on the domain of Lisp objects of type fix.

The example below presents how to carry out the addition of two Lisp fix's in C with conversion of the results into Lisp format in C.

```
In C:
LL_OBJECT tcclfix(i,j)
long i,j;
{
return(C_LL_FIX(i+j));
}

In Lisp:
? (defextern tcclfix (fix fix) t)
= tcclfix
? (tcclfix 32000 767)
= 32767
```

LL_C_FIX [*Fonction C*]

The function C, LL_C_FIX allows you to transform a Lisp object of type fix into a C integer. Because of the existence of the domain of C integer representations (always at least superior to the domain of Lisp object representations of type fix) overflow is not possible at the time of this conversion.

Example of the use of the function LL_C_FIX.

```
In C:
long tllcfix(i,j)
LL_OBJECT i,j;
{
return(LL_C_FIX(i) + LL_C_FIX(j));
}

In Lisp:
? (defextern tllcfix (t t) fix)
= tllcfix
? (tllcfix 32000 767)
= 32767
```

Floating point numbers can be implemented in various different ways according to host system architecture.

C_LL_FLOAT [*Fonction C*]

The function C, C_LL_FLOAT allows you to transform a C double floating point number into a Lisp object of type float.

Example of the use of the function C_LL_FLOAT.

```
In C:
LL_OBJECT tcllfloat(f)
double f;
{
return((LL_OBJECT)C_LL_FLOAT(f + 1.));
}

In Lisp:
? (defextern tcllfloat (float) t)
= tcllfloat
? (tcllfloat 2.)
= 3.
```

LL_C_FLOAT [*Fonction C*]

The function C, LL_C_FLOAT allows you to transform a Lisp object of type float into a C double floating point number.

Example of the use of the function LL_C_FLOAT

```
In C:
double tllcfloat(f)
LL_OBJECT f;
{
return(LL_C_FLOAT(f) + 1.);
}

? (defextern tllcfloat (t) float)
= tllcfloat
? (tllcfloat 2.)
= 3.
```

The LL_C_FLOAT C macro transforms LISP floating-point numbers into C floating-point numbers. The next function copies a vector of LISP floats into an array of C floats:

```
In C:
int ttabflt (size, tabf)
int size;
float tabf[];
{
int i;
for (i=0; i<size; i++)
tabf[i] = tabf[i] * tabf[i];
return size;
}


In Lisp:
? {(defextern ttabflt (fix floatvector))}
= {ttabflt}
? {(setq v \#[1. 2. 3. 4. 5.])}
= {\#[1. 2. 3. 4. 5.]}
? {(ttabflt 5 v)}
= {5}
? {v}
= {\#[1. 4. 9. 16. 25.]}
```

WARNING

The manipulation of LISP objects in C is a risky activity. Be careful when modifying the contents of LISP objects received as arguments. In particular, never touch the following fields: `ll_pname`, `ll_alink`, `ll_strobj`, `ll_vecobj`, `ll_strarr`, `ll_vecarr`, `ll_strtyp` or `ll_vectyp`. If you were to interfere with any of these fields, this could raise system errors.

Communication with FORTRAN takes place in a similar fashion, the only difference being the argument declarations on the FORTRAN side:

- An integer is declared `rfix` on the `defextern` side, and `integer` on the FORTRAN side.
- A float is declared `rfloat` on the `defextern` side, and `real*8` on the FORTRAN side.
- An array of integers is declared `fixvector` on the `defextern` side, and `integer (lg)` on the FORTRAN side.
- An array of floats is declared `floatvector` on the `defextern` side, and `real*4 (lg)` on the FORTRAN side.

As before, in the case of arrays, every modification carried out in FORTRAN is recovered in LISP when control returns to the `defextern`.

14.2.3 Calling Lisp from C

On certain machines—such as the Sm90, VAX, Sps7 and Sun—it is possible to call the LISP system from a C function that was itself called from LISP. Here is a sketch of this interaction:

1. The LISP system is called.
2. The C function is called from LISP by `defextern`.
3. The system is called from one of the the C functions.

These back-and-forth calls between LISP and C can be embedded until system resources are depleted.

Implementation

This dialog is carried out on the LISP side by using the `defextern` primitive described above, and on the C side by using three functions: `getsym`, `pusharg` and `lispcall`. These functions use the LISP object definitions from the `lelisp.h` file, and are defined as follows:

```
struct LL_SYMBOL *
getsym(s)
  char *s; {
  ...  /* Return the Lisp symbol that has the string s as its pname */
}

pusharg(type, val)
int type; any val;
{
  ...  /* Stack up the argument ival, which has type itype */
}

LL_OBJECT lispcall(typeres, narg, function)
  int typeres;               /* the type of the Lisp function's result */
  int narg;                  /* the number of args pushed by pusharg */
  struct LL_SYMBOL *function; /* the Lisp function to run */
  {
  ...  /* Call the Lisp function with the narg arguments on the stack.
          Return the result of this function. */
}
```

The argument types for the `pusharg`, as for the return from `lispcall`, can be chosen from the following symbolic types: `LLT_FIX`, `LLT_FLOAT`, `LLT_STRING`, `LLT_VECTOR` and `LLT_T` (which is a pointer to anything). As in the LISP to C call, automatic conversion of argument format is also carried out in the other direction, from C to LISP, as described in the table above.

Examples

Several examples of definitions of C functions that call the LISP system follow, along with sample uses of them.

Calling the cons function

This C function takes two LISP objects of any type as arguments, and returns a `cons` of these two objects:

```
struct LL_CONS * cons_in_c (a, b)
LL_OBJECT a, b;
{
  struct LL_SYMBOL *lisp_cons;

  lisp_cons = getsym("cons");

  pusharg(LLT_T, a);
  pusharg(LLT_T, b);

  return( (struct LL_CONS *) lispcall(LLT_T, 2, lisp_cons));
}
```

```
? (defextern cons_in_c (t t) t)
= cons_in_c

? (cons_in_c 1 2)
= (1 . 2)

? (trace cons)
= cons
? (cons_in_c 1 2)
cons ---> 1 2      ; the Lisp cons function is called, sure enough
cons <--- (1 . 2)
= (1 . 2)
```

N-ary functions

N-ary LISP functions can also be called from C. This example function transforms a vector into a list by calling the LISP `list` function.

Note that, in general, the `getsym` function is not called each time that the C function is called, but only once during an initialization pass.

```
#define null (struct LL_SYMBOL *) 0

struct LL_SYMBOL *lisp_list = null;

struct LL_CONS *vect_to_list (v, l)
int v[], l;
{
  int i;

  if(lisp_list == null)
    lisp_list = getsym("list");

  for (i = 0; i < l; i++)
    pusharg(LLT_T, v[i]);
```

```
  return( (struct LL_CONS *) lispcall(LLT_T, 1, lisp_list));
 }

? (defextern vect_to_list (vector fix) t)
= vect_to_list

? (vect_to_list #[1 2 3 4 -4 -3 -2 -1] 8)
= (1 2 3 4 -4 -3 -2 -1)
```

Recursion

Remember that the LISP and C inter-language calls can be recursively embedded. For example, you could write the Fibonnacci function in the form of mutually-recursive LISP and C functions:

```
? !cat fib.c
#include "lelisp.h"
struct LL_SYMBOL *lisp_fib;

init_c_fib()
{
  lisp_fib = getsym("fib");
}

int
c_fib (n)
  int n;{
  int x;

  if (n==1) return(1);
  else
  if (n==2) return(1);
  else{
  pusharg(LLT_FIX, n-1);
  x = (int) lispcall(LLT_FIX, 1, lisp_fib);

  pusharg(llt_fix, n-2);
  return(x+ (int) lispcall(llt_fix, 1, lisp_fib));
  }
}

= t

? ; Dynamic link of the fib.o module with the intepreter
? !cc -c fib.c
= t    ? (cload "fib.o")
= (17 . 16384)

? (defextern c_fib (fix) fix)
= c_fib    ? (defextern init_c_fib ())
= init_c_fib    ? (init_c_fib)
```

```
= 0
```

Here is the Lisp function that calls the C function in the midst of the recursion:

```
? (defun fib (n)
?     (cond ((eq n 1) 1)
?           ((eq n 2) 1)
?           (t (+ (c_fib (1- n))  ; the first recursive (C) call
?                 (c_fib (- n 2)))))) ; the second ...
= fib

? (fib 4)
= 3

? (trace fib c_fib)
= (fib c_fib)

? (fib 4)
fib ---> n=4
c_fib ---> arg106=3
fib ---> n=2
fib <--- 1
fib ---> n=1
fib <--- 1
c_fib <--- 2
c_fib ---> arg106=2
c_fib <--- 1
fib <--- 3
= 3
```

Launching Lisp from C

On certain machines, you can hand over the control of a LISP application to C. In such a situation, it is no longer LISP that calls C, but rather C that calls LISP. Some UNIX ports have a `llmain.c` file that has no other rôle than to act as the main program that launches LE-LISP. The user can execute all the code he wants to before launching LISP. After that, he can either recall C using the `defextern` process described earlier on, or return to his C environment of the main program. This mechanism is implemented by means of the UNIX system calls `setjmp` and `longjmp`. UNIX ports that offer this possibility contain demonstration files named `fromc.c` and `fromc.ll` in the LE-LISP directory called `lltest`.

Table of contents

14 External interfaces **14-1**

14.1 Interface functions 14-1

14.2 Links with external procedures under Unix 14-5

14.2.1 Principles 14-5

14.2.2 Calling functions written in C 14-5

14.2.3 Calling Lisp from C 14-17

Function Index

`(getglobal strg)` [*function with one argument*] 14-1

`(callextern address type` v_1 t_1 ... v_n t_n`)` [*function with a variable number of arguments*] 14-2

`(defextern symb ltype type)` [*special form*] 14-2

`(defextern-cache flag)` [*special form*] 14-5

`(cload string)` [*function with one argument*] 14-6

`C_LL_FIX` [*Fonction C*] 14-14

`LL_C_FIX` [*Fonction C*] 14-14

`C_LL_FLOAT` [*Fonction C*] 14-15

`LL_C_FLOAT` [*Fonction C*] 14-15

Chapter 15

Virtual terminal

To free the user from the explicit management of all the different kinds of alphanumeric TTYs that exist, LE-LISP has included the notion of a **virtual terminal**, which is composed of a set of variables and functions that describe the functionalities of a certain 'ideal' terminal type. A large number of physical terminals have LE-LISP virtual-terminal descriptions. The virtual terminal is initialized by the `initty` function, which loads a file from the library of LE-LISP virtual terminals. If a terminal has no LE-LISP virtual-terminal description, `initty` compiles the description using other terminal-description databases such as `termcap` and `terminfo`.

`(initty symb)` [*function with an optional argument*]

Initializes the virtual terminal for a terminal of type `symb`. If no argument is given, the physical terminal type is determined first by a call to `(getenv "TERM")`, and next by using the name of the system returned by `(system)`. If virtual terminal initialization fails, the `errvirtty` error is raised, and its default screen display is printed as follows:

```
** <fnt> : unknown terminal : <symb>
```

Here, `fnt` is the name of the function that caused the error—`termcap` or `terminfo`—and `symb` is the name of the unknown terminal. The following messages are displayed when a terminal type cannot be found:

```
? (initty 'alogon)
? ; attempt to make the virtty file using the termcap entry for: alogon
** termcap : unknown terminal type: alogon
? ; attempt to make the virtty file using the terminfo entry for: alogon
** terminfo : unknown terminal type: alogon
= alogon
```

`#:system:virtty-directory` [*variable*]

This variable contains the name of the directory where the LE-LISP virtual-terminal description files are located. The default extension attached to these files names is the value of the `#:system:lelisp-extension` variable.

#:system:termcap-file [*variable*]

This variable contains the name of the file with the `termcap`-format terminal descriptions. If the `TERMCAP` environment variable has a value, obtained from `(getenv "TERMCAP")`, then this is used instead of the value in `#:system:termcap-file`.

#:system:terminfo-directory [*variable*]

This variable contains the name of the directory where the `terminfo`-format terminal description files are stored. If the `TERMINFO` environment variable has a value, obtained from `(getenv "TERMINFO")`, then this is used instead of the value in `#:system:termcap-file`.

termcap [*feature*]

This feature indicates that the LE-LISP virtual terminal description-to-`termcap` format translator has been loaded into memory. This translator is found in the standard library under the name `termcap`.

terminfo [*feature*]

This feature indicates that the LE-LISP virtual terminal description-to-`terminfo` format translator has been loaded into memory. This translator is found in the standard library under the name `terminfo`.

15.1 Virtual terminal functions

The virtual terminal functions can be divided into three groups: the *standard* functions, the *required* functions, and the *optional* functions.

- The **standard functions** provide the minimal level of interaction with the LE-LISP system by implementing basic write-character-to-screen and read-character-from-keyboard functionalities. These functions are also used by the `bol`, `eol` and `flush` LISP functions to handle dialogue with the terminal.
- The **required functions** are used by full-screen tools such as the `pepe` full-screen editor, the multi-window interface and games.
- The **optional functions** perform more sophisticated video functions, but are not used by functions in the standard library.

The action of the functions described here varies according to the kind of terminal actually connected to the system. When a terminal cannot bring about the desired effect, the `tyerror` function is raised. This function returns `()` by default, and can be redefined to generate an error or to try to perform the desired action in another way.

(tyerror list) [*function with one argument*]

This function is automatically called by the virtual terminal functions when it is not possible to obtain a particular effect on a connected terminal. The argument `list` is the name of the function call that failed. The default behavior of `tyerror` is to return `()`, which is compatible with version 15.

15.1.1 Standard functions

(tyi) [*function with no arguments*]

Reads a character from the keyboard and returns its internal character code.

(tys) [*function with no arguments*]

Reads the keyboard. If a character is waiting to be read, it is returned in the same manner as with the `tyi` function. If there are no outstanding characters, `tys` will return `()` immediately.

For example, we might decide to increment a counter until a character is typed:

```
(let ((counter 0))
  (until (tys)
         (incr counter))
  counter)
```

The `tyi` can be defined in terms of `tys`:

```
(defun tyi ()
    (let ((c))
         (until (setq c (tys)))))
```

(tyinstring string) [*function with one argument*]

Reads a line from the keyboard and stores the input characters in `string`, a character string. It returns the number of characters read as its value. A line longer than the length of `string` will be truncated to its length. This function should be used when the `#:system:line-mode-flag` flag is set. (See the definition of the `bol` function, for example.)

(tycn cn) [*function with one argument*]

Displays the character whose internal character code is `cn`. The character is sent immediately to the screen.

(tystring string length) [*function with two arguments*]

Sends the `length` first characters of `string` to the screen.

In LE-LISP, `tystring` could be defined in the following manner:

```
(defun tystring (s l)
    (for (i 0 1 (sub1 (min (slen s) l)))
         (tycn (sref s i))))
```

(tynewline) [*function with no arguments*]

Sends an end-of-line marker to the screen.

(tyback cn) [*function with one argument*]

Erases the character whose internal code is `cn`, which is assumed to be immediately before the cursor on the screen, and moves the cursor back one space. If the character just before the cursor does not have internal code `cn`, the effect on the screen is undefined. This function is used by the `bol` function to erase a character when the user types an erase character like `delete` or `backspace`. The character code argument allows the line editor to work with proportionally-spaced characters.

For example, the following sequence leaves the screen unchanged:

```
(tycn #/a)
(tyback #/a)
```

Generally, the following definition produces the desired effect:

```
(defun tyback (cn)
    (tycn #\bs)
    (tycn #\sp)
    (tycn #\bs))
```

15.1.2 Required functions

The printing functions described in this section print characters to the virtual terminal's output channel. This channel is distinct from the ordinary terminal channel and has the 'number' `t` as its index. When the current channel is `t`, prints resulting from the action of the `prin`, `print` and `princn` functions and character emissions by the `tyo` and `tyod` functions occur in the same buffer.

The `t` channel is an output channel with the same status as the `()` channel. Its right margin is set just after the end of the buffer. Recall the familiar formula: `(1+ (slen (outbuf)))`. Therefore, there is never an `eol` programmable interrupt on the `t` channel, but only the `flush` interrupt when the write position reaches the end of the buffer.

The `tyflush` function triggers a `flush` interrupt on the channel of the virtual terminal. Furthermore, the buffer is emptied by the system after each read performed by a `tyi`, `tys` or `tyinstring` function.

(tyo o_1 ... o_n) [*function with a variable number of arguments*]

Prints the characters corresponding to the objects $o_1 \ldots o_n$ into the virtual terminal's buffer. Each object o_i can be one of the following:

- An internal character code.
- A character string.
- A list of internal character codes.

In LE-LISP, `tyo` could be defined in the following manner:

```
(defun tyo objs
    (with ((outchan t))
          (while objs (tyo1 (nextl objs)))))

(defun tyo1 (o1)
    (cond ((consp o1) (mapc 'princn o1))
          ((stringp o1) (prin o1))
          ((fixp o1) (princn o1))))
```

(tyflush) [*function with no arguments*]

Empties the virtual terminal's buffer by causing a `flush` programmable interrupt on the channel `t`. The `flush` function is invoked by this programmable interrupt, and prints the contents of the buffer to the screen.

In LE-LISP, `tyflush` could be defined in the following manner:

```
(defun tyflush ()
    (with ((outchan t))
          (itsoft 'flush ())))
```

(tyod n nc) [*function with two arguments*]

Prints the base-10 `nc`-character representation of the number `n` into the virtual terminal buffer.

The functions described next give full-screen editing control over the screen. They are required for the use of various tools furnished with the system; for example, the `pepe` full-screen editor. If these functions send escape sequences or other control characters or sequences to the screen, they must use the virtual terminal output channel via the `tyo` function to ensure the proper synchronization of full-page functions.

(typrologue) [*function with no arguments*]

Activates the terminal in full-page mode. This function must be called before any calls to the following functions.

(tyepilogue) [*function with no arguments*]

Returns the screen to normal scrolling mode.

(tyxmax) [*function with no arguments*]

Returns the zero-based number of columns currently printable on the terminal. Generally, `(tyxmax)` return 79 for 80-column terminals.

This function in fact returns the value of the `xmax` global variable of the `#:sys-package:tty` package.

In LE-LISP, `tyxmax` could be defined in the following manner:

```
(defun tyxmax ()
    (symeval (getsymb #:sys-package:tty 'xmax ())))
```

(tyymax) [*function with no arguments*]

Returns the zero-based number of lines currently printable on the terminal. Generally, `(tyymax)` return 23 for 24-column terminals.

(tycursor x y) [*function with two arguments*]

Places the cursor in column position `x` on line `y`. The upper-lefthand-most character position, in the corner of the screen, has the index `(0,0)`.

(tybs cn) [*function with one argument*]

Moves the cursor position back one place without erasing anything on the screen. The argument specifies the character immediately preceding the cursor on the screen. This lets programs back up over proportionally-spaced characters.

In general, the following definition produces the desired effect:

```
(defun tybs (cn)
    (tyo #\bs))
```

(tycr) [*function with no arguments*]

Places the cursor at the beginning of the current line.

In general, the following definition produces the desired effect:

```
(defun tycr ()
    (tyo #\cr))
```

(tycls) [*function with no arguments*]

Erases the entire screen.

(tybeep) [*function with no arguments*]

Triggers the alarm or bell of the terminal.

`(tyleftkey)` [*function with no arguments*]

Returns the key code associated with the *left arrow* '←' key. By default, this code is `#^B`.

`(tyrightkey)` [*function with no arguments*]

Returns key code associated with the *right arrow* '→' key. By default, this code is `#^F`.

`(tyupkey)` [*function with no arguments*]

Returns the key code associated with the *up arrow* '↑' key. By default, this code is `#^P`.

`(tydownkey)` [*function with no arguments*]

Returns the key code associated with the *down arrow* '↓' key. By default, this code is `#^N`.

15.1.3 Optional functions

`(tycleol)` [*function with no arguments*]

Erases the screen and places the cursor at the end-of-line position.

`(tycleos)` [*function with no arguments*]

Erases the screen and places the cursor at the end-of-screen position.

`(tyinscn cn)` [*function with one argument*]

`(tyinsch ch)` [*function with one argument*]

These two functions, which are identical in their behavior, insert the character whose internal character code is `cn` at the current cursor position.

`(tydelcn cn)` [*function with one argument*]

`(tydelch)` [*function with no arguments*]

These two functions erase the character at the current cursor position. The `tydelcn` function takes in addition a character code `cn` which specifies the character to be erased, to facilitate the treatment of proportionally-spaced characters. If the character at the current cursor position does not have internal code `cn`, the effect on the screen is undefined.

In LE-LISP, `tydelcn` could be defined in the following manner:

```
(defun tydelcn (cn)
    (tydelch))
```

(tyinsln)

[function with no arguments]

Inserts a new line at the current cursor position. Lines below the current line are scrolled down and the last line (if it was not empty) disappears.

(tydelln)

[function with no arguments]

Erases the line at the current cursor position. Lines below the current line are scrolled up and the last line on the screen becomes blank.

(tyattrib i)

[function with an optional argument]

If the `i` flag is true—that is, if it is different to `()`—`tyattrib` activates attribute mode. Otherwise, when it is equal to `()`, the attribute mode will be turned off. Called with no arguments, `tyattrib` returns the current value of attribute mode. Attribute mode highlights text on the screen and its exact nature depends on the terminal. It can be implemented as underlining, blinking or reverse video.

This function can be used with the `with` control structure to modify the attribute in a local context:

```
(with ((tyattrib t))
      (tyo "hello"))

(defun tyattrib x
    (ifn x
         (symeval (getsymb #:sys-package:tty 'tyattrib ()))
         (funcall (getfn #:sys-package:tty 'tyattrib ()) (car x))
         (set (getsymb #:sys-package:tty 'tyattrib ()) (car x))))
```

(tyshowcursor i)

[function with an optional argument]

If the `i` flag is true—that is, different to `()`—`tyshowcursor` activates cursor display on the terminal. If `i` is false—if it is equal to `()`—then `tyshowcursor` deactivates cursor display. Called with no arguments, the function returns the current value of the cursor display flag.

This function can be used with the `with` control structure to modify the cursor display in a local context:

```
(with ((tyshowcursor ()))
      (vdt))
```

(tyco x y cn_1 ... cn_n)

[function with a variable number of arguments]

The form (tyco x y cn_1 ... cn_n) is equivalent to

```
(progn (tycursor x y) (tyo cn1 ... cnN))
```

It places the cursor at position `(x, y)`, then prints the characters with codes $\texttt{cn}_1 \ldots \texttt{cn}_n$ into the terminal buffer using the `tyo` function.

(tycot `x y` $\texttt{cn}_1 \ldots \texttt{cn}_n$**)** [*function with a variable number of arguments*]

The form `(tycot x y` $\texttt{cn}_1 \ldots \texttt{cn}_n$`)` is equivalent to

```
(with ((tyattrib t))
      (tyco x y cn1 ... cnN))
```

It places the cursor at position `(x, y)`, shifts into attribute mode, prints the characters with codes $\texttt{cn}_1 \ldots \texttt{cn}_n$ into the terminal buffer using the `tyo` function, and shifts back into the previous mode.

15.2 Screen functions

A *screen* is a matrix of characters, with a width `w` and a height `h`, that is stored in the form of a character string. The following functions determine the differences between two screen images, so that asynchronous screen updates can be performed. They also let you place and manage sub-screens within a screen.

(redisplayscreen `sn so w h`**)** [*function with four or twelve arguments*]

Suppose that `sn` and `so` are character strings that hold screen representations that are `w` wide by `h` high. `redisplayscreen` will update the terminal, using the `tycursor` and `tyo` functions described above, by sending sequences representing the differences between the old screen, `so`, and the new one, `sn`. At the end of the update, `so` will have been physically modified to contain all the characters of `sn`, so that calls to `redisplayscreen` can be iterated.

It is possible to add eight more arguments, which have the same meaning here as they do in the context of the next funtion.

(bltscreen `sd ss w h`**)** [*function with four or twelve arguments*]

Moves characters from screen `ss` onto screen `sd`. The screens are `w` characters wide and `h` characters high. If this function is called with only four arguments, then the sizes of the screens are considered to be identical. If twelve arguments are used, a sub-screen can be placed anywhere on the main screen. The following is a description of these twelve arguments:

- `sd`: destination screen.
- `ss`: source screen.
- `wd`: width of the destination screen.
- `hd`: height of the destination screen.
- `ws`: width of the source screen.

- `hs`: height of the source screen.
- `xd`: x-coordinate of the destination sub-screen.
- `yd`: y-coordinate of the destination sub-screen.
- `xs`: x-coordinate of the source sub-screen.
- `ys`: y-coordinate of the source sub-screen.
- `wt`: width of the sub-screen moved.
- `ht`: height of the sub-screen moved.

Here is a demonstration of the combined use of the two screen-manipulation functions that have just been described:

```
(defun bltdemo ()
    (let ((nscreen (makestring (* 80 24) #\sp))
          (oscreen (makestring (* 80 24) #\sp)))
         (for (i 0 80 240)
              (bltstring nscreen i        "******")
              (bltstring nscreen (+ i 12)  "------")
              (bltstring nscreen (+ i 24) "......"))
         (tycls)
         (while t
                (redisplayscreen nscreen oscreen 80 24)
                (for (i 0 6 30)
                     (bltscreen nscreen nscreen
                          80 24 80 24
                          (random 0 80) (random 4 19)
                          i 0 6 4)))))
```

15.3 Using the virtual terminal

There are three demonstration programs for the virtual terminal:

- `hanoi` simulates the well-known Tower of Hanoi puzzle.
- `whanoi` does the same, but with asynchronous screen refresh.
- `vdt` simulates the difficult life of an earthworm.

They are found in the files `hanoi`, `whanoi` and `vdt` in the standard library. They are easy to read, and serve as guides to using the terminal directly, the virtual terminal and the asynchronous refresh functions.

`(hanoi n)` *[function with one argument]*

Animates the Towers of Hanoi puzzle using `n` disks. This `n` must lie between three and nine, inclusive. This function is loaded automatically, the first time it is called.

(hanoiend) [*function with no arguments*]

Reclaims space used by the functions comprising the game described immediately above.

(whanoi n) [*function with one argument*]

This function is similar to the `hanoi` function, except that it uses the asynchronous redisplay provided by the `redisplayscreen` function.

(whanoiend) [*function with no arguments*]

Reclaims space used by the functions comprising the game described immediately above.

(vdt) [*function with no arguments*]

Runs an animated game that simulates the growth of an earthworm. Use the arrow keys on the terminal to direct the growth of the worm. This function is loaded the first time that it is called.

(vdtend) [*function with no arguments*]

Reclaims space used by the functions comprising the game described immediately above.

15.4 Defining a virtual terminal

#:sys-package:tty [*variable*]

All the virtual-terminal functions call functions whose names are computed according to the following scheme:

```
(getfn #:sys-package:tty <name of the function> ())
```

The `tycn` function, for example, is defined as follows:

```
(defun tycn (cn)
    (funcall (getfn #:sys-package:tty 'tycn ()) cn))
```

At system initialization, the `#:sys-package:tty` variable has the value `tty`. The functions that implement the standard interface are therefore searched for in the `tty` package. The functions in this package—such as `#:tty:tycn`, `#:tty:tyi` and `#:tty:tyo`—are the functions that implement the physical interface to the user terminals.

There are two things that you have to do when defining a virtual terminal:

- Assign the appropriate value to the `#:sys-package:tty` global variable.
- Define the required functions in the package whose value is that of `#:sys-package:tty`.

The `#:sys-package:tty` variable must always point to a sub-package of the `tty` package, to permit the use of the virtual terminal default behaviors.

These default behaviors are implemented by the following functions:

- `tyi`, `tys` and `tyinstring`: empty the terminal buffer after reads.
- `tyo`, `tyod`, `tyco` and `tycot`: govern the use of the virtual terminal buffer.
- `tyinsch`, `tydelch` and other optional functions: call the `tyerror` function.
- `tyerror`: returns `()`.
- `redisplayscreen`: determines the difference between two screens, and sends sequences to the screen using the `tyo` and `tycursor` functions.

Usually, for a specific terminal, it suffices to redefine the functions that handle insertion and deletion of lines and characters. See, however, the example of a virtual terminal of type `#:tty:window` in the `virbitmap` file of the standard library. It redefines the `tyi`, `tys`, `tycn`, `tynewline` and `redisplayscreen` functions.

As an example, here is the definition of a virtual terminal for the H19 terminal:

```
; Le-Lisp version 15.2 : compilation of the virtual terminal : H19

(setq #:sys-package:tty '#:tty:h19)

(defvar #:tty:h19:xmax 79)

(defvar #:tty:h19:ymax 23)

(defun #:tty:h19:tycursor (col line)
    (#:tty:tyo 27 89 (+ 32 line) (+ 32 col)))

(defun #:tty:h19:tycls ()
    (#:tty:tyo 27 69))

(defun #:tty:h19:tycleol ()
    (#:tty:tyo 27 75))

(defun #:tty:h19:tycleos ()
    (#:tty:tyo 27 74))

(defun #:tty:h19:tydelch ()
    (#:tty:tyo 27 78))

(defun #:tty:h19:tyinsln ()
    (#:tty:tyo 27 76))

(defun #:tty:h19:tydelln ()
    (#:tty:tyo 27 77))

(defun #:tty:h19:tyattrib (x)
    (if x (#:tty:tyo 27 112) (#:tty:tyo 27 113)))

(defvar #:tty:h19:tyattrib ())
```

```
(defun #:tty:h19:tyinsch (arg)
    (#:tty:tyo 27 64 arg 27 79))

(defun #:tty:h19:typrologue ()
    (#:tty:h19:tycls))

(defun #:tty:h19:tyepilogue ()
    (tycursor 0 (sub1 #:tty:h19:ymax)))

(defvar #:tty:h19:tyshowcursor t)
```

Table of contents

15 Virtual terminal **15-1**

15.1 Virtual terminal functions 15-2

15.1.1 Standard functions 15-3

15.1.2 Required functions 15-4

15.1.3 Optional functions 15-7

15.2 Screen functions 15-9

15.3 Using the virtual terminal 15-10

15.4 Defining a virtual terminal 15-11

Function Index

`(initty symb)` [*function with an optional argument*] 15-1
`#:system:virtty-directory` [*variable*] 15-1
`#:system:termcap-file` [*variable*] 15-2
`#:system:terminfo-directory` [*variable*] 15-2
`termcap` [*feature*] 15-2
`terminfo` [*feature*] 15-2
`(tyerror list)` [*function with one argument*] 15-3
`(tyi)` [*function with no arguments*] 15-3
`(tys)` [*function with no arguments*] 15-3
`(tyinstring string)` [*function with one argument*] 15-3
`(tycn cn)` [*function with one argument*] 15-3
`(tystring string length)` [*function with two arguments*] 15-3
`(tynewline)` [*function with no arguments*] 15-4
`(tyback cn)` [*function with one argument*] 15-4
`(tyo` o_1 ... o_n`)` [*function with a variable number of arguments*] 15-4
`(tyflush)` [*function with no arguments*] 15-5
`(tyod n nc)` [*function with two arguments*] 15-5
`(typrologue)` [*function with no arguments*] 15-5
`(tyepilogue)` [*function with no arguments*] 15-5
`(tyxmax)` [*function with no arguments*] 15-6
`(tyymax)` [*function with no arguments*] 15-6
`(tycursor x y)` [*function with two arguments*] 15-6
`(tybs cn)` [*function with one argument*] 15-6
`(tycr)` [*function with no arguments*] 15-6
`(tycls)` [*function with no arguments*] 15-6
`(tybeep)` [*function with no arguments*] 15-7

`(tyleftkey)` [*function with no arguments*] 15-7
`(tyrightkey)` [*function with no arguments*] 15-7
`(tyupkey)` [*function with no arguments*] 15-7
`(tydownkey)` [*function with no arguments*] 15-7
`(tycleol)` [*function with no arguments*] 15-7
`(tycleos)` [*function with no arguments*] 15-7
`(tyinscn cn)` [*function with one argument*] 15-7
`(tyinsch ch)` [*function with one argument*] 15-7
`(tydelcn cn)` [*function with one argument*] 15-7
`(tydelch)` [*function with no arguments*] 15-7
`(tyinsln)` [*function with no arguments*] 15-8
`(tydelln)` [*function with no arguments*] 15-8
`(tyattrib i)` [*function with an optional argument*] 15-8
`(tyshowcursor i)` [*function with an optional argument*] 15-8
`(tyco x y` $\texttt{cn}_1$ `...` $\texttt{cn}_n$`)` [*function with a variable number of arguments*] 15-8
`(tycot x y` $\texttt{cn}_1$ `...` $\texttt{cn}_n$`)` [*function with a variable number of arguments*] 15-9
`(redisplayscreen sn so w h)` [*function with four or twelve arguments*] 15-9
`(bltscreen sd ss w h)` [*function with four or twelve arguments*] 15-9
`(hanoi n)` [*function with one argument*] 15-10
`(hanoiend)` [*function with no arguments*] 15-11
`(whanoi n)` [*function with one argument*] 15-11
`(whanoiend)` [*function with no arguments*] 15-11
`(vdt)` [*function with no arguments*] 15-11
`(vdtend)` [*function with no arguments*] 15-11
`#:sys-package:tty` [*variable*] 15-11

Chapter 16

Full-page editor

A virtual terminal—as described in the previous chapter—can be used to construct a **full-page editor**. In the context of LE-LISP, the file `pepe` in the standard library presents us with an editor of this kind. The use of LE-LISP as the implementation language for such an editor facilitates development and debugging, and gives rise to an *extensible* editor.

16.1 Functions to call the full-page editor

`pepe` [*feature*]

This flag indicates whether the full-page editor, called `pepe`, has been loaded into memory.

`(pepe f)` [*special form*]

This function calls the `pepe` editor on the object `f`, which is not evaluated. There is an easier way to do the same thing, using the macro character CONTROL-`E` (which we often write as `^E`). Typing `^E` and then `thing`—with no blank between `^E` and `thing`—corresponds to the call `(pepe thing)`.

- If the `thing` object is `()`, the current buffer is edited.
- If the object is `t`, editing begins on a new buffer named `tmp`.
- If the object is an *atom*, the file of the same name is edited.
- If the object is a *list*, it is evaluated and editing begins on a buffer named `tmp` that contains everything that was printed as a result of this evaluation.

Let us look at a few examples. To edit a disk file named `foo.ll`, type

```
^Efoo.ll
```

To edit the pretty-printed versions of the `foo` and `bar` functions, type

```
^E(pretty foo bar)
```

To edit the sorted list of system functions, type

```
^E(mapc 'print (sortl (maploblist 'typefn)))
```

(pepefile f) *[function with one argument]*

This function is similar to the previous one, except that it evaluates its argument.

(pepend) *[function with no arguments]*

Allows you to regain space taken up by the editor functions. The pepe flag disappears, and any succeeding call to pepe will reload these functions. The editor buffer is also deallocated.

16.2 Full-page editor commands

CONTROL-A		Go to the beginning of the line
CONTROL-B	←	Go back one character
CONTROL-C		Quit the pepe editor; reenter with ^E
CONTROL-D		Erase the current character
CONTROL-E		Go to the end of the line
CONTROL-F	→	Go forward one character
CONTROL-G		Kill the current command
CONTROL-K		Erase the line containing the cursor
CONTROL-L		Redisplay the entire screen
CONTROL-M	↩	Break the line at the cursor position
CONTROL-N	↓	Go to the next line
CONTROL-O		Break the line at the cursor level
CONTROL-P	↑	Go to the preceding line
CONTROL-S		Search for a string
CONTROL-V		Go to the next screen
CONTROL-Y		Insert, at the current cursor position, the last line erased with ^K
DELETE		Erase the character to the left of the cursor
ESCAPE-E		Execute the contents of the current buffer
ESCAPE-F		Change the name of the current file
ESCAPE-I		Insert the contents of another file
ESCAPE-R		Read a new file into the buffer
ESCAPE-S		Save the current buffer into the current file
ESCAPE-V		Go to the previous screen
ESCAPE-W		Write the current buffer into a file
ESCAPE-X		Call a pepe function directly
ESCAPE-Z		Save the current buffer onto disk and load it into memory
ESCAPE-)		Place the cursor on the next) character, in a LISP style
ESCAPE-]		Place the cursor on the next] character, in a LISP style
ESCAPE->		Go to the beginning of the buffer
ESCAPE-<		Go to the end of the buffer
ESCAPE-?		Display the present pepe help list

16.2.1 Extensions to the full-page editor

An interesting feature of `pepe` is its extensibility. Extensions can be made simply by assigning new operations to keystrokes. The following two functions associate LE-LISP expressions with keystrokes.

`(defkey key` e_1 ... e_n`)` [*special form*]

Associates the expressions e_1 ... e_n with the value of `key`.

`(defesckey key` e_1 ... e_n`)` [*special form*]

Associates the expressions e_1 ... e_n with the sequence of keystrokes ESCAPE-`key`.

Table of contents

16 Full-page editor **16-1**

16.1 Functions to call the full-page editor 16-1

16.2 Full-page editor commands 16-2

16.2.1 Extensions to the full-page editor 16-3

Function Index

`pepe` [*feature*] 16-1

`(pepe f)` [*special form*] 16-1

`(pepefile f)` [*function with one argument*] 16-2

`(pepend)` [*function with no arguments*] 16-2

`(defkey key` e_1 `...` e_n`)` [*special form*] 16-3

`(defesckey key` e_1 `...` e_n`)` [*special form*] 16-3

Chapter 17

Terminal-based line editor

The **terminal-based command-line editor** called `edlin` is available in interactive mode from the keyboard. Using this editor, you do not need to retype long or complicated commands. The editor facilitates all the usual editing operations on input lines, and it can carry out automatic searches for symbols.

The `edlin` tool is not a full-page editor, since it works on one line at a time. It does not use cursor-addressing functions, and it scrolls the screen when necessary. The following virtual-terminal primitives are used to insert and erase characters, erase end-of-lines, and move the cursor around within a line:

```
tyinscn
tydelcn
tycleol
tycr
tybs
tynewline
```

The `edlin` tool is written so that it can be used with proportionally-spaced fonts. The functions associated with this editor are found in the standard library named `edlin`.

17.1 Loading the terminal-based line editor

`edlin` [*feature*]

This flag indicates whether the `edlin` line editor has been loaded into memory.

`(edlin)` [*function with no arguments*]

If the `edlin` line editor has not already been loaded and activated, this function performs these tasks.

The editor—available as soon as this function has been called—manages the LE-LISP reader. This means that the editing capability of `edlin` can be available throughout an entire session, regardless of the application that carries out reading.

The `edlin` editor provides on-line help, which can be obtained by hitting the ESCAPE key followed by a question mark.

17.2 Terminal-based line-editor commands

Cursor movement of an EMACS kind:

CONTROL-F	Move forward one character
CONTROL-B	Move back one character
ESCAPE-F	Move forward one word
ESCAPE-B	Move back one word
CONTROL-A	Move to the beginning of the line
CONTROL-E	Move to the end of the line

Deletions (the DELETE key can be used in place of BACKSPACE):

CONTROL-D	Erase the next character
BACKSPACE	Erase the previous character
ESCAPE-D	Erase the next word
ESCAPE-BACKSPACE	Erase the previous word
CONTROL-U	Erase to the beginning of the line (see `^Y`)
CONTROL-K	Erase to the end of the line (see `^Y`)
CONTROL-Y	Reinsert characters erased by `^U` or `^K`

Command history:

CONTROL-I	Recall the previous command
CONTROL-N	Recall the next command
ESCAPE-<	Beginning of history
ESCAPE->	End of history
ESCAPE-H	Show history
ESCAPE-ESCAPE	Retrieve a previous command that begins like the current one

End of line:

RETURN *or* LINEFEED	Send the line to LISP and save it in the history
CONTROL-O	Send the line to LISP and skip to the next line in the history

Shortcuts:

ESCAPE-SPACEBAR	Complete the symbol preceding the cursor, or show all completion possibilities
ESCAPE-'	Quote the current symbol

Miscellaneous:

ESCAPE-*number command*	Repeat *command number* times
CONTROL-L	Redisplay the line
\	Quote character: forces literal interpretation of the following character
ESCAPE-CONTROL-G	Illegal sequence: useful when ESCAPE has been hit by mistake
ESCAPE-?	`edlin` help
CONTROL-T	Translate: that is, interchange the two previous characters
`(edlin)`	Run `edlin`
`(edlinend)`	Kill `edlin`

Commands that are also LE-LISP macro characters—such as `^L`, for example—are not interpreted as macros at the beginning of a command line. Consequently, the quote character \ does not need to be used in these cases.

Table of contents

17 Terminal-based line editor **17-1**

17.1 Loading the terminal-based line editor 17-1

17.2 Terminal-based line-editor commands 17-2

Function Index

`edlin` [*feature*] 17-1
`(edlin)` [*function with no arguments*] 17-1

Chapter 18

Virtual bitmap display

The LE-LISP system provides a portable graphics facility called the **Virtual Bitmap Display**, often referred to as VBD. It is capable of managing—independently of the underlying hardware—a high-resolution bit-mapped screen and a pointing device. This graphics facility provides LISP functions that construct windows, draw into these windows, and react to pointing-device movements. These three aspects are relatively independent. They are described in chapters 18, 19, and 20 of the present Reference Manual.

The Virtual Bitmap Display facility has been ported to a variety of hardware devices using different window systems, servers and managers.

A version running on classic alphanumeric terminals, based on the virtual terminal facility, is available in the standard LE-LISP system configured with the `bvtty` feature. This version is quite limited due to the low resolution of alphanumeric terminals.

18.1 Loading the description file

The VBD functions are parameterized by configuration files describing the various screen managers to which the system has been ported. These files are stored in the `virbitmap` directory of the standard LE-LISP distribution.

Before the VBD functions can be used, the description file corresponding to the relevant screen manager must be loaded. This is done by the `inibitmap` LISP function. This operation should be carried out only once per LISP session. It is possible, of course, to save a core image file after this operation has been performed. When the image is restored, the description file does not need to be reloaded.

On some systems, and in particular on most UNIX systems, loading a screen manager requires the use of a LISP executable that includes low-level primitives that read from and write to the screen and the pointing device. For more details, see the manual describing the LE-LISP implementation on your particular system.

`(inibitmap symbol)` [*function with an optional argument*]

Loads the screen manager description file. `symbol` is the name of the kind of screen manager to be loaded.

Here are the names that are currently recognized:

- `X11`, to load the X WINDOW System, versions `X11r2`, `X11r3` or `X11r4`.
- `mac`, to load the Macintosh ToolBox.
- `uis`, to load a VaxStation with UIS.
- `bvtty`, to load the minimal VBD for alphanumeric terminals.

If `symbol` is not given, the value of the `BITMAP` environment variable is used. (See the `getenv` function.)

Here, for example, is the loading of an X WINDOW System `11r`*x* description file:

```
? (inibitmap '|X11|)
= X11
```

Note that several different description files can be loaded in succession. This might be necessary if different screen types are to be managed during a single session.

```
? (inibitmap 'bvtty)
= bvtty
```

At this stage, `X11` and `bvtty` are both available in the LE-LISP environment.

18.2 Screen preparation

A so-called *screen* is a LISP structure that allows you to manipulate a physical screen device. A physical screen device normally includes a bitmapped video display tube, a keyboard and a mouse.

The VBD functions can build screens on different physical screen devices simultaneously, since these functions work on the logically current screen, which can be read from and written to by the `current-display` function.

18.2.1 Managing a single screen

If only a single screen is to be used in a LISP session, the first call to one of the VBD functions will automatically allocate a screen of the type loaded by `inibitmap`, and will set the current screen variable to point to it. In this case, it is not necessary to call the `bitprologue` and `current-display` functions, since the work they do will have already been done.

Here, for example, is the creation of a window:

```
? (create-window 'window 0 0 100 100 "fenestre" 0 1)
= #<window fenestre>
```

18.2.2 Managing several screens

If several different screens are to be managed simultaneously, `bitprologue` has to be called, to create them explicitly. The `current-display` function lets you move from one screen to another.

`(bitprologue name device)` [*function with one or two optional arguments*]

Creates, initializes and returns as its result a screen for the screen manager of type `name`. `name` is one of the symbols recognized by the `inibitmap` function. If `name` is not supplied, the name of the last window manager loaded by `inibitmap` is used.

The screen returned by this function can be used as an argument in the `screen` position in the functions described below.

`device` is a supplementary argument that depends on the screen manager being used. For the X WINDOW System environments (`X11`), this argument is a character string describing the connection in the standard X format. If the `device` argument is not supplied, the value of the `DISPLAY` environment variable is used.

`(current-display screen)` [*function with an optional argument*]

Returns the current screen. If an argument is supplied, the current display is changed to be the argument value. This current screen is involved in the following operations:

- Windows are created on the current screen by means of `create-window` and `make-window`.
- Icons are created on the current screen by means of `create-bitmap`.
- Fonts are created on the current screen by means of `load-font`.
- Colors are created on the current screen by means of `make-color`, `make-named-color` and `make-mutable-color`.
- Cursors are created on the current screen by means of `make-cursor`.
- Patterns are created on the current screen by means of `make-pattern`.
- Events are read on the current screen by means of `read-event`, `peek-event` and `eventp`.
- Windows are found on the current screen by means of `find-window`.

The current window is an implicit argument for the VBD drawing functions.

In the following examples, we are dealing with machines named `london` and `paris`. We first open the screen of the machine called `london`:

```
? (setq london (bitprologue '|X11| "london:0.0"))
= #<#:display:X11 X11 london:0.0>
```

We then open the screen on the machine called `paris`:

```
? (setq paris (bitprologue '|X11| "paris:0.0"))
= #<#:display:X11 X11 paris:0.0>
```

We now open a window on `london`:

```
? (current-display london)
= #<#:display:X11 X11 paris:0.0>
? (setq london-window (create-window 'window 0 0 100 100 "london" 0 1))
= #<window london>
```

Next, we open a window on `paris`:

```
? (with ((current-display paris))
?       (setq paris-window (create-window 'window 0 0 100 100 "paris" 0 1)))
= #<window paris>
```

We draw something on `london`:

```
? (with ((current-window london-window))
?       (draw-line 0 100 100 0))
= 0
```

Finally, we draw something on `paris`:

```
? (with ((current-display paris)
?        (current-window paris-window))
?       (draw-line 0 0 100 100))
= 0
```

(bitepilogue screen) *[function with an optional argument]*

Closes `screen` by killing all its open windows and all the icons, fonts, colors, cursors and patterns that might have been created within it. It also empties the queue of events waiting on this screen. You cannot use `screen` as an argument to the `current-display` function after this call. If the `screen` argument is not supplied, the current screen is closed.

(bitmap-save screen) *[function with an optional argument]*

Closes `screen` in the same way as the previous function, but stores in `screen` the set of objects that were created within it. It also returns the saved screen as its value. The saved `screen` can be passed later to the `bitmap-restore` function, in order to automatically reconstruct all the objects it contained before the save. Warning: The contents of windows are not saved, but each window receives a `repaint-window-event` event as the restore comes to completion. If the `screen` argument is not supplied, the current screen is saved.

Used in conjunction with the `save-core` function, `bitmap-save` makes it possible to save LISP core image files containing windows.

```
? (let ((saved-screen (bitmap-save)))
?      (save-core "image.core")
?      (bitmap-restore saved-screen))
```

(bitmap-restore screen) [*function with one argument*]

The `screen` argument is a screen saved by the previous function. The function initializes `screen` and reconstructs all the objects—windows, icons, fonts, cursors, colors and patterns—that were within it when it was saved. The contents of windows are not saved, but each window receives a `repaint-window-event` event as the restore comes to completion.

18.3 Functions on screens

The functions in this section report data concerning the current screen: its size, fonts, colors, cursors and patterns, and its predefined windows, if any.

These functions can take an optional `screen` argument. If this argument is not supplied, they operate on the current screen.

(bitxmax screen) [*function with an optional argument*]

(bitymax screen) [*function with an optional argument*]

These functions return the dimensions of the screen. The value of `bitxmax` is the width of the screen minus one, and the value of `bitymax` is its height minus one. The units are those of the elementary display points (pixels) of the device.

For an Apollo screen, these functions return the following values:

```
(bitxmax)  ⟹   1023
(bitymax)  ⟹   799
```

(root-window screen) [*function with an optional argument*]

Returns the window associated with the screen background. Drawing carried out in this window writes over all the other windows posted to the screen.

The root window can be used to follow the mouse on the whole screen: for example, when a (non-root) window is being moved or resized.

Operations in the root window should be as non-destructive as possible—using `xor` for drawing and then erasing images, for example—so that the contents of other windows will not be perturbed.

The following visually-striking function, for example, blinks the whole screen once:

```
(defun flash-screen ()
    (with ((current-window (root-window)))
          (with ((current-mode 6))
                (fill-rectangle 0 0 (bitxmax) (bitymax))
                (bitmap-flush)
                (fill-rectangle 0 0 (bitxmax) (bitymax))
                (bitmap-flush))))
```

(all-colors screen) [*function with an optional argument*]

Returns the list of colors that have been allocated by the `make-color`, `make-named-color` and `make-mutable-color` functions. (See the descriptions of these functions.)

(bitmap-refresh screen) [*function with an optional argument*]

Requests that `screen` be refreshed. This function can be used to clear the screen of stray bits that might be left after the execution of an incorrect graphics program.

If the host system does not preserve window contents (the X WINDOW system does not, for example), the windows will all be simply erased and sent `repaint-window-event` events.

(bitmap-flush screen) [*function with an optional argument*]

In some systems, for example in the X WINDOW system, the stream of graphic operations is buffered. These operations will be visible on the screen only when the buffer has been emptied. This buffer is automatically emptied by a call to one of the event manipulation functions: `read-event`, `peek-event` or `eventp`.

Sometimes it is necessary to explicitly empty the buffer. This happens, for instance, when frequent updates are required for animating a simulation, or when event reads are not being constantly done. At such times, `bitmap-flush` can be used to explicitly empty the buffer of surplus events.

In the following function, which displaces a character in the screen background, the use of `bitmap-flush` makes sure that the movement is regular:

```
(defun move-char (cn)
    (let ((x 0) (y 0) (dx 3) (dy 3))
         (with ((current-window (root-window)))
               (current-mode #:mode:xor)
               (current-font (large-roman-font))
               (while t
                   (draw-cn x y cn)
                   (setq x (+ x dx)
                         y (+ y dy))
                   (when (or (> x (bitxmax)) (< x 0))
                         (setq x (- x dx)
                               dx (- dx)))
                   (when (or (> y (bitymax)) (< y 0))
                         (setq y (- y dy)
                               dy (- dy)))
                   (draw-cn x y cn)
                   (bitmap-flush)))))
```

(bitmap-sync display) [*function with an optional argument*]

This function makes sure that all requests sent to the server have been executed, and that all events generated by these requests are in the events file. If the `display` argument is not supplied, the current display is used.

(display-synchronize display flag) [*function with one or two arguments*]

Certain systems, such as X11, can operate in two modes:

- Asynchronous: A graphic action might not be terminated when the function that triggered it returns control (the default option under X11).
- Synchronous: A graphic action is terminated when the function that triggered it returns control.

On systems operating in the two modes, **synchronous** and **asynchronous**, this function lets you know the current mode, or go from one to the other. The synchronous mode corresponds to a true value of the **flag** argument. The asynchronous mode corresponds to a false value. This function should only be used for **debug**, and must not be employed in normal use.

(display-depth display) [*function with one argument*]

This function returns the number of plans (bits by pixel) in the screen used by the **display** device. If the display device is monochrome, it therefore returns 1.

(default-window-type screen type) [*function with one or two arguments*]

If the **type** argument is not supplied, this function gives you the type used by default by the windows in **screen**. Supply **type** if you want to modify it.

(standard-roman-font screen) [*function with an optional argument*]

(standard-bold-font screen) [*function with an optional argument*]

(large-roman-font screen) [*function with an optional argument*]

(small-roman-font screen) [*function with an optional argument*]

These functions return the numeric indices of the preloaded fonts of **screen** or the current screen. (See the current-font function.) Four fixed-width fonts are always available:

- **standard-roman-font** is a regular font that can be used for text.
- **standard-bold-font** is a bolder font that can be used to highlight text.
- **large-roman-font** is a larger font that can be used for titles.
- **small-roman-font** is a smaller font that can be used for notes and captions.

(standard-foreground screen) [*function with an optional argument*]

(standard-background screen) [*function with an optional argument*]

These two functions return the initial foreground and background colors.
(See the `current-foreground` and `current-background` functions.)

(standard-foreground-pattern screen) [*function with an optional argument*]

(standard-background-pattern screen) [*function with an optional argument*]

(standard-light-gray-pattern screen) [*function with an optional argument*]

(standard-medium-gray-pattern screen) [*function with an optional argument*]

(standard-dark-gray-pattern screen) [*function with an optional argument*]

These functions return the numeric indices of the predefined textures (fill patterns) of `screen` or the current screen. (See the `current-pattern` function.)

The `standard-foreground-pattern` and `standard-background-pattern` are solids that use the current value returned by (`current-foreground`) and the current value returned by (`current-background`). Their effect therefore depends on the current colors.

The other predefined patterns are 'grays' that are independent of the current foreground and background colors. They provide a regular gradation between the initial foreground color given by the value of (`standard-foreground`) and the initial background color given by the value of (`standard-background`). Here are the details:

- `standard-medium-gray-pattern`:
  A medium 'grayed' value between these two colors.
- `standard-light-gray-pattern`:
  Between `standard-background` and `standard-medium-gray-pattern`.
- `standard-dark-gray-pattern`:
  Between `standard-foreground` and `standard-medium-gray-pattern`.

(standard-lelisp-cursor screen) [*function with an optional argument*]

(standard-gc-cursor screen) [*function with an optional argument*]

(standard-busy-cursor screen) [*function with an optional argument*]

These functions return the numeric indices of the predefined cursors of `screen` or the current screen.
(See the `current-cursor` function.)

These cursors can be used in one of the following capacities:

- `standard-lelisp-cursor` is displayed during normal LE-LISP operation.
- `standard-gc-cursor` is displayed during garbage-collection cycles. (See the `gcalarm` and `gc-before-alarm` functions.)
- `standard-busy-cursor` is displayed when LE-LISP is carrying out a long computation.

`(standard-foreground screen)` [*function with an optional argument*]

`(standard-background screen)` [*function with an optional argument*]

These function return initial foreground and background colors. These are the foreground and background colors which are set for each newly-created window.

In some systems these colors are selectable by the user at LE-LISP startup time. In the X WINDOW System, for example, this can be done in the `.Xdefaults` file.

18.4 Functions on windows

18.4.1 Global coordinates

A *global bitmap coordinate* is a pair of integers `(x, y)` that represents an address on a high-resolution screen.

The origins of this coordinate system are defined as follows:

- Upper left corner: `x=0`, `y=0`
- Lower left corner: `x=0`, `y=(bitymax)`
- Upper right corner: `x=(bitxmax)`, `y=0`
- Lower right corner: `x=(bitxmax)`, `y=(bitymax)`.

18.4.2 Local coordinates

The graphics environment attached to a window defines a local coordinate system in the window, defined as follows:

- Upper left corner of the window: `x=0`, `y=0`
- Lower right corner of the window: `x`=window height`-1`, `y`=window width`-1`.

The title and border regions are outside this coordinate system, but can be represented in it by using negative coordinates. The point `(-1, -1)`, for example, is in the border region of the window.

18.5 Windows

Windows are typed structured objects that have graphical representations on the screen. (See the explanation of structures.) These are rectangular frames that can have an optional title. A graphics environment is attached to each window. This environment supports various text, line, and zone drawing operations within the window. (See the section on virtual graphics.)

Window events are caused by mouse pointer movements and button clicks or selections that occur when the pointer is within a window. (See the `read-event` function.) Likewise, the displacement or resizing of windows by a non-LISP tool, such as the window manager running in an X WINDOW System environment, can cause events informing LE-LISP that the windows were modified or need to be redrawn.

The Virtual Bitmap Display facility does not assume that the contents of hidden parts of windows are saved by the system. When they are not saved, a `repaint-window-event` indicates the concealed areas that need to be redrawn.

Each window can contain any number of sub-windows, which have neither borders nor titles. They can cover each other in any manner, but they are all transparent, so that a sub-window does not obscure the contents of a daughter or sister sub-window that it covers. Moreover, sub-windows have no contents. The displacement or resizing of a sub-window never visually affects the screen.

Sub-windows are manipulated in the same way as windows. All the graphics and window-manipulation functions can be performed on sub-windows. It is even possible to divide sub-windows into sub-sub-windows. Graphics operations in sub-windows are carried out in the coordinate system of the sub-window, but the image is clipped (limited) to that part of the sub-window contained within the mother window. Sub-windows obtain mouse events with the priority due the super-window that contains them. However, they never receive `repaint-window-event` or `modify-window-event` events.

The total number of windows that can be displayed at a given moment might be limited in some systems. If this limit is surpassed, window creation functions raise the `errnmw` error, which has the following screen display:

```
** <fn>: no more windows available: ()
```

Some systems—such as virtual windows on the virtual terminal—do not have virtual graphics. Even so, these systems have a minimal graphics environment that allows only the display of text. This minimal graphics environment is described in the remainder of this chapter, and the full graphics environment is described in chapter 20.

The system recognizes a current window on which display-oriented functions occur. There is one current window per screen.

At system-window initialization—that is, after the call to the `bitprologue` function—the current window is the special window `()`, which indicates that the only available 'window' is the virtual terminal, and that graphics functions will have no effect.

18.5.1 Creating windows

The creation of a window or a sub-window usually happens with a call to the `make-window` function, which receives an instance of the `#:image:rectangle:window` structure as its argument. This structure must be allocated and its fields must be filled in with appropriate values before the call to `make-window`. The `create-window` and `create-subwindow` functions provide the means to allocate an instance of this structure and to call the `make-window` function in a single call.

`#:image:rectangle:window` [*structure*]

This LISP structure represents a window. It is defined as follows:

```
(defstruct #:image:rectangle
           x y w h)

(defstruct #:image:rectangle:window
           title
           hilited
           visible
           graph-env
           extend
           father
           properties
           (cursor 0)
           display
           subwindows
           events-list
           window-type
           graphic-properties
           state)
```

Some fields must be set by the user before a call to the `make-window` function is made. Other fields will be set automatically by the system.

The fields that must be filled are described next. They should never be modified directly after the call to `make-window` has been made. Instead, window manipulation functions should be used to modify them, although they can be directly examined (*i.e.*, read) at any time.

- `x`, `y` (short integers): the upper left corner of the window, expressed in the global coordinate system for windows, and in the local coordinate system of the mother window for sub-windows. The window can contain various borders and titles, according to the host system. (See the `hilited` and `visible` fields.) These will always be outside the window proper. The coordinate point `(0,0)` in the window will always be a point `(x,y)` in the global coordinate system for a window, or in the local coordinate system of the mother window for a sub-window.
- `w`, `h` (short integers): the width and height of the useful zone of the window, that is, not counting the border or title zones. Each value is indicated as a number of basic display points (pixels) for the device.

- `title` (character string): the optional title to be posted in the window border.
- `hilited` (short integer): indicates the highlight mode of the window. If this field has the value zero, the window is not highlighted. If it is not equal to zero, its value indicates the highlight mode: title highlighted, window colored, etc. The various highlight modes available depend on the system. By default, only one mode exists, coded by the value `1`.
- `visible` (short integer): indicates the window's visibility degree on the screen. If this flag has the value zero, the window is invisible. If it is not equal to zero, its value indicates the visibility degree: entirely visible, semi-visible, transparent, etc. The available visibility degrees depend on the system. By default, a single degree of visibility is available, coded by the value `1`. This represents opacity, meaning that the window obscures windows that it covers.
- `father` (window, or `()`): contains the window's parent window. If it has the special value `()`, then there is no parent window. Otherwise, this field represents a previously-constructed window, of which the current window is a sub-window.
- `display` (screen): contains the screen on which the window is displayed. If this field is not filled, the `make-window` function displays the window on the current screen. (See the `bitprologue` function.)
- `window-type`: the type of the window (transparent or opaque). In X WINDOW, these types correspond to `InputOnly` and `InputOutput`. Opaque windows can receive events of `redisplay` type, whereas transparent windows cannot.
- `state` (symbol): the current state of the window: `iconify`, `map` or `unmap`.

The other fields are maintained by the system. They can be read by the user at any time, but must not be modified.

- `graph-env` (structure `graph-env`): contains the graphics environment attached to the window.
- `extend`: the contents and type of this field are system-dependent and not of much interest to the user.
- `properties` (a list): contains the property A-list attached to the window. (See the `define-window-property-accessor` function.)
- `cursor` (short integer): contains the cursor currently attached to the window. (See the `current-cursor` function.)
- `subwindows` (list of windows): contains the list of the window's sub-windows.
- `events-list`: contains the list of authorized events of the window.
- `graphic-properties`: contains a list of internal graphic properties.

Windows can be instances of any sub-structure of the `#:image:rectangle:window` structure. This, together with the fact that window-manipulation functions work by sending messages to windows, means that the Virtual Bitmap Display facility can be easily extended. (See the `virbitmap` file of the standard distribution for more information on this subject.)

(make-window win) [*function with one argument*]

This function takes an instance of a window structure of a certain type as argument, and does everything necessary to create a window of this type described by the structure. It returns `win` as its value, with the system-maintained fields filled in, as described above. All the other fields of the `win` structure—`x`, `y`, `w`, `h`, `title`, `hilited`, `visible`, `father` and `display`—should be initialized before the call to `make-window`. To make a sub-window, set the `father` field of `win` to contain the structure of a previously-created window. In all other cases, `father` must contain `()`. If the `display` field contains `()`, then `make-window` fills it with the value of the current screen. (See the `current-display` function.)

The following function creates a window whose position and size are passed as arguments:

```
(defun build-window (x y w h)
    (let ((window (new '#:image:rectangle:window)))
         (#:image:rectangle:x window x)
         (#:image:rectangle:y window y)
         (#:image:rectangle:w window w)
         (#:image:rectangle:h window h)
         (#:image:rectangle:window:title window "fenestre")
         (#:image:rectangle:window:hilited window 1)
         (#:image:rectangle:window:visible window 1)
         (#:image:rectangle:window:display window (current-display))
         (make-window window)
         window))
```

(create-window type x y w h ti hi vi) [*function with eight arguments*]

Allocates a window of type `type`. This must be a sub-type of the `#:image:rectangle:window` type. The function initializes the fields of the window with the values passed as arguments, and calls the `make-window` function. `create-window` provides a convenient means for building a window without explictly allocating its structure instance.

`create-window` has the same behavior as the following function:

```
(defun create-window (type x y w h ti hi vi)
    (let ((window (new type)))
         (#:image:rectangle:x window x)
         (#:image:rectangle:y window y)
         (#:image:rectangle:w window w)
         (#:image:rectangle:h window h)
         (#:image:rectangle:window:title window ti)
         (#:image:rectangle:window:hilited window hi)
         (#:image:rectangle:window:visible window vi)
         (#:image:rectangle:window:display window (current-display))
         (make-window window)
         window))
```

(create-subwindow type x y w h ti hi vi fa) [*function with nine arguments*]

This function is similar to the last one, except that it creates a sub-window of the `fa` parent window. The `x` and `y` coordinates should be in the local coordinate system of `fa`.

18.5.2 Drawing in windows

The `make-window` call adds a `repaint-window-event` event to the event queue as soon as the window has actually been created. This event must be received and handled before any drawing is done in the window. Requests to draw in the window that occur before this event might in fact be lost.

Well-structured programs using the VBD should therefore create windows and then enter an event-handling loop, which primarily waits on `repaint-window-event` events and paints the windows, both initially and then after changes are made to windows during the execution of the program.

The following example shows how such an event-handling loop might be written. The `nice-picture` function draws something in the window. The `open-window` function creates a window, then enters an event-handling loop: the `process-events` function. This loop reacts to `repaint-window-event` events by calling `nice-picture` to draw or redraw the window contents. The loop terminates and kills the window as soon as a mouse click occurs. The `process-events` loop is a typical event-handling procedure that is presented here as a model for writing other loops of a similar nature. Chapter 20 describes in detail the kind of behavior that should be associated with the various types of VBD facility events.

```
(defun open-window ()
    (let ((window (create-window '#:image:rectangle:window 0 0 200 200 "target" 1 1)))
         (protect
             (process-events)
             (kill-window window)
             (bitmap-flush))))

(defun process-events ()
    (untilexit done
       (let ((event (read-event)))
            (selectq (#:event:code event)
                (repaint-window-event
                   (with ((current-window (#:event:window event)))
                         (current-clip (#:event:x event)
                                       (#:event:y event)
                                       (#:event:w event)
                                       (#:event:h event))
                         (nice-picture)))
                (down-event
                     (exit done))
                (modify-window-event
                     (update-window (#:event:window event)
                                    (#:event:x event)
                                    (#:event:y event)
                                    (#:event:w event)
                                    (#:event:h event)))))))

(defun nice-picture ()
    (unless nice-colors (nice-init))
    (let ((colors (apply 'cirlist nice-colors)))
         (for (i (div (#:image:rectangle:w (current-window)) 2)
                 -10 1)
              (with ((current-foreground (nextl colors)))
                    (fill-circle
```

```
                        (div (#:image:rectangle:w (current-window)) 2)
                        (div (#:image:rectangle:h (current-window)) 2)
                        i)))))

(defvar nice-colors ())

(defun nice-init ()
    (setq nice-colors
          (list
             (make-color 32000 24000 16000)
             (make-color 26000 24000 10000)
             (make-color 26000 28000 16000)
             (make-color 24000 28000 24000)
             (make-color 22000 24000 32000))))
```

18.5.3 Attaching properties to windows

It is possible to attach named properties to each instance of the `#:image:rectangle:window` type. These properties are manipulated like fields, but they are stored in an A-list in the window's `properties` field.

`(define-window-property-accessor prop)` [*macro*]

This macro defines the `#:image:rectangle:window:prop` function to be an access function to the window property named `prop`, which must be a symbol.

`(#:image:rectangle:window:prop w v)` [*function with one or two arguments*]

Examines or sets (if the second argument is supplied) the value associated with the property named `prop` belonging to the window `w`. If the window does not have the `prop` property, `()` is returned. The property is stored in the A-list in the properties field of window `w`. To distinguish between the absence of the property and its presence with the value `()`, examine this A-list directly.

Properties only take up memory space in a window structure if the window actually has the properties in question.

Here, for example, is the definition of a window property named `contents`:

```
? (define-window-property-accessor contents)
= #:image:rectangle:window:contents
```

For the `win1` window, this property has the value `"in win1"`:

```
? (#:image:rectangle:window:contents win1 "in win1")
= in win1
```

Let us look at this value:

```
? (#:image:rectangle:window:contents win1)
= in win1
```

Finally, let us look at the property lists of `win1` and of another window called `win2`:

```
? (#:image:rectangle:window:properties win1)
= ((contents . in win1))
? (#:image:rectangle:window:properties win2)
= ()
```

18.5.4 Functions on windows

The following functions let programs act on windows created by the `make-window`, `create-window` or `create-subwindow` functions. Their rôle is to simply transmit messages to the window specified in their arguments. It is the primitive window functions, described in the next section, which act directly on windows. Generally, only the functions described in this section are used.

`(current-window win)` *[function with an optional argument]*

The window passed as the argument will become the current window on the screen to which it is attached. All ensuing graphics functions will take place in this window. If no argument is furnished `current-window` returns the current window. The current window has no rôle other than to be the implicit argument for drawing functions.

If the argument is `()`, the screen's current window becomes `()`, and the graphics printing functions will not be effective on this screen.

If the argument is a window which has been killed, the system raises the `errnotawindow` error, with the following display:

```
** <fn>: not a window: <e>
```

Whenever an argument is supplied, the `uncurrent-window` message is sent to the current window before the current window is changed.

A window can be temporarily selected by using `current-window` within a `with` control structure. Here, for example, is a way to draw a line in a window:

```
(with ((current-window window))
      (draw-line 0 0 100 100))
```

`(current-keyboard-focus-window win)` *[function with an optional argument]*

Examines or modifies (if the second argument is given) the current owner of the keyboard. The owner of the keyboard is the window that receives `ascii-event` events. (See the `read-event` function.) After a call to this function, keyboard strokes will result in `ascii-event` events being sent to the `win` window.

The host windowing system (or a window manager) can sometimes change the keyboard owner independently of LE-LISP. In this case `keyboard-focus-event` events will be generated and placed in the event queue; LE-LISP structures can be kept up to date by treating them appropriately.

(modify-window win x y w h ti hi vi) [*function with eight arguments*]

The `win` agument is a window previously created by the `make-window` function. The `x`, `y`, `w`, `h`, `ti`, `hi` and `vi` arguments are similar to the arguments of the same name of the `create-window` function. This function provides a means of modifying the parameters of a window. Its position, title or size can be changed, it can be highlighted or unhighlighted, and it can be made visible or invisible. The function returns the modified window structure as its value. By convention, if one of the arguments `x`, `y`, `w`, `h`, `ti`, `hi` or `vi` has the value `()`, the corresponding parameter remains unchanged.

Window size modifications do not affect the size of sub-windows.

As an example, let us create a window entitled `Lisp` and a window entitled `Foo`:

```
(defvar wlisp (create-window 'window 10 10 400 400 "Lisp" 0 1)
        wfoo  (create-window 'window 30 30 500 500 "Foo"  0 1))
```

Move the window named `Lisp`:

```
(modify-window wlisp 20 20 () () () () ())
```

Reduce the size of the window named `Foo`:

```
(modify-window wfoo () () 50 100 () () ())
```

Modify the title of the window named `Lisp`:

```
(modify-window wlisp () () () () "New Title" () ())
```

Highlight the title of the `wfoo` window:

```
(modify-window wfoo () () () () () 1 ())
```

Erase the `wlisp` window from the screen, making it invisible:

```
(modify-window wlisp () () () () () () 0)
```

Cause this same window to reappear:

```
(modify-window wlisp () () () () () () 1)
```

(kill-window win) [*function with one argument*]

Kills the window given as its argument, as well as all its sub-windows, if it has any. If the window is visible, it will be erased from the screen. Any further use of this window will raise the `errnotawindow` error. The `windowp` predicate applied to `win` will subsequently return `()`. Note that `win` can be rebuilt using the `make-window` function, but this would not rebuild any sub-windows it might have had.

If the current window is killed by this function, the window `()` becomes the current window. The window that was current before `kill-window` was called receives an `uncurrent-window` message before being killed.

(move-window win x y) [*function with three arguments*]

Move the `win` window at the coordinate point `x`, `y`.

(resize-window win w h) [*function with three arguments*]

Resize the `win` window with a width of `w` and a height of `h`.

(move-resize-window win x y w h) [*function with five arguments*]

Perform a `(move-window win x y)` followed by a `(resize-window win w h)`.

(window-events-list win events) [*function with one or two arguments*]

Consult or set, if the `events` argument is present, the event list attached to the `win` window. `events`, when present, is a list of events among `down-event`, `up-event`, `ascii-event` ...

(window-title win title) [*function with one or two arguments*]

Consult or set, if the `title` argument is present, the title of the `win` window. `title`, when present, must be a character string. Only main windows can have a title.

(window-state win state) [*function with one or two arguments*]

Consult or set, if the `state` argument is present, the state of the `win` window. `state`, when present, must be a symbol among `map`, `unmap`, `lower`, `raise` and `iconify`. This function can iconify a window and later raise it.

(window-background win color) [*function with one or two arguments*]

Consult or set, if the `color` argument is present, the background color of the `win` window. `color`, when present, must be a valid instance of the `color` type.

(window-border win border) [*function with one or two arguments*]

Consult or set, if the `border` argument is present, the size (in pixels) of the border of the `win` window.

(window-clear-region win x y w h) [*function with five arguments*]

Clear the rectangle `x`, `y`, `w`, `h` of the `win` window.

(windowp win) [*function with one argument*]

This predicate is true if `win` is an instance of the `#:image:rectangle:window` type, or of one of

its sub-types, and if `win` has been created by `create-window`, `create-subwindow` or `make-window`, and has not yet been killed by `kill-window`.

The next two functions provide means to manage the display order of windows displayed on the screen. They modify the 'stacking order' of the windows, but not their `(x, y)` coordinates.

`(pop-window win)` [*function with one argument*]

Puts the `win` window in the foreground.

`(move-behind-window win1 win2)` [*function with two arguments*]

Puts the `win1` window behind the `win2` window on the screen. This does not necessarily mean that `win1` is *directly* behind `win2`. The stacking position of `win2` is not changed.

The next two functions provide means to use absolute coordinates when dealing with windows.

`(find-window x y)` [*function with two arguments*]

Returns the lowest LISP window in the stacking hierarchy that contains the screen point `(x, y)`, considered as a point in the global coordinate system. If this point does not appear in any window, `find-window` returns `()`.

Points in border and title areas count for `find-window` as being included in their windows.

`(map-window win x y symbx symby)` [*function with five arguments*]

The `win` argument is a window. `x` and `y` are numbers designating a point in the global coordinate system of the screen. `symbx` and `symby` are symbols. When `map-window` returns, these symbols contain as their values the coordinates, in the local coordinate system of `win`, of the global point `(x, y)`.

Return values from `map-window` are only meaningful for `(x. y)` points that are in the window `win` or in its border or title regions. Given points in the border or title regions, `map-window` will just return local coordinates that are not contained in the window area proper.

```
? (defvar w (create-window 'window 10 10 400 400 "w" 0 1)
?         x 0
?         y 0)
= 0
? (eq w (find-window 200 200))
= t
? (map-window w 200 200 'x 'y)
? x
= 190
? y
= 190
```

18.5.5 Primitive functions on windows

These functions are the primitive functions of the virtual multi-windowing system. They are the methods for the `current-window`, `uncurrent-window`, `modify-window`, `kill-window`, `move-behind-window` and `map-window` messages for objects of type `#:image:rectangle:window`.

Normally, you will never need to use them directly. They are included here merely for reasons of completeness at a documentary level.

`(#:image:rectangle:window:current-window win)` [*function with one argument*]

`(#:image:rectangle:window:current-keyboard-focus-window w)` [*function with one argument*]

`(#:image:rectangle:window:uncurrent-window win)` [*function with one argument*]

`(#:image:rectangle:window:modify-window win left top w h ti hi vi)` [*function with eight arguments*]

`(#:image:rectangle:window:kill-window win)` [*function with one argument*]

`(#:image:rectangle:window:pop-window win)` [*function with one argument*]

`(#:image:rectangle:window:move-behind-window win1 win2)` [*function with two arguments*]

`(#:image:rectangle:window:map-window win x y symbx symby)` [*function with five arguments*]

18.5.6 Minimal graphics primitives

The Virtual Bitmap Display facility defines a set of minimal graphics primitives that work on every implementation, even on alphanumeric terminals. Extended graphics primitives, available on workstations with high-resolution screens, are described in Chapter 20.

All the drawing functions operate in the current window on the current screen. (See the `current-window` function.)

(clear-graph-env) [*function with no arguments*]

Erases the screen. In other words, the screen is uniformly filled with the `(standard-background)` color. (See the description of this attribute.)

(draw-cursor `x y i`**)** [*function with three arguments*]

This function provides a way to display and erase a cursor at the point `(x, y)` in the current window. `i` is a binary flag. If it has the value `()`, the cursor is erased. If it has the value `t`, the cursor is displayed. The actual form of the cursor that is displayed depends, of course, on the system.

This cursor is generally used to tell the user that they should type a character. Here is the standard technique for reading a character from the keyboard and echoing it at the position `(x, y)`, which it takes as its argument:

```
(defun wtyi-echo (x y)
    (draw-cursor x y t)
    (let ((c (tyi)))
         (draw-cursor x y ())
         (draw-cn x y c)))
```

(draw-cn `x y cn`**)** [*function with three arguments*]

Writes the character whose internal code is `cn` at the `(x, y)` position of the current window. The character is in fact a two-colored rectangular tile,
the letter being drawn in the `(current-foreground)` color, and the background in the `(current-background)` color.

In LE-LISP, `draw-cn` could be defined in the following manner:

```
(defun draw-cn (x y cn)
    (draw-substring x y (string (ascii cn)) 0 1)
```

(draw-string `x y s`**)** [*function with three arguments*]

(draw-substring `x y s start length`**)** [*function with five arguments*]

The `draw-substring` function writes `length` characters of the string `s`, taken from the position `start`, at the `(x, y)` position in the current graphics environment.

The `draw-string` macro writes the complete string `s` at the point `(x, y)`.

Here, for example, we write the string `"foothebar"` at local coordinate point `(10, 10)` in the current window:

```
(draw-substring 10 10 "foothebar" 2 5)
```

In LE-LISP, `draw-string` could be defined in the following manner:

```
(dmd draw-string (x y s)
     `(let ((s ,s))
           (draw-substring ,x ,y s 0 (slength s))))
```

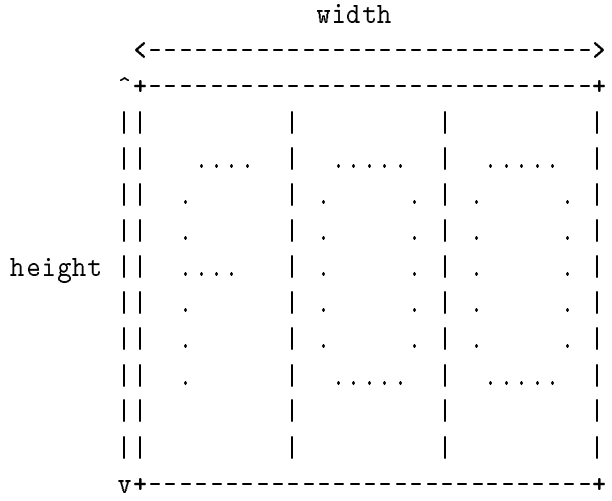
18.5.7 Character strings

A character string on the screen has a *display rectangle*, a *base point* and an *increment point*.

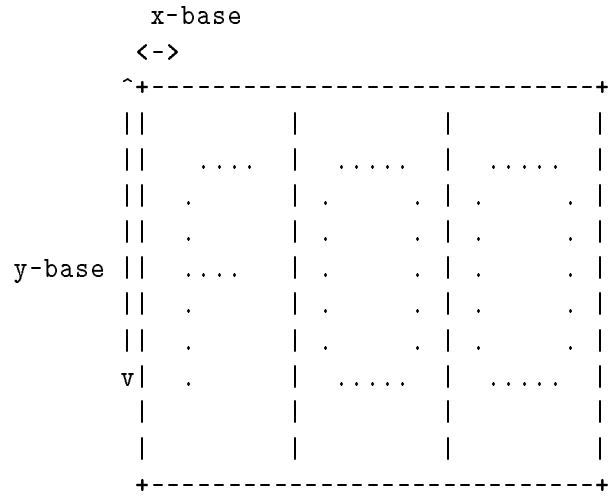
- The display rectangle is the screen zone occupied by the string.
- The base point is the point within the display rectangle at the lower-left corner of the first character of the string. It is passed to the `draw-string` function, which displays the string at the point `(x, y)`. (See the description of `draw-string`.)
- The increment point is a point of the display rectangle that can be used as the base point of the next character string, to be displayed after the current string.

The following diagrams illustrate these various quantities.

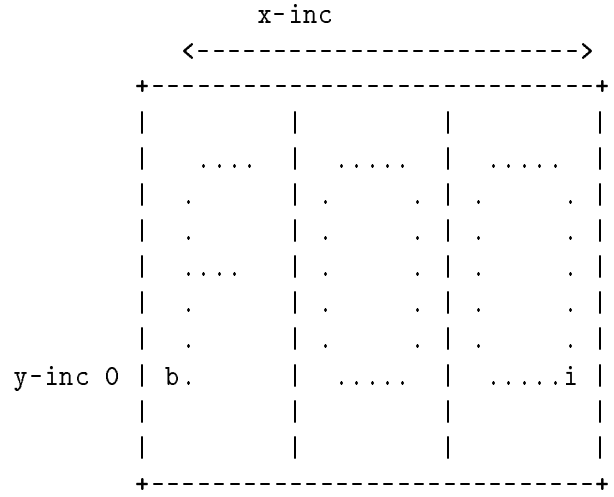
The *display rectangle* (width and height):



The *base point* (in the local coordinates of the display rectangle):



The *increment point* (relative to the base point):



In the above example, the increment point and the base point have the same y-coordinate. The value of `y-inc` is therefore zero.

These six quantities, representing the visual aspects of a character string, obviously depend on the display attributes such as body, boldface and font. They can be computed by the following functions:

(width-substring `string start length`**)** [*function with three arguments*]

(height-substring `string start length`**)** [*function with three arguments*]

(x-base-substring `string start length`**)** [*function with three arguments*]

(y-base-substring `string start length`**)** [*function with three arguments*]

(x-inc-substring `string start length`**)** [*function with three arguments*]

(y-inc-substring `string start length`**)** [*function with three arguments*]

These functions return the values described above for the `length` characters of the string `string` beginning in position `start`. The values returned take account of the current window's font. (See the `current-font` function.)

The following function can be used, for example, to determine if the point `(px, py)` is in the graphics representation of the string `"foo"` at position `(x, y)`:

```
(defun in-stringp (px py s x y)
    (let ((lx (- x (x-base-substring s 0 (slength s))))
          (ty (- y (y-base-substring s 0 (slength s)))))
         (and (> px lx)
              (> py ty)
              (< (- px lx) (width-substring s 0 (slength s)))
              (< (- py ty) (height-substring s 0 (slength s))))))
```

The next function displays `n` strings on the screen, one after the other.

```
(defun draw-n-strings (x y string . strings)
    (draw-string x y string 1)
    (while strings
       (draw-string
           (incr x (x-inc-substring string 0 (slength string)))
           (incr y (y-inc-substring string 0 (slength string)))
           (nextl strings string))))
```

(width-space) [*function with no arguments*]

(height-space) [*function with no arguments*]

(x-base-space) [*function with no arguments*]

(y-base-space) [*function with no arguments*]

These four functions calculate the width, height and base-point coordinates of the space character. They are particularly useful for manipulating fixed-width text.

In LE-LISP, `width-space` could be defined in the following manner:

```
(defun width-space ()
    (width-substring " " 0 1))
```

(font-height) [*function with no arguments*]

Returns the difference between the ordinates (y-axis) of two successive lines of text in the current type font.

(font-ascent) [*function with no arguments*]

Returns the `ascent` of the current type font. This is the greatest height of a character above the base line.

(display-get-font-names `display max pattern`) [*function with two or three arguments*]

Takes a `pattern` character string that can contain the characters `*` (any series of characters, even empty) and `?` (any character whatsoever) and returns the names of the loadable type fonts that correspond to this expression. The `pattern` argument is optional, and is equal to `"*"` by default. The `max` argument lets you limit the number of replies, because there could be a very large number of type fonts in the system.

In the following example, we look for all the type fonts whose names start with `"adobe"`:

```
? (display-get-font-names (current-display) 100 "adobe*")
```

(display-get-font-info `display font-name font-info`) [*function with two or three arguments*]

Returns an instance of the `font-info` class whose fields provide metric data concerning the type font called `font-name`. If the `font-info` argument is supplied, this function does not allocate a new instance of `font-info`, but uses the `font-info` argument, which must be of `font-info` type. This function does not require the type font called `font-name` to be already loaded. The function is useful, for example, to find out whether a type font is of fixed or proportional spacing. This function provides the following metric data:

- General data on the font:
  `ascent` and `descent` fields.
- Data concerning the smallest character:
  `minlbearing`, `minrbearing`, `minascent`, `mindescent` and `minwidth` fields.
- Data concerning the largest character:
  `maxlbearing`, `maxrbearing`, `maxascent`, `maxdescent` and `maxwidth` fields.
- An angle for the slope of italic fonts.
- A width, `Iweight`, for boldface fonts.

The result depends upon the screen system, and certain fields can remain empty. If the type font called `font-name` does not exist, the function returns `()`.

Table of contents

18 Virtual bitmap display **18-1**

18.1 Loading the description file 18-1

18.2 Screen preparation 18-2

18.2.1 Managing a single screen 18-2

18.2.2 Managing several screens 18-3

18.3 Functions on screens 18-5

18.4 Functions on windows 18-9

18.4.1 Global coordinates 18-9

18.4.2 Local coordinates 18-9

18.5 Windows 18-10

18.5.1 Creating windows 18-11

18.5.2 Drawing in windows 18-14

18.5.3 Attaching properties to windows 18-15

18.5.4 Functions on windows 18-16

18.5.5 Primitive functions on windows 18-20

18.5.6 Minimal graphics primitives 18-21

18.5.7 Character strings 18-22

Function Index

`(inibitmap symbol)` [*function with an optional argument*] 18-1
`(bitprologue name device)` [*function with one or two optional arguments*] 18-3
`(current-display screen)` [*function with an optional argument*] 18-3
`(bitepilogue screen)` [*function with an optional argument*] 18-4
`(bitmap-save screen)` [*function with an optional argument*] 18-4
`(bitmap-restore screen)` [*function with one argument*] 18-5
`(bitxmax screen)` [*function with an optional argument*] 18-5
`(bitymax screen)` [*function with an optional argument*] 18-5
`(root-window screen)` [*function with an optional argument*] 18-5
`(all-colors screen)` [*function with an optional argument*] 18-6
`(bitmap-refresh screen)` [*function with an optional argument*] 18-6
`(bitmap-flush screen)` [*function with an optional argument*] 18-6
`(bitmap-sync display)` [*function with an optional argument*] 18-6
`(display-synchronize display flag)` [*function with one or two arguments*] 18-7
`(display-depth display)` [*function with one argument*] 18-7
`(default-window-type screen type)` [*function with one or two arguments*] 18-7
`(standard-roman-font screen)` [*function with an optional argument*] 18-7
`(standard-bold-font screen)` [*function with an optional argument*] 18-7
`(large-roman-font screen)` [*function with an optional argument*] 18-7
`(small-roman-font screen)` [*function with an optional argument*] 18-7
`(standard-foreground screen)` [*function with an optional argument*] 18-7
`(standard-background screen)` [*function with an optional argument*] 18-8
`(standard-foreground-pattern screen)` [*function with an optional argument*] 18-8
`(standard-background-pattern screen)` [*function with an optional argument*] 18-8
`(standard-light-gray-pattern screen)` [*function with an optional argument*] 18-8
`(standard-medium-gray-pattern screen)` [*function with an optional argument*] 18-8

(standard-dark-gray-pattern screen) [*function with an optional argument*] 18-8
(standard-lelisp-cursor screen) [*function with an optional argument*] 18-8
(standard-gc-cursor screen) [*function with an optional argument*] 18-8
(standard-busy-cursor screen) [*function with an optional argument*] 18-8
(standard-foreground screen) [*function with an optional argument*] 18-9
(standard-background screen) [*function with an optional argument*] 18-9
#:image:rectangle:window [*structure*] 18-11
(make-window win) [*function with one argument*] 18-13
(create-window type x y w h ti hi vi) [*function with eight arguments*] 18-13
(create-subwindow type x y w h ti hi vi fa) [*function with nine arguments*] 18-13
(define-window-property-accessor prop) [*macro*] 18-15
(#:image:rectangle:window:prop w v) [*function with one or two arguments*] 18-15
(current-window win) [*function with an optional argument*] 18-16
(current-keyboard-focus-window win) [*function with an optional argument*] 18-16
(modify-window win x y w h ti hi vi) [*function with eight arguments*] 18-17
(kill-window win) [*function with one argument*] 18-17
(move-window win x y) [*function with three arguments*] 18-18
(resize-window win w h) [*function with three arguments*] 18-18
(move-resize-window win x y w h) [*function with five arguments*] 18-18
(window-events-list win events) [*function with one or two arguments*] 18-18
(window-title win title) [*function with one or two arguments*] 18-18
(window-state win state) [*function with one or two arguments*] 18-18
(window-background win color) [*function with one or two arguments*] 18-18
(window-border win border) [*function with one or two arguments*] 18-18
(window-clear-region win x y w h) [*function with five arguments*] 18-18
(windowp win) [*function with one argument*] 18-18
(pop-window win) [*function with one argument*] 18-19
(move-behind-window win1 win2) [*function with two arguments*] 18-19
(find-window x y) [*function with two arguments*] 18-19
(map-window win x y symbx symby) [*function with five arguments*] 18-19
(#:image:rectangle:window:current-window win) [*function with one argument*] ... 18-20
(#:image:rectangle:window:current-keyboard-focus-window w) [*function with one argument*] 18-20
(#:image:rectangle:window:uncurrent-window win) [*function with one argument*] . 18-20

`(#:image:rectangle:window:modify-window win left top w h ti hi vi)` [*function with eight arguments*] 18-20

`(#:image:rectangle:window:kill-window win)` [*function with one argument*] 18-20

`(#:image:rectangle:window:pop-window win)` [*function with one argument*] 18-20

`(#:image:rectangle:window:move-behind-window win1 win2)` [*function with two arguments*] 18-20

`(#:image:rectangle:window:map-window win x y symbx symby)` [*function with five arguments*] 18-20

`(clear-graph-env)` [*function with no arguments*] 18-21

`(draw-cursor x y i)` [*function with three arguments*] 18-21

`(draw-cn x y cn)` [*function with three arguments*] 18-21

`(draw-string x y s)` [*function with three arguments*] 18-21

`(draw-substring x y s start length)` [*function with five arguments*] 18-22

`(width-substring string start length)` [*function with three arguments*] 18-23

`(height-substring string start length)` [*function with three arguments*] 18-23

`(x-base-substring string start length)` [*function with three arguments*] 18-24

`(y-base-substring string start length)` [*function with three arguments*] 18-24

`(x-inc-substring string start length)` [*function with three arguments*] 18-24

`(y-inc-substring string start length)` [*function with three arguments*] 18-24

`(width-space)` [*function with no arguments*] 18-24

`(height-space)` [*function with no arguments*] 18-24

`(x-base-space)` [*function with no arguments*] 18-24

`(y-base-space)` [*function with no arguments*] 18-24

`(font-height)` [*function with no arguments*] 18-25

`(font-ascent)` [*function with no arguments*] 18-25

`(display-get-font-names display max pattern)` [*function with two or three arguments*] 18-25

`(display-get-font-info display font-name font-info)` [*function with two or three arguments*] 18-25

Chapter 19

Virtual mouse

The LE-LISP system provides the means to manage a device that selects discrete points on a high-resolution screen. This device, called the **virtual mouse**, can be an actual physical mouse with one or several buttons, an optical pen, a graphics tablet, a touch screen, or any other pointing device.

The binding of the cursor associated with the virtual mouse is always performed by the underlying graphics system.

The pointing device is always multiplexed with the user keyboard. This chapter describes all the pointing-device and keyboard manipulation functions.

The Virtual Bitmap Display device includes events that are enqueued to reflect changes in the state of the underlying graphics system. Pointing-device actions, keyboard actions, actions on windows brought about by agents outside the LE-LISP system, and refresh requests for particular zones of the screen, all result in events being queued. In addition, the use of structures to represent events makes the event system easy to extend.

19.1 Events

An *event* is an information unit emanating from the Virtual Bitmap Display facility. This information might be generated because of one of the following 'happenings':

- a physical mouse action selecting points in a window or sub-window in various ways;
- a keystroke;
- an action on windows carried out via the underlying window manager;
- a window refresh request.

The set of events actually generated depends upon the system being used. The list of event types provided later on in this chapter is complete enough to deal with most of the well-known graphics systems. It could easily be enriched and extended to deal with other systems.

Events are stored in a queue that a program can consult at any time. There is one single event queue, and all events are multiplexed on a single channel.

A convenient way of managing events consists of the using the *event loop* described in chapter 6 of this manual. This approach generalizes the input/output notion by harmonizing input from the

keyboard and the mouse with all other kinds of input such as the reading of a process or a UNIX pipe.

19.2 Structure and types of events

event [*structure*]

This structure describes LISP objects that represent events in the queue. It has the following definition:

```
(defstruct event
           code window detail gx gy x y w h)
```

The `code` field contains a symbol describing the event type. The `window` field indicates the window affected by the event, which is usually the lowest window in the hierarchy that contained the mouse when the event was triggered. (See, however, the `grab-event` function.)

The meaning of the other fields depends on the type that the `code` field contains. The list of event types is not exhaustive and, on some systems, supplementary codes could be implemented. Also, some systems do not generate certain event codes. Only the `ascii-event` and `down-event` events are implemented in all cases. In general, the `drag-event`, `up-event` and `move-event` events, with several values of the `detail` field, are also fully implemented.

Note: From this point on, an event name such as `drag-event` will often be used to refer both to the event type and to an *instance* of that event. This enables us to avoid awkward phrases such as '`drag-event` event'.

The various event types are described below.

move-event [*event*]

The `move-event` describes a pointing-device movement when no buttons are pushed. The `gx` and `gy` fields in the associated event structure represent the global coordinates of the mouse at the moment of the event, while `x` and `y` are the mouse coordinates relative to the local `window` coordinate system.

down-event [*event*]

drag-event [*event*]

up-event [*event*]

`down-event` indicates that a mouse button has been pushed. It can be followed by any number of `drag-event`s indicating that the mouse was moved while a button was held down.

The numbers `gx`, `gy`, `x` and `y` play the same rôle as in the case of the `move-event` event. `detail` is a numeric code describing the action of pushing a button. It encodes information about which button was pushed, whether the button single-clicked, double-clicked or triple-clicked, whether a key was being held down at the same time that the button was pushed, etc. The values of `detail` lie between zero and some positive integer, depending upon the system. The zero value corresponds to the simplest manner in which a button can be pushed in the underlying graphics system.

The system guarantees that, after a `down-event` occurs in a window, it receives a matching `up-event`.

`ascii-event` [*event*]

This event indicates that a character has been typed on the keyboard. `detail` represents the ASCII code of the typed character. The window affected by this event is the one that was designated as the recipient of keyboard events, either by the system or by a call to the `current-keyboard-focus-window` function. (See the description of this function.)

`modify-window-event` [*event*]

This event indicates that an agent outside the LE-LISP system has modified the size or the position of `window`. `gx`, `gy`, `w` and `h` represent the new position and size of the window. After an event of this type occurs, the `update-window` function must be called, so that the LISP data structures representing the window are kept up to date.

`kill-window-event` [*event*]

This event indicates that an agent external to the LE-LISP system is requesting the destruction of `window`. After this type of event occurs, and when the program 'agrees' to kill the window, the `kill-window` function must be called to actually carry out this task.

`enterwindow-event` [*event*]

`leavewindow-event` [*event*]

These events indicate that the mouse entered or left a window. `window` is the window concerned.

`repaint-window-event` [*event*]

This event indicates that the system needs to refresh a part of `window`, and is asking LE-LISP to do the work. `x`, `y`, `w` and `h` represent the rectangular zone of the window to be repainted. `x` and `y` are given in the window's local coordinates. When this event occurs, it is possible that the zone has been erased, that is, filled with the `(standard-background)` color.

The standard reaction to this event is to call a function to redraw the window's contents by

drawing the clipping region in the specified refresh zone. This can be done in the following manner:

```
(defun repaint-window (event)
    (let ((window (#:event:window event))
          #:clip:x #:clip:y #:clip:w #:clip:h)
         (with ((current-window window))
               (current-clip) ; get the current clip
               ; clip to the zone
               (current-clip (#:event:x event) (#:event:y event)
                             (#:event:w event) (#:event:h event))
          <repaint the window here>
               ; reposition the old clip
               (current-clip #:clip:x #:clip:y #:clip:w #:clip:h))))
```

This event also makes it possible to synchronize LE-LISP and the underlying window system. In some systems—the X WINDOW system, for example—an arbitrary time can pass between the return of the `create-window` function and the actual creation of the window on the screen. The first `repaint-window-event` indicates that the window has actually been created.

All operations carried out on a window between the occurrence of the `create-window` event and the `repaint-window-event` event are in general lost. It is therefore important to wait for the refresh event before doing anything in the window. For a better understanding of event handling, see the `process-events` function in the *Drawing in windows* section of Chapter 18, as well as the `process-events` function in the examples of `grab-event` in this chapter.

keyboard-focus-event [*event*]

This event indicates that the host system has changed the owner of the keyboard. `window` contains either the new owner or `()` if the owner is no longer a window managed by LE-LISP. After an event of this type, the `current-keyboard-focus-window` function must be called so that the LISP data structures representing the window are kept up to date.

map-window [*event*]

This event occurs when a window comes to the top of all the other windows.

unmap-window [*event*]

This event occurs when a window disappears from the screen or is iconified.

visibility-change [*event*]

This event occurs when the level of visibility of a window changes. The `detail` field of the event is a positive integer with the following possible values:

- `0`: totally visible

- `1`: partly visible
- `2`: totally invisible.

`client-message` [*event*]

This is a communication-type event whose `detail` field contains the message sent, if it is a character string.

19.3 Mouse modes

The mouse can function in several modes, which are determined by the settings of two independent flags, the abbreviated mouse mode flag and the mouse interrupt mode flag.

If the *abbreviated mouse mode* flag is set, the insertion of a new `move-event` in the event queue proceeds according to the following rules:

- If the last event in the queue is also a `move-event`, or a `drag-event`, the new event *replaces* the previous one.
- Otherwise it is *added* to the queue, as in the normal case.

This mechanism allows the size of the event queue to be kept small.

If the *mouse interrupt mode* flag is set, a programmable interrupt named `event` is raised just before any event is added to the queue.

`(event-mode mode)` [*function with an optional argument*]

`mode` becomes the new mouse mode. Without an `event-mode` argument, this function returns the current mouse mode. Used with the `with` control structure, this function can temporarily modify the mouse mode.

The codes are defined as follows:

Mode	Mouse interrupt mode*	Abbreviated mouse mode
0		
2		×
4	×	
6	×	×

* This is the case on some systems only.

At the initialization of the system, the mouse mode is zero. The `event` interrupts are not raised, and the queue does not use abbreviated mouse mode.

19.4 Events queue

Events are instances of the `event` structure, and are placed onto the event queue.

The length of the event queue depends on the particular system. If the queue overflows, the oldest events are lost or, in the worst of cases—as with some X WINDOW implementations—LE-LISP is abruptly killed.

`(eventp)` *[function with no arguments]*

This predicate is true if there is at least one event waiting for service in the queue. This function never modifies the contents of the queue.

`(read-event event)` *[function with an optional argument]*

`(peek-event event)` *[function with an optional argument]*

These functions provide ways to read and examine the next event in the queue. The `read-event` function removes the first event from the queue, while `peek-event` just examines it without removing it.

These functions return an instance of the `event` structure whose fields have been filled with the descriptors of the event being read or examined.

If an argument is provided, it must be an instance of the `event` structure, and it is this structure, duly updated, that is returned. Otherwise, a common global structure instance is always returned, since successive calls to these functions update its contents.

Both `read-event` and `peek-event` call the `bitmap-flush` function before reading the event queue. This guarantees that display operations are actually carried out before user actions are handled.

The following function prints a trace of mouse events:

```
(defun trace-events ()
   (let (e)
    (loop
        (setq e (read-event))
        (print "place " (#:event:x e) ", " (#:event:y e)
               " event " (#:event:code e))))))
```

In general, every program should contain an event-handling routine. The `process-events` function of Chapter 18 is an example of such a routine.

`(flush-event)` *[function with no arguments]*

Empties all waiting events from the events queue.

`(add-event event)` *[function with one argument]*

(add-event gx gy code) [*function with three arguments*]

If there is a single argument, `add-event` copies `event` onto the head of the event queue. This function is particularly useful for extending the available types of events.

When there are three arguments, `add-event` behaves like the following function:

```
(defun add-event3 (x y code)
    (let ((event (#event:make)))
         (#:event:gx event x)
         (#:event:gy event y)
         (#:event:code event code)
         (add-event event)))
```

(grab-event window) [*function with an optional argument*]

Given a window argument, `grab-event` makes it the owner of all the events in the queue. This means that all mouse events (`down-event`, `drag-event`, `up-event` and `move-event`) and keyboard events (`ascii-event`) will then have `window` as their destination. On some systems, where the mouse is shared by several programs, this function also makes LE-LISP the owner of the mouse. Consequently, programs other than LE-LISP no longer receive mouse events.

If the argument has the value `()`, the mouse is treated normally.

Without an argument, `grab-event` returns the LISP window that owns the queue, or `()` if no LISP window has called the `grab-event` function.

This function is particularly useful for forcing the user to confirm that he really wants a certain operation to be performed. A typical example might involve a question from the machine, such as: *Do you really want to erase all your files?*

Making LISP the owner of the mouse might render the system unusable in the case of a LISP error. If this were to happen, it would no longer be possible to select another window with the mouse in order to kill LE-LISP. This function should therefore be used with care.

Let us look at an example. The following function creates a window and blocks all operations—even those meant for or coming from other programs that share the screen with LISP—until the user clicks the mouse in the window. Notice that the `grab-event` function is not called until the occurrence of the first `repaint-window-event` for the confirmation window. This message indicates that the window has actually been created, and it makes it possible to synchronize the 'grabbing' of the mouse with the creation of the window.

```
(defun confirm (string)
    (let ((window (create-window 'window 100 100 100 100 "" 0 1)))
         (protect
             (process-events window string)
             (grab-event ())
             (kill-window window)
             (bitmap-flush)))))

(defun process-events (window string)
```

```
(untilexit done
   (let ((event (read-event)))
        (selectq (#:event:code event)
                 (down-event (exit done))
                 (repaint-window-event
                      (when (eq (#:event:window event) window)
                            (grab-event window)
                            (with ((current-window window))
                                  (draw-string 10 50 string))))
                 ))))
```

(allow-event display event) [*function with two arguments*]

This function lets you add `event` to the list of events handled by the display device called `display`. All windows created after this function is called, will ask the system to receive this new type of event, except if they have specified their own event lists. The function returns the event.

Example:

```
(allow-event (current-display) 'functionkey-event)
```

(disallow-event display event) [*function with two arguments*]

This function lets you remove `event` of the list of the events handled by the display device called `display`. All windows created after this function is called, will ask the system to receive this new type of event, except if they have specified their own event lists. The function returns the event.

Example:

```
(disallow-event (current-display) 'functionkey-event)
```

(allowed-event-p display event) [*function with two arguments*]

This predicate indicates whether `event` belongs to the events handled by the display device called `display`. If this is the case, it returns the event. Otherwise, it returns `()`.

19.5 Programmable interrupts

It must be pointed out that the `event` programmable interrupt is not available in some systems.

event [*programmable interrupt*]

When the mouse interrupt flag is set, the `event` programmable interrupt is triggered just before the posting of an event onto the event queue. (See the `event-mode` function.) In such a case, the event is not added to the event queue. Instead, the interrupt receives as an argument the event that triggered the interrupt.

This interrupt has the same status as the `break` and `clock` interrupts. It is inhibited by the function `with-no-interrupts` and, during the handling of `event` interrupts, all other interrupts are blocked by an implicit call to this same function. The events that occur during the handling of an `event` interrupt are added normally to the queue. The function called by this interrupt to actually handle events must empty the queue before termination.

The default handling of this interrupt consists of adding the event to the queue by using the `add-event` function, as described in the function below.

`(event event)` [*function with one argument*]

This is the standard event-handling function associated with the `event` programmable interrupt.

In LE-LISP, `event` could be defined in the following manner:

```
(defun event (event)
    (add-event event))
```

19.6 Synchronous mouse tracking

The next function informs you, at all times, of the position and state of the mouse.

`(read-mouse event)` [*function with an optional argument*]

Returns an instance of the `event` structure whose fields describe the current position and state of the mouse. If an argument is supplied, it is this structure, duly updated, that is returned. Otherwise, the same structure instance is always returned, since successive calls to `read-mouse` update its contents.

The fields of the structure have the following meanings:

- `gx`, `gy`: global coordinates of the mouse.
- `window`: 'lowest' window (in stacking order) appearing under the current mouse position.
- `x`, `y`: window-local mouse coordinates.
- `detail`: state of the mouse buttons. A zero value indicates that all buttons are up.

19.7 Virtual menus

Virtual menus allow the user to make a choice from among a set of character strings.

A menu proposes a choice in one or several lists of character strings called *selection lists*. The operations actually carried out on the screen in order to post choices, and then validate the particular choice, depend on the system being used. Selection lists can be implemented in one of several ways:

- Pull-down menus hung off a menu bar, in an Apple Macintosh style.
- Wandering menus, as in the Sun Microsystems SunViews product.
- A deck of cards, as in the XMenu package.

(create-menu ti n_1 v_1 ... n_n v_n) [*function with a variable number of arguments*]

Creates and initializes a menu containing the items n_i with values v_i in a selection list named `ti`. The n_i and v_i arguments have the same meanings as those of the `string` and `value` arguments in the `menu-insert-item` function. The selection list and the menu items placed in the menu are all active. Here is an example of the creation of a menu:

```
? (setq menu (create-menu "tty menu"
?                         "aidapaint"  0
?                         "calculator" 1
?                         "function"   2
?                         "file"       3
?                         "help"       4
?                         "icon"       5
?                         "lhoblist"   6
?                         "object"     7
?                         "structure"  8
?                         "end"        9))
= #<menu tty menu>
```

(kill-menu menu) [*function with one argument*]

Destroys `menu`.

(activate-menu menu x y) [*function with three arguments*]

Displays and activates `menu` at the global coordinate point `(x, y)`. This function returns either the value associated with the chosen item, or `()` if the user did not make a choice.

(menu-insert-item-list menu choice string active) [*function with four arguments*]

Inserts a new selection list, named `string`, in the `choice` position of the selection list of `menu`. If `choice` has the value 0, the list becomes the first selection list. If `choice` is greater than or equal to the number of selection lists in the menu, it is added in the last position.

The short integer `active` indicates whether the selection list to be added is selectable (value `1`) or not (value `0`).

The `menu-insert-item-list` function returns the index of the inserted selection list as its value.

(menu-insert-item menu choice item string active value) [*function with six arguments*]

Adds an item to a menu in the `item` position of the `choice` list. `item` indicates the item's row in the menu, counting from zero. `value` is the value associated with the item, and must be a LISP object not equal to `()`. The `activate-menu` function returns this value when the item has been chosen. `active` is a short integer acting as a flag, indicating whether the item is selectable (`active = 1`) or not (`active = 0`).

The `menu-insert-item` function returns the index of the inserted item as its value.

`(menu-delete-item-list menu choice)` [*function with two arguments*]

Removes the selection list in the `choice` row, counting from zero, in `menu`.

`(menu-delete-item menu choice item)` [*function with three arguments*]

Removes `item` from the `choice` selection list of `menu`.

`(menu-modify-item-list menu choice string active)` [*function with four arguments*]

Modifies the selection list whose index is `choice` in `menu`. If either `string` or `active` has the value `()`, it is not modified.

`(menu-modify-item menu choice item string active value)` [*function with six arguments*]

Modifies a menu item. If either `string`, `value` or `active` has the value `()`, it is not modified.

19.8 Cut and Paste

The virtual bitmap proposes two cut-&-paste functions that can work in conjunction with other applications when the screen system allows it.

`(display-store-selection display buffer)` [*function with two arguments*]

Takes the `buffer` character string and copies it into a buffer associated with the display device called `display`. If the display device allows it, `buffer` is shared with all non-LE-LISP applications that use this display device.

`(display-get-selection display)` [*function with one argument*]

This function retrieves a character string representing the current selection in the display device called `display`. If the display device offers this characteristic (this is true for `X11`), this function lets you retrieve a character string coming from another application using the same display device.

Table of contents

19 Virtual mouse **19-1**

19.1 Events 19-1

19.2 Structure and types of events 19-2

19.3 Mouse modes 19-5

19.4 Events queue 19-6

19.5 Programmable interrupts 19-8

19.6 Synchronous mouse tracking 19-9

19.7 Virtual menus 19-9

19.8 Cut and Paste 19-11

Function Index

`event` [*structure*] 19-2
`move-event` [*event*] 19-2
`down-event` [*event*] 19-2
`drag-event` [*event*] 19-2
`up-event` [*event*] 19-2
`ascii-event` [*event*] 19-3
`modify-window-event` [*event*] 19-3
`kill-window-event` [*event*] 19-3
`enterwindow-event` [*event*] 19-3
`leavewindow-event` [*event*] 19-3
`repaint-window-event` [*event*] 19-3
`keyboard-focus-event` [*event*] 19-4
`map-window` [*event*] 19-4
`unmap-window` [*event*] 19-4
`visibility-change` [*event*] 19-4
`client-message` [*event*] 19-5
`(event-mode mode)` [*function with an optional argument*] 19-5
`(eventp)` [*function with no arguments*] 19-6
`(read-event event)` [*function with an optional argument*] 19-6
`(peek-event event)` [*function with an optional argument*] 19-6
`(flush-event)` [*function with no arguments*] 19-6
`(add-event event)` [*function with one argument*] 19-6
`(add-event gx gy code)` [*function with three arguments*] 19-7
`(grab-event window)` [*function with an optional argument*] 19-7
`(allow-event display event)` [*function with two arguments*] 19-8
`(disallow-event display event)` [*function with two arguments*] 19-8

`(allowed-event-p display event)` [*function with two arguments*] 19-8

`event` [*programmable interrupt*] 19-8

`(event event)` [*function with one argument*] 19-9

`(read-mouse event)` [*function with an optional argument*] 19-9

`(create-menu ti` n_1 v_1 ... n_n v_n`)` [*function with a variable number of arguments*] 19-10

`(kill-menu menu)` [*function with one argument*] 19-10

`(activate-menu menu x y)` [*function with three arguments*] 19-10

`(menu-insert-item-list menu choice string active)` [*function with four arguments*]19-10

`(menu-insert-item menu choice item string active value)` [*function with six arguments*] 19-11

`(menu-delete-item-list menu choice)` [*function with two arguments*] 19-11

`(menu-delete-item menu choice item)` [*function with three arguments*]..... 19-11

`(menu-modify-item-list menu choice string active)` [*function with four arguments*]19-11

`(menu-modify-item menu choice item string active value)` [*function with six arguments*] 19-11

`(display-store-selection display buffer)` [*function with two arguments*] 19-11

`(display-get-selection display)` [*function with one argument*] 19-11

Chapter 20

Graphics primitives

This chapter describes the **portable graphics aspects** of the Virtual Bitmap Display (VBD) facility. These primitives constitute a virtual graphics interface that provides you with the means to draw and fill figures and to display text on color bitmap screens.

20.1 Graphics environments

The *graphics environment* is the set of graphics attributes employed by a drawing function. The following attributes are retained by the VBD facility:

- cursor: `current-cursor`
- foreground color: `current-foreground`
- background color: `current-background`
- character font: `current-font`
- fill pattern or texture: `current-pattern`
- line style: `current-line-style`
- drawing (combination) mode: `current-mode`
- clipping zone: `current-clip`.

A separate graphics environment is attached to each window. All the graphics operations use the graphics attributes defined in the graphics environment of the current window.

A set of functions, described below, allows the graphics environment of the current window to be modified.

20.1.1 Current font

A font is a collection of figures used by the `draw-cn`, `draw-string` and `draw-substring` functions to represent characters on the screen. In the Virtual Bitmap Display, a font is represented by a small integer called the font identifier.

A number of fonts is initally available in the system. Some of these fonts have standard uses. (See the `standard-roman-font` function.) A new font can be added to the system at any time—the system simply allocates a new font identifier for it. The name of the fonts available for loading depends of course on the system being used.

`(current-font font)` *[function with an optional argument]*

`font` becomes the font for the current graphics environment. Called without an argument, `current-font` returns the font for the current graphics environment. `font` is a font identifier, that is, an integer between zero and `(font-max)`. The number of utilizable fonts, their image on the screen and their sizes all depend on the system being used.

`(font-max)` *[function with no arguments]*

Returns the biggest legal argument usable in a call to the `current-font` function, that is also equal to the number of fonts available minus one. This number is always greater than or equal to one.

`(load-font string)` *[function with one argument]*

Loads the font described in the system by the character string `string`. It returns a small integer which can be used as a parameter to the `current-font` function. Subsequent calls to the `font-max` function will reflect the fact that this new font was loaded into the system. If the character font named `string` was already loaded, `load-font` will return its font identifier.

The value of the string called `string` depends on the system and is absolutely not portable.

If the font does not exist in the system being used, the `erroob` error is raised, with the following screen display:

```
** load-font: out of bounds : <string>
```

`(font-name font)` *[function with one argument]*

Returns the character string that describes the font whose identifier is `font`, an integer between 0 and `(font-max)`.

20.1.2 Foreground and background colors

The foreground color is the color used for drawing lines and characters and for filling zones when the current fill pattern is `standard-foreground-pattern`. The background is used as the background of character designs and to fill zones when the current fill pattern is `standard-background-pattern`.

Two colors are initally available, `standard-foreground` and `standard-background`. (See the function of the same names.) New colors can be constructed by either using the RGB system of color definition, or by using a Virtual Bitmap Display facility database which allows colors to be named symbolically. Since colors are expensive resources, and since only a few are usually needed, it is suggested that they be freed as soon as they are no longer being used.

One can also allocate dynamically modifiable colors. In general, these are even more expensive resources than the regular colors just described.

(current-foreground color) [*function with an optional argument*]

(current-background color) [*function with an optional argument*]

These function examine or, if an argument is provided, set the foreground and background colors of the current window.

(name-to-rgb name) [*function with one argument*]

Takes the name of a color, `name`, and returns the corresponding RGB values (red, green and blue) in the form of a vector of three integers. If this color does not exist, `()` is returned.

(get-rgb-values pixel) [*function with one argument*]

Takes a color pixel value, `pixel` (of the kind that would be returned by the `byteref` function), and returns the corresponding RGB values (red, green and blue) in the form of a vector of three integers. If this pixel value does not correspond to a color, `()` is returned.

(make-color red green blue) [*function with three arguments*]

`red`, `green` and `blue` are integers in the range of zero to `32767`. `make-color` returns a new color with RGB components equal to the argument values. A value of `32767` indicates saturation. Black and white are represented, therefore, by the following colors:

```
(setq black (make-color 0 0 0))

(setq white (make-color 32767 32767 32767))
```

(make-named-color name) [*function with one argument*]

Returns a new color whose RGB components are those specified in the system color database. A certain system-dependent number of colors are predefined in this database.

```
(setq red (make-named-color "red"))

(setq gr (make-named-color "goldenrod"))
```

(make-mutable-color red green blue) [*function with three arguments*]

This function is similar to the last one but returns a modifiable color, whose RGB components can be changed in the course of a session. In certain cases, this enables you to carry out efficient animations. (See the example below.)

(kill-color color) [*function with one argument*]

Frees the color `color`. A color can no longer be used after this function has been called on it.

(red-component color value) [*function with one or two arguments*]

(green-component color value) [*function with one or two arguments*]

(blue-component color value) [*function with one or two arguments*]

Given a single argument these functions return the value of the corresponding (red, green, or blue) component of `color`. If two arguments are supplied, `color` must be a modifiable color. (See the `make-mutable-color` function.) These functions then modify the value of the corresponding (red, green or blue) component of the color. `value` has the same meaning here as in the argument lists of the `make-color` and `make-mutable-color` functions.

Here is an example of the use of modifiable colors in an animation:

```
(defun modifiable-colors (n)
    ; return a list of n modifiable colors
    (let ((res ()))
         (repeat n
                 (newl res (make-mutable-color 0 0 0)))
         res))

(defun draw-bar (colors)
    ; draw the form to animate,
    ; fill it with the foreground color
    (current-pattern (standard-foreground-pattern))
    (let ((circolors (apply 'cirlist colors)))
         (for (i 150 -3 1)
              (current-foreground (nextl circolors))
              (fill-circle 150 50 i))))

(defun animate-bar (colors)
    ; animate the colors in the color list
    (let ((circolors (apply 'cirlist colors)))
         (while t
                (red-component (nextl circolors) 0)
                (red-component (car circolors) 32767)
                (bitmap-flush)
                (flush-event))))

(defun animation (n)
    (with ((current-window (create-window '#:image:rectangle:window
                                          100 100 300 100 "tube" 1 1)))
          (let ((colors (modifiable-colors n)))
               (protect
                     (progn (draw-bar colors)
                            (animate-bar colors))
                     (mapc 'kill-color colors)
```

```
(kill-window (current-window))
(bitmap-flush)))))
```

Test the example with `(animation 5)` or even `(animation 45)`. Quit by typing CONTROL-C.

20.1.3 Cursor

The cursor is the mouse's image on the screen. Three predefined cursors—`standard-lelisp-cursor`, `standard-gc-cursor` and `standard-busy-cursor`—are available.

`(current-cursor cursor)` [*function with an optional argument*]

`cursor` becomes the cursor for the current screen. Called without an argument, `current-cursor` returns the current cursor. `cursor` is a small positive integer which designates a cursor. The number of different cursors available and their image on the screen depend on the system being used.

`(cursor-max)` [*function with no arguments*]

Returns the largest legal argument usable in a call to the function `current-cursor`.

`(make-cursor b1 b2 x y)` [*function with four arguments*]

Returns a new cursor identifier. `b1` and `b2` are bitmaps that hold the foreground drawing and the background pattern of the cursor. `x` and `y` are the coordinates of the 'hot point' of the cursor.

`(make-named-cursor name)` [*function with one argument*]

This function loads a new cursor whose name, `name`, must be a character string from the following list:

- `circle`: transparent circle
- `cross`: crossroads
- `crosshair`: thin cross
- `diamond-cross`: diamond composed of four triangles
- `dot`: full circle
- `exchange`: icon representing an exchange of two entities
- `compass`: north/south/east/west icon.
- `left-hand`: handing pointing left
- `heart`: heart

- `x-cross`: x-shaped cross
- `left-ptr`: arrow pointing left
- `mouse`: computer mouse
- `pencil`: pencil
- `pirate`: skull and crossbones
- `plus`: addition symbol
- `question`: question mark
- `sizing`: icon representing a size change
- `spray`: spray can
- `watch`: watch indicating wait time
- `i-beam`: vertical insertion bar.

The new cursor is only inserted into the list of available cursors if it was not there already. The named cursors are portable, since they are similar on all Virtual Bitmap implementations.

`(cursor-name num)` *[function with one argument]*

Takes a cursor number (such as the `current-cursor` function would supply), and returns either the character string corresponding to the name of this cursor, if it is a named cursor, or `()` if this is not the case. If this number does not correspond to a loaded cursor, an error is raised.

`(move-cursor x y)` *[function with two arguments]*

Displaces the current cursor to the global coordinate-relative point `(x,y)` on the screen.

20.1.4 Line style

The line style describes the way that lines are drawn. The `draw-polyline`, `draw-rectangle`, `draw-circle` and `draw-ellipse` functions all use lines that are affected by the line style. A number of different styles—such as dotted and double-dashed—are initally available, depending on the system being used.

`(current-line-style line-style)` *[function with an optional argument]*

`line-style` becomes the line style for the current window. Called without an argument, `current-line-style` returns the current line style. `line-style` is a small positive integer which designates a line style. The number of different available styles and their images on the screen depend on the system being used.

(line-style-max) [*function with no arguments*]

Returns the biggest legal argument usable in a call to the `current-line-style` function.

(make-line-style `width style`) [*function with two arguments*]

Creates a new line-drawing style and returns its number. The new style is defined by `width`, expressed in pixels. and `style`, which is a positive integer that will interpreted by the particular kind of display device. For example, in X11, there are three possible styles: `0`, `1` and `2`.

20.1.5 Fill patterns (or textures)

The fill functions `fill-area`, `fill-rectangle`, `fill-circle` and `fill-ellipse` can use single-color or two-color patterns. A pattern or texture is located by use of a pattern identifier, a number between zero and the value of `(pattern-max)`.

Single-color patterns called `standard-foreground-pattern` and `standard-background-pattern` can be used to fill forms with the current foreground or background color.

The other patterns have two color values which are independent of the foreground and background colors. A number of two-color patterns are initally available which provide a set of 'gray-tones', or mixtures of colors. (See the `standard-gray-pattern` function.)

New two-color patterns can be created using icons as the starting point. The pattern will do a fill by regularly repeating the icon image on the screen.

(current-pattern `pattern`) [*function with an optional argument*]

`pattern` becomes the pattern for the current graphics environment. Called without an argument, `current-pattern` returns the current pattern. `pattern` is a small positive integer which designates a pattern. The number of different patterns available and their image on the screen depends on the system being used.

(pattern-max) [*function with no arguments*]

Returns the biggest legal value which can be used in a call to the `current-pattern` function.

(make-pattern `bitmap`) [*function with one argument*]

Returns a new pattern identifier. The use of this pattern in a fill command will result in the duplification of the icon `bitmap` to fill the zone in question. Patterns made in this way will use the icon's colors and so are independent of the foreground and background colors.

20.1.6 Drawing (combination) mode

In drawing operations, the points drawn onto the screen (the *source points* or *source image*) can be combined in various ways with the points already displayed on the screen (the *destination points*

or *destination image*). The combination, or drawing, mode does not have much significance when more than two colors are being used. This mode only works, then, with drawing operations using the predefined `standard-foreground` and `standard-background` colors.

(current-mode mode) [*function with an optional argument*]

`mode` becomes the drawing mode for the current graphics environment. Called without an argument, `current-mode` returns the current drawing mode. `mode` is a small integer which designates a drawing, or combination, mode. Sixteen modes are available, numbered from zero to `15`. Each mode indicates the manner in which the drawing primitives will combine a pixel from the graphics environment (stored images, textures, etc.) and a pixel displayed on the screen.

The modes are defined as a function of the source pixel, `S`, and the destination pixel, `D`, in the following manner:

S	D	Resulting pixel															
Mode		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

#:mode:set [*variable*]

#:mode:or [*variable*]

#:mode:xor [*variable*]

#:mode:not [*variable*]

These variables contain the most often-used drawing modes. They have the following values:

```
#:mode:set 3
#:mode:or  7
#:mode:xor 6
#:mode:not 12
```

20.1.7 Clipping zone

All graphics operations will draw only that part of the requested image which, based on the position at which the image is to be attached to the current window, will be included therein. One can

furthermore specify a rectangular zone within the window to which images will be limited. This is particularly useful for treating `repaint-window-event`s, when it is only necessary to redraw the part of the image effected by the event. (See the `read-event` function.)

`(current-clip x y w h)` *[function with zero or four arguments]*

The rectangle with upper left-hand corner at the point `(x, y)`, with width `w` and height `h` becomes the current clipping rectangle. Called without an argument, `current-mode` returns the current clipping rectangle in the global variables `#:clip:x`, `#:clip:y`, `#:clip:w` and `#:clip:h`. All display operations are limited by this clipping rectangle. So, only those operations requested to be carried out within this region will be performed.

`#:clip:x` *[variable]*

`#:clip:y` *[variable]*

`#:clip:w` *[variable]*

`#:clip:h` *[variable]*

These variables contain the coordinates of the current clipping rectangle after a call to the `current-clip` function without arguments.

20.2 Graphics primitives

The absolutely lowest-level primitives are the four functions defined in the GKS graphics standard, these being `draw-polyline`, `draw-polymarker`, `fill-area` and `draw-substring`. For efficiency reasons, extensions to these primitives are furnished in LE-LISP. These extensions for the most part are dedicated to the display and filling of rectangular zones, operations which are often very rapid on bitmap screens. These functions could, of course, be defined using GKS primitives.

`(draw-polyline n vx vy)` *[function with three arguments]*

`vx` and `vy` are two vectors of length greater than or equal to `n`. This function draws the `n` vectors defined by the sequence of points `(vx[0], vy[0]) ... (vx[n-1], vy[n-1])`. The line style used to draw the vectors is determined by the value of the current `line-style` parameter. The current drawing mode determines how the vectors will be combined with objects already drawn on the screen.

For example, the following call draws a triangle:

```
(draw-polyline 4 #[0 10 20 0] #[0 10 0 0])
```

(draw-polymarker n vx vy) [*function with three arguments*]

This function is similar to the previous one, but draws only the extremities—the endpoints—of the vectors. The current drawing mode determines how the vector endpoints will be combined with objects already present on the screen.

For example, the following call draws the vertices of a triangle:

```
(draw-polymarker 4 #[0 10 20 0] #[0 10 0 0])
```

(fill-area n vx vy) [*function with three arguments*]

`vx` and `vy` are two vectors of length greater than or equal to `n`. This function fills the polygon defined by the `n` points whose `(x, y)` coordinates are given by the vectors `vx` and `vy`. The polygon is filled by the pattern defined by the current value of the `pattern` parameter. The current drawing mode determines how the polygon will be combined with objects already drawn on the screen.

For example, the following call fills a triangular zone:

```
(fill-area 3 #[0 10 20] #[0 10 0])
```

(draw-substring x y s start length) [*function with five arguments*]

See Chapter 18 and the description there of the minimal graphics environment.

(draw-ellipse x y rx ry) [*function with four arguments*]

Draws an ellipse whose axes are parallel to the two coordinate axes. The coordinate point `(x, y)` is the center of the ellipse, and `rx` and `ry` are integers representing the two half-axes of the ellipse.

(fill-ellipse x y rx ry) [*function with four arguments*]

This function is similar to the last one, except that it fills the ellipse's outline with the current pattern instead of drawing it.

These two ellipse functions are not, strictly speaking, GKS primitives, but most GKS systems provide similar functionality by means of escape functions.

(clear-graph-env) [*function with no arguments*]

See Chapter 18 for a description of the minimal graphics environment.

20.3 Extended graphics functions

20.3.1 Line-drawing functions

These functions draw lines using the line current style, given by `(line-style)`. We give their definitions in terms of GKS or GKS-compatible primitives.

(draw-point x y) [*function with two arguments*]

In LE-LISP, `draw-point` could be defined in the following manner:

```
(defun draw-point (x y)
    (draw-polymarker 1 (vector x) (vector y)))
```

(draw-line x0 y0 x1 y1) [*function with four arguments*]

In LE-LISP, `draw-line` could be defined in the following manner:

```
(defun draw-line (x0 y0 x1 y1)
    (draw-polyline 2 (vector x0 x1) (vector y0 y1)))
```

(draw-rectangle x y w h) [*function with four arguments*]

In LE-LISP, `draw-rectangle` could be defined in the following manner:

```
(defun draw-line (x0 y0 w h)
    (draw-polyline 5 (vector x0 (+ x0 w) (+ x0 w) x0 y0)
         (vector y0 y0 (+ y0 h) (+ y0 h) y0)))
```

(draw-circle x y r) [*function with four arguments*]

In LE-LISP, `draw-circle` could be defined in the following manner:

```
(defun draw-circle (x y r)
    (draw-ellipse x y r r))
```

(draw-arc x y w h angle1 angle2) [*function with six arguments*]

This function draws an arc of a circle or an ellipse in the current window. This arc is defined by the rectangle centered at `x` and `y`, of height `h` and width `w`. It draws only the portion of the ellipse between the two angles `angle1` and `angle2`, expressed in degrees in a trigonometrical sense (with the origin at 3 o'clock). Here is an example that draws the half-circle at `(10, 10)` with a radius of `20`:

```
(draw-arc 10 10 40 40 0 180)
```

(draw-segments n vx1 vy1 vx2 vy2) [*function with five arguments*]

As the outcome of a single call, this function draws `n` segments in the current window. Segment number `i` is defined by its two extremities: `(vx[i], vy[i])` and `(vx2[i], vy2[i])`. The `vx1`, `vy1`, `vx2` and `vy2` arguments are therefore objects of `vector` type including at least `n` elements.

(draw-rectangles n vx vy vw vh) [*function with five arguments*]

As the outcome of a single call, this function draws `n` rectangles. Rectangle number `i` is defined by `(vx[i], vy[i], vw[i], vh[i])`. The `vx`, `vy`, `vw` and `vh` arguments are vectors of integers. If one or several values are identical for all the rectangles, it is possible to pass a single integer rather than a vector whose values all identical.

20.3.2 Fill functions

These functions fill zones using the current fill pattern, given by `(pattern)`. We give their definitions in terms of GKS primitives.

(fill-rectangle x y w h) [*function with four arguments*]

In LE-LISP, `fill-rectangle` could be defined in the following manner:

```
(defun fill-rectangle (x y w h)
    (fill-area 4 (vector x0 (+ x0 w) (+ x0 w) x0)
                 (vector y0 y0 (+ y0 h) (+ y0 h))))
```

(fill-circle x y r) [*function with three arguments*]

In LE-LISP, `fill-circle` could be defined in the following manner:

```
(defun fill-circle (x y r)
    (fill-ellipse x y r r))
```

(fill-arc x y w h angle1 angle2) [*function with six arguments*]

This function fills in a portion of a pie chart (circle graph) in the current window. The arc that is filled in is defined by the rectangle centered at `x` and `y`, of height `h` and width `w`. It only fills in the portion of the ellipse between the two angles `angle1` and `angle2`, expressed in degrees in a trigonometrical sense (with the origin at 3 o'clock). Here is an example that draws the half-circle with its center at `(10, 10)`, and a radius of `20`:

```
(fill-arc 10 10 20 20 0 180)
```

(fill-rectangles n vx vy vw vh) [*function with five arguments*]

Fills `n` rectangles in a single call. Rectangle number `i` is defined by `(vx[i], vy[i], vw[i], vh[i])`. The `vx`, `vy`, `vw` and `vh` arguments are vectors of integers. If one or several values are identical for all the rectangles, it is possible to pass a single integer rather than a vector whose values all identical.

(highlight-rectangle x y w h) [*function with four arguments*]

This function inverses a rectangle, defined as `(x, y, w, h)`, by interchanging the background color, `current-background`, with the drawing color, `current-foreground`, and vice versa. It therefore causes inversion in a color-screen system. In a monochrome system, it has the same effect as `fill-rectangle` in `#:mode:xor` mode. Points that start out with neither the background nor the foreground color end up with an indeterminate color.

20.3.3 Displaying text

These functions are defined in Chapter 18. They are mentioned here to remind the reader of their existence and form.

(draw-string x y s) [*function with three arguments*]

(draw-cn x y cn) [*function with three arguments*]

(width-substring string start length) [*function with three arguments*]

(height-substring string start length) [*function with three arguments*]

(x-base-substring string start length) [*function with three arguments*]

(y-base-substring string start length) [*function with three arguments*]

(x-inc-substring string start length) [*function with three arguments*]

(y-inc-substring string start length) [*function with three arguments*]

20.3.4 Bitmaps

bitmap [*structure*]

A bitmap is a two-dimensional bit array implemented in the form of a LISP structure as defined below.

The current version of the Virtual Bitmap Display facility manipulates only tow-color bitmaps, with set bits (value `1`) painted with the foreground color and the cleared bits (value `0`) painted with the background color. The values of these two colors are determined, for any particular bitmap, at the moment the bitmap is created.

```
(defstruct bitmap
    w
    h
    extend
    display)
```

The `w` and `h` fields contain the width and height of the bitmap. These are short integers. The contents of the `extend` field are system-dependent, although it often contains a pointer to an external memory zone containing the bitmap bits. This field cannot be manipulated by the user. The `w` and `h` fields can only be examined via the `#:bitmap:w` and `#:bitmap:h` functions. The bitmap's set of image points can be examined and affected by the `#:bitmap:bits` function, described below.

(create-bitmap w h bits) [*function with two or three arguments*]

Creates and returns a new bitmap. The optional third argument is analogous to the second argument of the `#:bitmap:bits` function. It allows the bitmap's set of image points to be initalized from the point store.

(window-bitmap window) [*function with one argument*]

Returns the bitmap store of the graphics environment of `window`. This bitmap store is shared with the graphics environment of `window`. All operations on one of these are therefore reflected in the other.

(#:bitmap:w bitmap) [*function with one argument*]

(#:bitmap:h bitmap) [*function with one argument*]

These two functions report the values of the `w` and `h` fields of the bitmap passed as their argument.

(#:bitmap:bits bitmap bits) [*function with one or two arguments*]

Given a single argument, `#:bitmap:bits` returns a LISP object representing the set of image points of `bitmap`. With two arguments, it copies the set of points described by the LISP object `bits` into `bitmap`. This function is particularly useful for reading and writing bitmaps, and for loading a set of predefined bitmaps, for example a set of icons used by another programming language processor.

A set of points is represented by a vector of character strings, these being of type `bitvector`. Each string represents a line of the bitmap. The first character of the first string describes the first eight bits of the first line of the bitmap. The most significant bit in this string corresponds to the first (leftmost) bit in the bitmap.

If a string does not contain enough characters to fill a line of the bitmap, the remaining bits are set to a system-dependent 'garbage' value. If a string contains more characters than needed to fill a line, the surplus characters are ignored and do no harm.

#* [#-macro]

This macro allows character strings of type `bitvector` to be filled with hexadecimal values.

For example, `#*aa` is read as a character string of type `bitvector` containing the eight bits `10101010`. And `#*012a10` is read as a character string of type `bitvector` containing the 24 bits `000000010010101000010000`.

(#:bitvector:prin bitvector) [*function with one argument*]

(#:bitmap:prin bitmap) [*function with one argument*]

These functions are used by the system to print a bitmap. If the `#:system:print-for-read` flag is set, the bits of the bitmap are also be printed. This makes it possible for bitmaps to be easily saved into files.

The format used to write a bitmap is determined by the flag that we now present.

#:system:compressed-icon [*variable*]

If this variable is not equal to `()`, bitmaps will be printed using a compressed mode which gains a space factor of around five. This is reflected in proportionally faster read times, too. If this flag is not set, then bitmaps are simply printed as they appear.

It is not necessary to set this flag when reading in bitmaps, for the read format is automatically determined from the read source. It is strongly suggested that the flag be set when doing I/O on very large bitmaps.

#b w h bits [#-macro]

This `#`-macro reads in a bitmap. The required parameters `w` and `h` indicate the width and height of the bitmap. The optional `bits` parameter is the LISP representation of the image points of the `bits` bitmap. It must be a vector of character strings, each of which is a string of type `bitvector`.

```
#B(24 13)           is read as     #.(create-bitmap 24 13)
#B(2 2 #[#*8 #*4])  is read as     #.(create-bitmap 2 2 #[#*8 #*4])
```

The latter example refers to the following little 2×2 bitmap:

```
+-+-+
|*| |
+-+-+
| |*|
+-+-+
```

```
(defsharp |B| ()
     (ncons (apply 'create-bitmap (read))))
```

(kill-bitmap bitmap) [*function with one argument*]

`kill-bitmap` liberates the memory space used by the bitmap passed to it as an argument. This function is necessary for bitmaps that contain pointers to a memory zone external to the LE-LISP system. A bitmap should never be used again after this function has been applied to it.

```
(defun kill-bitmap (bitmap)
    (to-bitmap 'kill-bitmap bitmap))
```

(bmref bitmap x y) [*function with three arguments*]

(bmset bitmap x y bit) [*function with four arguments*]

These functions are used to examine and modify the bit at coordinate `(x, y)` in `bitmap`.

(bitblit b1 b2 x1 y1 x2 y2 w h) [*function with eight arguments*]

Copies the rectangle whose upper left-hand corner is `(x2, y2)`, with width `w` and height `h`, from the bitmap `b2` into the bitmap `b1` at coordinate point `(x1, y1)`.

The points of the source zone in `b2` combine with the destination bits in `b1` in one or other of the following modes:

- Current combination mode (see the `current-mode` function) if the bitmap `b1` is a window.
- Otherwise, in global (top window) mode.

The value returned by `bitblit` has no particular meaning.

`b1` and `b2` can represent the same bitmap, and this makes it possible to perform animation, scrolling and other essential graphics operations.

In the following example, the `joke` function scrolls the screen—the bitmap of the root window—from left to right:

```
(defun joke ()
    (let ((b (window-bitmap (root-window))))
         (for (i 0 1 (bitxmax))
              (bitblit b b i 0 0 0 (bitxmax) (bitymax))
              (bitmap-flush))))
```

#:bitmap:bytemap [*structure*]

The `bytemap` type is a sub-type of `bitmap` type, with no additional fields. It lets you manipulate bitmaps in 256 colors.

You can create a color bitmap by calling the `create-bytemap` function.

(create-bytemap `w h bytes table`) [*function with two to four arguments*]

Allocates a `bytemap` structure whose dimensions are `w` and `h`. The third argument, if it is supplied, is the vector of byte strings indicating the index of the color of each point (from `0` to `255`). The fourth argument lets you specify the color table to be used to initialize the bitmap. For example, we can write the following code:

```
? (create-bytemap 10 10)
= #BC(10 10 #[])
?
? (create-bytemap 10 10
        #[#*00010203040500010203
          #*      ...
                  ...               ]
              '(#[0 0 0 0 ()]
                #[1 24576 24576 24576 ()]
                #[2 0 24576 24576 ()]
                #[3 24576 0 0 ()]
                #[4 0 0 24576 ()]
                #[5 0 24576 0 t]))
= #BC(10 10 #[])
```

Each vector of the color table is composed of the following values, in this order:

- Color number
- Red component
- Green component
- Blue component
- Indicator of a mutable color.

Each occurrence of a color number will be replaced, if need be, by the the closest color in the current color table (obtained through the `all-colors` function). Mutable colors are replaced, if possible, by the closest mutable colors in the color table. The 'closeness' between two colors is evaluated by obtaining the sum of differences, component by component.

#BC [*#-macro*]

The `B` macro-character lets you define color bitmaps. Consider the following examples of code:

```
#BC(10 10)
```

and

```
(create-bytemap 10 10)
```

They are equivalent.

(#:bitmap:bytes bytemap bytes) [*function with one or two arguments*]

This function lets you set or consult the totality of `bytemap`.

(byteref bytemap i j) [*function with three arguments*]

For the point whose coordinates are `i` and `j`, this function returns the index of its color in the color table.

(byteset bytemap i j col) [*function with four arguments*]

This function sets the point whose coordinates are `i` and `j` to the color whose index is `col` in the color table.

(substitute-color bytemap oldcol newcol) [*function with three arguments*]

This function lets you change the `oldcol` color to `newcol` throughout the entire `bytemap`.

(subst-colors bytetmap l) [*function with two arguments*]

The `l` argument is a list of the form `((old1. new1)... (oldn. newn))`. Here, `oldi` and `newi` are instances of the `color` class. This function lets you replace, in `bytemap`, all the points whose color is `oldi` by the color `newi`. This function has no effect on a monochrome bitmap. The `subst-colors` function returns the obsolete `substitute-color` function.

(send '#:bitmap:bytemap:prin bitmap) [*method with one argument*]

This function is used by the system to print a color bitmap. If the `#:system:print-for-read` indicator is set, the bits of the bitmap are also printed, as well as the current color table.

20.3.5 Compatibility between types of bitmaps

The functions described above, as well as the similar functions that operate on black and white bitmaps (`bmset`, `bmref` ...), can be applied both to objects of `bitmap` type and to those of `bytemap` type.

You can define and use color bitmaps in a monochrome environment. The `create-bytemap` function then creates `bitmap` objects, and all the functions that operate on `bytemap` objects behave in the same way as those that operate on `bitmap` objects.

Table of contents

20 Graphics primitives **20-1**

20.1 Graphics environments 20-1

20.1.1 Current font 20-1

20.1.2 Foreground and background colors 20-2

20.1.3 Cursor 20-5

20.1.4 Line style 20-6

20.1.5 Fill patterns (or textures) 20-7

20.1.6 Drawing (combination) mode 20-7

20.1.7 Clipping zone 20-8

20.2 Graphics primitives 20-9

20.3 Extended graphics functions 20-10

20.3.1 Line-drawing functions 20-10

20.3.2 Fill functions 20-12

20.3.3 Displaying text 20-13

20.3.4 Bitmaps 20-13

20.3.5 Compatibility between types of bitmaps 20-18

Function Index

`(current-font font)` [*function with an optional argument*] 20-2
`(font-max)` [*function with no arguments*] 20-2
`(load-font string)` [*function with one argument*] 20-2
`(font-name font)` [*function with one argument*] 20-2
`(current-foreground color)` [*function with an optional argument*] 20-3
`(current-background color)` [*function with an optional argument*] 20-3
`(name-to-rgb name)` [*function with one argument*] 20-3
`(get-rgb-values pixel)` [*function with one argument*] 20-3
`(make-color red green blue)` [*function with three arguments*] 20-3
`(make-named-color name)` [*function with one argument*] 20-3
`(make-mutable-color red green blue)` [*function with three arguments*] 20-3
`(kill-color color)` [*function with one argument*] 20-4
`(red-component color value)` [*function with one or two arguments*] 20-4
`(green-component color value)` [*function with one or two arguments*] 20-4
`(blue-component color value)` [*function with one or two arguments*] 20-4
`(current-cursor cursor)` [*function with an optional argument*] 20-5
`(cursor-max)` [*function with no arguments*] 20-5
`(make-cursor b1 b2 x y)` [*function with four arguments*] 20-5
`(make-named-cursor name)` [*function with one argument*] 20-5
`(cursor-name num)` [*function with one argument*] 20-6
`(move-cursor x y)` [*function with two arguments*] 20-6
`(current-line-style line-style)` [*function with an optional argument*] 20-6
`(line-style-max)` [*function with no arguments*] 20-7
`(make-line-style width style)` [*function with two arguments*] 20-7
`(current-pattern pattern)` [*function with an optional argument*] 20-7
`(pattern-max)` [*function with no arguments*] 20-7

(make-pattern bitmap) [*function with one argument*] 20-7
(current-mode mode) [*function with an optional argument*] 20-8
#:mode:set [*variable*] 20-8
#:mode:or [*variable*] 20-8
#:mode:xor [*variable*] 20-8
#:mode:not [*variable*] 20-8
(current-clip x y w h) [*function with zero or four arguments*] 20-9
#:clip:x [*variable*] 20-9
#:clip:y [*variable*] 20-9
#:clip:w [*variable*] 20-9
#:clip:h [*variable*] 20-9
(draw-polyline n vx vy) [*function with three arguments*] 20-9
(draw-polymarker n vx vy) [*function with three arguments*] 20-10
(fill-area n vx vy) [*function with three arguments*] 20-10
(draw-substring x y s start length) [*function with five arguments*] 20-10
(draw-ellipse x y rx ry) [*function with four arguments*] 20-10
(fill-ellipse x y rx ry) [*function with four arguments*] 20-10
(clear-graph-env) [*function with no arguments*] 20-10
(draw-point x y) [*function with two arguments*] 20-11
(draw-line x0 y0 x1 y1) [*function with four arguments*] 20-11
(draw-rectangle x y w h) [*function with four arguments*] 20-11
(draw-circle x y r) [*function with four arguments*] 20-11
(draw-arc x y w h angle1 angle2) [*function with six arguments*] 20-11
(draw-segments n vx1 vy1 vx2 vy2) [*function with five arguments*] 20-11
(draw-rectangles n vx vy vw vh) [*function with five arguments*] 20-12
(fill-rectangle x y w h) [*function with four arguments*] 20-12
(fill-circle x y r) [*function with three arguments*] 20-12
(fill-arc x y w h angle1 angle2) [*function with six arguments*] 20-12
(fill-rectangles n vx vy vw vh) [*function with five arguments*] 20-12
(highlight-rectangle x y w h) [*function with four arguments*] 20-13
(draw-string x y s) [*function with three arguments*] 20-13
(draw-cn x y cn) [*function with three arguments*] 20-13
(width-substring string start length) [*function with three arguments*] 20-13
(height-substring string start length) [*function with three arguments*] 20-13

`(x-base-substring string start length)` [*function with three arguments*] 20-13
`(y-base-substring string start length)` [*function with three arguments*] 20-13
`(x-inc-substring string start length)` [*function with three arguments*] 20-13
`(y-inc-substring string start length)` [*function with three arguments*] 20-13
`bitmap` [*structure*] 20-13
`(create-bitmap w h bits)` [*function with two or three arguments*] 20-14
`(window-bitmap window)` [*function with one argument*] 20-14
`(#:bitmap:w bitmap)` [*function with one argument*] 20-14
`(#:bitmap:h bitmap)` [*function with one argument*] 20-14
`(#:bitmap:bits bitmap bits)` [*function with one or two arguments*] 20-14
`#*` [*#-macro*] 20-15
`(#:bitvector:prin bitvector)` [*function with one argument*] 20-15
`(#:bitmap:prin bitmap)` [*function with one argument*] 20-15
`#:system:compressed-icon` [*variable*] 20-15
`#b w h bits` [*#-macro*] 20-15
`(kill-bitmap bitmap)` [*function with one argument*] 20-16
`(bmref bitmap x y)` [*function with three arguments*] 20-16
`(bmset bitmap x y bit)` [*function with four arguments*] 20-16
`(bitblit b1 b2 x1 y1 x2 y2 w h)` [*function with eight arguments*] 20-16
`#:bitmap:bytemap` [*structure*] 20-16
`(create-bytemap w h bytes table)` [*function with two to four arguments*] 20-17
`#BC` [*#-macro*] 20-17
`(#:bitmap:bytes bytemap bytes)` [*function with one or two arguments*] 20-18
`(byteref bytemap i j)` [*function with three arguments*] 20-18
`(byteset bytemap i j col)` [*function with four arguments*] 20-18
`(substitute-color bytemap oldcol newcol)` [*function with three arguments*] 20-18
`(subst-colors bytetmap l)` [*function with two arguments*] 20-18
`(send '#:bitmap:bytemap:prin bitmap)` [*method with one argument*] 20-18

Index of concepts

***** Fatal error : no room for code. 7-20
***** Fatal error : no room for fixs. 7-20
***** Fatal error : no room for floats. 7-20
***** Fatal error : no room for heap. 7-20
***** Fatal error : no room for lists. 7-19
***** Fatal error : no room for strings. 7-19
***** Fatal error : no room for symbols. 7-20
***** Fatal error : no room for vectors. 7-19
***** Fatal error : stack overflow. 7-20
***** Fatal error : stack underflow. 3-40
** fn : bad address : s 12-2
** fn : bad arguments list : a 2-6
** fn : bad definition : symb 3-17
** fn : bad parameter : e 3-17
** fn : bad parameter : s 2-10
** fn : bad {} abbreviation : e 5-13
** fn : bad type for an hash table : s 3-105
** fn : can't compute : larg 4-2
** fn : illegal bind : (p v) 2-10
** fn : inactive lexical scope : e 3-33
** fn : me'thode inde'finie : l 5-9
** fn : no lexical scope : e 3-33
** fn : not a fix : s 4-13
** fn : not a float : s 4-22
** fn : not a list : e 3-2
** fn : not a number : s 4-25
** fn : not a string : e 3-2
** fn : not a structure : s 5-3
** fn : not a symbol : e 3-2
** fn : not a variable : e 3-2
** fn : not a vector : s 3-99
** fn : not an abbreviation : e 5-13
** fn : not an atom : e 3-1
** fn : not an Hash Table : s 3-105
** fn : out of bounds : s 3-87 3-99
** fn : } not within {}: e 5-13
** fn : redefined function : symb 3-17
** fn : I/O error : n 6-48
** fn : string expected : s 3-86
** fn : string too long : s 3-87

** fn : syntax error : msg 6-3
** fn : undefined function : symb 2-7
** fn : undefined method : l 5-9
** fn : undefined tag : symb 3-38
** fn : undefined variable : symb 2-5
** fn : wrong number of arguments : n 2-8
** fn : wrong number of arguments : s 2-9
** fn: no more windows available: () 18-10
** fn: not a window: e 18-16
** fnt : unknown terminal : symb 15-1
** function : division by zero. 4-1
** load-font: argument hors limite : string 20-2
** load-font: out of bounds : string 20-2
** toplevel : undefined tag : symb 3-38

-A

a 3-1
abbreviated mode (mouse) 19-5
ackermann (function) 3-26
addr 12-1
Aida 6-57
al 3-65
a-link 2-2
A-list 3-65
allcdr
    (l) *Function* 3-8
anonymous function 2-7
arrays 3-103
association list 3-65
autoload 6-54
autoload function 6-54
autoload mode 6-54

-B

backspace (character) 6-8
begin-comment 6-14
#:bitmap:bytemap:prin
    (bitmap) *Method* 20-18
boolean (value) 3-42
boolean value 3-42

-C

car 2-5
cdr 2-5
ch 3-1 6-1
chan 6-47
channel 6-47
character (in Le-Lisp) 3-97
character strings 2-4 6-10
circular list 3-61
circular or shared list 9-12
classes of objects 5-7
clipping zone 20-8
C_LL_FIX 14-13
C_LL_FLOAT 14-14
closed compiler macros 13-3
cn 3-1
code zone 12-1
Color bitmaps 20-16
colors 20-2
comment delimiter 6-14
comments 6-13
compiler macros 13-3
compilers 13-1
complex numbers 10-1
complex standard library file 10-1
Complice (compiler) 13-1
Complice errors 13-19
Complice warnings 13-22
conditional read 6-27
control character 6-8
core-image file 7-11
cpmac (standard library file) 13-3
cursor 20-5
c-val 2-2

-D

debug loop 11-8
debug mode 11-7
debugging 11-7
debugging tools 11-1
dec prl 10-1
delete (character) 6-8
dmacro functions 2-12
dmsubr 2-9
down arrow (keyboard key) 15-7

dynamic execution environment 7-5

-E

edlin (standard library file) 17-1
embedded comments 6-27
end-comment 6-14
end-of-line 6-2
err0dv 4-1
errbal 2-6
errbdf 3-17
errbht 3-105
errbpa 2-10 3-17
errfcns 7-19
errfcod 7-20
errffix 7-20
errfflt 7-20
errfhep 7-20
errfstk 7-20
errfstr 7-19
errfsud 3-40
errfsym 7-20
errfvec 7-19
errgen 4-2
errilb 2-10
errios 6-48
errnaa 3-1
errnab 3-33
errnda 12-2
errnfa 4-22
errnht 3-105
errnia 4-13
errnla 3-2
errnmw 18-10
errnna 4-25
errnotanabbrev 5-13
errnotawindow 18-16
errnsa 3-2 3-86
errnva 3-2
erroob 3-87 3-99 20-2
errstc 5-2
errstl 3-87
errsxt 6-3
errsxtacc 5-13
errsxtclosingacc 5-13

errsym 3-2
errudf 2-7
errudm 5-9
errudt 3-38
errudv 2-5
errvec 3-99
errvirtty 15-1
errwna 2-8 2-9
errxia 3-33
escape 3-38
evaluation of atomic objects 2-5
event loop 6-57
events 19-1
events loop 6-57
evloop 6-57
evloop file in the standard library 6-57
expr 2-9

-F

feature 6-74
fexpr 2-11
file 6-47
file specification 6-47
fill patterns (or textures) 20-7
flambda expression 2-11
fn 3-1
font (of characters) 20-1
form (capable of evaluation) 2-6
format (standard library file) 9-1
formatted output 9-1
Franz Lisp 0-1
fsubr 2-8
f-type 2-2
function call 2-6
function definition 2-13
functions 2-6
f-val 2-2

-G

garbage collector 7-19
global variable 2-6
Graphics environment 20-1
graphics primitives 20-9

-H

hanoi (demonstration file) 15-10
hash tables 3-104
hash-coding 2-3
high 12-1
$HOME/.lelisp 1-4

-I

incremental execution 11-12
inheritance of methods 5-7
input buffer 6-41
input file 6-50
input stream 6-50
Inria 0-1
inspection loop 11-8
installation directory 1-4
internal character codes 3-97
interrupt character 7-3
interrupt mode (mouse) 19-5
intrinsic property 2-2
invisible bit 3-59

-L

l 3-1
labelled cons 2-5 3-59
lambda expression 2-9
lap 12-1
lch 6-2
lcn 6-2
left arrow (keyboard key) 15-7
left print margin 6-44
LeLisp 0-1
lexical analysis 6-14
lexical environment for execution 7-11
libcir (standard library file) 9-12
libraries 6-56
library 6-54
line editor 6-8
line style 20-6
lines 6-1
Lisp Assembly Program 12-1
Lisp Machine Lisp 0-1
list cell 2-5

LL_C_FIX 14-13
LL_C_FLOAT 14-14
llm3 12-1
llm3 instruction 12-5
llm3 label 12-5
llm3 operands 12-5
local variable 2-6
lock function 3-39
low 12-1

-M

machine interrupts 7-3
Maclisp 0-1
macro character 6-18
macro functions 2-12
managing interrupts 7-3
memory access 12-1
memory address 12-1
memory allocator 7-19
methods 5-7
mlambda expression 2-12
msubr 2-8
multi-tasks 7-5

-N

n 3-1
NIL (New Implementation of Lisp) 0-1
nil special symbol 2-5
non-local branch 3-38
nsubr 2-7
numbers 2-3 6-11

-O

object-oriented programming 5-7
open compiler macros 13-4
output channel of the virtual terminal 15-4
output file 6-51
output stream 6-51
o-val 2-2
own variable 2-6

-P

packages 5-7
path 6-63
pathname 6-63
pckgcell 2-3
p-list 2-2
p-name 2-3
programmable interrupts 7-1
prompt string 6-9
p-type 2-2

-Q

Q standard library file 10-1
quote 6-20
quote character 6-14

-R

-r (argument to Le-Lisp command) 1-4
ratio standard library file 10-1
rational numbers 10-1
rationals (numbers) 10-1
read errors 6-3
read table 6-14
reading a character 6-5
reading a line 6-4
reading S-expressions 6-3
real-time clock interrupt 7-4
records (à la Pascal) 5-1
right arrow (keyboard key) 15-7
right print margin 6-45
rubout (character) 6-8

-S

s 3-1
I/O streams 6-47
scheduler standard library file 7-5
sequencer 7-5
S-expressions 2-1
single-step mode 11-12
I/O (input/output) 6-1
snobol 4 3-65
special form 2-8
special symbol ||3-45

standard functions 2-6
standard reader 6-9
startup (standard library file) 7-14
stepwise execution 11-12
strg 3-1 3-86
string delimiter 6-15
string is the default character string type 3-86
stringio 6-40
stringio file in the standard library 6-40
structures 5-1
subr 2-7
switch 3-30
symb 3-1
symbol delimiter 6-10
symbols 2-1

-T

t (symbolic constant) 3-42
tables (association) 3-65
tcons 3-59
termcap (virtual terminal description database) 15-1
terminal input 6-8
terminal output channel 6-47
Terminal Virtuel 6-57
terminfo (virtual terminal description database) 15-1
top-level 7-18
trace 11-1
tracing 11-1
true 3-42
type of a character string 3-86
type of a vector of S-expressions 3-99
typed vector of S-expressions 3-98
types of characters 6-14

-U

^U (character) 6-8
up arrow (keyboard key) 15-7
user interrupt 7-3

-V

variable-functions 3-24
vdt (demonstration file) 15-10

vect 3-1
vector of S-expressions 3-98
vectors of S-expressions 2-5
virtual bitmap 6-60
virtual files 6-63
virtual mouse 19-1
virtual pointing device 19-1
virtual terminal 15-1
Vlisp 0-1
VMS 6-61

-W

whanoi (demonstration file) 15-10

-X

^X (character) 6-8

Index of the LLM3 Machine

`a1` [*LLM3 operand*] 12-6

`a2` [*LLM3 operand*] 12-6

`a3` [*LLM3 operand*] 12-6

`a4` [*LLM3 operand*] 12-6

`(addadr addr1 addr2)` [*function with two arguments*] 12-2

`adjstk op` [*LLM3 instruction*] 12-11

`alink accu` [*LLM3 operand*] 12-13

`alink accu` [*LLM3 operand*] 12-7

`bfcons op lab` [*LLM3 instruction*] 12-12

`bffix op lab` [*LLM3 instruction*] 12-14

`bffloat op lab` [*LLM3 instruction*] 12-16

`bfnil op lab` [*LLM3 instruction*] 12-12

`bfstrg op lab` [*LLM3 instruction*] 12-18

`bfsymb op lab` [*LLM3 instruction*] 12-12

`bfvar op lab` [*LLM3 instruction*] 12-13

`bfvect op lab` [*LLM3 instruction*] 12-17

`bra lab` [*LLM3 instruction*] 12-9

`bri op` [*LLM3 instruction*] 12-9

`brx llab op` [*LLM3 instruction*] 12-9

`btcons op lab` [*LLM3 instruction*] 12-11

`btfix op lab` [*LLM3 instruction*] 12-14

`btfloat op lab` [*LLM3 instruction*] 12-16

`btnil op lab` [*LLM3 instruction*] 12-12

`btstrg op lab` [*LLM3 instruction*] 12-18

`btsymb op lab` [*LLM3 instruction*] 12-12

`btvar op lab` [*LLM3 instruction*] 12-13

`btvect op lab` [*LLM3 instruction*] 12-17

`cabeq op1 op2 lab` [*LLM3 instruction*] 12-9

`cabne op1 op2 lab` [*LLM3 instruction*] 12-9

`calli op` [*LLM3 instruction*] 12-10

`(calln addr l)` [*function with two arguments*] 12-4

`(call addr a1 a2 a3)` [*function with four arguments*] 12-4

`call lab` [*LLM3 instruction*] 12-10
`car accu` [*LLM3 operand*] 12-12
`car accu` [*LLM3 operand*] 12-6
`cdr accu` [*LLM3 operand*] 12-12
`cdr accu` [*LLM3 operand*] 12-6
`cfbeq op1 op2 lab` [*LLM3 instruction*] 12-17
`cfbge op1 op2 lab` [*LLM3 instruction*] 12-17
`cfbgt op1 op2 lab` [*LLM3 instruction*] 12-17
`cfble op1 op2 lab` [*LLM3 instruction*] 12-17
`cfblt op1 op2 lab` [*LLM3 instruction*] 12-17
`cfbne op1 op2 lab` [*LLM3 instruction*] 12-17
`cnbeq op1 op2 lab` [*LLM3 instruction*] 12-15
`cnbge op1 op2 lab` [*LLM3 instruction*] 12-15
`cnbgt op1 op2 lab` [*LLM3 instruction*] 12-15
`cnble op1 op2 lab` [*LLM3 instruction*] 12-15
`cnblt op1 op2 lab` [*LLM3 instruction*] 12-15
`cnbne op1 op2 lab` [*LLM3 instruction*] 12-15
`cvalq symb` [*LLM3 operand*] 12-13
`cvalq symb` [*LLM3 operand*] 12-7
`cval accu` [*LLM3 operand*] 12-13
`cval accu` [*LLM3 operand*] 12-6
`decr op` [*LLM3 instruction*] 12-14
`diff op1 op2` [*LLM3 instruction*] 12-14
`endl` [*LLM3 pseudo-instruction*] 12-8
`end` [*LLM3 pseudo-instruction*] 12-8
`entry symb ftype` [*LLM3 pseudo-instruction*] 12-8
`eval expr` [*LLM3 operand*] 12-8
`eval e` [*LLM3 pseudo-instruction*] 12-8
`fdiff op1 op2` [*LLM3 instruction*] 12-16
`fentry symb ftype` [*LLM3 pseudo-instruction*] 12-8
`fplus op1 op2` [*LLM3 instruction*] 12-16
`fquo op1 op2` [*LLM3 instruction*] 12-16
`ftimes op1 op2` [*LLM3 instruction*] 12-16
`fvalq symb` [*LLM3 operand*] 12-13

`fvalq symb` [*LLM3 operand*] 12-7
`fval accu` [*LLM3 operand*] 12-13
`fval accu` [*LLM3 operand*] 12-6
`(gtadr addr1 addr2)` [*function with two arguments*] 12-3
`hbmovx op strg index` [*LLM3 instruction*] 12-19
`hbxmov strg index op` [*LLM3 instruction*] 12-19
`hgsize op1 op2` [*LLM3 instruction*] 12-19
`hpmovx op vect n` [*LLM3 instruction*] 12-18
`hpxmov vect n op` [*LLM3 instruction*] 12-18
`(incradr addr n)` [*function with two arguments*] 12-2
`incr op` [*LLM3 instruction*] 12-14
`jcall symb` [*LLM3 instruction*] 12-11
`jmp symb` [*LLM3 instruction*] 12-9
`lab` [*LLM3 operand*] 12-7
`land op1 op2` [*LLM3 instruction*] 12-15
`ld-codop` [*constant*] 12-19
`ld-dir` [*constant*] 12-19
`ld-ind` [*constant*] 12-20
`#:ld:shared-strings` [*variable*] 12-5
`#:ld:special-case-loader` [*variable*] 12-4
`(loader l i)` [*function with two arguments*] 12-4
`loader` [*feature*] 12-4
`local symb` [*LLM3 pseudo-instruction*] 12-8
`(loc s)` [*function with one argument*] 12-3
`lor op1 op2` [*LLM3 instruction*] 12-16
`lshift op1 op2` [*LLM3 instruction*] 12-16
`lxor op1 op2` [*LLM3 instruction*] 12-16
`(memory addr n)` [*function with one or two arguments*] 12-3
`movxsp op1 op2` [*LLM3 instruction*] 12-11
`mov op1 op2` [*LLM3 instruction*] 12-9
`negate op` [*LLM3 instruction*] 12-14
`nil` [*LLM3 operand*] 12-6
`nop` [*LLM3 instruction*] 12-10
`@ lab` [*LLM3 operand*] 12-7

`& n` [*LLM3 operand*] 12-7
`oval accu` [*LLM3 operand*] 12-13
`oval accu` [*LLM3 operand*] 12-7
`pkgc accu` [*LLM3 operand*] 12-13
`pkgc accu` [*LLM3 operand*] 12-7
`plist accu` [*LLM3 operand*] 12-13
`plist accu` [*LLM3 operand*] 12-6
`plus op1 op2` [*LLM3 instruction*] 12-14
`pname accu` [*LLM3 operand*] 12-13
`pname accu` [*LLM3 operand*] 12-7
`pop op` [*LLM3 instruction*] 12-11
`push op` [*LLM3 instruction*] 12-11
`quote exp` [*LLM3 operand*] 12-6
`quo op1 op2` [*LLM3 instruction*] 12-14
`rem op1 op2` [*LLM3 instruction*] 12-15
`return` [*LLM3 instruction*] 12-11
`sobgez op lab` [*LLM3 instruction*] 12-10
`sstack op` [*LLM3 instruction*] 12-10
`stack op` [*LLM3 instruction*] 12-10
`(subadr addr1 addr2)` [*function with two arguments*] 12-2
`(#:system:ccode addr)` [*function with an optional argument*] 12-2
`(#:system:ecode)` [*function with no arguments*] 12-2
`times op1 op2` [*LLM3 instruction*] 12-14
`title symb` [*LLM3 pseudo-instruction*] 12-8
`typ accu` [*LLM3 operand*] 12-18
`typ accu` [*LLM3 operand*] 12-19
`typ accu` [*LLM3 operand*] 12-7
`(vag addr)` [*function with one argument*] 12-3
`val accu` [*LLM3 operand*] 12-18
`val accu` [*LLM3 operand*] 12-19
`val accu` [*LLM3 operand*] 12-7
`xspmov op1 op2` [*LLM3 instruction*] 12-11

Index of LE-LISP entities

`(abbrevp symb)` [*function with one argument*] 5-14
`abbrev` [*feature*] 5-13
`(abs x)` [*function with one argument*] 4-5
`#^` [*#-macro*] 6-25
`~*` [*print format*] 9-9
`~?` [*print format*] 9-10
`~A` [*print format*] 9-3
`~B` [*print format*] 9-5
`~C` [*print format*] 9-6
`~D` [*print format*] 9-4
`~E` [*print format*] 9-6
`~F` [*print format*] 9-7
`~G` [*print format*] 9-8
`~O` [*print format*] 9-5
`~P` [*print format*] 9-5
`~R` [*print format*] 9-4
`~S` [*print format*] 9-4
`~T` [*print format*] 9-9
`~X` [*print format*] 9-5
`~[;]` [*print format*] 9-10
`~#\lf` [*print format*] 9-9
`~%` [*print format*] 9-8
`~{}` [*print format*] 9-11
`~~` [*print format*] 9-8
`~^` [*print format*] 9-11
`^` [*macro character*] 6-20
`^A` [*macro character*] 6-31
`^E` [*macro character*] 6-31
`^F` [*macro character*] 6-31
`^L` [*macro character*] 6-30
`^P` [*macro character*] 6-32

(acons s1 s2 al) [*function with three arguments*] 3-65
(acosh z) [*function with one argument*] 10-10
(acos x) [*function with one argument*] 4-10
(activate-menu menu x y) [*function with three arguments*] 19-10
(add-event event) [*function with one argument*] 19-6
(add-event gx gy code) [*function with three arguments*] 19-7
(add-feature symb) [*function with one argument*] 6-75
(add1 x) [*function with one argument*] 4-13
(addprop pl pval ind) [*function with three arguments*] 3-76
(add x y) [*function with two arguments*] 4-13
(adjoin item list eq-func) [*function with two or three arguments*] 3-107
(all-colors screen) [*function with an optional argument*] 18-6
(allcar l) [*function with one argument*] 3-8
(allow-event display event) [*function with two arguments*] 19-8
(allowed-event-p display event) [*function with two arguments*] 19-8
(alphalessp str1 str1) [*function with two arguments*] 3-91
(and s_1 ... s_n) [*special form*] 3-28
(any fn l_1 ... l_n) [*function with a variable number of arguments*] 3-11
(append1 l s) [*function with two arguments*] 3-55
(append l_1 ... l_n) [*function with a variable number of arguments*] 3-54
(apply fn s_1 ... s_n l) [*function with a variable number of arguments*] 3-7
(aref array i_1 ... i_n) [*function with a variable number of arguments*] 3-103
(arg n) [*function with an optional argument*] 3-5
array [*feature*] 3-103
ascii-event [*event*] 19-3
(asciip cn) [*function with one argument*] 3-98
(ascii cn) [*function with one argument*] 3-97
(aset array i_1 ... i_n e) [*function with a variable number of arguments*] 3-104
(asinh z) [*function with one argument*] 10-9
(asin x) [*function with one argument*] 4-10
(assoc s al) [*function with two arguments*] 3-66
(assq symb al) [*function with two arguments*] 3-65
(at-end return-code) [*function with an optional argument*] 7-17
at-end [*programmable interrupt*] 7-17

(atanh z) [*function with one argument*] 10-10
(atan x) [*function with one argument*] 4-10
(atomp s) [*function with one argument*] 3-43
(atom s) [*function with one argument*] 3-43
(autoload file sym$_1$ ... sym$_n$) [*special form*] 6-54
(backtrack symb l e$_1$ ... e$_n$) [*special form*] 3-41
#<base>r [*#-macro*] 6-25
#BC [*#-macro*] 20-17
(bitblit b1 b2 x1 y1 x2 y2 w h) [*function with eight arguments*] 20-16
(bitepilogue screen) [*function with an optional argument*] 18-4
(bitmap-flush screen) [*function with an optional argument*] 18-6
(bitmap-refresh screen) [*function with an optional argument*] 18-6
(bitmap-restore screen) [*function with one argument*] 18-5
(bitmap-save screen) [*function with an optional argument*] 18-4
(bitmap-sync display) [*function with an optional argument*] 18-6
(#:bitmap:bits bitmap bits) [*function with one or two arguments*] 20-14
(send '#:bitmap:bytemap:prin bitmap) [*method with one argument*] 20-18
#:bitmap:bytemap [*structure*] 20-16
(#:bitmap:bytes bytemap bytes) [*function with one or two arguments*] 20-18
(#:bitmap:h bitmap) [*function with one argument*] 20-14
(#:bitmap:prin bitmap) [*function with one argument*] 20-15
(#:bitmap:w bitmap) [*function with one argument*] 20-14
bitmap [*structure*] 20-13
(bitprologue name device) [*function with one or two optional arguments*] 18-3
(#:bitvector:prin bitvector) [*function with one argument*] 20-15
(bitxmax screen) [*function with an optional argument*] 18-5
(bitymax screen) [*function with an optional argument*] 18-5
(block symb e$_1$ ... e$_n$) [*special form*] 3-33
(bltscreen sd ss w h) [*function with four or twelve arguments*] 15-9
(bltstring str1 n1 str2 n2 n3) [*function with four or five arguments*] 3-94
(bltvector vect1 n1 vect2 n2 n3) [*function with five arguments*] 3-102
(blue-component color value) [*function with one or two arguments*] 20-4
(bmref bitmap x y) [*function with three arguments*] 20-16
(bmset bitmap x y bit) [*function with four arguments*] 20-16

(boblist n) [*function with an optional argument*] 3-85
(bol) [*function with no arguments*] 6-43
bol [*programmable interrupt*] 6-42
(boundp symb) [*function with one argument*] 3-69
(break) [*function with no arguments*] 11-7
(byteref bytemap i j) [*function with three arguments*] 20-18
(byteset bytemap i j col) [*function with four arguments*] 20-18
#b w h bits [*#-macro*] 20-15
(c----r l) [*functions of one argument*] 3-49
(callextern address type v_1 t_1 ... v_n t_n) [*function with a variable number of arguments*] 14-2
(cartesian-product set1 set2) [*function with two arguments*] 3-109
(car l) [*function with one argument*] 3-49
(cascii ch) [*function with one argument*] 3-97
(cassoc s al) [*function with two arguments*] 3-67
(cassq symb al) [*function with two arguments*] 3-66
(catch-all-but l s_1 ... s_n) [*special form*] 3-41
(catcherror i s_1 ... s_n) [*special form*] 7-8
(catenate str_1 ... str_n) [*function with a variable number of arguments*] 3-91
cbcom [*character type*] 6-14
cdot [*character type*] 6-14
(cdr l) [*function with one argument*] 3-49
cecom [*character type*] 6-14
(ceiling n) [*function with one argument*] 4-4
(channel chan) [*function with an optional argument*] 6-50
(chrnth n strg) [*function with two arguments*] 3-93
(chrpos cn strg pos) [*function with two or three arguments*] 3-93
(chrset n strg cn) [*function with three arguments*] 3-94
(circopy e) [*function with one argument*] 9-13
(cirequal e1 e2) [*function with two arguments*] 9-13
(cirlist e_1 ... e_n) [*function with a variable number of arguments*] 3-62
(cirnequal e1 e2) [*function with two arguments*] 9-13
(cirprinflush e) [*function with one argument*] 9-13
(cirprint e) [*function with one argument*] 9-13
(cirprin e) [*function with one argument*] 9-13

(cis r) [*function with one argument*] 10-9
(clear-graph-env) [*function with no arguments*] 18-21
(clear-graph-env) [*function with no arguments*] 20-10
client-message [*event*] 19-5
#:clip:h [*variable*] 20-9
#:clip:w [*variable*] 20-9
#:clip:x [*variable*] 20-9
#:clip:y [*variable*] 20-9
(cload string) [*function with one argument*] 14-6
(clockalarm n) [*function with one argument*] 7-4
(clock) [*function with no arguments*] 7-4
clock [*programmable interrupt*] 7-4
(close chan) [*function with an optional argument*] 6-50
(closure lvar fn) [*function with two arguments*] 3-22
clpar [*character type*] 6-14
(clrhash ht) [*function with one argument*] 3-106
cmacro [*character type*] 6-15
cmsymb [*character type*] 6-16
cnull [*character type*] 6-14
(combine-pathnames path1 path2) [*function with two arguments*] 6-70
(comline strg) [*function with one argument*] 7-24
(comment e_1 ... e_n) [*special form*] 3-6
(compile-all-in-core print loader) [*function with zero, one or two optional arguments*] . 13-2
(compilefiles source object) [*function with two arguments*] 13-3
(compilemodule module ind) [*function with one or two arguments*] 13-17
#:compiler:open-p [*variable*] 13-4
(compiler source status print loader) [*function with four arguments*] 13-2
compiler [*feature*] 13-1
(compile source status print loader) [*special form with one, two, three or four arguments*]13-2
(complexp c) [*function with one argument*] 10-6
complex [*feature*] 10-1
#:complice:no-warning [*variable*] 13-19
#:complice:parano-flag [*variable*] 13-18
#:complice:warning-flag [*flag*] 13-19

`complice` [*feature*] 13-1

`(concat str`$_1$ `...` `str`$_n$`)` [*function with a variable number of arguments*] 3-83

`(cond l`$_1$ `...` `l`$_n$`)` [*special form*] 3-29

`(conjugate c)` [*function with one argument*] 10-8

`(consp s)` [*function with one argument*] 3-45

`(constantp s)` [*function with one argument*] 3-43

`(cons s1 s2)` [*function with two arguments*] 3-52

`(control-file-pathname name path)` [*function with one or two arguments*] 6-72

`(copyfile ofile nfile)` [*function with two arguments*] 6-53

`(copylist l)` [*function with one argument*] 3-55

`(copy s)` [*function with one argument*] 3-56

`(core-init-std msg)` [*function with one argument*] 7-15

`(cosh z)` [*function with one argument*] 10-9

`(cos x)` [*function with one argument*] 4-10

`cpkgc` [*character type*] 6-15

`cpname` [*character type*] 6-15

`cquote` [*character type*] 6-14

`(create-bitmap w h bits)` [*function with two or three arguments*] 20-14

`(create-bytemap w h bytes table)` [*function with two to four arguments*] 20-17

`(create-directory dir)` [*function with one argument*] 6-53

`(create-menu ti n`$_1$ `v`$_1$ `...` `n`$_n$ `v`$_n$`)` [*function with a variable number of arguments*] 19-10

`(create-subwindow type x y w h ti hi vi fa)` [*function with nine arguments*] 18-13

`(create-window type x y w h ti hi vi)` [*function with eight arguments*] 18-13

`crpar` [*character type*] 6-14

`(csend fndef symb o p`$_1$ `...` `p`$_n$`)` [*function with a variable number of arguments*] 5-11

`csep` [*character type*] 6-15

`csplice` [*character type*] 6-15

`(cstack)` [*function with no arguments*] 7-11

`cstring` [*character type*] 6-15

`csymb` [*character type*] 6-15

`(curread)` [*function with no arguments*] 6-7

`(current-background color)` [*function with an optional argument*] 20-3

`(current-clip x y w h)` [*function with zero or four arguments*] 20-9

`(current-cursor cursor)` [*function with an optional argument*] 20-5

(current-directory path) [*function with an optional argument*] 6-73
(current-display screen) [*function with an optional argument*] 18-3
(current-font font) [*function with an optional argument*] 20-2
(current-foreground color) [*function with an optional argument*] 20-3
(current-keyboard-focus-window win) [*function with an optional argument*] 18-16
(current-language language) [*function with an optional argument*] 9-15
(current-line-style line-style) [*function with an optional argument*] 20-6
(current-mode mode) [*function with an optional argument*] 20-8
(current-pattern pattern) [*function with an optional argument*] 20-7
(current-window win) [*function with an optional argument*] 18-16
(cursor-max) [*function with no arguments*] 20-5
(cursor-name num) [*function with one argument*] 20-6
C_LL_FIX [*Fonction C*] 14-13
C_LL_FLOAT [*Fonction C*] 14-14
#C [*#-macro*] 10-6
(debug-command cn) [*function with one argument*] 11-12
(debugend) [*function with no arguments*] 11-1
(debug s) [*special form*] 11-7
debug [*feature*] 11-1
(decr symb n) [*special form*] 3-74
(defabbrev sym1 sym2) [*macro*] 5-13
(default-language language) [*function with an optional argument*] 9-15
default-pathname-defaults [*variable*] 6-69
(default-window-type screen type) [*function with one or two arguments*] 18-7
(defesckey key e_1 ... e_n) [*special form*] 16-3
(defextern-cache flag) [*special form*] 14-5
(defextern symb ltype type) [*special form*] 14-2
(define-window-property-accessor prop) [*macro*] 18-15
(defkey key e_1 ... e_n) [*special form*] 16-3
(defmacro-open name ... fval) [*special form*] 13-5
(defmacro symb lvar s_1 ... s_n) [*special form*] 3-19
(defmake type fn (field$_1$ ... field$_n$)) [*macro*] 5-17
(defmessage message def$_1$ ... def$_n$) [*macro*] 9-17
(defprop pl pval ind) [*special form*] 3-77

(defrecord symbol field$_1$ ... field$_n$) [*macro*] 5-16
(defsetf access lparams store . body) [*macro*] 3-24
(defsharp ch larg s1 ... sN) [*special form*] 6-24
(defstruct struct field$_1$ ... field$_n$) [*macro*] 5-2
defstruct [*feature*] 5-1
(deftclass symbol field$_1$ ... field$_n$) [*macro*] 5-16
(defun symb lvar s$_1$ ... s$_n$) [*special form*] 3-18
(defvar symb e) [*special form*] 3-70
(delete-directory dir) [*function with one argument*] 6-53
(deletefile file) [*function with one argument*] 6-53
(delete s l) [*function with two arguments*] 3-64
#\ [*#-macro*] 6-24
(delq symb l) [*function with two arguments*] 3-64
(demethod {tclass}:name (obj p$_1$ ... p$_n$) (f$_1$ ... f$_n$) . body) [*macro*] 5-18
(denominator f) [*function with one argument*] 10-5
(deposit-byte n np nl m) [*function with four arguments*] 4-18
(deposit-field n np nl m) [*function with four arguments*] 4-18
(desetq l1 l2) [*special form*] 3-72
(deset l1 l2) [*function with two arguments*] 3-72
(device-namestring path) [*function with one argument*] 6-70
(de symb lvar s$_1$ ... s$_n$) [*special form*] 3-18
(df symb lvar s$_1$ ... s$_n$) [*special form*] 3-18
(difference x$_1$ ... x$_n$) [*function with a variable number of arguments*] 4-25
(differ x$_1$ ... x$_n$) [*function with a variable number of arguments*] 4-25
(digitp cn) [*function with one argument*] 3-98
(directory-namestring path) [*function with one argument*] 6-70
(directoryp path) [*function with one argument*] 6-73
(disallow-event display event) [*function with two arguments*] 19-8
(displace l ln) [*function with two arguments*] 3-60
(display-depth display) [*function with one argument*] 18-7
(display-get-font-info display font-name font-info) [*function with two or three arguments*] 18-25
(display-get-font-names display max pattern) [*function with two or three arguments*] . 18-25
(display-get-selection display) [*function with one argument*] 19-11

(display-store-selection display buffer) [*function with two arguments*] 19-11
(display-synchronize display flag) [*function with one or two arguments*] 18-7
(divide x y) [*function with two arguments*] 4-26
(div x y) [*function with two arguments*] 4-13
(dmc ch () s_1 ... s_n) [*special form*] 6-18
(dmd symb lvar s_1 ... s_n) [*special form*] 3-19
(dms ch () s_1 ... s_n) [*special form*] 6-19
(dm symb lvar s_1 ... s_n) [*special form*] 3-19
(do* lv lr ec_1 ... ec_n) [*macro*] 3-37
(dont-compile exp_1 ... exp_n) [*special form*] 13-3
down-event [*event*] 19-2
(do lv lr ec_1 ... ec_n) [*macro*] 3-35
drag-event [*event*] 19-2
(draw-arc x y w h angle1 angle2) [*function with six arguments*] 20-11
(draw-circle x y r) [*function with four arguments*] 20-11
(draw-cn x y cn) [*function with three arguments*] 18-21
(draw-cn x y cn) [*function with three arguments*] 20-13
(draw-cursor x y i) [*function with three arguments*] 18-21
(draw-ellipse x y rx ry) [*function with four arguments*] 20-10
(draw-line x0 y0 x1 y1) [*function with four arguments*] 20-11
(draw-point x y) [*function with two arguments*] 20-11
(draw-polyline n vx vy) [*function with three arguments*] 20-9
(draw-polymarker n vx vy) [*function with three arguments*] 20-10
(draw-rectangles n vx vy vw vh) [*function with five arguments*] 20-12
(draw-rectangle x y w h) [*function with four arguments*] 20-11
(draw-segments n vx1 vy1 vx2 vy2) [*function with five arguments*] 20-11
(draw-string x y s) [*function with three arguments*] 18-21
(draw-string x y s) [*function with three arguments*] 20-13
(draw-substring x y s start length) [*function with five arguments*] 18-22
(draw-substring x y s start length) [*function with five arguments*] 20-10
(ds symb type adr) [*special form*] 3-20
(duplstring n strg) [*function with two arguments*] 3-92
E.0 [*Complice error message*] 13-20
E.1 [*Complice error message*] 13-20

E.2 [*Complice error message*] 13-20
E.3 [*Complice error message*] 13-20
E.4 [*Complice error message*] 13-21
E.5 [*Complice error message*] 13-21
E.6 [*Complice error message*] 13-21
(edlin) [*function with no arguments*] 17-1
edlin [*feature*] 17-1
(end return-code) [*function with an optional argument*] 7-17
(enough-namestring path default) [*function with one or two arguments*] 6-71
enterwindow-event [*event*] 19-3
(eof chan) [*function with one argument*] 6-52
eof [*programmable interrupt*] 6-52
(eol) [*function with no arguments*] 6-46
eol [*programmable interrupt*] 6-45
(eprogn l) [*function with one argument*] 3-3
(eqn x y) [*function with two arguments*] 4-14
(eqstring str1 str2) [*function with two arguments*] 3-91
(equal-pathname path1 path2) [*function with two arguments*] 6-65
(equal s1 s2) [*function with two arguments*] 3-47
(eqvector vect1 vect2) [*function with two arguments*] 3-101
(eq s1 s2) [*function with two arguments*] 3-46
(error s1 s2 s3) [*function with three arguments*] 7-8
(errset e i) [*macro*] 7-9
(err s_1 ... s_n) [*special form*] 7-9
(eval-when moments . expressions) [*function with a variable number of arguments*] 13-13
(eval s env) [*function with one or two arguments*] 3-2
(even? z) [*function with one argument*] 10-4
(evenp x) [*function with one argument*] 4-14
event-loop [*feature*] 6-57
(event-mode mode) [*function with an optional argument*] 19-5
(eventp) [*function with no arguments*] 19-6
(event event) [*function with one argument*] 19-9
event [*programmable interrupt*] 19-8
event [*structure*] 19-2

(every fn l_1 ... l_n) [*function with a variable number of arguments*] 3-11
(evexit s s_1 ... s_n) [*special form*] 3-38
(evlis l) [*function with one argument*] 3-2
(evlis l) [*function with one argument*] 3-61
(evloop-add-display display manage-fct) [*function with two arguments*] 6-60
(evloop-add-input fd manage-fct arg-to-use) [*function with three arguments*] 6-58
(evloop-add-output fd manage-fct arg-to-use) [*function with three arguments*] 6-59
(evloop-allow-tty-input) [*function with no arguments*] 6-58
(evloop-change-manage-function fd new-manage-fct new-arg-to-use) [*function with three arguments*] 6-58
(evloop-change-output-manage-function fd new-manage-fct new-arg-to-use) [*function with three arguments*] 6-60
(evloop-disallow-tty-input) [*function with no arguments*] 6-58
(evloop-display-managed-p display) [*function with one argument*] 6-60
(evloop-initialized-p) [*function with no arguments*] 6-59
(evloop-init) [*function with no arguments*] 6-57
(evloop-input-managedp fd) [*function with one argument*] 6-59
(evloop-readp fd) [*function with one argument*] 6-59
(evloop-remove-display display) [*function with one argument*] 6-60
(evloop-remove-input fd) [*function with one argument*] 6-58
(evloop-remove-output fd) [*function with one argument*] 6-60
(evloop-restart) [*function with no arguments*] 6-58
(evloop-select) [*function with no arguments*] 6-58
(evloop-set-timeout-handler handler) [*function with one argument*] 6-59
(evloop-set-timeout secs millisecs) [*function with two arguments*] 6-59
(evloop-stop) [*function with no arguments*] 6-58
(evloop-wait) [*function with no arguments*] 6-59
(evtag s s_1 ... s_n) [*special form*] 3-38
(ex* x y z) [*function with three arguments*] 4-21
(ex+ x y) [*function with two arguments*] 4-20
(ex- x) [*function with one argument*] 4-20
(ex/ x y) [*function with two arguments*] 4-21
(ex1+ x) [*function with one argument*] 4-20
#:ex:mod [*variable*] 4-7

#:ex:regret [*variable*] 4-20
(ex? x y) [*function with two arguments*] 4-22
(exchstring strg1 strg2) [*function with two arguments*] 3-88
(exchvector vect1 vect2) [*function with two arguments*] 3-102
(exit symb s_1 ... s_n) [*special form*] 3-38
(expand-pathname path) [*function with one argument*] 6-67
(explodech s) [*function with one argument*] 6-39
(explode s) [*function with one argument*] 6-39
(exp x) [*function with one argument*] 4-11
(fact n) [*function with one argument*] 10-4
(fadd x y) [*function with two arguments*] 4-22
(false e_1 ... e_n) [*function with a variable number of arguments*] 3-42
(fdiv x y) [*function with two arguments*] 4-23
(featurep symb) [*function with one argument*] 6-75
(feqn x y) [*function with two arguments*] 4-23
(fge x y) [*function with two arguments*] 4-23
(fgt x y) [*function with two arguments*] 4-23
(fib n) [*function with one argument*] 10-4
(field-list symbol) [*function with one argument*] 5-17
(file-namestring path) [*function with one argument*] 6-69
(fill-arc x y w h angle1 angle2) [*function with six arguments*] 20-12
(fill-area n vx vy) [*function with three arguments*] 20-10
(fill-circle x y r) [*function with three arguments*] 20-12
(fill-ellipse x y rx ry) [*function with four arguments*] 20-10
(fill-rectangles n vx vy vw vh) [*function with five arguments*] 20-12
(fill-rectangle x y w h) [*function with four arguments*] 20-12
(fillstring strg n1 cn n2) [*function with three or four arguments*] 3-95
(fillvector vect n1 e n2) [*function with three or four arguments*] 3-102
(find-window x y) [*function with two arguments*] 18-19
(findfn s) [*function with one argument*] 3-79
(firstn n l) [*function with two arguments*] 3-57
(fixp x) [*function with one argument*] 4-2
(fix x) [*function with one argument*] 4-3
(flambda l s_1 ... s_n) [*special form*] 3-6

(flet l s_1 ... s_n) [*special form*] 3-23
(fle x y) [*function with two arguments*] 4-24
(floatp x) [*function with one argument*] 4-3
(float x) [*function with one argument*] 4-4
(floor n) [*function with one argument*] 4-3
(flt x y) [*function with two arguments*] 4-24
(flush-event) [*function with no arguments*] 19-6
(flush) [*function with no arguments*] 6-46
flush [*programmable interrupt*] 6-46
(fmax x y) [*function with two arguments*] 4-24
(fmin x y) [*function with two arguments*] 4-24
(fmul x y) [*function with two arguments*] 4-22
(fneqn x y) [*function with two arguments*] 4-23
(font-ascent) [*function with no arguments*] 18-25
(font-height) [*function with no arguments*] 18-25
(font-max) [*function with no arguments*] 20-2
(font-name font) [*function with one argument*] 20-2
(format dest cntrl e_1 ... e_n) [*function with two or more arguments*] 9-1
format [*feature*] 9-1
(for (var in ic ntl e_1 ... e_m) s_1 ... s_n) [*special form*] 3-32
(freecons cons) [*function with one argument*] 7-22
(freetree tree) [*function with one argument*] 7-23
(fsub x y) [*function with two arguments*] 4-22
(funcall fn s_1 ... s_n) [*function with a variable number of arguments*] 3-7
(function fn) [*special form*] 3-5
(gc-before-alarm) [*function with no arguments*] 7-22
gc-before-alarm [*programmable interrupt*] 7-22
(gcalarm) [*function with no arguments*] 7-22
gcalarm [*programmable interrupt*] 7-22
(gcd z_1 ... z_n) [*function with a variable number of arguments*] 10-4
(gcinfo i) [*function with an optional argument*] 7-20
(gc i) [*function with an optional argument*] 7-20
(genarith) [*arithmetic interrupt*] 4-2
(gensym) [*function with no arguments*] 3-84

(get-abbrev symb) [*function with one argument*] 5-14
(get-all-messages language) [*function with one argument*] 9-17
(get-message-p message) [*function with one argument*] 9-16
(get-message message) [*function with one argument*] 9-16
(get-rgb-values pixel) [*function with one argument*] 20-3
(getdefmodule defmod key) [*function with two arguments*] 13-16
(getdef symb) [*function with one argument*] 3-80
(getenv strg) [*function with one argument*] 7-24
(getfn1 pkgc fn) [*function with two arguments*] 5-7
(getfn1 pkgc symb) [*function with two arguments*] 3-82
(getfn2 pkgc1 pkgc2 fn) [*function with three arguments*] 5-8
(getfn pkgc symb lastpkgc) [*function with two or three arguments*] 3-82
(getfn pkgc1 fn pkgc2) [*function with two or three arguments*] 5-8
(getglobal strg) [*function with one argument*] 14-1
(gethash key ht default) [*function with two or three arguments*] 3-105
(getl pl l) [*function with two arguments*] 3-76
(getprop pl ind) [*function with two arguments*] 3-75
(get pl ind) [*function with two arguments*] 3-75
(ge x y) [*function with two arguments*] 4-15
(go symb) [*special form*] 3-34
(grab-event window) [*function with an optional argument*] 19-7
(green-component color value) [*function with one or two arguments*] 20-4
(gt x y) [*function with two arguments*] 4-15
(hanoiend) [*function with no arguments*] 15-11
(hanoi n) [*function with one argument*] 15-10
(has-an-abbrev symb) [*function with one argument*] 5-14
(hash-table-count ht) [*function with one argument*] 3-106
(hash-table-p obj) [*function with one argument*] 3-105
(hash strg) [*function with one argument*] 3-90
hash [*feature*] 3-104
(height-space) [*function with no arguments*] 18-24
(height-substring string start length) [*function with three arguments*] 18-23
(height-substring string start length) [*function with three arguments*] 20-13
(herald) [*function with no arguments*] 7-16

(highlight-rectangle x y w h) [*function with four arguments*] 20-13
(host-namestring path) [*function with one argument*] 6-70
(ibase n) [*function with an optional argument*] 6-12
(identity s) [*function with one argument*] 3-5
(ifn s_1 s_2 s_3 ... s_n) [*special form*] 3-27
(if s_1 s_2 s_3 ... s_n) [*special form*] 3-26
(#:image:rectangle:window:current-keyboard-focus-window w) [*function with one argument*] 18-20
(#:image:rectangle:window:current-window win) [*function with one argument*] 18-20
(#:image:rectangle:window:kill-window win) [*function with one argument*] 18-20
(#:image:rectangle:window:map-window win x y symbx symby) [*function with five arguments*] 18-20
(#:image:rectangle:window:modify-window win left top w h ti hi vi) [*function with eight arguments*] 18-20
(#:image:rectangle:window:move-behind-window win1 win2) [*function with two arguments*] 18-20
(#:image:rectangle:window:pop-window win) [*function with one argument*] 18-20
(#:image:rectangle:window:prop w v) [*function with one or two arguments*] 18-15
(#:image:rectangle:window:uncurrent-window win) [*function with one argument*] 18-20
#:image:rectangle:window [*structure*] 18-11
(imagpart c) [*function with one argument*] 10-8
(imax x y) [*function with two arguments*] 4-16
(imin x y) [*function with two arguments*] 4-15
(implodech s) [*function with one argument*] 6-40
(implode ln) [*function with one argument*] 6-39
(inbuf n cn) [*function with one or two optional arguments*] 6-42
(inchan chan) [*function with an optional argument*] 6-49
(incr symb n) [*special form*] 3-74
(index str1 str2 n) [*function with two or three arguments*] 3-95
(inibitmap symbol) [*function with an optional argument*] 18-1
(initty symb) [*function with an optional argument*] 15-1
(inmax n) [*function with an optional argument*] 6-42
(inpos n) [*function with an optional argument*] 6-42
(input file) [*function with one argument*] 6-50
(integerp z) [*function with one argument*] 10-3

(intersection list1 list2 eq-func) [*function with two or three arguments*] 3-107
(itsoft symb larg) [*function with two arguments*] 7-2
keyboard-focus-event [*event*] 19-4
(kill-bitmap bitmap) [*function with one argument*] 20-16
(kill-color color) [*function with one argument*] 20-4
(kill-menu menu) [*function with one argument*] 19-10
kill-window-event [*event*] 19-3
(kill-window win) [*function with one argument*] 18-17
(kwote s) [*function with one argument*] 3-54
(lambda l s_1 ... s_n) [*special form*] 3-6
(large-roman-font screen) [*function with an optional argument*] 18-7
(lastn n l) [*function with two arguments*] 3-57
(last s) [*function with one argument*] 3-51
leavewindow-event [*event*] 19-3
(length s) [*function with one argument*] 3-51
(let* lv s_1 ... s_n) [*special form*] 3-15
(letn symb lv s_1 ... s_n) [*special form*] 3-16
(lets lv s_1 ... s_n) [*special form*] 3-15
(letterp cn) [*function with one argument*] 3-98
(letvq lvar lval s_1 ... s_n) [*special form*] 3-15
(letv lvar lval s_1 ... s_n) [*special form*] 3-14
(let lv s_1 ... s_n) [*special form*] 3-13
(le x y) [*function with two arguments*] 4-15
(lhoblist strg) [*function with one argument*] 3-85
(libautoload file sym_1 ... sym_n) [*special form*] 6-57
#:libcir:package-parano [*variable*] 9-13
libcir [*feature*] 9-12
(libloadfile file i) [*function with two arguments*] 6-56
(libload file i) [*special form*] 6-56
(line-style-max) [*function with no arguments*] 20-7
(list-features) [*function with no arguments*] 6-75
(listp s) [*function with one argument*] 3-45
(list s_1 ... s_n) [*function with a variable number of arguments*] 3-53
(llcp-std name) [*function with one argument*] 7-14

LL_C_FIX [*Fonction C*] 14-13
LL_C_FLOAT [*Fonction C*] 14-14
(lmargin n) [*function with an optional argument*] 6-44
(load-byte-test n np nl) [*function with three arguments*] 4-19
(load-byte n np nl) [*function with three arguments*] 4-18
(load-cpl im min ed env ld cmp) [*function with one to six arguments*] 7-15
(load-font string) [*function with one argument*] 20-2
(load-std im min ed env ld cmp) [*function with one to six arguments*] 7-14
(load-stm im min ed env ld cmp) [*function with one to six arguments*] 7-14
(loadfile file i) [*function with two arguments*] 6-54
(loadmodule module reload-p interpreted-p) [*function with one, two or three arguments*] 13-16
(loadobjectfile file) [*function with one argument*] 13-12
(load file i) [*special form*] 6-54
(lock fn s_1 ... s_n) [*special form*] 3-39
(log10 x) [*function with one argument*] 4-11
(logand x y) [*function with two arguments*] 4-16
(lognot x) [*function with one argument*] 4-16
(logor x y) [*function with two arguments*] 4-17
(logshift x y) [*function with two arguments*] 4-17
(logxor x y) [*function with two arguments*] 4-17
(log x) [*function with one argument*] 4-11
(lowercase cn) [*function with one argument*] 3-98
(lt x y) [*function with two arguments*] 4-15
(macro-openp name) [*function with one argument*] 13-5
(macroexpand1 s) [*function with one argument*] 3-20
(macroexpand s) [*function with one argument*] 3-21
(make-color red green blue) [*function with three arguments*] 20-3
(make-cursor b1 b2 x y) [*function with four arguments*] 20-5
(make-hash-table-equal) [*function with no arguments*] 3-105
(make-hash-table-eq) [*function with no arguments*] 3-105
(make-line-style width style) [*function with two arguments*] 20-7
(make-macro-open name fval) [*function with two arguments*] 13-5
(make-mutable-color red green blue) [*function with three arguments*] 20-3
(make-named-color name) [*function with one argument*] 20-3

(make-named-cursor name) [*function with one argument*] 20-5
(make-pathname element$_1$ ... element$_n$) [*function with a variable number of arguments*] ... 6-69
(make-pattern bitmap) [*function with one argument*] 20-7
(make-window win) [*function with one argument*] 18-13
(makearray a$_1$ ... a$_n$ s) [*function with a variable number of arguments*] 3-103
(makecomplex c i) [*function with two arguments*] 10-7
(makedef symb ftyp fval) [*function with three arguments*] 3-80
(makelist n s) [*function with two arguments*] 3-54
(makestring n cn) [*function with two arguments*] 3-92
(makevector n s) [*function with two arguments*] 3-99
(makunbound symb) [*function with one argument*] 3-71
(map-window win x y symbx symby) [*function with five arguments*] 18-19
map-window [*event*] 19-4
(mapcan fn l$_1$ ... l$_n$) [*function with a variable number of arguments*] 3-10
(mapcar fn l$_1$ ... l$_n$) [*function with a variable number of arguments*] 3-9
(mapcoblist fn) [*function with one argument*] 3-12
(mapcon fn l$_1$ ... l$_n$) [*function with a variable number of arguments*] 3-9
(mapc fn l$_1$ ... l$_n$) [*function with a variable number of arguments*] 3-8
(maphash fnt ht) [*function with two arguments*] 3-106
(maplist fn l$_1$ ... l$_n$) [*function with a variable number of arguments*] 3-9
(maploblist fn) [*function with one argument*] 3-12
(mapl fn l$_1$ ... l$_n$) [*function with a variable number of arguments*] 3-8
(mapoblist fn) [*function with one argument*] 3-12
(mapvector fn vect) [*function with two arguments*] 3-11
(map fn l$_1$ ... l$_n$) [*function with a variable number of arguments*] 3-8
(mask-field n np nl) [*function with three arguments*] 4-18
(max x$_1$ ... x$_n$) [*function with a variable number of arguments*] 4-7
(mcons s$_1$ ... s$_n$) [*function with a variable number of arguments*] 3-52
(member s l) [*function with two arguments*] 3-50
(memq symb l) [*function with two arguments*] 3-49
(menu-delete-item-list menu choice) [*function with two arguments*] 19-11
(menu-delete-item menu choice item) [*function with three arguments*] 19-11
(menu-insert-item-list menu choice string active) [*function with four arguments*] .. 19-10

(menu-insert-item menu choice item string active value) [*function with six arguments*] 19-11
(menu-modify-item-list menu choice string active) [*function with four arguments*] .. 19-11
(menu-modify-item menu choice item string active value) [*function with six arguments*] 19-11
(merge-pathnames path default-path) [*function with two arguments*] 6-70
(message-languages) [*function with no arguments*] 9-16
messages [*feature*] 9-14
microceyx [*feature*] 5-15
(minusp x) [*function with one argument*] 4-10
(min x_1 ... x_n) [*function with a variable number of arguments*] 4-7
(mlambda l s_1 ... s_n) [*special form*] 3-6
#:mode:not [*variable*] 20-8
#:mode:or [*variable*] 20-8
#:mode:set [*variable*] 20-8
#:mode:xor [*variable*] 20-8
modify-window-event [*event*] 19-3
(modify-window win x y w h ti hi vi) [*function with eight arguments*] 18-17
#:module:compiled-list [*variable*] 13-17
#:module:interpreted-list [*variable*] 13-17
(modulo x y) [*function with two arguments*] 4-6
(move-behind-window win1 win2) [*function with two arguments*] 18-19
(move-cursor x y) [*function with two arguments*] 20-6
move-event [*event*] 19-2
(move-resize-window win x y w h) [*function with five arguments*] 18-18
(move-window win x y) [*function with three arguments*] 18-18
(mul x y) [*function with two arguments*] 4-13
#M [*sharp macro*] 9-16
#m [*sharp macro*] 9-16
(#n=) [*#-macro*] 9-12
(name-to-rgb name) [*function with one argument*] 20-3
(namestring path) [*function with one argument*] 6-65
(nconc1 l s) [*function with two arguments*] 3-62
(nconc l_1 ... l_n) [*function with a variable number of arguments*] 3-61

(ncons s) [*function with one argument*] 3-52
(neqn x y) [*function with two arguments*] 4-14
(nequal s1 s2) [*function with two arguments*] 3-48
(neq s1 s2) [*function with two arguments*] 3-47
(newl symb s) [*special form*] 3-73
(newr symb s) [*special form*] 3-73
(new struct) [*function with one argument*] 5-2
(nextl sym1 sym2) [*special form*] 3-73
(nintersection list1 list2 eq-func) [*function with two or three arguments*] 3-108
(nlistp s) [*function with one argument*] 3-46
! [*macro character*] 6-32
' [*macro character*] 6-20
(** n m) [*function with two arguments*] 10-5
(* x_1 ... x_n) [*function with a variable number of arguments*] 4-5
(+ x_1 ... x_n) [*function with a variable number of arguments*] 4-4
(- x_1 ... x_n) [*function with a variable number of arguments*] 4-5
(// x y) [*function with one or two arguments*] 4-6
(/= x y) [*function with two arguments*] 4-8
(/ x y) [*function with one or two arguments*] 4-6
(1+ x) [*function with one argument*] 4-5
(1- x) [*function with one argument*] 4-5
(2** n) [*function with one argument*] 4-17
(<= x_1 x_2 ... x_n) [*function with a variable number of arguments*] 4-9
(<> x y) [*function with two arguments*] 4-8
(<?> x y) [*function with two arguments*] 4-8
(< x_1 x_2 ... x_n) [*function with a variable number of arguments*] 4-9
(= x_1 x_2 ... x_n) [*function with a variable number of arguments*] 4-8
(>= x_1 x_2 ... x_n) [*function with a variable number of arguments*] 4-8
(> x_1 x_2 ... x_n) [*function with a variable number of arguments*] 4-9
,. [*macro character*] 6-21
,@ [*macro character*] 6-21
, [*macro character*] 6-21
[[*macro character*] 10-6
[[*macro character*] 6-20

#" [#-macro] 6-26
#<> [non-Lisp pointer] 6-37
#' [#-macro] 6-26
#([#-macro] 6-27
#* [#-macro] 20-15
#+ [#-macro] 6-27
#- [#-macro] 6-27
#. [#-macro] 6-26
#/ [#-macro] 6-24
#: [#-macro] 6-26
#[[#-macro] 6-27
#% [#-macro] 6-25
#| [#-macro] 6-27
[macro character] 6-23
` [macro character] 6-20
| [inside #-macro] 6-26
(not s) [function with one argument] 3-43
(nreconc l s) [function with two arguments] 3-63
(nreverse l) [function with one argument] 3-62
(nset-difference list1 list2 eq-func) [function with two or three arguments] 3-108
(nset-exclusive-or list1 list2 eq-func) [function with two or three arguments] 3-108
(nsubst s1 s2 l) [function with three arguments] 3-63
(nthcdr n l) [function with two arguments] 3-50
(nth n l) [function with two arguments] 3-51
(null s) [function with one argument] 3-42
(numberp s) [function with one argument] 3-44
(numberp x) [function with one argument] 4-2
(numerator f) [function with one argument] 10-5
(nunion list1 list2 eq-func) [function with two or three arguments] 3-107
(#n#) [#-macro] 9-12
(obase n) [function with an optional argument] 6-38
(objval symb s) [function with one or two arguments] 3-82
(oblist pkgc symb) [function with zero, one or two arguments] 3-84
(oddp x) [function with one argument] 4-14

(ogetq tclass-or-record field obj) [*macro*] 5-17
(omakeq type field$_1$ val$_1$ ... field$_n$ val$_n$) [*macro*] 5-17
(omatchq tclass obj) [*macro*] 5-18
(openab file) [*function with one argument*] 6-49
(opena file) [*function with one argument*] 6-49
(openib file) [*function with one argument*] 6-49
(openi file) [*function with one argument*] 6-48
(openob file) [*function with one argument*] 6-49
(openo file) [*function with one argument*] 6-49
(oputq tclass-or-record field obj val) [*macro*] 5-18
(or s$_1$ ... s$_n$) [*special form*] 3-28
(outbuf n cn) [*function with one or two optional arguments*] 6-45
(outchan chan) [*function with an optional argument*] 6-50
(outpos n) [*function with an optional argument*] 6-45
(output file) [*function with one argument*] 6-51
(packagecell symb pkgc) [*function with one or two arguments*] 3-82
(pairlis l1 l2 al) [*function with three arguments*] 3-65
(parallelvalues e$_1$ ... e$_n$) [*special form*] 7-5
(parallel e$_1$ ... e$_n$) [*special form*] 7-5
(pathname-device path) [*function with one argument*] 6-67
(pathname-directory path) [*function with one argument*] 6-68
(pathname-host path) [*function with one argument*] 6-67
(pathname-name path) [*function with one argument*] 6-68
(pathname-type path) [*function with one argument*] 6-68
(pathname-version path) [*function with one argument*] 6-68
#:pathname:current [*indicator*] 6-66
(#:pathname:prin path) [*function with one argument*] 6-65
#:pathname:up [*indicator*] 6-66
#:pathname:wild [*indicator*] 6-67
(pathnamep s) [*function with one argument*] 6-64
(pathname string) [*function with one argument*] 6-64
pathname [*feature*] 6-63
pathname [*structure*] 6-63
(pattern-max) [*function with no arguments*] 20-7

#$8000 [*unnameable number*] 6-36

#$ [*#-macro*] 6-25

(peek-event event) [*function with an optional argument*] 19-6

(peekch) [*function with no arguments*] 6-5

(peekcn) [*function with no arguments*] 6-5

(pepefile f) [*function with one argument*] 16-2

(pepend) [*function with no arguments*] 16-2

(pepe f) [*special form*] 16-1

pepe [*feature*] 16-1

(pgcd z0 z1) [*function with two arguments*] 10-4

(phase c) [*function with one argument*] 10-8

(placdl l s) [*function with two arguments*] 3-61

(plength strg) [*function with one argument*] 3-89

(plist pl l) [*function with one or two arguments*] 3-75

(plusp x) [*function with one argument*] 4-9

(plus x_1 ... x_n) [*function with a variable number of arguments*] 4-25

(pname strg) [*function with one argument*] 3-89

(pop-window win) [*function with one argument*] 18-19

portable-pathname [*variable*] 6-74

(portable-pathname-p path) [*function with one argument*] 6-74

(power-set set) [*function with one argument*] 3-108

(power x y) [*function with two arguments*] 4-11

(pprint s) [*function with one argument*] 8-2

(pprin s) [*function with one argument*] 8-2

(pratom atom) [*function with one argument*] 6-39

(precision n) [*function with an optional argument*] 10-2

(precompile exp1 result exp2 operand) [*special form*] 13-3

#:pretty:quotelength [*variable*] 8-3

#:pretty:quotelevel [*variable*] 8-3

(prettyend) [*function with no arguments*] 8-2

(prettyf file sym_1 ... sym_n) [*special form*] 8-2

(pretty sym_1 ... sym_n) [*special form*] 8-2

pretty [*feature*] 8-2

(princh ch n) [*function with one or two arguments*] 6-35

(princn cn n) [*function with one or two arguments*] 6-34
(prinflush s_1 ... s_n) [*function with a variable number of arguments*] 6-34
(prinf <cntrl> <e1> ... <eN>) [*SUBR à 1 or N arguments*] 9-2
(print-to-string s) [*function with one argument*] 6-41
(printdefmodule defmod module) [*function with two arguments*] 13-16
(printerror symb s1 s2) [*function with three arguments*] 7-8
(printf <cntrl> <e1> ... <eN>) [*SUBR à 1 or N arguments*] 9-2
(printlength n) [*function with an optional argument*] 6-35
(printlevel n) [*function with an optional argument*] 6-35
(printline n) [*function with an optional argument*] 6-36
(printstack n s) [*function with one or two optional arguments*] 11-12
(print s_1 ... s_n) [*function with a variable number of arguments*] 6-33
(prin s_1 ... s_n) [*function with a variable number of arguments*] 6-33
(probefile file) [*function with one argument*] 6-53
(probepathf file) [*function with one argument*] 6-56
(probepathm module) [*function with one argument*] 13-16
(probepatho file) [*function with one argument*] 13-12
(prog* l ec_1 ... ec_n) [*macro*] 3-35
(prog1 s_1 ... s_n) [*special form*] 3-3
(prog2 s_1 s_2 ... s_n) [*special form*] 3-4
(progn s_1 ... s_n) [*special form*] 3-4
(prog l ec_1 ... ec_n) [*macro*] 3-34
(prompt strg) [*function with an optional argument*] 6-9
(protect s_1 s_2 ... s_n) [*special form*] 3-39
(psetq sym_1 s_1 ... sym_n s_n) [*special form*] 3-71
(ptype symb n) [*function with one or two arguments*] 6-38
(put-abbrev sym1 sym2) [*function with two arguments*] 5-13
(put-message message language string) [*function with three arguments*] 9-17
(puthash key ht value) [*function with three arguments*] 3-105
(putprop pl pval ind) [*function with three arguments*] 3-77
#p [*#-macro*] 6-65
(quomod x y) [*function with two arguments*] 4-7
(quote s) [*special form*] 3-5
(quotient x y) [*function with two arguments*] 4-6

(quo x y) [*function with two arguments*] 4-6
q [*feature*] 10-1
(random x y) [*function with two arguments*] 4-19
(rassoc s al) [*function with two arguments*] 3-67
(rassq symb al) [*function with two arguments*] 3-66
(rationalp q) [*function with one argument*] 10-3
ratio [*feature*] 10-1
(read-delimited-list cn) [*function with one argument*] 6-6
(read-event event) [*function with an optional argument*] 19-6
(read-from-string string) [*function with one argument*] 6-40
(read-mouse event) [*function with an optional argument*] 19-9
(readch) [*function with no arguments*] 6-5
(readcn) [*function with no arguments*] 6-5
(readdefmodule module) [*function with one argument*] 13-16
(readline) [*function with no arguments*] 6-4
(readstring) [*function with no arguments*] 6-4
(read) [*function with no arguments*] 6-3
(realpart c) [*function with one argument*] 10-8
(realp r) [*function with one argument*] 10-6
(record-language language) [*function with one argument*] 9-15
(record-namep symbol) [*function with one argument*] 5-17
(red-component color value) [*function with one or two arguments*] 20-4
(redisplayscreen sn so w h) [*function with four or twelve arguments*] 15-9
(rem-abbrev symb) [*function with one argument*] 5-14
(rem-feature symb) [*function with one argument*] 6-75
(remfn symb) [*function with one argument*] 3-80
(remhash key ht) [*function with two arguments*] 3-106
(remob symb) [*function with one argument*] 3-86
(remove-language language) [*function with one argument*] 9-15
(remove-macro-open name) [*function with one argument*] 13-6
(remove-message message language) [*function with two arguments*] 9-17
(remove s l) [*function with two arguments*] 3-58
(remprop pl ind) [*function with two arguments*] 3-77
(remq symb l) [*function with two arguments*] 3-58

(rem x y) [*function with two arguments*] 4-14
(renamefile ofile nfile) [*function with two arguments*] 6-53
repaint-window-event [*event*] 19-3
(repeat m s_1 ... s_n) [*special form*] 3-32
(reread lcn) [*function with one argument*] 6-5
(resetfn symb ftype fval) [*function with three arguments*] 3-79
(resize-window win w h) [*function with three arguments*] 18-18
(restore-core file) [*function with one argument*] 7-12
(resume env) [*function with one argument*] 7-5
(return-from symb e) [*special form*] 3-33
(return e) [*special form*] 3-33
(reverse s) [*function with one argument*] 3-55
(revert symb) [*function with one argument*] 3-81
(rmargin n) [*function with an optional argument*] 6-45
(root-window screen) [*function with an optional argument*] 18-5
(round n d) [*function with two arguments*] 4-4
(rplaca l s) [*function with two arguments*] 3-60
(rplacd l s) [*function with two arguments*] 3-60
(rplac l s1 s2) [*function with three arguments*] 3-60
(runtime) [*function with no arguments*] 7-23
(save-core file) [*function with one argument*] 7-12
(save-std name msg fnt-sav fnt-rest) [*function of two, three or four arguments*] 7-16
(scale x y z) [*function with three arguments*] 4-14
(scanstring str1 str2 n) [*function with two or three arguments*] 3-96
(schedule fnt e_1 ... e_n) [*special form*] 7-5
(search-in-path path file) [*function with two arguments*] 6-55
(selectq s l_1 ... l_n) [*special form*] 3-29
(send-error symb rest) [*function with two arguments*] 5-9
(send-super type symb o p_1 ... p_n) [*function with a variable number of arguments*] 5-10
(send2 symb o1 o2 p_1 ... p_n) [*function with a variable number of arguments*] 5-11
(sendfq message par_1 ... par_n) [*macro*] 5-19
(sendf message par_1 ... par_n) [*macro*] 5-19
(sendq message object par_1 ... par_n) [*macro*] 5-18
(send message objet par_1 ... par_n) [*function with a variable number of arguments*] 5-18

(send symb o p_1 ... p_n) [*function with a variable number of arguments*] 5-9
(set-difference list1 list2 eq-func) [*function with two or three arguments*] 3-108
(set-equal list1 list2 eq-func) [*function with two or three arguments*] 3-109
(set-exclusive-or list1 list2 eq-func) [*function with two or three arguments*] 3-108
(set-pathname-device path device) [*function with two arguments*] 6-68
(set-pathname-directory path dir) [*function with two arguments*] 6-68
(set-pathname-host path host) [*function with two arguments*] 6-68
(set-pathname-name path name) [*function with two arguments*] 6-68
(set-pathname-type path type) [*function with two arguments*] 6-68
(set-pathname-version path version) [*function with two arguments*] 6-69
(setdemodule defmod key value) [*function with three arguments*] 13-16
(setfn symb ftype fval) [*function with three arguments*] 3-79
(setf loc_1 val_1 ... loc_n val_n) [*macro*] 3-24
(setqq sym_1 e_1 ... sym_n e_n) [*special form*] 3-71
(setq sym_1 s_1 ... sym_n s_n) [*special form*] 3-71
sets [*feature*] 3-106
(set symb s) [*function with two arguments*] 3-71
#:sharp:value [*property*] 6-25
(signum c) [*function with one argument*] 10-8
(sinh z) [*function with one argument*] 10-9
(sin x) [*function with one argument*] 4-10
(sleep n) [*function with one argument*] 7-24
(slength strg) [*function with one argument*] 3-89
(slen strg) [*function with one argument*] 3-87
(slet lv s_1 ... s_n) [*special form*] 3-15
(small-roman-font screen) [*function with an optional argument*] 18-7
(sortl l) [*function with one argument*] 3-68
(sortn l) [*function with one argument*] 3-69
(sortp l) [*function with one argument*] 3-69
(sort fn l) [*function with two arguments*] 3-68
sort [*feature*] 3-68
(spanstring str1 str2 n) [*function with two or three arguments*] 3-97
(sqrt x) [*function with one argument*] 4-11
(srandom x) [*function with an optional argument*] 4-19

(sref strg n) [*function with two arguments*] 3-87
(sset strg n cn) [*function with three arguments*] 3-88
(standard-background-pattern screen) [*function with an optional argument*] 18-8
(standard-background screen) [*function with an optional argument*] 18-8
(standard-background screen) [*function with an optional argument*] 18-9
(standard-bold-font screen) [*function with an optional argument*] 18-7
(standard-busy-cursor screen) [*function with an optional argument*] 18-8
(standard-dark-gray-pattern screen) [*function with an optional argument*] 18-8
(standard-foreground-pattern screen) [*function with an optional argument*] 18-8
(standard-foreground screen) [*function with an optional argument*] 18-7
(standard-foreground screen) [*function with an optional argument*] 18-9
(standard-gc-cursor screen) [*function with an optional argument*] 18-8
(standard-lelisp-cursor screen) [*function with an optional argument*] 18-8
(standard-light-gray-pattern screen) [*function with an optional argument*] 18-8
(standard-medium-gray-pattern screen) [*function with an optional argument*] 18-8
(standard-roman-font screen) [*function with an optional argument*] 18-7
(stepeval e env) [*function with two arguments*] 7-11
stepeval [*programmable interrupt*] 7-10
(step s) [*special form*] 11-13
(stratom n strg i) [*function with three arguments*] 6-3
(stringp s) [*function with one argument*] 3-45
(string s) [*function with one argument*] 3-89
(#:struct:field o e) [*function with one or two arguments*] 5-3
(#:struct:make) [*function with no arguments*] 5-2
(structurep o) [*function with one argument*] 5-3
(sub1 x) [*function with one argument*] 4-13
(sublis al s) [*function with two arguments*] 3-67
(subsetp list1 list2 eq-func) [*function with two or three arguments*] 3-109
(subst-colors bytetmap l) [*function with two arguments*] 20-18
(substitute-color bytemap oldcol newcol) [*function with three arguments*] 20-18
(substring-equal size strg1 pos1 strg2 pos2) [*function with five arguments*] 3-96
(substring strg n1 n2) [*function with two or three arguments*] 3-92
(subst s1 s2 s) [*function with three arguments*] 3-57
(subtypep type1 type2) [*function with two arguments*] 5-6

(subversion) [*function with no arguments*] 7-17
(sub x y) [*function with two arguments*] 4-13
(super-itsoft package symb larg) [*function with three arguments*] 7-2
(suspend) [*function with no arguments*] 7-5
(symbolp s) [*function with one argument*] 3-44
(symbol pkgc strg) [*function with two arguments*] 3-83
{symb} [*macro character*] 5-14
(symeval symb) [*function with one argument*] 3-70
(synonymq sym1 sym2) [*special form*] 3-81
(synonym sym1 sym2) [*function with two arguments*] 3-81
#:sys-package:colon [*variable*] 6-30
#:sys-package:genarith [*variable*] 4-2
#:sys-package:itsoft [*variable*] 7-1
#:sys-package:sharp [*variable*] 6-23
#:sys-package:tty [*variable*] 15-11
(syserror symb s1 s2) [*function with three arguments*] 7-8
syserror [*programmable interrupt*] 7-7
#:system:compressed-icon [*variable*] 20-15
#:system:core-directory [*variable*] 7-12
#:system:core-extension [*variable*] 7-12
#:system:debug-line [*variable*] 11-12
#:system:defstruct-all-access-flag [*variable*] 5-2
#:system:editor [*variable*] 6-31
#:system:in-read-flag [*variable*] 6-7
#:system:inibitmap-after-restore-flag [*variable*] 7-16
#:system:initty-after-restore-flag [*variable*] 7-15
#:system:lelisp-extension [*variable*] 6-48
#:system:line-mode-flag [*variable*] 6-9
#:system:llib-directory [*variable*] 6-48
#:system:llub-directory [*variable*] 6-48
#:system:loaded-from-file [*property*] 3-18
#:system:loaded-from-file [*variable*] 3-18
#:system:loaded-from-file [*variable*] 3-70
#:system:obj-extension [*variable*] 13-12

#:system:path [*variable*] 6-55
#:system:previous-def-flag [*variable*] 3-17
#:system:previous-def [*property*] 3-17
#:system:print-case-flag [*variable*] 6-37
#:system:print-for-read [*variable*] 6-37
#:system:print-msgs [*variable*] 6-48
#:system:print-package-flag [*variable*] 6-37
#:system:print-with-abbrev-flag [*variable*] 5-14
#:system:read-case-flag [*variable*] 6-10
#:system:real-terminal-flag [*variable*] 6-9
#:system:redef-flag [*variable*] 3-17
#:system:stack-depth [*variable*] 11-12
#:system:termcap-file [*variable*] 15-2
#:system:terminfo-directory [*variable*] 15-2
(#:system:toplevel-tag) [*escape*] 7-19
#:system:unixp [*variable*] 7-17
#:system:virtty-directory [*variable*] 15-1
(system) [*function with no arguments*] 7-16
(tagbody ec_1 ... ec_n) [*special form*] 3-33
(tag symb s_1 ... s_n) [*special form*] 3-38
(tailp s l) [*function with two arguments*] 3-50
(tanh z) [*function with one argument*] 10-9
(tclass-namep symbol) [*function with one argument*] 5-16
(tconscl s) [*function with one argument*] 3-59
(tconsmk s) [*function with one argument*] 3-59
(tconsp s) [*function with one argument*] 3-59
(tcons s1 s2) [*function with two arguments*] 3-59
(temporary-file-pathname name) [*function with one argument*] 6-73
(teread) [*function with no arguments*] 6-7
termcap [*feature*] 15-2
terminfo [*feature*] 15-2
(terpri n) [*function with an optional argument*] 6-34
(times x_1 ... x_n) [*function with a variable number of arguments*] 4-25
(time e) [*function with one argument*] 7-24

`#:toplevel:eval` [*variable*] 7-18

`#:toplevel:read` [*variable*] 7-18

`#:toplevel:status` [*variable*] 7-18

`(toplevel)` [*function with no arguments*] 7-18

`toplevel` [*programmable interrupt*] 7-18

`#:trace:arg1` [*variable*] 11-3

`#:trace:arg2` [*variable*] 11-3

`#:trace:arg3` [*variable*] 11-3

`#:trace:not-in-trace-flag` [*variable*] 11-4

`#:trace:trace` [*variable*] 11-5

`#:trace:value` [*variable*] 11-4

`(traceval e env)` [*function with one or two arguments*] 7-11

`(trace` $\texttt{trace}_1$ ... $\texttt{trace}_n$`)` [*special form*] 11-2

`(transitive-closure fn list eq-func)` [*function with two or three arguments*] 3-110

`(true-pathname path)` [*function with one argument*] 6-71

`(true` $\texttt{e}_1$ ... $\texttt{e}_n$`)` [*function with a variable number of arguments*] 3-42

`(truncate x)` [*function with one argument*] 4-3

`(tryinparallel` $\texttt{e}_1$ ... $\texttt{e}_n$`)` [*special form*] 7-5

`(tyattrib i)` [*function with an optional argument*] 15-8

`(tyback cn)` [*function with one argument*] 15-4

`(tybeep)` [*function with no arguments*] 15-7

`(tybs cn)` [*function with one argument*] 15-6

`(tycleol)` [*function with no arguments*] 15-7

`(tycleos)` [*function with no arguments*] 15-7

`(tycls)` [*function with no arguments*] 15-6

`(tycn cn)` [*function with one argument*] 15-3

`(tycot x y` $\texttt{cn}_1$ ... $\texttt{cn}_n$`)` [*function with a variable number of arguments*] 15-9

`(tyco x y` $\texttt{cn}_1$ ... $\texttt{cn}_n$`)` [*function with a variable number of arguments*] 15-8

`(tycr)` [*function with no arguments*] 15-6

`(tycursor x y)` [*function with two arguments*] 15-6

`(tydelch)` [*function with no arguments*] 15-7

`(tydelcn cn)` [*function with one argument*] 15-7

`(tydelln)` [*function with no arguments*] 15-8

`(tydownkey)` [*function with no arguments*] 15-7

(tyepilogue) [*function with no arguments*] 15-5
(tyerror list) [*function with one argument*] 15-3
(tyflush) [*function with no arguments*] 15-5
(tyinsch ch) [*function with one argument*] 15-7
(tyinscn cn) [*function with one argument*] 15-7
(tyinsln) [*function with no arguments*] 15-8
(tyinstring string) [*function with one argument*] 15-3
(tyi) [*function with no arguments*] 15-3
(tyleftkey) [*function with no arguments*] 15-7
(tynewline) [*function with no arguments*] 15-4
(tyod n nc) [*function with two arguments*] 15-5
(tyo o_1 ... o_n) [*function with a variable number of arguments*] 15-4
(type-of s) [*function with one argument*] 5-5
(typech ch symb) [*function with one or two arguments*] 6-16
(typecn cn symb) [*function with one or two arguments*] 6-16
(typefn symb) [*function with one argument*] 3-78
(typep s type) [*function with two arguments*] 5-6
(typestring strg symb) [*function with one or two arguments*] 3-88
(typevector vect symb) [*function with one or two arguments*] 3-101
(typrologue) [*function with no arguments*] 15-5
(tyrightkey) [*function with no arguments*] 15-7
(tyshowcursor i) [*function with an optional argument*] 15-8
(tystring string length) [*function with two arguments*] 15-3
(tys) [*function with no arguments*] 15-3
(tyupkey) [*function with no arguments*] 15-7
(tyxmax) [*function with no arguments*] 15-6
(tyymax) [*function with no arguments*] 15-6
(unexit symb s_1 ... s_n) [*special form*] 3-39
(union list1 list2 eq-func) [*function with two or three arguments*] 3-107
(unless s_1 s_2 ... s_n) [*special form*] 3-27
unmap-window [*event*] 19-4
(unstep s_1 ... s_n) [*special form*] 11-13
(untilexit symb e_1 ... e_n) [*special form*] 3-39
(until s s_1 ... s_n) [*special form*] 3-31

(untrace sym$_1$ ... sym$_n$) [*special form*] 11-2
(unwind n s$_1$ ... s$_n$) [*special form*] 3-40
up-event [*event*] 19-2
(uppercase cn) [*function with one argument*] 3-97
(user-homedir-pathname) [*function with no arguments*] 6-72
(user-interrupt) [*function with no arguments*] 7-4
user-interrupt [*programmable interrupt*] 7-4
#u [*#-macro*] 6-65
(valfn symb) [*function with one argument*] 3-79
(variablep s) [*function with one argument*] 3-44
(vdtend) [*function with no arguments*] 15-11
(vdt) [*function with no arguments*] 15-11
(vectorp s) [*function with one argument*] 3-45
(vector s$_1$ ... s$_n$) [*function with a variable number of arguments*] 3-99
(version) [*function with no arguments*] 7-17
visibility-change [*event*] 19-4
(vlength vect) [*function with one argument*] 3-100
(vref vect n) [*function with two arguments*] 3-100
(vset vect n e) [*function with three arguments*] 3-100
W.0 [*Complice warning message*] 13-22
W.10 [*Complice warning message*] 13-24
W.11 [*Complice warning message*] 13-25
W.2 [*Complice warning message*] 13-22
W.3 [*Complice warning message*] 13-22
W.4 [*Complice warning message*] 13-23
W.5 [*Complice warning message*] 13-23
W.6 [*Complice warning message*] 13-23
W.7 [*Complice warning message*] 13-23
W.8 [*Complice warning message*] 13-24
W.9 [*Complice warning message*] 13-24
(whanoiend) [*function with no arguments*] 15-11
(whanoi n) [*function with one argument*] 15-11
(when s$_1$ s$_2$ ... s$_n$) [*special form*] 3-27
(while s s$_1$ ... s$_n$) [*special form*] 3-31

(width-space) [*function with no arguments*] 18-24
(width-substring string start length) [*function with three arguments*] 18-23
(width-substring string start length) [*function with three arguments*] 20-13
(wildcard path) [*function with one argument*] 6-67
(window-background win color) [*function with one or two arguments*] 18-18
(window-bitmap window) [*function with one argument*] 20-14
(window-border win border) [*function with one or two arguments*] 18-18
(window-clear-region win x y w h) [*function with five arguments*] 18-18
(window-events-list win events) [*function with one or two arguments*] 18-18
(window-state win state) [*function with one or two arguments*] 18-18
(window-title win title) [*function with one or two arguments*] 18-18
(windowp win) [*function with one argument*] 18-18
(with-input-from-string string . body) [*macro*] 6-40
(with-output-to-string string n . body) [*macro*] 6-40
(without-interrupts e_1 ... e_n) [*special form*] 7-3
(with l s_1 ... s_n) [*special form*] 3-25
(x-base-space) [*function with no arguments*] 18-24
(x-base-substring string start length) [*function with three arguments*] 18-24
(x-base-substring string start length) [*function with three arguments*] 20-13
(x-inc-substring string start length) [*function with three arguments*] 18-24
(x-inc-substring string start length) [*function with three arguments*] 20-13
(xcons s1 s2) [*function with two arguments*] 3-52
(y-base-space) [*function with no arguments*] 18-24
(y-base-substring string start length) [*function with three arguments*] 18-24
(y-base-substring string start length) [*function with three arguments*] 20-13
(y-inc-substring string start length) [*function with three arguments*] 18-24
(y-inc-substring string start length) [*function with three arguments*] 20-13
(zerop x) [*function with one argument*] 4-9