Programming Language Eulisp

Version 0.99
Programming Language EuLisp, version 0.99

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword</td>
<td>1</td>
</tr>
<tr>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>1 Language Structure</td>
<td>2</td>
</tr>
<tr>
<td>2 Scope</td>
<td>3</td>
</tr>
<tr>
<td>3 Conformance Definitions</td>
<td>3</td>
</tr>
<tr>
<td>4 Error Definitions</td>
<td>3</td>
</tr>
<tr>
<td>5 Compliance</td>
<td>4</td>
</tr>
<tr>
<td>6 Conventions</td>
<td>4</td>
</tr>
<tr>
<td>6.1 Layout and Typography</td>
<td>4</td>
</tr>
<tr>
<td>6.2 Meta-Language</td>
<td>6</td>
</tr>
<tr>
<td>6.3 Naming</td>
<td>6</td>
</tr>
<tr>
<td>7 Definitions</td>
<td>6</td>
</tr>
<tr>
<td>8 Syntax</td>
<td>9</td>
</tr>
<tr>
<td>8.1 Whitespace and Comments</td>
<td>9</td>
</tr>
<tr>
<td>8.2 Objects</td>
<td>9</td>
</tr>
<tr>
<td>9 Modules</td>
<td>10</td>
</tr>
<tr>
<td>9.1 Directives</td>
<td>10</td>
</tr>
<tr>
<td>9.2 Export Directive</td>
<td>10</td>
</tr>
<tr>
<td>9.3 Import Directive</td>
<td>10</td>
</tr>
<tr>
<td>9.4 Expose Directive</td>
<td>11</td>
</tr>
<tr>
<td>9.5 Syntax Directive</td>
<td>11</td>
</tr>
<tr>
<td>9.6 Definitions and Expressions</td>
<td>11</td>
</tr>
<tr>
<td>9.7 Module Processing</td>
<td>12</td>
</tr>
<tr>
<td>9.8 Module Definition</td>
<td>12</td>
</tr>
<tr>
<td>10 Objects</td>
<td>13</td>
</tr>
<tr>
<td>10.1 System Defined Classes</td>
<td>13</td>
</tr>
<tr>
<td>10.2 Single Inheritance</td>
<td>14</td>
</tr>
<tr>
<td>10.3 Defining Classes</td>
<td>14</td>
</tr>
<tr>
<td>10.4 Defining Generic Functions and Methods</td>
<td>15</td>
</tr>
<tr>
<td>10.5 Specializing Methods</td>
<td>17</td>
</tr>
<tr>
<td>10.6 Method Lookup and Generic Dispatch</td>
<td>18</td>
</tr>
<tr>
<td>10.7 Creating and Initializing Objects</td>
<td>18</td>
</tr>
<tr>
<td>10.8 Accessing Slots</td>
<td>19</td>
</tr>
<tr>
<td>11 Concurrency</td>
<td>19</td>
</tr>
<tr>
<td>11.1 Threads</td>
<td>20</td>
</tr>
<tr>
<td>11.2 Locks</td>
<td>21</td>
</tr>
<tr>
<td>12 Conditions</td>
<td>22</td>
</tr>
<tr>
<td>12.1 Condition Classes</td>
<td>22</td>
</tr>
<tr>
<td>12.2 Condition Handling</td>
<td>24</td>
</tr>
</tbody>
</table>
Programming Language EuLisp, version 0.99

13 Expressions, Definitions and Control Forms ........................................ 26
  13.1 Simple Expressions ................................................................. 26
  13.2 Functions: creation, definition and application .............................. 27
  13.3 Destructive Operations ............................................................. 28
  13.4 Conditional Expressions ........................................................... 29
  13.5 Variable Binding and Sequences .................................................. 30
  13.6 Events ...................................................................................... 33
  13.7 Quasiquotation Expressions .......................................................... 33
  13.8 Summary of Level-0 Expressions and Definitions ............................. 34
    13.8.1 Syntax of Level-0 defining forms ........................................ 34
    13.8.2 Syntax of Level-0 expressions ........................................... 35
    13.8.3 Syntax of Level-0 macros .................................................. 35

Annexes
A Level-0 Module Library ........................................................................... 37
  A.1 Characters .................................................................................... 37
  A.2 Collections .................................................................................... 39
  A.3 Comparison .................................................................................... 43
  A.4 Conversion .................................................................................... 45
  A.5 Copying .......................................................................................... 46
  A.6 Double Precision Floats ................................................................. 47
  A.7 Elementary Functions ..................................................................... 49
  A.8 Floating Point Numbers ................................................................... 51
  A.9 Fixed Precision Integers .................................................................. 52
  A.10 Formatted-IO ................................................................................. 53
  A.11 Integers ....................................................................................... 55
  A.12 Lists ............................................................................................. 56
  A.13 Numbers ....................................................................................... 59
  A.14 Streams ....................................................................................... 62
  A.15 Strings ......................................................................................... 66
  A.16 Symbols ....................................................................................... 68
  A.17 Tables ........................................................................................... 70
  A.18 Vectors ......................................................................................... 71
  A.19 Syntax of Level-0 objects ............................................................. 72
B Programming Language EuLisp, Level-1 .................................................. 74
  B.1 Classes and Objects ......................................................................... 74
  B.2 Generic Functions ............................................................................ 75
  B.3 Methods ......................................................................................... 76
  B.4 Object Introspection ....................................................................... 78
  B.5 Class Introspection ......................................................................... 78
  B.6 Slot Introspection ........................................................................... 79
  B.7 Generic Function Introspection ....................................................... 79
  B.8 Method Introspection ...................................................................... 80
  B.9 Class Initialization .......................................................................... 81
  B.10 Slot Description Initialization ....................................................... 82
  B.11 Generic Function Initialization ...................................................... 82
  B.12 Method Initialization ..................................................................... 83
  B.13 Inheritance Protocol ...................................................................... 83
  B.14 Slot Access Protocol ...................................................................... 87
  B.15 Method Lookup and Generic Dispatch ......................................... 91
  B.16 Low Level Allocation Primitives .................................................... 92
  B.17 Dynamic Binding .......................................................................... 93
  B.18 Exit Extensions ............................................................................. 94
  B.19 Summary of Level-1 Expressions and Definitions ........................... 95
    B.19.1 Syntax of Level-1 defining forms ...................................... 95
    B.19.2 Syntax of Level-1 expressions .......................................... 95
    B.19.3 Syntax of Level-1 macros ................................................. 95

Indexes ...................................................................................................... 98
  Class Index ......................................................................................... 98
  Condition Index ................................................................................. 99
  Constant Index ................................................................................... 100
  Function Index .................................................................................. 101
  Macro Index ....................................................................................... 102
  Generic Function Index ....................................................................... 103
  Method Index ..................................................................................... 104
Programming Language EuLisp, version 0.99

General Index .......................................................... 105

Figures
1  Example of module directives ................................. 11
2  Example module using expose ................................. 11
3  Level-0 initial class hierarchy .............................. 13
4  Level-0 initial condition class hierarchy ................. 23
5  Illustration of handler actions .............................. 25
6  Example using let/cc .............................................. 31
7  Interaction of unwind-protect with non-local exits ....... 33
A.1 Level-0 number class hierarchy ............................ 59
B.1 Level-1 class hierarchy ........................................ 78
B.2 Initialization Call Structure ................................ 82

Tables
1  Modules comprising eulisp-level-0 .......................... 3
2  Minimal character set ............................................ 9
3  Syntax of objects for reading and writing ................ 9
4  Module syntax ..................................................... 12
5  defstruct syntax ................................................ 14
6  defgeneric rewrite rules ....................................... 16
A.1 Character Syntax ............................................... 37
A.2 Summary of comparison functions .......................... 44
A.3 Floating Point Syntax ......................................... 51
A.4 Integer Syntax .................................................. 55
A.5 Pair and List Syntax .......................................... 56
A.6 Examples of string literals ................................... 66
A.7 String Syntax .................................................... 66
A.8 Identifier/Symbol Syntax ..................................... 69
A.9 Vector Syntax .................................................... 71
Programming Language EuLisp, version 0.99

Foreword

The EuLISP group first met in September 1985 at IRCAM in Paris to discuss the idea of a new dialect of Lisp, which should be less constrained by the past than Common Lisp and less minimalist than Scheme. Subsequent meetings formulated the view of EuLISP that was presented at the 1986 ACM Conference on Lisp and Functional Programming held at MIT, Cambridge, Massachusetts 12 and at the European Conference on Artificial Intelligence (ECAI-86) held in Brighton, Sussex 18. Since then, progress has not been steady, but happening as various people had sufficient time and energy to develop part of the language. Consequently, although the vision of the language has in the most part been shared over this period, only certain parts were turned into physical descriptions and implementations. For a nine month period starting in January 1989, through the support of INRIA, it became possible to start writing the EuLISP definition. Since then, affairs have returned to their previous state, but with the evolution of the implementations of EuLISP and the background of the foundations laid by the INRIA-supported work, there is convergence to a consistent and practical definition.

The acknowledgments for this definition fall into three categories: intellectual, personal, and financial.

The ancestors of EuLISP (in alphabetical order) are Common Lisp7, InserLISP9, LE-LISP 4, LISP/VM 1, Scheme 6, and T 14 16. Thus, the authors of this report are pleased to acknowledge both the authors of the manuals and definitions of the above languages and the many who have dissected and extended those languages in individual papers. The various papers on Standard ML 11 and the draft report on Haskell 8 have also provided much useful input.

The writing of this report has, at various stages, been supported by Bull S.A., Gesellschaft für Mathematik und Datenverarbeitung (GMD, Sankt Augustin), Ecole Polytechnique (LIX), ILOG S.A., Institut National de Recherche en Informatique et en Automatique (INRIA), University of Bath, and Université Paris VI (LITP). The authors gratefully acknowledge this support. Many people from European Community countries have attended and contributed to EuLISP meetings since they started, and the authors would like to thank all those who have helped in the development of the language.

In the beginning, the work of the EuLISP group was supported by the institutions or companies where the participants worked, but in 1987 DG XIII (Information technology directorate) of the Commission of the European Communities agreed to support the continuation of the working group by funding meetings and providing places to meet. It can honestly be said that without this support EuLISP would not have reached its present state. In addition, the EuLISP group is grateful for the support of: British Council in France (Alliance programme), British Council in Spain (Acciones Integradas programme), British Standards Institute, Deutscher Akademischer Austauschdienst (DAAD), Departament de Lenguatges i Sistemes Informàtics (LSI, Universitat Politècnica de Catalunya), Fraunhofer Gesellschaft Institut für Software und Systemtechnik, Gesellschaft für Mathematik und Datenverarbeitung (GMD), ILOG S.A., Insiders GmbH, Institut National de Recherche en Informatique et en Automatique (INRIA), Institut de Recherche et de Coordinación AcústicaMusique (IRCAM), Rank Xerox France, Science and Engineering Research Council (UK), Siemens AG, University of Bath, University of Technology, Delft, University of Edinburgh, Universitat Erlangen and Université Paris VI (LITP).


The editors of the EuLISP definition wish particularly to acknowledge the work of Harley Davis on the first versions of the description of the object system. The second version was largely the work of Harry Bretthauer, with the assistance of Jürgen Kopp, Harley Davis and Keith Playford.

Julian Padget (jmp@maths.bath.ac.uk)
School of Mathematical Sciences
University of Bath
Bath, Avon, BA2 7AY, UK

Greg Nuyens (nuyens@ilog.com)
ILOG SA
2, Avenue Galliéni
94353 Genilly CEDEX, FRANCE

The editors.
Introduction

EuLisp is a dialect of Lisp and as such owes much to the great body of work that has been done on language design in the name of Lisp over the last thirty years. The distinguishing features of EuLisp are (i) the integration of the classical Lisp type system and the object system into a single class hierarchy (ii) the complementary abstraction facilities provided by the class and the module mechanism (iii) support for concurrent execution.

Here is a brief summary of the main features of the language.

- Classes are first-class objects. The class structure integrates the primitive classes describing fundamental datatypes, the predefined classes and user-defined classes.

- Modules together with classes are the building blocks of both the EuLisp language and of applications written in EuLisp. The module system exists to limit access to objects by name. That is, modules allow for hidden definitions. Each module defines a fresh, empty, lexical environment.

- Multiple control threads can be created in EuLisp and the concurrency model has been designed to allow consistency across a wide range of architectures. Access to shared data can be controlled via locks (semaphores). Event-based programming is supported through a generic waiting function.

- Both functions and continuations are first-class in EuLisp, but the latter are not as general as in Scheme because they can only be used in the dynamic extent of their creation. That implies they can only be used once.

- A condition mechanism which is fully integrated with both classes and threads, allows for the definition of generic handlers and which supports both propagation of conditions and continuous handling.

- Dynamically scoped bindings can be created in EuLisp, but their use is restricted, as in Scheme, EuLisp enforces a strong distinction between lexical bindings and dynamic bindings by requiring the manipulation of the latter via special forms.

EuLisp does not claim any particular Lisp dialect as its closest relative, although parts of it were influenced by features found in Common Lisp, InterLisp, LISP, LISP/VM, Scheme, and T. EuLisp both introduces new ideas and takes from these Lisps. It also extends or simplifies their ideas as necessary. But this is not the place for a detailed language comparison. That can be drawn from the rest of this text.

EuLisp breaks with Lisp tradition in describing all its types (in fact, classes) in terms of an object system. This is called The EuLisp Object System, or ELOS. ELOS incorporates elements of the Common Lisp Object System (CLOS) 2, ObjJLisp 7, Oaklisp 9, MicroCeyx 5, and MCS 3.

1 Language Structure

The EuLisp definition comprises the following items:

- **Level-0** comprises all the level-0 functions, macros and special forms, which is this text minus Annex B. The object system can be extended by user-defined structure classes, and generic functions.

- **Level-1** extends level-0 with the functions, macros and special forms defined in Annex B. The object system can be extended by user-defined classes and metaclasses. The implementation of level 0 is not necessarily written or writable as a conforming level-0 program.

A level-0 function is a (generic) function defined in this text to be part of a conforming processor for level-0. A function defined in terms of level-0 operations is an example of a level-0 application.

Much of the functionality of EuLisp is defined in terms of modules. These modules might be available (and used) at any level, but certain modules are required at a given level. Whenever a module depends on the operations available at a given level, that dependency will be specified.

EuLisp level-0 is provided by the module eulisp-level-0. This module imports and re-exports the modules specified in Table 1.

This definition is organized in three parts:

- Sections 8-13 describe the core of level-0 of EuLisp, covering modules, simple classes, objects and generic functions, threads, conditions, control forms and events. These sections contain the information about EuLisp that characterizes the language.

- Annex A describes the additional classes required at level-0 and the operations defined on instances of those classes. The annex is organized by class in alphabetical order. These sections contain information about the predefined classes in EuLisp that are necessary to make the language usable, but is not central to an appreciation of the language.

- Annex B describes the reflective aspects of the object system and how to program the metaobject protocol and some additional control forms.

Prior to these, sections 2-6 define the scope of the text and error definitions and typographical and layout conventions used in this text.
Table 1 — Modules comprising eulisp-level-0

<table>
<thead>
<tr>
<th>Module</th>
<th>Section(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>A.1</td>
</tr>
<tr>
<td>collection</td>
<td>A.2</td>
</tr>
<tr>
<td>compare</td>
<td>A.3</td>
</tr>
<tr>
<td>condition</td>
<td>12</td>
</tr>
<tr>
<td>convert</td>
<td>A.4</td>
</tr>
<tr>
<td>copy</td>
<td>A.5</td>
</tr>
<tr>
<td>double-float</td>
<td>A.6</td>
</tr>
<tr>
<td>elementary-functions</td>
<td>A.7</td>
</tr>
<tr>
<td>event</td>
<td>13.6</td>
</tr>
<tr>
<td>fixed-precision-integer</td>
<td>A.9</td>
</tr>
<tr>
<td>formatted-io</td>
<td>A.10</td>
</tr>
<tr>
<td>function</td>
<td>13.2</td>
</tr>
<tr>
<td>lock</td>
<td>11.2</td>
</tr>
<tr>
<td>null</td>
<td>A.12</td>
</tr>
<tr>
<td>number</td>
<td>A.13</td>
</tr>
<tr>
<td>object-0</td>
<td>10</td>
</tr>
<tr>
<td>pair</td>
<td>A.12</td>
</tr>
<tr>
<td>stream</td>
<td>A.14</td>
</tr>
<tr>
<td>string</td>
<td>A.15</td>
</tr>
<tr>
<td>symbol</td>
<td>A.16</td>
</tr>
<tr>
<td>syntax-0</td>
<td>13.7</td>
</tr>
<tr>
<td>table</td>
<td>A.17</td>
</tr>
<tr>
<td>thread</td>
<td>11.1</td>
</tr>
<tr>
<td>vector</td>
<td>A.18</td>
</tr>
</tbody>
</table>

2 Scope

This text specifies the syntax and semantics of the computer programming language EU LISP by defining the requirements for a conforming EU LISP processor and a conforming EU LISP program (the textual representation of data and algorithms).

This text does not specify:

a) The size or complexity of a EU LISP program that will exceed the capacity of any specific configuration or processor, nor the actions to be taken when those limits are exceeded.

b) The minimal requirements of a configuration that is capable of supporting an implementation of a EU LISP processor.

c) The method of preparation of a EU LISP program for execution or the method of activation of this EU LISP program once prepared.

d) The method of reporting errors, warnings or exceptions to the client of a EU LISP processor.

e) The typographical representation of a EU LISP program for human reading.

f) The means to map module names to the filing system or other object storage system attached to the processor.

To clarify certain instances of the use of English in this text the following definitions are provided:

must — a verbal form used to introduce a required property. All conforming processors must satisfy the property.

should — A verbal form used to introduce a strongly recommended property. Implementors of processors are urged (but not required) to satisfy the property.

3 Conformance Definitions

The following terms are general in that they could be applied to the definition of any programming language. They are derived from ISO/IEC TR 10034: 1990.

3.1 configuration: Host and target computers, any operating systems(s) and software (run-time system) used to operate a language processor.

3.2 conformity clause: Statement that is not part of the language definition but that specifies requirements for compliance with the language standard.

3.3 conforming program: Program which is written in the language defined by the language standard and which obeys all the conformity clauses for programs in the language standard.

3.4 conforming processor: Processor which processes conforming programs and program units and which obeys all the conformity clauses for processors in the language standard.

3.5 error: Incorrect program construct or incorrect functioning of a program as defined by the language standard.

3.6 extension: Facility in the processor that is not specified in the language standard but that does not cause any ambiguity or contradiction when added to the language standard.

3.7 implementation-defined: Specific to the processor, but required by the language standard to be defined and documented by the implementer.

3.8 processor: Compiler, translator or interpreter working in combination with a configuration.

4 Error Definitions

Errors in the language described in this definition fall into one of the following three classes:

4.1 dynamic error: An error which is detected during the execution of a EU LISP program or which is a violation of the defined dynamic behaviour of EU LISP. Dynamic errors have two classifications:

a) Where a conforming processor is required to detect the erroneous situation or behaviour and report it. This is signified by the phrase an error is signalled. The condition class to be signalled is specified. Signalling an error consists of identifying the condition class related to the error and allocating an instance of it. This instance is initialized with information dependent on the condition class. A conforming EU LISP program can rely on the fact that this condition will be signalled.
b) Where a conforming processor might or might not detect and report upon the error. This is signified by the phrase . . . is an error. A processor should provide a mode where such errors are detected and reported where possible.

4.2 environmental error: An error which is detected by the configuration supporting the EULISP processor. The processor must signal the corresponding dynamic error which is identified and handled as described above.

4.3 static error: An error which is detected during the preparation of a EULISP program for execution, such as a violation of the syntax or static semantics of EULISP by the program under preparation.

NOTE — The violation of the syntactic or static semantic requirements is not an error, but an error might be signalled by the program performing the analysis of the EULISP program.

All errors specified in this definition are dynamic unless explicitly stated otherwise.

5 Compliance

An EULISP processor can conform at either of the two levels defined under Language Structure in the Introduction. Thus a level-0 conforming processor must support all the basic expressions, classes and class operations defined at level-0. A level-1 conforming processor must support all the basic expressions, classes, class operations and modules defined at level-1 and at level-0.

The following two statements govern the conformance of a processor at a given level.

a) A conforming processor must correctly process all programs conforming both to the standard at the specified level and the implementation-defined features of the processor.

b) A conforming processor should offer a facility to report the use of an extension which is statically determinable solely from inspection of a program, without execution. (It is also considered desirable that a facility to report the use of an extension which is only determinable dynamically be offered.)

A level-0 conforming program is one which observes the syntax and semantics defined for level-0. A level-0 conforming program might not conform at level-1. A strictly-conforming level-0 program is one that also conforms at level-1. A level-1 conforming program observes the syntax and semantics defined for level-1.

In addition, a conforming program at any level must not use any extensions implemented by a language processor, but it can rely on implementation-defined features.

The documentation of a conforming processor must include:

a) A list of all the implementation-defined definitions or values.

b) A list of all the features of the language standard which are dependent on the processor and not implemented by this processor due to non-support of a particular facility, where such non-support is permitted by the standard.

c) A list of all the features of the language implemented by this processor which are extensions to the standard language.

d) A statement of conformity, giving the complete reference of the language standard with which conformity is claimed, and, if appropriate, the level of the language supported by this processor.

6 Conventions

This section defines the conventions employed in this text, how definitions will be laid out, the typeface to be used, the meta-language used in descriptions and the naming conventions. A later section (?) contains definitions of the terms used in this text.

6.1 Layout and Typography

Both layout and fonts are used to help in the description of EULISP. A language element is defined as an entry with its name as the heading of a clause, coupled with its classification. Examples of several kinds of entry are now given. Some subsections of entries are optional and are only given where it is felt necessary.

6.1.1 a-special-form

6.1.1.1 Syntax

(a-special-form form1 ... formn)

6.1.1.2 Arguments

form1: description of structure and role of form1.

:::

formn: description of structure and role of formn.

6.1.1.3 Result

A description of the result.

6.1.1.4 Remarks

Any additional information defining the behaviour of a-special-form

6.1.1.5 Examples

Some examples of use of the special form and the behaviour that should result.

6.1.1.6 See also: Cross references to related entries.
6.1.2 a-function

6.1.2.1 Arguments

\textit{argument-a}: information about the class or classes of \textit{argument-a}.

\ldots

[\textit{argument-a}]: information about the class or classes of the optional argument \textit{argument-n}.

6.1.2.2 Result

A description of the result and, possibly, its class.

6.1.2.3 Remarks

Any additional information about the actions of \textit{a-function}.

6.1.2.4 Examples

Some examples of calling the function with certain arguments and the result that should be returned.

6.1.2.5 See also: Cross references to related entries.

6.1.3 a-generic

6.1.3.1 Generic Arguments

\begin{quote}
\begin{enumerate}
\item \textit{(argument-a <class-a>)}: means that \textit{argument-a} of a-generic must be an instance of \textit{<class-a>} and that \textit{argument-a} is one of the arguments on which a-generic specializes. Furthermore, each method defined on a-generic may specialize only on a subclass of \textit{<class-a>} for \textit{argument-a}.
\item \textit{argument-n}: means that (i) \textit{argument-n} is an instance of \textit{<object>}, \textit{i.e.} it is unconstrained, (ii) a-generic does not specialize on \textit{argument-n}, (iii) no method on a-generic can specialize on \textit{argument-n}.
\end{enumerate}
\end{quote}

6.1.3.2 Result

A description of the result and, possibly, its class.

6.1.3.3 Remarks

Any additional information about the actions of a-generic. This will probably be in general terms, since the actual behaviour will be determined by the methods.

6.1.3.4 See also: Cross references to related entries.

6.1.4 a-generic

(A method on a-generic with the following specialized arguments.)

6.1.4.1 Specialized Arguments

\begin{quote}
\begin{enumerate}
\item \textit{(argument-a <class-a>)}: means that \textit{argument-a} of the method must be an instance of \textit{<class-a>}. Of course, this argument must be one which was defined in a-generic as being open to specialization and \textit{<class-a>} must be a subclass of the class.
\item \textit{argument-n}: means that (i) \textit{argument-n} is an instance of \textit{<object>}, \textit{i.e.} it is unconstrained, (ii) a-generic does not specialize on \textit{argument-n}, (iii) no method on a-generic can specialize on \textit{argument-n}.
\end{enumerate}
\end{quote}

6.1.4.2 Result

A description of the result and, possibly, its class.

6.1.4.3 Remarks

Any additional information about the actions of this method attached to a-generic.

6.1.4.4 See also: Cross references to related entries.

6.1.5 a-condition

6.1.5.1 Initialization Options

\begin{quote}
\begin{enumerate}
\item \textit{initarg-a value-a}: means that an instance of \textit{<a-condition>} has a slot called \textit{initarg-a} which should be initialized to \textit{value-a}, where \textit{value-a} is often the name of a class, indicating that \textit{value-a} should be an instance of that class and a description of the information that \textit{value-a} is supposed to provide about the exceptional situation that has arisen.
\item \textit{initarg-n value-n}: As for \textit{initarg-a}.
\end{enumerate}
\end{quote}

6.1.5.2 Remarks

Any additional information about the circumstances in which the condition will be signalled.

6.1.6 <class-name>

6.1.6.1 Initialization Options

\begin{quote}
\begin{enumerate}
\item \textit{initarg-a value-a}: means that \textit{<class-name>} has an initarg whose name is \textit{initarg-a} and the description will usually say of what class (or classes) \textit{value-a} should be an instance. This initarg is required.
\item [\textit{initarg-n value-n}]: The enclosing square brackets denote that this initarg is optional. Otherwise the interpretation of the definition is as for \textit{initarg-a}.
\end{enumerate}
\end{quote}

6.1.6.2 Remarks

A description of the role of \textit{<class-name>}.


6.2 Meta-Language

The terms used in the following descriptions are defined in Annex 7.

A standard function denotes an immutable top-lexical binding of the defined name. All the definitions in this text are bindings in some module except for the special form operators, which have no definition anywhere. All bindings and all the special form operators can be renamed.

NOTE — A description making mention of “an x” where “x” is the name a class usually means “an instance of <x>”.

Frequently, a class-descriptive name will be used in the argument list of a function description to indicate a restriction on the domain to which that argument belongs. In the case of a function, it is an error to call it with a value outside the specified domain. A generic function can be defined with a particular domain and/or range. In this case, any new methods must respect the domain and/or range of the generic function to which they are to be attached. The use of a class-descriptive name in the context of a generic function definition defines the intention of the definition, and is not necessarily a policed restriction.

If it is required to indicate repetition, the notation: expression* and expression+ will be used for zero or more and one or more occurrences, respectively. The arguments in some function descriptions are enclosed in square brackets—graphic representation “[ ] and ‘’”. This indicates that the argument is optional. The accompanying text will explain what default values are used.

The result-class of an operation, except in one case, refers to a primitive or a defined class described in this definition. The exception is for predicates. Predicates are defined to return either the empty list—written nil—representing the boolean value false, or any value other than nil, representing true. Although the class containing exactly this set of values is not defined in the language, notation is abused for convenience and boolean is defined, for the purposes of this report, to mean that set of values. If the true value is a useful value, it is specified precisely in the description of the function.

6.3 Naming

Naming conventions are applied in the descriptions of primitive and defined classes as well as in the choice of other function names. Here is a list of the conventions and some examples of their use.

6.3.1 “<name>” wrapping: By convention, classes have names which begin with “<” and end with “>”.

6.3.2 “binary-” prefix: The two argument version of a n-ary argument function. There is not always a correspondence between the root and the name of the n-ary function, for example binary-plus is the two argument (generic) function corresponding to the n-ary argument + function.

6.3.3 “<class>” suffix: The name of a metaclass of a set of related classes. For example, <function-class>, which is the class of <simple-function>, <generic-function> and any of their subclasses and <condition-class> is the class of all condition classes. The exception is <class> itself which is the default metaclass. The prefix should describe the general domain of the classes in question, but not necessarily any particular class in the set.

6.3.4 “generic-” prefix: The generic version of the function named by the stem.

6.3.5 hyphenation: Function and class names made up of more than one word are hyphenated, for example: compute-specialized-slot-description-class.

6.3.6 “p” suffix: A predicate function is named by a “p” suffix if the function or class name (after removing the enclosing < and >) is not hyphenated, for instance, consp, and is named by a “-p” suffix if it is, for instance double-float-p.

7 Definitions

This set of definitions, which are used throughout this definition, is self-consistent but might not agree with notions accepted in other language definitions. The terms are defined in alphabetical rather than dependency order and where a definition uses a term defined elsewhere in this section it is written in italics. Some of the terms defined here are redundant. Names in courier font refer to entities defined in the language.

7.1 boolean: A boolean value is either false, which is represented by the empty list—written () — and also the value of nil—or true, which is represented by any other value than ().

7.2 class: A class is an object which describes the structure and behaviour of a set of objects which are its instances. A class object contains inheritance information and a set of slot descriptions which define the structure of its instances. A class object is an instance of a metaclass. All classes in EuLisp are subclasses of <object>, and all instances of <class> are classes.

7.3 defining form: Any form or any macro expression expanding into a form whose operator is one of defclass, defcondition, defconstant, defgeneric, deflocal, defmacro, defstruct, defun, or defvar.

7.4 direct instance: A direct instance of a class class1 is any object whose most specific class is class1.

7.5 direct subclass: A class1 is a direct subclass of class2 if class1 is a subclass of class2, class1 is not identical to class2, and there is no other class3 which is a superclass of class1 and a subclass of class2.

7.6 direct superclass: A direct superclass of a class class1 is any class for which class1 is a direct subclass.
Programming Language EuLisp, version 0.99

7.7 dynamic environment: The inner and top dynamic environment, taken together, are referred to as the dynamic environment.

7.8 function: A function is either a continuation, a simple function or a generic function.

7.9 generic function: Generic functions are functions for which the method executed depends on the class of its arguments. A generic function is defined in terms of methods which describe the action of the generic function for a specific set of argument classes called the method’s domain.

7.10 indirect instance: An indirect instance of a class class1 is any object whose class is an indirect subclass of class1.

7.11 indirect subclass: A class2 is an indirect subclass of class1 if class2 is a subclass of class2, class1 is not identical to class2, and there is at least one other class3 which is a superclass of class1 and a subclass of class2.

7.12 inheritance graph: A directed labelled acyclic graph whose nodes are classes and whose edges are defined by the direct subclass relations between the nodes. This graph has a distinguished root, the class object, which is a superclass of every class.

7.13 inherited slot description: A slot description is inherited for a class if the slot description is defined for another class which is a direct or indirect superclass of class1.

7.14 initarg: A symbol used as a keyword in an initlist to mark the value of some slot or additional information. Used in conjunction with make and the other object initialization functions to initialize the object. An initarg may be declared for a slot in a class definition form using the initarg slot option or the initargs class option.

7.15 initform: A form which is evaluated to produce a default initial slot value. Initforms are closed in their lexical environments and the resulting closure is called an initfunction. An initform may be declared for a slot in a class definition form using the initform slot option.

7.16 initfunction: A function of no arguments whose result is used as the default value of a slot. Initfunctions capture the lexical environment of an initform declaration in a class definition form.

7.17 initlist: A list of alternating keywords and values which describes some not-yet instantiated object. Generally the keywords correspond to the initargs of some class.

7.18 inner dynamic: Inner dynamic bindings are created by dynamic-let, referenced by dynamic and modified by dynamic-setq. Inner dynamic bindings extend—and can shadow—the dynamically enclosing dynamic environment.

7.19 inner lexical: Inner lexical bindings are created by lambda and let/cc, referenced by variables and modified by setq. Inner lexical bindings extend—and can shadow—the lexically enclosing lexical environment. Note that let/cc creates an immutable binding.

7.20 instance: Every object is the instance of some class. An instance thus describes an object in relation to its class. An instance takes on the structure and behaviour described by its class. An instance can be either direct or indirect.

7.21 instantiation graph: A directed graph whose nodes are objects and whose edges are defined by the instance relations between the objects. This graph has only one cycle, an edge from class to itself. The instantiation graph is a tree and class is the root.

7.22 lexical environment: The inner and top lexical environment of a module are together referred to as the lexical environment except when it is necessary to distinguish between them.

7.23 metaclass: A metaclass is a class object whose instances are themselves classes. All metaclasses in EuLisp are instances of class, which is an instance of itself. All metaclasses are also subclasses of class. class is a metaclass.

7.24 method: A method describes the action of a generic function for a particular list of argument classes called the method’s domain. A method is thus said to add to the behaviour of each of the classes in its domain. Methods are not functions but objects which contain, among other information, a function representing the method’s behaviour.

7.25 method specificity: A domain domain1 is more specific than another domain2 if the first class in domain1 is a subclass of the first class in domain2, or, if they are the same, the rest of domain2 is more specific than the rest of domain1.

7.26 multi-method: A method which specializes on more than one argument.

7.27 new instance: A newly allocated instance, which is distinct, but can be isomorphic to other instances.

7.28 reflective: A system which can examine and modify its own state is said to be reflective. EuLisp is reflective to the extent that it has explicit class objects and metaclasses, and user-extendible operations upon them.

7.29 self-instantiated class: A class which is an instance of itself. In EuLisp, class is the only example of a self-instantiated class.

7.30 setter function: The function associated with the function that accesses a place in an entity, which changes the value stored in that place.
7.31 simple function: A function comprises at least: an expression, a set of identifiers, which occur in the expression, called the parameters and the closure of the expression with respect to the lexical environment in which it occurs, less the parameter identifiers. Note: this is not a definition of the class <simple-function>.

7.32 slot: A named component of an object which can be accessed using the slot's accessor. Each slot of an object is described by a slot description associated with the class of the object. When we refer to the structure of an object, this usually means its set of slots.

7.33 slot description: A slot description describes a slot in the instances of a class. This description includes the slot's name, its logical position in instances, and a way to determine its default value. A class's slot descriptions may be accessed through the function class-slot-descriptions. A slot description can be either direct or inherited.

7.34 slot option: A keyword and its associated value applying to one of the slots appearing in a class definition form, for example: the accessor keyword and its value, which defines a function used to read or write the value of a particular slot.

7.35 slot specification: A list of alternating keywords and values (starting with a keyword) which represents a not-yet-created slot description during class initialization.

7.36 special form: A special form is a semantic primitive of the language. In consequence, any processor (for example, a compiler or a code-walker) need be able to process only the special forms of the language and compositions of them.

7.37 specialize: A verbal form used to describe the creation of a more specific version of some entity. Normally applied to classes, slot-descriptions and methods.

7.38 specialize on: A verbal form used to describe relationship of methods and the classes specified in their domains.

7.39 subclass: The behaviour and structure defined by a class class1 are inherited by a set of classes which are termed subclasses of class1. A subclass can be either direct or indirect or itself.

7.40 superclass: A class1 is a superclass of class2 iff class2 is a subclass of class1. A superclass can be either direct or indirect or itself.

7.41 top dynamic: Top dynamic bindings are created by defvar, referenced by dynamic and modified by dynamic-setq. There is only one top dynamic environment.

7.42 top lexical: Top lexical bindings are created in the top lexical environment of a module by

defclass, defcondition, defconstant, defgeneric, defmacro, defstruct, defun.

All these bindings are immutable. deflocal creates a mutable top-lexical binding. All such bindings are referenced by variables and these made by deflocal are modified by setq. Each module defines its own distinct top lexical environment.
8 Syntax

Case is distinguished in each of characters, strings and identifiers, so that variable-name and Variable-name are different, but where a character is used in a positional number representation (e.g. A#3Ad) the case is ignored. Thus, case is also significant in this definition and, as will be observed later, all the special form and standard function names are lower case. In this section, and throughout this text, the names for individual character glyphs are those used in ISO/IEC DIS 646:1990.

The minimal character set to support EULISP is defined in Table 2. The language as defined in this text uses only the characters given in this table. Thus, left hand sides of the productions in this table define and name groups of characters which are used later in this definition: decimal digit, upper letter, lower letter, letter, other character and special character.

### Table 2 — Minimal character set

#### decimal digit

- = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

#### upper letter

- = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'

#### lower letter

- = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'

#### letter

- = upper letter | lower letter;

#### other character

- = '!' | '$' | '%' | '&' | '(' | ')' | '*' | '+' | '-' | ':' | ' '; #[1]

#### special character

- = '/' | '?' | ':.' | '5' | '9' | '@' | '#;' | '(' | ')' | '[?]

#### level 0 character

- = decimal digit | letter | other character | special character;

8.2 Objects

The syntax of the classes of objects that can be read by EULISP is defined in the section of this definition corresponding to the class as defined in Table 3. The syntax for identifiers corresponds to that for symbols.

### Table 3 — Syntax of objects for reading and writing

<table>
<thead>
<tr>
<th>object</th>
<th>character</th>
<th>float</th>
<th>integer</th>
<th>list</th>
<th>string</th>
<th>symbol</th>
<th>vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>(• A.1 •)</td>
<td>(• A.8 •)</td>
<td>(• A.11 •)</td>
<td>(• A.12 •)</td>
<td>(• A.15 •)</td>
<td>(• A.16 •)</td>
<td>(• A.18 •)</td>
</tr>
</tbody>
</table>

8.1 Whitespace and Comments

Whitespace characters are space and newline. The newline character is also used to represent end of record for configurations providing such an input model, thus a reference to newline in this definition should also be read as a reference to end of record. The only use of whitespace is to improve the legibility of programs for human readers. Whitespace separates tokens and is only significant in a string or when it occurs escaped within an identifier.

A comment is introduced by the comment-begin glyph, called semicolon (;) and continues up to, but does not include, the end of the line. Hence, a comment cannot occur in the middle of a token because of the whitespace in the form of the newline. Thus a comment is equivalent to whitespace.

**NOTE** — There is no notation in EULISP for block comments.
9 Modules

The **EULISP** module scheme has several influences: LeLisp's module system and module compiler (complice), Haskell, ML 10, MIT-Scheme's *make-environment* and 'T's locales.

All bindings of objects in **EULISP** reside in some module somewhere. Also, all programs in **EULISP** are written as one or more modules. Almost every module imports a number of other modules to make its definition meaningful. These imports have two purposes, which are separated in **EULISP**: firstly the bindings needed to process the syntax in which the module is written, and secondly the bindings needed to resolve the free variables in the module after syntax expansion. These bindings are made accessible by specifying which modules are to be imported for which purpose in a directive at the beginning of each module. The names of modules are bound in a disjoint binding environment which is only accessible via the module definition form. That is to say, modules are not first-class. The body of a module definition comprises a list of directives followed by a sequence of definitions, expressions and export forms.

The module mechanism provides abstraction and security in a form complementary to that provided by the object system. Indeed, although objects do support data abstraction, they do not support all forms of information hiding and they are usually conceptually smaller units than modules. A module defines a mapping between a set of names and either local or imported bindings of those names. Most such bindings are immutable. The exception are those bindings created by *deflocal* which may be modified by both the defining and importing modules. There are no implicit imports into a module—not even the special forms are available in a module that imports nothing. A module exports nothing by default. Mutually referential modules are not possible because a module must be defined before it can be used. Hence, the importation dependencies form a directed acyclic graph.

NOTE — The issue of mutually referential modules will be addressed in a future version of the definition of **EULISP**.

The processing of a module definition uses three environments, which are initially empty. These are the top-lexical, the external and the syntax environments of the module. The top-lexical environment comprises all the locally defined bindings and all the imported bindings. The external environment comprises all the exposed bindings—bindings from modules being exposed by this module but not necessarily imported—and all the exported bindings, which are either local or imported. Thus, the external environment might not be a subset of the top-lexical environment because, by virtue of an expose directive, it can contain bindings from modules which have not been imported. This is the environment received by any module importing this module. The syntax environment comprises all the bindings available for the syntax expansion of the module. Each binding is represented as a pair of a local-name and a module-name. It is a static error if any two instances of local-name in any one of these environments have different module-names. This is called a name clash. These environments do not all need to exist at the same time, but it is simpler for the purposes of definition to describe module processing as if they do.

9.1 Directives

The list of module directives is a sequence of keywords and forms, where the keywords indicate the interpretation of the forms. This representation allows for the addition of further keywords at other levels of the definition and also for implementation-defined keywords. For the keywords given here, there is no defined order of appearance, nor is there any restriction on the number of times that a keyword can appear. Multiple occurrences of any of the directives defined here are treated as if there is a single directive whose form is the combination of each of the occurrences. This definition describes the processing of four keywords, which are now described in detail. The syntax of all the directives is given in Table 4 and an example of their use appears in Figure 1.

9.2 Export Directive

This is denoted by the keyword *export* followed by a list of names of top-lexical bindings defined in this module and has the effect of making those bindings accessible to any module importing this module by adding them to the external environment of the module. A name clash can arise in the external environment from interaction with exposed modules.

9.3 Import Directive

The purpose of this directive is to specify the imported bindings which constitute part of the top-lexical environment of a module. These are the explicit run-time dependencies of the module. Additional run-time dependencies may arise as a result of syntax expansion. These are called implicit run-time dependencies.

The import directive is a sequence of *module-descriptors*, being module names or the filters except, only and rename, which denotes the union of all the names generated by each element of the sequence. A filter can, in turn, be applied to a sequence of module descriptors, and so the effect of different kinds of filters can be combined by nesting them. An import directive specifies either the importation of a module in its entirety or the selective importation of specified bindings from a module.

In processing import directives, every name should be thought of as a pair of a *module-name* and a *local-name*. Intuitively, a namelist of such pairs is generated by reference to the module name and then filtered by except, only and rename. In an import directive, when a namelist has been filtered, the names are regarded as being defined in the top-lexical environment of the module into which they have been imported. A name clash can arise in the top-lexical environment from interaction between different imported modules. Elements of an import directive are interpreted as follows:

*except* — Filters the names from each *module-descriptor* discarding these specified and keeping all other names. The except directive is convenient when almost all of the names exported by a module are required, since it is only necessary to name those few that are not wanted to exclude them.

*module-name* — All the exported names from *module-name*.

*only* — Filters the names from each *module-descriptor* keeping only those names specified and discarding all other names. The only directive is convenient when only a few
Diagrams and code snippets are presented in the text.
Table 4 — Module syntax

<table>
<thead>
<tr>
<th>module</th>
</tr>
</thead>
<tbody>
<tr>
<td>= (? (defmodule?, module name, module directives, (module form), ?);</td>
</tr>
<tr>
<td>module name</td>
</tr>
<tr>
<td>= identifier: (* A:16 *)</td>
</tr>
<tr>
<td>module directives</td>
</tr>
<tr>
<td>= (? (module directive), ?);</td>
</tr>
<tr>
<td>module directive</td>
</tr>
<tr>
<td>= 'export', (? (identifier), ?)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>level 0 module form</td>
</tr>
<tr>
<td>= (? 'export', (identifier), ?)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>module descriptor</td>
</tr>
<tr>
<td>= module name</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>module filter</td>
</tr>
<tr>
<td>= (? 'except', (? (identifier), ?), module descriptor, ?)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>rename pair</td>
</tr>
<tr>
<td>= (? (identifier, identifier, ?));</td>
</tr>
</tbody>
</table>

NOTE — Additional implementation-defined steps may be added here, such as compilation.

**initialization** — The top-lexical bindings of the module (created above) are initialized by evaluating the forms in the body of the module in the order they appear.

NOTE — In this sense, a module can be regarded as a generalization of the labels form of this and other Lisp dialects.

### 9.8 Module Definition

**9.8.1 defmodule**

The syntax of a module and its constituents is defined in Table 4.

**9.8.1.2 Arguments**

- **module-name**: A symbol used to name the module.
- **module-directive**: A form specifying the exported names, exposed modules, imported modules and syntax modules used by this module.
- **module-form**: A sequence of defining forms, expressions and export forms.

**9.8.1.3 Remarks**

The defmodule form defines a module named by module-name and associates the name module-name with a module object in the module binding environment.

NOTE — Intentionally, nothing is defined about any relationship between modules and files.

**9.8.1.4 Examples**

An example module definition with explanatory comments is given in Figure 1.

### 9.7 Module Processing

The following steps summarize the module definition process:

**directive processing** — This is described in detail in Section 9.1. This step creates and initializes the top-lexical, syntax and external environments.

**syntax expansion** — The body of the module is expanded according to the operators defined in the syntax environment constructed from the syntax directive.

NOTE — The semantics of syntax expansion are still under discussion and will be described fully in a future version of the EuLisp definition. In outline, however, it is intended that the mechanism should provide for hygienic expansion of forms in such a way that the programmer need have no knowledge of the expansion-time or run-time dependencies of the syntax defining module.

**static analysis** — The expanded body of the module is analyzed. Names referenced in export forms are added to the external environment. Names defined by defining forms are added to the top-lexical environment. It is a static error, if a free identifier in an expression or defining form does not have a binding in the top-lexical environment.
10 Objects

In EuLisp, every object in the system has a specific class. Classes themselves are first-class objects. In this respect EuLisp differs from statically-typed object-oriented languages such as C++ and PASCAL. The EuLisp object system is called TELOS. The facilities of the object system are split across the two levels of the definition. Level-0 supports the definition of generic functions, methods and structures. Level-1 provides the reflective system which supports the meta-object protocol (MOP), introspection, the definition of new metaclasses and the specialization of classes other than structures. Metaclasses control the structure and behaviour of their instances and the representation of their metainstances. Extensions at level-1, such as multiple inheritance, support for the change-class functionality of CLOS, and persistent objects can be supported through metaclasses. In addition, metaclasses can provide new kinds of classes with reduced power but increased efficiency; the class <structure-class> is an example. No metaclass nor any operation which could return a metaclass as a result, e.g. class-of, are accessible at level-0. That supports the clear distinction between object level and metaclass level programming required for many optimizations.

Programs written using TELOS typically involve the design of a class hierarchy, where each class represents a category of entities in the problem domain, and a protocol, which defines the operations on the objects in the problem domain.

A class defines the structure and behaviour of its instances. Structure is the information contained in the class's instances and behaviour is the way in which the instances are treated by the protocol defined for them.

The components of an object are called its slots. Each slot of an object is defined by its class.

A protocol defines the operations which can be applied to instances of a set of classes. This protocol is typically defined in terms of a set of generic functions, which are functions whose application behaviour depends on the classes of the arguments. The particular class-specific behaviour is partitioned into separate units called methods. A method is not a function itself, but is a closed expression which is a component of a generic function.

Generic functions replace the send construct found in many object-oriented languages. In contrast to sending a message to a particular object, which it must know how to handle, the method executed by a generic function is determined by all of its arguments. Methods which specialize on more than one of their arguments are called multi-methods.

Inheritance is provided through classes. Slots and methods defined for a class will also be defined for its subclasses but a subclass may specialize them. In practice, this means that an instance of a class will contain all the slots defined directly in the class as well as all of those defined in the class’s superclasses. In addition, a method specialized on a particular class will be applicable to direct and indirect instances of this class. The inheritance rules, the applicability of methods and the generic dispatch are described in detail later in this section.

Classes are defined using the defstruct (10.3.1) and defcondition (12.1.8) defining forms.

Generic functions are defined using the defgeneric defining form, which creates a named generic function in the top-lexical environment of the module in which it appears and generic-lambda, which creates an anonymous generic function. These forms are described in detail later in this section.

Methods can either be defined at the same time as the generic function, or else defined separately using the defmethod macro, which adds a new method to an existing generic function. This macro is described in detail later in this section.

10.1 System Defined Classes

The basic classes of EuLisp are elements of the object system class hierarchy, which is shown in Figure 3. Indentation indicates a subclass relationship to the class under which the line has been indented, for example, <condition> is a subclass of <object>. The names given here correspond to the bindings of names to classes as they are exported from the level-0 modules. Classes directly relevant to the object system are described in this section while others are described in corresponding sections, e.g. <condition> is described in the conditions section.

In this definition, unless otherwise specified, classes declared to be subclasses of other classes may be indirect subclasses. Classes not declared to be in a subclass relationship are disjoint. Furthermore, unless otherwise specified, all objects declared to be of a certain class may be indirect instances of that class.

10.1.1 <object> class

The root of the inheritance hierarchy. <object> defines the basic methods for initialization and external representation of objects. No initialization options are specified for <object>.
10.1.2 <structure> class

The default superclass of structure classes. All classes defined using the defstruct form are direct or indirect subclasses of <structure>. Thus, this class is specializable by user defined classes at level-0. No initoptions are specified for <structure>.

10.1.3 telos-condition condition

This is the general condition class for conditions arising from operations in the object system.

10.2 Single Inheritance

TELOS level-0 provides only single inheritance, meaning that a class can have exactly one direct superclass—but indefinitely many direct subclasses. In fact, all classes in the level-0 class inheritance tree have exactly one direct superclass except the root class <object> which has no direct superclass.

Each class has a class precedence list (CPL), a linearized list of all its superclasses, which defines the classes from which the class inherits structure and behaviour. For single inheritance classes, this list is defined recursively as follows:

a) the CPL of <object> is a list of one element containing <object> itself;

b) the CPL of any other class is a list of classes beginning with the class itself followed by the elements of the CPL of its direct superclass which is <object> by default.

The class precedence list controls system-defined protocols concerning:

a) inheritance of slot and class options when initializing a class,

b) method lookup and generic dispatch when applying a generic function.

10.3 Defining Classes

10.3.1 defstruct defining form

10.3.1.1 Syntax

The syntax for defstruct is given in Table 5.

10.3.1.2 Arguments

class-name: A symbol naming a binding to be initialized with the new structure class. The binding is immutable.

superclass-name: A symbol naming a binding of a class to be used as the direct superclass of the new structure class.

Table 5 — defstruct syntax

| defstruct form | = '(', defstruct, class name, superclass name, slot description list, (class option), ')'; |
| class name | = identifier; (+ A.16 *) |
| superclass name | = identifier; (+ A.16 *) |
| slot description list | = '(' (slot description), ');'; |
| slot description | = slot name |
| | ?(' slot name, (slot option), ')'; |
| slot name | = identifier; (+ A.16 *) |
| slot option | = 'initarg' identifier (+ A.16 *) |
| | 'initform' level 0 expression (+ 13 *) |
| | 'reader' identifier (+ A.16 *) |
| | 'writer' identifier (+ A.16 *) |
| | 'accessor' identifier; (+ A.16 *) |
| class option | = 'initargs', '(', (identifier), ')'; |
| | 'constructor', '(', (identifier, (initarg name), ')'); |
| | 'predicate' identifier; (+ A.16 *) |

(slot-spec*): A list of slot specifications (see below), comprising either a slot-name or a list of slot-name followed by some slot-options.

class-option*: A sequence of keys and values (see below) which, taken together, apply to the class as a whole.

10.3.1.3 Remarks

defstruct creates a new structure class. Structure classes support single inheritance as described above. Neither class redefinition nor changing the class of an instance is supported by structure classes.1)

The slot-options are interpreted as follows:

initarg identifier: The value of this option is an identifier naming a symbol, which is the name of an argument to be supplied in the init-options of a call to make on the new class. The value of this argument in the call to make is the initial value of the slot. This option must only be specified once for a particular slot. The same initarg name may be used for several slots, in which case they will share the same initial value if the initarg is given to make. Subclasses inherit the initarg. Each slot must have at most one initarg including the inherited one. That means, a subclass can not shadow or add a new initarg, if a superclass has already defined one.

initform form: The value of this option is a form, which is evaluated as the default value of the slot, to be used if no initarg is defined for the slot or given to a call to make. The form is evaluated in the lexical environment of the call to defstruct and the dynamic environment of the call to

1)In combination with the guarantee that the behaviour of generic functions cannot be modified once it has been defined, due to no support for method removal nor method combination, this implies level-0 programs with static semantics.
make. The form is evaluated each time make is called and
the default value is called for. The order of evaluation of
the initforms in all the slots is determined by initialize.
This option must only be specified once for a particular
slot. Subclasses inherit the initform. However, a more
specific form may be specified in a subclass, which will
shadow the inherited one.

reader identifier: The value is the identifier of the vari-
able to which the reader function will be bound. The
binding is immutable. The writer function is a means to
access the slot. The reader function is a function of one
argument, which should be an instance of the new class.
No writer function is automatically bound with this op-
tion. This option can be specified more than once for a
slot, creating several bindings for the same reader func-
tion. It is a static error to specify the same reader, writer,
or accessor name for two different slots.

writer identifier: The value is the identifier of the vari-
able to which the writer function will be bound. The bind-
ing is immutable. The writer function is a means to change
the slot value. The creation of the writer is analogous to
that of the reader function. The writer function is a func-
tion of two arguments, the first should be an instance of
the new class and the second can be any new value for the
slot. This option can be specified more than once for a
slot. It is a static error to specify the same reader, writer,
or accessor name for two different slots.

accessor identifier: The value is the identifier of the vari-
able to which the reader function will be bound. In ad-
dition, the use of this slot-option causes the writer function
to be associated to the reader via the setter mechanism.
This option can be specified more than once for a slot.
It is a static error to specify the same reader, writer, or
accessor name for two different slots.

The class options are interpreted as follows:

initargs list: The value of this option is a list of ident-
fiers naming symbols, which extend the inherited names
of arguments to be supplied in the call to make on the new class. Initargs are inherited by union.
The values of all legal arguments in the call to make are
the initial values of corresponding slots if they name a slot
initarg or are ignored by the default initialize method,
otherwise. This option must only be specified once for a
class.

constructor constructor-spec: Creates a constructor
function for the new class. The constructor specification
gives the name to which the constructor function will be
bound, followed by a sequence of legal initargs for the class.
The new function creates an instance of the class and fills
the slots according to the match between the specified
initargs and the given arguments to the constructor func-
tion. This option may be specified any number of times
for a class.

predicate identifier: Creates a function which tests
whether an object is an instance of the new class. The
predicate specification gives the name to which the predi-
cate function will be bound. This option may be specified
any number of times for a class.

10.4 Defining Generic Functions and Methods

10.4.1 defgeneric defining form

<table>
<thead>
<tr>
<th>defgeneric form</th>
</tr>
</thead>
<tbody>
<tr>
<td>= '(defgeneric, gf name, gf lambda list, (level 0 init option), ?)</td>
</tr>
<tr>
<td>gf name</td>
</tr>
<tr>
<td>= identifier: (A.16)</td>
</tr>
<tr>
<td>gf lambda list</td>
</tr>
<tr>
<td>= specialized lambda list:</td>
</tr>
<tr>
<td>level 0 init option</td>
</tr>
<tr>
<td>= method?, method description:</td>
</tr>
<tr>
<td>method description</td>
</tr>
<tr>
<td>= '(specialized lambda list, {form}, ?)</td>
</tr>
<tr>
<td>specialized lambda list</td>
</tr>
<tr>
<td>= '(specialized parameter, specialized parameter,</td>
</tr>
<tr>
<td>()), identifier, ?)</td>
</tr>
<tr>
<td>specialized parameter</td>
</tr>
<tr>
<td>= '(identifier, class name, ?)</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

10.4.2 Arguments

gf-name: One of a symbol, or a form denoting a setter
function or a converter function.
gen-lambda-list: The parameter list of the generic func-
tion, which may be specialized to restrict the domain of
methods to be attached to the generic function.
level-0-init-option*: Options as specified below.

10.4.3 Remarks

This defining form defines a new generic function. The result-
ging generic function will be bound to gf-name. The sec-
ond argument is the formal parameter list. The method's
specialized lambda list must be congruent to that of the
generic function. Two lambda lists are said to be congru-
ent iff:
a) both have the same number of formal parameters, and
b) if one lambda list has a rest formal parameter then the
other lambda list has a rest formal parameter too, and vice
versa.

An error is signalled (condition class: <non-congruent-lambda-lists>) if any method defined
on this generic function does not have a lambda list
congruent to that of the generic function.

An error is signalled (condition class: < incompatible-method-domain>) if the method's specialized lambda list widens the domain
of the generic function. In other words, the lambda lists of all
methods must specialize on subclasses of the classes in the
lambda list of the generic function.

An error is signalled (condition class: <method-domain-clash>) if any methods defined on
this generic function have the same domain. These conditions
tions apply both to methods defined at the same time as the
gen function and to any methods added subsequently by
defmethod. An init-option is an identifier followed by a cor-
responding value.

An error is signalled (condition class: <non-applicable-method>) if an attempt is made to apply
gen function which has no applicable methods for the
classes of the arguments supplied.

The init-option is:

method method-spec: This option is followed by a
method description. A method description is a list com-
prising the specialized lambda list of the method, which
denotes the domain, and a sequence of forms, denoting
the method body. The method body is closed in the lex-
ical environment in which the gen function definition
appears. This option may be specified more than once.

The rewrite rules for defgeneric are given in Table 6.

### 10.4.1.4 Examples

In the following example of the use of defgeneric a gen-
function named gf-0 is defined with three methods attached
to it. The domain of gf-0 is constrained to be <object>
× <class-a>. In consequence, each method added to the
gen function, both here and later (by defmethod), must
have a domain which is a subclass of <object> × <class-a>,
which is to say that <class-c>, <class-e> and <class-g>
must all be subclasses of <class-a>.

(defgeneric gf-0 (arg1 (arg2 <class-a>))
  method (((m1-arg1 <class-b>) (m1-arg2 <class-c>)) ...)
  method (((m2-arg1 <class-d>) (m2-arg2 <class-e>)) ...)
  method (((m3-arg1 <class-f>) (m3-arg2 <class-g>)) ...))

### 10.4.1.5 See also: defmethod, generic-lambda.

<table>
<thead>
<tr>
<th>Table 6 — defgeneric rewrite rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>(defgeneric identifier gf lambda list</td>
</tr>
<tr>
<td>(level 0 init option)) ==</td>
</tr>
<tr>
<td>(defconstant identifier</td>
</tr>
<tr>
<td>(generic-lambda gf lambda list (level 0 init option)))</td>
</tr>
<tr>
<td>(defgeneric (setter identifier) gf lambda list</td>
</tr>
<tr>
<td>(level 0 init option)) ==</td>
</tr>
<tr>
<td>((setter setter) identifier</td>
</tr>
<tr>
<td>(generic-lambda gf lambda list (level 0 init option)))</td>
</tr>
<tr>
<td>(defgeneric (converter identifier) gf lambda list</td>
</tr>
<tr>
<td>(level 0 init option)) ==</td>
</tr>
<tr>
<td>((converter converter) identifier</td>
</tr>
<tr>
<td>(generic-lambda gf lambda list level 0 init option)))</td>
</tr>
</tbody>
</table>

### 10.4.2 defmethod

macro

### 10.4.2.1 Syntax

defmethod form
  = '(?, 'defmethod', gf locator, |
    specialized lambda list, |
    {form}, ?);

### 10.4.2.2 Remarks

This macro is used for defining new methods on generic
functions. A new method object is defined with the spec-
ified body and with the domain given by the specialized
lambda list. This method is added to the generic func-
tion bound to gf-name, which is an identifier, or a form
denoting a setter function or a converter function. If
the specialized lambda-list is not congruent with that of
the generic function, an error is signalled (condition class:
<non-congruent-lambda-lists>). An error is signalled
(condition class: <incompatible-method-domain>) if the
method's specialized lambda list would widen the domain
of the generic function. If there is a method with the same
domain already defined on this generic function, an error is
signalled (condition class: <method-domain-clash>).

### 10.4.3 generic-lambda

macro

### 10.4.3.1 Syntax

generic lambda form
  = '(?, 'generic-lambda', gf lambda list, |
    {level 0 init option}, ?));

### 10.4.3.2 Remarks

generic-lambda creates and returns an anonymous generic function
that can be applied immediately, much like the nor-
mal lambda. The gen-lambda-list and the init-options are
interpreted exactly as for the level-0 definition of defgeneric.

### 10.4.3.3 Examples

In the following example an anonymous version of gf-0 (see
defgeneric above) is defined. In all other respects the re-
sulting object is the same as gf-0.

generic-lambda ((arg1 <object>) (arg2 <class-a>))
  method (((m1-arg1 <class-b>) (m1-arg2 <class-c>)) ...)
  method (((m2-arg1 <class-d>) (m2-arg2 <class-e>)) ...)
  method (((m3-arg1 <class-f>) (m3-arg2 <class-g>)) ...)

10.4.4 defgenerate

macro

10.4.4.1 Syntax

defgenerate form
  = '(?, 'defgenerate', gf lambda list, |
    {form}, ?);

10.4.4.2 Remarks

This macro is used to generate class-level definitions. It
is provided for backwards compatibility. It does not
accept init-options and it is an error to use.

10.4.4.3 Examples

In the following example anonymous version of generate
above is defined. In all other respects the re-
sulting object is the same as generate.
method (((m3-arg1 <class-f>) (m3-arg2 <class-g>)) ...)

10.4.3.4 See also defgeneric.

10.4.4 no-applicable-method telos-condition

10.4.4.1 Initialization Options

generic function: The generic function which was applied.

arguments list: The arguments of the generic function which was applied.

10.4.4.2 Remarks

Signalled by a generic function when there is no method which is applicable to the arguments.

10.4.5 incompatible-method-domain telos-condition

10.4.5.1 Initialization Options

generic function: The generic function to be extended.

method method: The method to be added.

10.4.5.2 Remarks

Signalled by one of defgeneric, defmethod or generic-lambda if the domain of the method would not be a subdomain of the generic function’s domain.

10.4.6 non-congruent-lambda-lists telos-condition

10.4.6.1 Initialization Options

generic function: The generic function to be extended.

method method: The method to be added.

10.4.6.2 Remarks

Signalled by one of defgeneric, defmethod or generic-lambda if the lambda list of the method is not congruent to that of the generic function.

10.4.7 method-domain-clash telos-condition

10.4.7.1 Initialization Options

generic function: The generic function to be extended.

methods list: The methods with the same domain.

10.4.7.2 Remarks

Signalled by one of defgeneric, defmethod or generic-lambda if there would be methods with the same domain attached to the generic function.

10.5 Specializing Methods

The following two operators are used to specialize more general methods. The more specialized method can do some additional computation before calling these operators and can then carry out further computation before returning. It is an error to use either of these operators outside a method body. Argument bindings inside methods are immutable. Therefore an argument inside a method retains its specialized class throughout the processing of the method.

10.5.1 call-next-method special form

10.5.1.1 Syntax

(call-next-method)

10.5.1.2 Result

The result of calling the next most specific applicable method.

10.5.1.3 Remarks

The next most specific applicable method is called with the same arguments as the current method. An error is signalled (condition class: <no-next-method>) if there is no next most specific method.

10.5.2 next-method-p special form

10.5.2.1 Syntax

(next-method-p)

10.5.2.2 Result

If there is a next most specific method, next-method-p returns a non-() value, otherwise, it returns ().

10.5.3 no-next-method telos-condition

10.5.3.1 Initialization Options

method method: The method which called call-next-method.

operand-list list: A list of the arguments to have been passed to the next method.

10.5.3.2 Remarks

Signalled by call-next-method if there is no next most specific method.
10.6 Method Lookup and Generic Dispatch

The system defined method lookup and generic function dispatch is purely class based. eql methods known from CLOS are excluded.

The application behaviour of a generic function can be described in terms of method lookup and generic dispatch. The method lookup determines

a) which methods attached to the generic function are applicable to the supplied arguments, and

b) the linear order of the applicable methods with respect to classes of the arguments and the argument precedence order.

A class \(C_1\) is called "more specific than class \(C_2\) with respect to \(C_3\) if \(C_1\) appears before \(C_2\) in the class precedence list (CPL) of \(C_3\)\(^1\).

Two additional concepts are needed to explain the processes of method lookup and generic dispatch: (i) whether a method is applicable, (ii) how specific it is in relation to the other applicable methods. The definitions of each of these terms is now given.

A method with the domain \(D_1 \times \ldots \times D_m \times \{<\text{list}>\}\) is applicable to the arguments \(a_1 \ldots a_m [a_{m+1} \ldots a_n]\) if the class of each argument, \(C_i\), is a subclass of \(D_i\), which is to say, \(D_i\) is a member of \(C_i\)'s class precedence list.

A method \(M_i\) with the domain \(D_{i1} \times \ldots \times D_{im} \times \{<\text{list}>\}\) is more specific than a method \(M_j\) with the domain \(D_{j1} \times \ldots \times D_{jm} \times \{<\text{list}>\}\) with respect to the arguments \(a_1 \ldots a_m [a_{m+1} \ldots a_n]\) iff there exists an \(i \in \{1 \ldots m\}\) so that \(D_{ji}\) is more specific than \(D_{ji}\) with respect to \(C_i\), the class of \(a_i\), and for all \(j = 1 \ldots i - 1, D_{ji}\) is not more specific than \(D_{ji}\) with respect to \(C_i\), the class of \(a_i\).

Now, with the above definitions, we can describe the application behaviour of a generic function (if \(a_1 \ldots a_m [a_{m+1} \ldots a_n]\)):

a) Select the methods applicable to \(a_1 \ldots a_m [a_{m+1} \ldots a_n]\) from all methods attached to \(f\).

b) Sort the applicable methods \(M_1 \ldots M_n\) into decreasing order of specificity using left to right argument precedence order to resolve otherwise equally specific methods.

c) If call-next-method appears in one of the method bodies, make the sorted list of applicable methods available for it.

d) Apply the most specific method on \(a_1 \ldots a_m [a_{m+1} \ldots a_n]\).

e) Return the result of the previous step.

The first two steps are usually called "method lookup" and the first four are usually called "generic dispatch.

10.7 Creating and Initializing Objects

Objects can be created by calling

- constructors (predefined or user defined) or
- make, the general constructor function or
- allocate, the general allocator function.

10.7.1 make

function

10.7.1.1 Arguments

\textit{class}: The class of the object to create.

\textit{key}: obj ... key_n obj_n: Initialization arguments.

10.7.1.2 Result

An instance of \textit{class}.

10.7.1.3 Remarks

The general constructor \textit{make} creates a new object calling allocate and initialises it by calling \textit{initialize}. \textit{make} returns whatever allocate returns as its result.

10.7.2 allocate

function

10.7.2.1 Arguments

\textit{class}: A structure class.

\textit{initlist}: A list of initialization arguments.

10.7.2.2 Result

A new uninitialized direct instance of the first argument.

10.7.2.3 Remarks

The \textit{class} must be a structure class, the \textit{initlist} is ignored. The behaviour of allocate is extended at level-1 for classes not accessible at level-0. The level-0 behaviour is not affected by the level-1 extension.

10.7.3 initialize

generic function

10.7.3.1 Generic Arguments

\textit{object <objects>}: The object to initialize.

\textit{initlist}: The list of initialization arguments.

10.7.3.2 Result

The initialized object.
10.7.3.3 Remarks
Initializes an object and returns the initialized object as the result. It is called by \texttt{make} on a new uninitialized object created by calling \texttt{allocate}.

Users may extend initialize by defining methods specializing on newly defined classes, which are structure classes at level-0.

10.7.4 initialize method

10.7.4.1 Specialized Arguments

\texttt{(object object)}: The object to initialize.

\texttt{initlist}: The list of initialization arguments.

10.7.4.2 Result

The initialized object.

10.7.4.3 Remarks

This is the default method attached to initialize. This method performs the following steps:

a) Checks if the supplied initargs are legal and signals an error otherwise. Legal initargs are those specified in the class definition directly or inherited from a superclass. An initarg may be specified as a slot option or as a class option.

b) Initializes the slots of the object according to the initarg, if supplied, or according to the most specific \texttt{initform}, if specified. Otherwise, the slot remains “unbound”.

Legal initargs which do not initialize a slot are ignored by the default \texttt{initialize} method. More specific methods may handle these initargs and call the default method by calling \texttt{call-next-method}.

10.8 Accessing Slots

Object components (slots) can be accessed using reader and writer functions (accessors) only. For system defined object classes there are predefined readers and writers. Some of the writers are accessible using the \texttt{setter} function. If there is no writer for a slot, its value cannot be changed. When users define new classes, they can specify which readers and writers should be accessible in a module and by which binding. Accessor bindings are not exported automatically when a class (binding) is exported. They can only be exported explicitly.

11 Concurrency

The basic elements of parallel processing in EuLisp are processes and mutual exclusion, which are provided by the classes \texttt{<thread>} and \texttt{<lock>} respectively.

A thread is allocated and initialized, by calling \texttt{make}. The initarg of a thread specifies the initial function, which is where execution starts the first time the thread is dispatched by the scheduler. In this discussion four states of a thread are identified: \texttt{new}, \texttt{running}, \texttt{aborted} and \texttt{finished}. These are for conceptual purposes only and a EuLisp program cannot distinguish between new and running or between aborted and finished. (Although accessing the result of a thread would permit such a distinction retrospectively, since an aborted thread will cause a condition to be signalled on the accessing thread and a finished thread will not.) In practice, the running state is likely to have several internal states, but these distinctions and the information about a thread's current state can serve no useful purpose to a running program, since the information may be incorrect as soon as it is known. The initial state of a thread is new. The union of the two final states is known as \texttt{determined}. Although a program can find out whether a thread is determined or not by means of \texttt{wait} with a timeout of \texttt{t} (denoting a poll), the information is only useful if the thread has been determined.

A thread is made available for dispatch by starting it, using the function \texttt{thread-start}, which changes its state from new to running. After running a thread becomes either finished or aborted. When a thread is finished, the result of the initial function may be accessed using \texttt{thread-value}. If a thread is aborted, which can only occur as a result of a signal handled by the default handler (installed when the thread is created), then \texttt{thread-value} will signal the condition that aborted the thread on the thread accessing the value. Note that \texttt{thread-value} suspends the calling thread if the thread whose result is sought is not determined.

While a thread is running, its progress can be suspended by accessing a lock, by a stream operation or by calling \texttt{thread-value} on an undetermined thread. In each of these cases, \texttt{thread-reschedule} is called to allow another thread to execute. This function may also be called voluntarily. Progress can resume when the lock becomes unlocked, the input/output operation completes or the undetermined thread becomes determined.

The actions of a thread can be influenced externally by signal. This function registers a condition to be signalled no later than when the specified thread is rescheduled for execution—when \texttt{thread-reschedule} returns. The condition must be an instance of \texttt{thread-condition}. Conditions are delivered to the thread in order of receipt. This ordering requirement is only important in the case of a thread sending more than one signal to the same thread, but in other circumstances the delivery order cannot be verified. A signal on a determined thread has no discernable effect on either the signalled or signalling thread unless the condition is not an instance of \texttt{<thread-condition>}, in which case an error is signalled on the signalling thread. See also Section 12.

A lock is an abstract data type protecting a binary value which denotes whether the lock is locked or unlocked. The operations on a lock are \texttt{lock} and \texttt{unlock}. Executing a lock operation will eventually give the calling thread exclusive control of a lock. The unlock operation unlocks the lock so that either a thread subsequently calling \texttt{lock} or one of the
threads which has already called `lock` on the lock can gain exclusive access.

NOTE — It is intended that implementations of locks based on spin-locks, semaphores or channels should all be capable of satisfying the above description. However, to be a conforming implementation, the use of a spin-lock must observe the fairness requirement, which demands that between attempts to acquire the lock, control must be ceded to the scheduler.

The programming model is that of concurrently executing threads, regardless of whether the configuration is a multiprocessor or not, with some constraints and some weak fairness guarantees.

a) A processor is free to use run-to-completion, timeslicing and/or concurrent execution.

b) A conforming program must assume the possibility of concurrent execution of threads and will have the same semantics in all cases — see discussion of fairness which follows.

c) The default condition handler for a new thread, when invoked, will change the state of the thread to aborted, save the signalled condition and reschedule the thread.

d) A continuation must only be called from within its dynamic extent. This does not include threads created within the dynamic extent. An error is signalled (condition class: `<wrong-thread-continuation>`), if a continuation is called on a thread other than the one on which it was created.

e) The lexical environment (inner and top) associated with the initial function may be shared, as is the top-dynamic environment, but each thread has a distinct inner-dynamic environment. In consequence, any modifications of bindings in the lexical environment or in the top-dynamic environment should be mediated by locks to avoid non-deterministic behaviour.

f) The creation and starting of a thread represent changes to the state of the processor and as such are not affected by the processor’s handling of errors signalled subsequently on the creating/starting thread (cf. streams). That is to say, a non-local exit to a point dynamically outside the creation of the subsidiary thread has no default effect on the subsidiary thread.

g) The behaviour of `io` on the same stream by multiple threads is undefined unless it is mediated by explicit locks.

The parallel semantics are preserved on a sequential run-to-completion implementation by requiring communication between threads to use only thread primitives and shared data protected by locks — both the thread primitives and locks will cause rescheduling, so other threads can be assumed to have a chance of execution.

There is no guarantee about which thread is selected next. However, a fairness guarantee is needed to provide the illusion that every other thread is running. A strong guarantee would ensure that every other thread gets scheduled before a thread which reschedules itself is scheduled again. Such a scheme is usually called “round-robin”. This could be stronger than the guarantee provided by a parallel implementation or the scheduler of the host operating system and cannot be mandated in this definition.

A weak but sufficient guarantee is that if any thread reschedules infinitely often then every other thread will be scheduled infinitely often. Hence if a thread is waiting for shared data to be changed by another thread and is using a lock, the other thread is guaranteed to have the opportunity to change the data. If it is not using a lock, the fairness guarantee ensures that in the same scenario the following loop will exit eventually:

```lisp
(while (< data 0) (thread-reschedule))
```

### 11.1 Threads

The defined name of this module is `thread`. This section defines the operations on threads.

11.1.1 `<thread>`  

The class of all instances of `<thread>`.

11.1.1.1 Initialization Options

- `init-function`: an instance of `<function>` which will be called when the resulting thread is started by `thread-start`.

11.1.2 `threadp`  

Function

11.1.2.1 Arguments

- `object`: An object to examine.

11.1.2.2 Result

The supplied argument if it is an instance of `<thread>`, otherwise `()`.

11.1.3 `thread-reschedule`  

Function

This function takes no arguments.

11.1.3.1 Result

The result is `()`.  

11.1.3.2 Remarks

This function is called for side-effect only and may cause the thread which calls it to be suspended, while other threads are run. In addition, if the thread’s condition queue is not empty, the first condition is removed from the queue and signalled on the thread. The resume continuation of the signal will be one which will eventually call the continuation of the call to `thread-reschedule`.

11.1.3.3 See also: `thread-value`, `signal` and Section 12 for details of conditions and signalling.
11.1.4 current-thread function

This function takes no arguments.

11.1.4.1 Result
The thread on which current-thread was executed.

11.1.5 thread-start function

11.1.5.1 Arguments

thread: the thread to be started, which must be new. If thread is not new, an error is signalled (condition class: <thread-already-started>).

obj ... objn: values to be passed as the arguments to the initial function of thread.

11.1.5.2 Result
The thread which was supplied as the first argument.

11.1.5.3 Remarks
The state of thread is changed to running. The values obj to objn will be passed as arguments to the initial function of thread.

11.1.6 thread-value function

11.1.6.1 Arguments

thread: the thread whose finished value is to be accessed.

11.1.6.2 Result
The result of the initial function applied to the arguments passed from thread-start. However, if a condition is signalled on thread which is handled by the default handler the condition will now be signalled on the thread calling thread-value—that is the condition will be propagated to the accessing thread.

11.1.6.3 Remarks
If thread is not determined, each thread calling thread-value is suspended until thread is determined, when each will either get the thread's value or signal the condition.

11.1.6.4 See also: thread-reschedule, signal.

11.1.7 wait method

11.1.7.1 Specialized Arguments

(thread <thread>): The thread on which to wait.

(timeout <object>): The timeout period which is specified by one of (), t, and non-negative integer.

11.1.7.2 Result
Result is either thread or (). If timeout is (), the result is thread if it is determined. If timeout is t, thread suspends until thread is determined and the result is guaranteed to be thread. If timeout is a non-negative integer, the call blocks until either thread is determined, in which case the result is thread, or until the timeout period is reached, in which case the result is (), whichever is the sooner. The units for the non-negative integer timeout are the number of clock ticks to wait. The implementation-defined constant ticks-per-second is used to make timeout periods processor independent.

11.1.7.3 See also: wait and ticks-per-second (Section 12).

11.1.8 thread-condition condition

11.1.8.1 Initialization Options

current-thread thread: The thread which is signalling the condition.

11.1.8.2 Remarks
This is the general condition class for all conditions arising from thread operations.

11.1.9 wrong-thread-continuation thread-condition

11.1.9.1 Initialization Options

continuation continuation: A continuation.

thread thread: The thread on which continuation was created.

11.1.9.2 Remarks
Signalled if the given continuation is called on a thread other than the one on which it was created.

11.1.10 thread-already-started thread-condition

11.1.10.1 Initialization Options

thread thread: A thread.

11.1.10.2 Remarks
Signalled by thread-start if the given thread has been started already.

11.2 Locks

The defined name of this module is lock.
11.2.1 <lock>  

The class of all instances of <lock>. This class has no init- 
options. The result of calling make on <lock> is a new, open  
lock.

11.2.2 lockp  

function

11.2.2.1 Arguments

object: An object to examine.

11.2.2.2 Result

The supplied argument if it is an instance of lock, otherwise 0.

11.2.3 lock  

function

11.2.3.1 Arguments

lock: the lock to be acquired.

11.2.3.2 Result

The lock supplied as argument.

11.2.3.3 Remarks

Executing a lock operation will eventually give the calling  
thread exclusive control of lock. A consequence of calling  
lock is that a condition from another thread may be sign- 
alled on this thread. Such a condition will be signalled  
before lock has been acquired, so a thread which does not  
handle the condition will not lead to starvation; the con- 
dition will be signalled continually so that the process of  
aquireing the lock may continue after the condition has  
been handled.

11.2.3.4 See also: unlock and Section 12 for details of con- 
ditions and signalling.

11.2.4 unlock  

function

11.2.4.1 Arguments

lock: the lock to be released.

11.2.4.2 Result

The lock supplied as argument.

11.2.4.3 Remarks

The unlock operation unlocks lock so that either a thread  
subsequently calling lock or one of the threads which has  
already called lock on the lock can gain exclusive access.

11.2.4.4 See also: lock.

12 Conditions

The defined name of this module is condition.

The condition system was influenced by the Common Lisp  
error system,13 and the Standard ML exception mechanism.  
It is a simplification of the former and an extension of the  
latter. Following standard practice, this text defines the ac-
tions of functions in terms of their normal function. Where an  
exceptional behaviour might arise, this has been defined in  
terms of a condition. However, not all exceptional situations  
are errors. Following Pitman, we use condition to be a kind  
of occasion in a program when an exceptional situation has  
been signalled. An error is a kind of condition—error and  
condition are also used as terms for the objects that repre-
sent exceptional situations. A condition can be signalled  
continually by passing a continuation for the resumption to  
signal. If a continuation is not supplied then the condition  
cannot be continued.

These two categories are characterized as follows:

a) A condition might be signalled when some limit has  
been transgressed and some corrective action is needed  
before processing can resume. For example, memory zone  
exhaustion on attempting to heap-allocate an item can  
be corrected by calling the memory management scheme  
to recover dead space. However, if no space was recov-
ered a new kind of condition has arisen. Another example  
arises in the use of IEEE floating point arithmetic, where  
a condition might be signalled to indicate divergence of  
an operation. A condition should be signalled continually  
when there is a strategy for recovery from the condition.

b) Alternatively, a condition might be signalled when  
some catastrophic situation is recognized, such as the  
memory manager being unable to allocate more memory or  
unable to recover sufficient memory from that already al- 
located. A condition should be signalled non-continually  
when there is no reasonable way to resume processing.

A condition class is defined using defcondition (see Sec- 
ton 12.1.8). The definition of a condition causes the creation  
of a new class of condition. A condition is signalled using  
the function signal, which has two required arguments and  
one optional argument: an instance of a condition, a re- 
sume continuation or the empty list—the latter signifying a  
non-continuable signal—and a thread. A condition can be  
handled using the special form with-handler, which takes a  
function—the handler function—and a sequence of forms to  
be protected. The initial condition class hierarchy is shown  
in Figure 4.

12.1 Condition Classes

12.1.1 <condition>  

class

12.1.1.1 Initialization Options

message <string>: A string, containing information  
which should pertain to the situation which caused the  
condition to be signalled.

12.1.1.2 Remarks

The class which is the superclass of all condition classes.
12.1.6.2 Result

Returns object if it is a condition, otherwise ()

12.1.7 initialize

method

12.1.7.1 Specialized Arguments

(condition <condition>): a condition.

init-list: A list of initialization options as follows:

message string: A string, containing information which should pertain to the situation which caused the condition to be signalled.

12.1.7.2 Result

A new, initialized condition.

12.1.7.3 Remarks

First calls call-next-method to carry out initialization specified by superclasses then does the <condition> specific initialization. The following init-option is recognized by this method:

message string: A string which should contain information about the condition that has arisen.

12.1.8 defcondition

defining form

12.1.8.1 Syntax

(defcondition condition-class-name superclass-name init-option")

12.1.8.2 Arguments

condition-class-name: A symbol naming a binding to be initialized with the new condition class.

superclass-name: A symbol naming a binding of a class to be used as the superclass of the new condition class.

init-option": A sequence of symbols and expressions to be passed to then generic functions allocate and initialize.

12.1.8.3 Remarks

This defining form defines a new condition class. The first argument is the name to which the new condition class will be bound. The second is the name of the superclass of the new condition class and an init-option is an identifier followed by its (default) initial value. If superclass-name is (), the superclass is taken to be <condition>. Otherwise superclass-name must be <condition> or the name of one of its subclasses.
12.2 Condition Handling

Conditions are handled with a function called a handler. Handlers are established dynamically and have dynamic scope and extent. Thus, when a condition is signalled, the processor will call the dynamically closest handler. This can accept, resume or decline the condition (see with-handler for a full discussion and definition of this terminology). If it declines, then the next dynamically closest handler is called, and so on, until a handler accepts or resumes the condition. It is the first handler accepting the condition that is used and this may not necessarily be the most specific. Handlers are established by the special form with-handler.

12.2.1 signal

<table>
<thead>
<tr>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition: The condition to be signalled.</td>
</tr>
<tr>
<td>function: The function to be called if the condition is handled and resumed, that is to say, the condition is continuant, or () otherwise.</td>
</tr>
<tr>
<td>[thread]: If this argument is not supplied, the condition is signalled on the thread which called signal, otherwise, thread indicates the thread on which condition is to be signalled.</td>
</tr>
</tbody>
</table>

12.2.1.2 Result

signal should never return. It is an error to call signal's continuation.

12.2.1.3 Remarks

Called to indicate that a specified condition has arisen during the execution of a program.

If the third argument is not supplied, signal calls the dynamically closest handler with condition and continuation. If the second argument is a subclass of function, it is the resume continuation to be used in the case of a handler deciding to resume from a continuant condition. If the second argument is (), it indicates that the condition was signalled as a non-continuant condition—in this way the handler is informed of the signaler’s intention.

If the third argument is supplied, signal registers the specified condition to be signalled on thread. The condition must be an instance of the condition class <thread-condition>, otherwise an error is signalled (condition class: <wrong-condition-class>) on the thread calling signal. A signal on a determined thread has no effect on either the signalled or signalling thread except in the case of the above error.

12.2.1.4 See also: thread-reschedule, thread-value, with-handler.

12.2.2 wrong-condition-class

<table>
<thead>
<tr>
<th>thread-condition</th>
</tr>
</thead>
</table>

12.2.2.1 Initialization Options

condition condition: A condition.

Signalled by signal if the given condition is not an instance of the condition class <thread-condition>.

| special form |

12.2.3 with-handler

12.2.3.1 Syntax

(with-handler handler-function protected-form)

12.2.3.2 Arguments

handler-function: A function or a generic function which will be called if a condition is signalled during the dynamic extent of protected-forms. A handler function takes two arguments—a condition, and a resume function/continuation. The condition is the condition object that was passed to signal as its first argument. The resume continuation is the continuation (or ()) that was given to signal as its second argument.

protected-form* : The sequence of forms whose execution is protected by the handler-function specified above.

12.2.3.3 Result

The value of the last form in the sequence of protected-forms.

12.2.3.4 Remarks

A with-handler form is evaluated in four steps:

a) The new handler-function is constructed and identifies the dynamically closest handler.

b) The dynamically closest handler is shadowed by the establishment of the new handler-function.

c) The sequence of protected-forms is evaluated in order and the value of the last one is returned as the result of the with-handler expression.

d) the handler-function is disestablished, and the previous handler is no longer shadowed.

The above is the normal behaviour of with-handler. The exceptional behaviour of with-handler happens when there is a call to signal during the evaluation of protected-form. signal calls the dynamically closest handler-function passing on the first two arguments given to signal. The handler-function is executed in the dynamic extent of the call to signal. However, any signals occurring during the execution of handler-function are dealt with by the dynamically closest handler outside the extent of the form which established handler-function. A handler-function takes one of three actions:

a) Return. This causes the next-closest enclosing handler-function to be called, passing on the condition and the resume continuation. This is termed declining the condition. The situation when there is no next closest enclosing handler is discussed later.

b) Call the resume continuation. This action might be taken if the condition is recognised by the handler function
Programming Language EuLisp, version 0.99

(1st/cc accept
 (with-handler
   (generic-lambda
    ((condition (condition)) (resume (function)))
    method
    ((<condition>) resume)
    (cond
     (:seriousp c)
     ;; serious error, exit from with-handler (accept)
     (accept))
     (:fixablep c)
     ;; fixable error (resume)
     (resume (fix c)))
     (t
      ;; otherwise, by omission, let another handler deal
      ;; with it (decline)
      ()))
    ;; the protected expression
    (something-which-might-signal-an-error)))

Figure 5 — Illustration of handler actions

and might be preceded by some corrective action. This is termed resuming the condition.

c) Not return and not call the resume continuation. This action might be taken if the condition is recognized by the handler function and might be preceded by some corrective action before some kind of transfer of control. This is termed accepting the condition.

It is an error if the condition is declined and there is no next closest enclosing handler. In this circumstance the identified error is delivered to the configuration to be dealt with in an implementation-defined way. Errors arising in the dynamic extent of the handler function are signalled in the dynamic extent of the original signal but are handled in the enclosing dynamic extent of the handler.

12.2.5 error function

12.2.5.1 Arguments

error-message: a string containing relevant information.
condition-class: the class of condition to be signalled.
init-option*: a sequence of options to be passed to initialize-instance when making the instance of condition.

12.2.5.2 Result
The result is ()

12.2.5.3 Remarks
The error function signals a non-continuable error. It calls signal with an instance of a condition of condition-class initialized from init-options, the error-message and a resume continuation value of (), signifying that the condition was not signalled continuably.

12.2.4 error function

12.2.4.1 Arguments

error-message: a string containing relevant information.
condition-class: the class of condition to be signalled.
init-option*: a sequence of options to be passed to initialize-instance when making the instance of condition.

12.2.4.2 Result
The result is ()

12.2.4.3 Remarks
The error function signals a non-continuable error. It calls
13 Expressions, Definitions and Control Forms

This section gives the syntax of well-formed expressions and describes the semantics of the special-forms, functions and macros of the level-0 language. In the case of level-0 macros, the description includes a set of expansion rules. However, these descriptions are not prescriptive of any processor and a conforming program cannot rely on adherence to these expansions.

13.1 Simple Expressions

13.1.1 constant

There are two kinds of constants, literal constants and defined constants. Only the first kind are considered here. A literal constant is a number, a string, a character, or the empty list. The result of processing such a literal constant is the constant itself—that is, it denotes itself.

13.1.1.1 Examples

() the empty list
123 a fixed precision integer
#\a a character
"abc" a string

13.1.2 defconstant

13.1.2.1 Syntax

\begin{verbatim}
defconstant form
 = '(', 'defconstant', identifier, form, ')
\end{verbatim}

13.1.2.2 Arguments

identifier: A symbol naming a binding containing the value of form.
form: The form whose value will be stored in the binding of identifier.

13.1.2.3 Remarks

The value of form is stored in the top-lexical binding of identifier. It is a static error to attempt to modify the binding of a defined constant.

13.1.3 nil

13.1.3.1 Remarks

The symbol nil is defined to be immutably bound to the empty list, which is represented as (). The empty list is used to denote the abstract boolean value false.

13.1.4 t

13.1.4.1 Remarks

The symbol t is defined to be immutably bound to the symbol t. This may be used to denote the abstract boolean value true, but so may any other value than ()

13.1.5 symbol

The current lexical binding of symbol is returned. A symbol can also name a defined constant—that is, an immutable top-lexical binding.

13.1.6 deflocal

13.1.6.1 Syntax

\begin{verbatim}
deflocal form
 = '(', 'deflocal', identifier, form, ')
\end{verbatim}

13.1.6.2 Arguments

identifier: A symbol naming a binding containing the value of form.
form: The form whose value will be stored in the binding of identifier.

13.1.6.3 Remarks

The value of form is stored in the top-lexical binding of identifier. The binding created by a deflocal form is mutable.

13.1.6.4 See also: setq.

13.1.7 quote

13.1.7.1 Syntax

\begin{verbatim}
quote form
 = '(', 'quote', object, ')
\end{verbatim}

13.1.7.2 Arguments

object: the object to be quoted.

13.1.7.3 Result

The result is object.
13.1.7.4 Remarks
The result of processing the expression (quote object) is object. The object can be any object having an external representation. The special form quote can be abbreviated using apostrophe—graphic representation —so that (quote a) can be written 'a. These two notations are used to incorporate literal constants in programs. It is an error to modify a literal expression.

13.2 Functions: creation, definition and application

13.2.1 lambda

13.2.1.1 Syntax

\[
\text{lambda form} = (\text{'lambda}, \text{lambda list}, \{\text{form}\}, ')\; \\
\text{lambda list} = \text{identifier} \quad (\text{*- A.16 *}) \\
\text{simple list} \\
\text{rest list} \\
\text{simple list} = (\text{'identifier}, ')?; \quad (\text{*- A.16 *}) \\
\text{rest list} = (\text{'identifier}, '?, 'identifier, ')?; \\
\]

13.2.1.2 Arguments

\text{lambda-list:} The parameter list of the function conforming to the syntax specified in Table ??.
\text{form:} An expression.

13.2.1.3 Result

A function with the specified \text{lambda-list} and sequence of \text{forms}.

13.2.1.4 Remarks

The function construction operator is \text{lambda}. Access to the lexical environment of definition is guaranteed. The syntax of \text{lambda-list} is defined by the grammar in Table ??.

If \text{lambda-list} is an \text{identifier}, it is bound to a newly allocated list of the actual parameters. This binding has lexical scope and indefinite extent. If \text{lambda-list} is a \text{simple-list}, the arguments are bound to the corresponding \text{identifiers}. Otherwise, \text{lambda-list} must be a \text{rest-list}. In this case, each \text{identifier} preceding the dot is bound to the corresponding argument and the \text{identifier} succeeding the dot is bound to a newly allocated list whose elements are the remaining arguments. These bindings have lexical scope and indefinite extent. It is a static error if the same \text{identifier} appears more than once in a \text{lambda-list}. It is an error to modify \text{rest-list}.

13.2.2 defmacro

13.2.2.1 Syntax

\[
\text{defmacro form} = (\text{'defmacro}, \text{macro name}, \text{lambda list}, \{\text{form}\}, ')\; \\
\]

13.2.2.2 Arguments

\text{macro-name:} A symbol naming an immutable top-lexical binding to be initialized with a function having the specified \text{lambda-list} and \text{body}.
\text{lambda-list:} The parameter list of the function conforming to the syntax specified under \text{lambda}.
\text{body:} A sequence of forms.

13.2.2.3 Remarks

The \text{defmacro} form defines a function named by \text{macro-name} and stores the definition as the top-lexical binding of \text{macro-name}. The interpretation of the \text{lambda-list} is as defined for \text{lambda} (see Table ??).

NOTE — A macro is automatically exported from the the module which defines it. A macro cannot be used in the module which defines it.

13.2.2.4 See also: lambda.

13.2.3 defun

13.2.3.1 Syntax

\[
\text{defun form} = \text{simple defun} \\
\text{setter defun} \\
\text{simple defun} = (\text{'defun}, \text{function name}, \text{lambda list}, \{\text{form}\}, ')\; \\
\text{setter defun} = (\text{'defun}, 'setter', \text{function name}, ')\; \\
\]

13.2.3.2 Arguments

\text{function-name:} A symbol naming an immutable top-lexical binding to be initialized with a function having the specified \text{lambda-list} and \text{body}.
\text{setter function-name:} An expression denoting the setter function to correspond to \text{function-name}.
\text{lambda-list:} The parameter list of the function conforming to the syntax specified under \text{lambda}.
\text{body:} A sequence of forms.
13.2.3 Remarks
The defun form defines a function named by function-name and stores the definition (i) as the top-lexical binding of function-name or (ii) as the setter function of function-name. The interpretation of the lambda-list is as defined for lambda. The rewrite rules for defun are given below.

| (defun identifier | (defconstant identifier lambda-list lambda-list body body) body) |
| (defun (setter identifier) lambda-list lambda-list body body) |

13.2.4 function call

13.2.4.1 Syntax

function call form

- ?'(?, operator, (?operand, ?))

13.2.4.2 Arguments

operator: This may be a symbol—being either the name of a special form, or a lexical variable—or a function call, which must result in an instance of <function>.

An error is signalled (condition class: <invalid-operator>) if the operator is not a function.

operand*: Each operand must be either an atomic expression, a literal expression or a function call.

13.2.4.3 Result

The result is the value of the application of operator to the evaluation of operand*.

13.2.4.4 Remarks

The operand expressions are evaluated in order from left to right. The operator expression may be evaluated at any time before, during or after the evaluation of the operands.

NOTE — The above rule for the evaluation of function calls was finally agreed upon for this version since it is in line with one strand of common practice, but it may be revised in a future version.

13.2.4.5 See also: constant, symbol, quote.

13.2.5 invalid-operator

13.2.5.1 Initialization Options

invalid-operator object: The object which was being used as an operator.

operand-list list: The operands prepared for the operator.

13.2.5.2 Remarks

Signalled by function call if the operator is not an instance of <function>.

13.2.6 apply

13.2.6.1 Syntax

apply form

- ?'(?, apply', function, (form), ?)'

13.2.6.2 Arguments

function: An expression which must evaluate to an instance of <function>.

form, ... form_n-1: A sequence of expressions, which will be evaluated according to the rules given in function call.

form_n: An expression which must evaluate to a proper list. It is an error if obj_n is not a proper list.

13.2.6.3 Result

The result is the result of calling function with the actual parameter list created by appending form_n to a list of the arguments form_0 through form_n-1. An error is signalled (condition class: <invalid-operator>) if the first argument is not an instance of <function>.

13.2.6.4 See also: function call, <invalid-operator>.

13.3 Destructive Operations

An assignment operation modifies the contents of a binding named by a identifier—that is, a variable.

13.3.1 setq

13.3.1.1 Syntax

setq form

- ?'(?, setq', identifier, form, ?)'

13.3.1.2 Arguments

identifier: The identifier whose lexical binding is to be updated.

form: An expression whose value is to be stored in the binding of identifier.
13.3.1.3 Result
The result is the value of form.

13.3.1.4 Remarks
The form is evaluated and the result is stored in the closest lexical binding named by identifier. It is a static error to modify an immutable binding.

13.3.2 setter function

13.3.2.1 Arguments
reader: An expression which must evaluate to an instance of <function>.

13.3.2.2 Result
The writer corresponding to reader.

13.3.2.3 Remarks
A generalized place update facility is provided by setter. Given reader, setter returns the corresponding update function. If no such function is known to setter, an error is signaled (condition class: <no-setter>). Thus (setter car) returns the function to update the car of a pair. New update functions can be added by using setter's update function, which is accessed by the expression (setter setter). Thus ((setter setter) a-reader a-writer) installs the function which is the value of a-writer as the writer of the reader function which is the value of a-reader. All writer functions in this definition and user-defined writers have the same immutable status as other standard functions, such that attempting to redefine such a function, for example ((setter setter) car a-new-value), signals an error (condition class: <cannot-update-setter>)

13.3.2.4 See also: defgeneric, defmethod, defstruct, defun.

13.3.3 no-setter execution-condition

13.3.3.1 Initialization Options
object object: The object given to setter.

13.3.3.2 Remarks
Signalled by setter if there is no updater for the given function.

13.3.4 cannot-update-setter execution-condition

13.3.4.1 Initialization Options
accessor object: The given accessor object.
updater object: The given updater object.

13.3.4.2 Remarks
Signalled by (setter setter) if the updater of the given accessor is immutable.

13.3.4.3 See also: setter.

13.4 Conditional Expressions

13.4.1 if special form

13.4.1.1 Syntax
if form
   = '(% (if 'if antecedent consequent alternative)?)

13.4.1.2 Arguments
antecedent: A form.
consequent: A form.
alternative: A form.

13.4.1.3 Result
Either the value of consequent or alternative depending on the value of antecedent.

13.4.1.4 Remarks
The antecedent is evaluated. If the result is true the consequence is evaluated, otherwise the alternative is evaluated. Both consequence and alternative must be specified. The result of if is the result of the evaluation of whichever of consequence or alternative is chosen.

13.4.2 cond macro

13.4.2.1 Syntax
cond macro
   = '(% (cond 'cond antecedent consequent alternative)?)

13.4.2.2 Remarks
The cond macro provides a convenient syntax for collections of if-then-elseif..else expressions. The rewrite rules for cond are given below.

29
13.5 Variable Binding and Sequences

13.5.1 let/cc

special form

let/cc form = 'let/cc? identifier, (form), ');

13.5.1.2 Arguments

identifier: To be bound to the continuation of the let/cc form.

body: A sequence of forms.

13.5.1.3 Result

The result of evaluating the last form in body or the value of the argument given to the continuation bound to identifier.

13.5.1.4 Remarks

The identifier is bound to a new location, which is initialized with the continuation of the let/cc form. This binding is immutable and has lexical scope and indefinite extent. Each form in body is evaluated in order in the environment extended by the above binding. It is an error to call the continuation outside the dynamic extent of the let/cc form that created it. The continuation is a function of one argument. Calling the continuation causes the restoration of the lexical environment and dynamic environment that existed before entering the let/cc form.

13.5.1.5 Examples

An example of the use of let/cc is given in Figure 6. The function path-open takes a list of paths, the name of a file and list of options to pass to open. It tries to open the file by appending the name to each path in turn. Each time open fails, it signals a condition that the file was not found which is trapped by the handler function. That calls the continuation bound to fail to cause it to try the next path in the list. When open does find a file, the continuation bound to succeed is called with the stream as its argument, which is subsequently returned to the caller of path-open. If the path list is exhausted, map (section A.2) terminates and an error (condition class: <cannot-open-path>) is signalled.

13.5.1.6 See also block, return-from.
Programming Language EuLisp, version 0.99

13.5.2 block
macro

13.5.2.1 Syntax

```
(block macro)
   = ?(?, 'block', identifier, {form}, ?);
```

13.5.2.2 Remarks

The block expression is used to establish a statically scoped binding of an escape function. The block variable is bound to the continuation of the block. The continuation can be invoked anywhere within the block by using return-from. The forms are evaluated in sequence and the value of the last one is returned as the value of the block form. See also let/cc. The rewrite rules for block are given below.

The rewrite for block, does not prevent the block being exited from anywhere in its dynamic extent, since the function bound to identifier is a first-class item and can be passed as an argument like other values.

```
(block identifier)  ⇒ ()
(block identifier form*)  ⇒ (let/cc identifier form*)
```

13.5.2.3 See also: return-from.

13.5.3 return-from
macro

13.5.3.1 Syntax

```
(return-from macro)
   = ?(?, 'return-from', identifier, [form], ?);
```

13.5.3.2 Remarks

In return-from, the identifier names the continuation of the (lexical) block from which to return. return-from is the invocation of the continuation of the block named by identifier. The form is evaluated and the value is returned as the value of the block named by identifier. The rewrite rules for return-from are given below.

```
(return-from identifier)  ⇒ (identifier ())
(return-from identifier form)  ⇒ (identifier form)
```

13.5.3.3 See also: block.

13.5.4 labels
special form

13.5.4.1 Syntax

```
labels form
   = ?(?, 'labels', ?, (function definition), ?),
      (form), ?);
function definition
   = ?(?, identifier, lambda list, [form], ?);
```

13.5.4.2 Arguments

identifier: A symbol naming a new inner-lexical binding to be initialized with the function having the lambda-list and body-specified.

lambda-list: The parameter list of the function conforming to the syntax specified below.

body: A sequence of forms.

labels-body: A sequence of forms.

13.5.4.3 Result

The labels operator provides for local mutually recursive function creation. Each identifier is bound to a new inner-lexical binding initialized with the function constructed from lambda-list and body. The scope of the identifiers is the entire labels form. The lambda-list is either a single variable or a list of variables—see lambda. Each form in labels-body is evaluated in order in the lexical environment extended with the bindings of the identifiers. The result of evaluating the last form in labels-body is returned as the result of the labels form.

13.5.5 let
macro

13.5.5.1 Syntax
13.5.5.2 Remarks

The optional identifier denotes that the let form can be called from within its body. This is an abbreviation for labels form in which identifier is bound to a function whose parameters are the identifiers of the bindings of the let, whose body is that of the let and whose initial call passes the values of the initializing form of the bindings. A binding is specified by either an identifier or a two element list of an identifier and an initializing form. All the initializing forms are evaluated in order from left to right in the current environment and the variables named by the identifiers in the bindings are bound to new locations holding the results. Each form in body is evaluated in order in the environment extended by the above bindings. The result of evaluating the last form in body is returned as the result of the let form. The rewrite rules for let are given below.

\[
\begin{align*}
&\text{(let () } \text{form}) \equiv (\text{progn form}) \\
&\text{(let (id1 form1) (id2 form2) ... ) } \text{form} \equiv (\text{lambda (id1 id2 id3 ... ) form}) \\
&\text{(let id0 (id1 form1) (id2 form2) ... ) } \text{form} \equiv (\text{labels (id1 id2 ... ) form}) \\
&\text{(let* (form)}
\end{align*}
\]

13.5.6 let*

13.5.6.1 Syntax

\[
\text{let star macro}
\]

13.5.6.2 Remarks

A binding is specified by a two element list of a variable and an initializing form. The first initializing form is evaluated in the current environment and the corresponding variable is bound to a new location containing that result. Subsequent bindings are processed in turn, evaluating the initializing form in the environment extended by the previous binding. Each form in body is evaluated in order in the environment extended by the above bindings. The result of evaluating the last form is returned as the result of the let* form. The rewrite rules for let* are given below.

\[
\begin{align*}
&\text{(let* () } \text{form} \equiv (\text{progn form}) \\
&\text{(let* (var1 form1) var2 form2 ... ) } \text{form} \equiv \text{let* ((var1 form1)} \\
&\text{ unprotected form)
\end{align*}
\]
The defined name of this module is event.

13.6.1 wait  
**generic function**

13.6.1.1 Generic Arguments

- `obj`: An object.

- `(timeout <object>)`: One of `()`, `t` or a non-negative integer.

13.6.1.2 Result

Returns `()` if `timeout` was reached, otherwise a non-`()` value.

13.6.2 ticks-per-second <double-float>

The number of time units in a second expressed as a double precision floating point number. This value is implementation-defined.

13.7 Quasiquote Expressions

13.7.1 quasiquote  
**macro**

13.7.1.1 Syntax

```
quasiquote macro  
  = `((`, `quasiquote`, `, `skeleton`, `, `))
```

13.7.1.2 Remarks

Quasiquote is also known as backquoting. A quasiquoted expression is a convenient way of building a structure. The skeleton describes the shape and, generally, many of the entries in the structure but some holes remain...
to be filled. The quasiquote macro can be abbreviated by using the glyph called grave accent (`,) so that (quasiquote expression) can be written `expression.

13.7.2 unquote syntax

13.7.2.1 Syntax

unquote syntax
= `(`, `unquote`, form, `)`
| `,`, form;

13.7.2.2 Remarks
See unquote-splicing.

13.7.3 unquote-splicing syntax

13.7.3.1 Syntax

unquote-splicing syntax
= `(`, `unquote-splicing`, form, `)`
| `,`, form;

13.7.3.2 Remarks
The holes in a quasiquoted expression are identified by unquote expressions of which there are two kinds—forms whose value is to be inserted at that location in the structure and forms whose value is to be spliced into the structure at that location. The former is indicated by an unquote expression and the latter by an unquote-splicing expression. In unquote-splice the form must result in a proper list. The insertion of the result of an unquote-splice expression is as if the opening and closing parentheses of the list are removed and all the elements of the list are appended in place of the unquote-splice expression.

The syntax forms unquote and unquote-splicing can be abbreviated respectively by using the glyph called comma (,) preceding an expression and by using the diphthong comma followed by the glyph called commercial at (\&) preceding a form. Thus, (unquote a) may be written ,a and (unquote-splicing a) can be written ,@a.

13.7.3.3 Examples

`(a ,@((list 1 2) b))  =  (a (1 2) b)
`(a ,@((list 1 2) b))  =  (a 1 2 b)

13.8 Summary of Level-0 Expressions and Definitions

This section gives the syntax of all level-0 expressions and definitions together. The syntax of data objects is given in the section pertaining to the class and is summarized in section A.19.
Programming Language EuLisp, version 0.99

13.8.2 Syntax of Level-0 expressions

generic lambda form

lambda form

defgeneric form

defmethod form

defconstatnt form

deflocal form

defmacro form

defun form

13.8.3 Syntax of Level-0 macros

cond macro

and macro

or macro
block macro
   = `(block?, identifier, (form), ?);

return from macro
   = `(return-from?, identifier, (form), ?);

let macro
   = `(let?, [identifier], `(binding), ?),
      (form), ?);
    binding
      = identifier
        | `(identifier, form, ?);

let star macro
   = `(let*, [identifier], `(binding), ?), (form), ?);

quasiquot e macro
   = `(quasiquote?, skeleton, ?);

unquote syntax
   = `(unquote?, form, ?)
     | ?,?, form;

unquote splicing syntax
   = `(unquote-splicing?, form, ?)
     | ?,@?, form;
Annex A
(normative)
Level-0 Module Library

A.1 Characters

The defined name of this module is character.

A.1.1 character syntax

Character literals are denoted by the extension glyph, called hash (#), followed by the character-extension glyph, called reverse solidus (\), followed by the name of the character. The syntax for the external representation of characters is defined in Table A.1. For most characters, their name is the same as the glyph associated with the character, for example: the character “a” has the name “a” and has the external representation #\a. Certain characters in the group named special (see Table 2 and also Table ??) have symbolic names, for example: the newline character has the name newline and has the external representation #\newline. These special cases are the characters in the production special character token in Table A.1.

Any character which does not have a name, and thereby an external representation dealt with by cases described so far is represented by #\x followed by up to four hexadecimal digits. The value of the hexadecimal number represents the position of the character in the current character set. Examples of such character literals are #\x0 and #\xabc0, which denote, respectively, the characters at position 0 and at position 43081 in the character set current at the time of reading or writing. The syntax for the external representation of characters is defined in Table A.1.

NOTE — This text refers to the “current character set” but defines no means of selecting alternative character sets. This is to allow for future extensions and implementation-defined extensions which support more than one character set.

A.1.2 <character> class

The class of all characters.

A.1.3 characterp function

A.1.3.1 Arguments

object: Object to examine.

A.1.3.2 Result

Returns object if it is a character, otherwise 0.

A.1.4 equal method

A.1.4.1 Specialized Arguments

(character; <character>): A character.

A.1.5 binary< method

A.1.5.1 Specialized Arguments

(character1; <character>): A character. (character2 <character>): A character.

A.1.5.2 Result

If both characters denote uppercase alphabetic or both denote lowercase alphabetic, the result is defined by alphabetical order. If both characters denote a digit, the result is defined by numerical order. In these three cases, if the comparison is true, the result is character1, otherwise the result is 0. Any other comparison is an error and the result of such comparisons is undefined.

A.1.5.3 Examples
A.1.5.4 See also: Method on binary< (A.3) for strings (A.15).

A.1.6 as-lowercase  generic function

A.1.6.1 Generic Arguments

(object <object>): An object to convert to lower case.

A.1.6.2 Result
An instance of the same class as object converted to lower case according to the actions of the appropriate method for the class of object.

A.1.6.3 See also: Another method is defined on as-lowercase for strings (A.15).

A.1.7 as-lowercase  method

A.1.7.1 Specialized Arguments

(character <character>): A character.

A.1.7.2 Result
If character denotes an upper case character, a character denoting its lower case counterpart is returned. Otherwise the result is the argument.

A.1.8 as-uppercase  generic function

A.1.8.1 Generic Arguments

(object <object>): An object to convert to upper case.

A.1.8.2 Result
An instance of the same class as object converted to upper case according to the actions of the appropriate method for the class of object.

A.1.8.3 See also: Another method is defined on as-uppercase for strings (A.15).

A.1.9 as-uppercase  method

A.1.9.1 Specialized Arguments

(character <character>): A character.

A.1.9.2 Result
If character denotes an lower case character, a character denoting its upper case counterpart is returned. Otherwise the result is the argument.

A.1.10 generic-prin  method

A.1.10.1 Specialized Arguments

(character <character>): Character to be output on stream.

(stream <stream>): Stream on which character is to be output.

A.1.10.2 Result
The character character.

A.1.10.3 Remarks
Output the interpretation of character on stream.

A.1.11 generic-write  method

A.1.11.1 Specialized Arguments

(character <character>): Character to be output on stream.

(stream <stream>): Stream on which character is to be output.

A.1.11.2 Result
The character character.

A.1.11.3 Remarks
Output external representation of character on stream in the format #\name as described at the beginning of this section.
A.2 Collections

The defined name of this module is collection. A collection is defined as an instance of one of <list>, <string>, <vector>, <table> or any user-defined class for which a method is added to any of the collection manipulation functions. Collection does not name a class and does not form a part of the class hierarchy. This module defines a set of operations on collections as generic functions and default methods with the behaviours given here.

When iterating over a single collection, the order in which elements are processed might not be important, but as soon as more than one collection is involved, it is necessary to specify how the collections are aligned so that it is clear which elements of the collections will be processed together. This is quite straightforward in the cases of <list>, <string> and <vector>, since there is an intuitive natural order for the elements which allows them to be identified by a non-negative integer. Thus, when iterating over a combination of any of these, all the elements at index position $i$ will be processed together, starting with the elements at position 0 and finishing with those at position $n - 1$ where $n$ is the size of the smallest collection in the combination. The subset of collections which have natural order is called sequence and members of this set can be identified by the predicate sequencep, while collections in general can be identified by collectionp.

Collection alignment is more complicated when tables are involved since they use explicit keys rather than natural order to identify their elements. In any iteration over a combination of collections including a table or some tables, the set of keys used is the intersection of the keys of the tables and the implicit keys of the other collection classes present; this identifies the elements of the collections with common keys. Thus, for an iteration to process any elements from the combination of a collection with natural order and a table, the table must have some integer keys and they must be in the range $[0 .. size]$ of the collection with natural order.

A conforming level-0 implementation must define methods on these functions to support operations on lists (A.12), strings (A.15), tables (A.17), vector (A.18) and any combination of these.

A.2.1 collection-condition

This is the condition class for all collection processing conditions.

A.2.2 accumulate

A.2.2.1 Generic Arguments

(function <function>): A function of two arguments.

(obj <object>): The object which is the initial value for the accumulation operation.

(collection <object>): The collection which is the subject of the accumulation operation.

A.2.2.2 Result

The result is the result of the application of function to the accumulated result and successive elements of collection. The initial value of the accumulated result is supplied by obj.

A.2.2.3 Examples

Note that the order of the elements in the result of the second example depends on the hashing algorithm of the implementation and does not prescribe the result that any particular implementation must give.

\[
\begin{align*}
(\text{accumulate } * 1 #((1 2 3 4 5))) & \Rightarrow 120 \\
(\text{accumulate}
\begin{align*}
& (\lambda (a v) \ (\text{cons } v \ a)) \\
& ()
\end{align*}
\Text{make } <table>
\begin{align*}
& \text{entries}
\end{align*}
\end{align*}
\text{entries}
\begin{align*}
& ?((1 . b) (0 . a) (2 . c)))
\end{align*}
\]

A.2.3 accumulate
generic function

A.2.3.1 Generic Arguments

(function <function>): A function of two arguments.

(collection <object>): The collection which is the subject of the accumulation operation.

A.2.3.2 Result

The result is the result of the application of function to the accumulated result and successive elements of collection starting with the second element. The initial value of the accumulated result is the first element of collection. The terms first and second correspond to the appropriate elements of a natural order collection, but no elements in particular of an explicit key collection. If collection is empty, the result is ()

A.2.3.3 Examples

\[
\begin{align*}
(\text{accumulate}
\begin{align*}
& \text{entries}
\end{align*}
\begin{align*}
& ?((1 . b) (0 . a) (2 . c)))
\end{align*}
\end{align*}
\]

A.2.4 anyp
generic function

A.2.4.1 Generic Arguments

(function <function>): A function to be used as a predicate on the elements of the collection(s).

(collection <object>): A collection.

[more-collections]: More collections.

A.2.4.2 Result

The function is applied to argument lists constructed from corresponding successive elements of collection and more-collections. If the result is true, the result of anyp is true and there are no further applications of function to elements of collection and more-collections. If any of the collections is exhausted, the result of any is ()

39
A.2.4.3 Examples

\begin{verbatim}
(anyp even #((1 2 3 4))) => true
(anyp #((lambda (a b) (= (% a b) 0))
#((3 2) '(1 0)))
\end{verbatim}

A.2.5 collectionp

generic function

A.2.5.1 Generic Arguments

\begin{verbatim}
(object <object>)): An object to examine.
\end{verbatim}

A.2.5.2 Result

Returns true if object is a collection, otherwise ()

A.2.5.3 Remarks

This predicate does not return object because () is a collection.

A.2.6 concatenate

generic function

A.2.6.1 Generic Arguments

\begin{verbatim}
(collection <object>): A collection.
[more-collections]: More collections.
\end{verbatim}

A.2.6.2 Result

The result is an object of the same class as collection.

A.2.6.3 Remarks

The contents of the result object depend on whether collection has natural order or not:

a) If collection has natural order then the size of the result is the sum of the sizes of collection and more-collections. The result collection is initialized with the elements of collection followed by the elements of each of more-collections taken in turn. If any element cannot be stored in the result collection, for example, if the result is a string and some element is not a character, an error is signalled (condition class: collection-condition).

b) If collection does not have natural order, then the result will contain associations for each of the keys in collection and more-collections. If any key occurs more than once, the associated value in the result is the value of the last occurrence of that key after processing collection and each of more-collections taken in turn.

A.2.6.4 Examples

\begin{verbatim}
(concatenate #((1) '(2) "bc") => #((1 2 #\b #\c))
(concatenate "a" #(<\b>) => "ab"
(concatenate #(<\b>) => #(<\b>
'(a b)
"c")
(make <table>)
'(a b)
"c")
\end{verbatim}

A.2.7 do

generic function

A.2.7.1 Generic Arguments

\begin{verbatim}
(function <function>): A function.
(collection <object>): A collection.
[more-collections]: More collections.
\end{verbatim}

A.2.7.2 Result

The result is (). This operator is used for side-effect only. The function is applied to argument lists constructed from corresponding successive elements of collection and more-collections and the result is discarded. Application stops if any of the collections is exhausted.

A.2.7.3 Examples

\begin{verbatim}
(do prin "(1 b #\c)) => 1bc
(do write "(1 b #\c)) => 1b#\c
\end{verbatim}

A.2.8 element

generic function

A.2.8.1 Generic Arguments

\begin{verbatim}
(collection <object>): The object to be accessed or updated.
(key <object>): The object identifying the key of the element in collection.
\end{verbatim}

A.2.8.2 Result

The value associated with key in collection.

A.2.8.3 Examples

\begin{verbatim}
(element "abc" 1) => #\b
(element '(a b c) 1) => b
(element #((a b c) 1) => b
(element (make <table> 'fill-value 'b)
'(a b )
(1))
\end{verbatim}

A.2.9 (setter element)

setter

A.2.9.1 Generic Arguments

\begin{verbatim}
(collection <object>): The object to be accessed or updated.
(key <object>): The object identifying the key of the element in collection.
(value <object>): The object to replace the value associated with key in collection (for setter).
\end{verbatim}

A.2.9.2 Result

The argument supplied as value, having updated the association of key in collection to refer to value.
Programming Language EuLisp, version 0.99

A.2.10 emptyp

Generic function

A.2.10.1 Generic Arguments

(collection <object>): The object to be examined.

A.2.10.2 Result

Returns true if collection is the object identified with the empty object for that class of collection.

A.2.10.3 Examples

(emptyp "") => true
(emptyp ()) => true
(emptyp #()) => true
(emptyp (make <table>)) => true

A.2.11 fill

Generic function

A.2.11.1 Generic Arguments

(collection <object>): A collection to be (partially) filled.

(object <object>): The object with which to fill collection.

[<keys>]: The keys with which object is to be associated.

A.2.11.2 Result

The result is ()

A.2.11.3 Remarks

This function side-effects collection by updating the values associated with each of the specified keys with obj. If no keys are specified, the whole collection is filled with obj. Otherwise, the key specification can take two forms:

a) A collection, in which case the values of the collection are taken to be the keys of collection to be associated with obj.

b) Two fixed precision integers, denoting the start and end keys, respectively, in a natural order collection to be associated with obj. An error is signalled (condition class: collection-condition) if collection does not have natural order. It is an error if the start and end do not specify an ascending sub-interval of the interval \([0, size]\), where size is that of collection.

A.2.12 map

Generic function

A.2.12.1 Generic Arguments

(function <function>): A function.

(collection <object>): A collection.

[more-collections]: More collections.

A.2.12.2 Result

The result is an object of the same class as collection. The elements of the result are computed by the application of function to argument lists constructed from corresponding successive elements of collection and more-collections. Application stops if any of the collections is exhausted.

A.2.12.3 Examples

(map cons #1 2 3) => #((1 . 3))
(map (lambda (f) (f 1 2)) (+ - *) %)

A.2.13 member

Generic function

A.2.13.1 Generic Arguments

(object <object>): The object to be searched for in collection.

(collection <object>): The collection to be searched.

([test <function>]): The function to be used to compare object and the elements of collection.

A.2.13.2 Result

Returns true if there is an element of collection such that the result of the application of test to object and that element is true. If test is not supplied, eql is used by default. Note that true denotes any value that is not () and that the class of the result depends on the class of collection. In particular, if collection is a list, the result of member is a list.

A.2.13.3 Examples

(member #\b "abc") => true
(member #\b #\(a b c\)) => (b c)
(member #\b #\(a b c\)) => true
(member #\b
(make <table> entries \#((1 . b) (0 . a) (2 . c))))

A.2.14 reverse

Generic function

A.2.14.1 Generic Arguments

(collection <object>): A collection.

A.2.14.2 Result

The result is an object of the same class as collection whose elements are the same as those in collection, but in the reverse order with respect to the natural order of collection. If collection does not have natural order, the result is equal to the argument.

A.2.14.3 Examples

(reverse "abc") => "cba"
(reverse #\(1 2 3\)) => (3 2 1)
(reverse #\(a b c\)) => #\(c b a\)
A.2.15 sequence  

**generic function**

A.2.15.1 Generic Arguments

\((\text{object} <\text{object}>):\) An object to examine.

A.2.15.2 Result

Returns true if \(\text{object}\) is a sequence (has natural order), otherwise \(()\).

A.2.15.3 Remarks

This predicate does not return \(\text{object}\) because \(()\) is a sequence.

A.2.16 size  

**generic function**

A.2.16.1 Generic Arguments

\((\text{collection} <\text{object}>):\) The object to be examined.

A.2.16.2 Result

An integer which denotes the size of \(\text{collection}\) according to the method for the class of \(\text{collection}\).

A.2.16.3 Examples

\begin{align*}
\text{(size "")} & \Rightarrow 0 \\
\text{(size ())} & \Rightarrow 0 \\
\text{(size ())} & \Rightarrow 0 \\
\text{(size (make <table>))} & \Rightarrow 0 \\
\text{(size "abc")} & \Rightarrow 3 \\
\text{(size (cons 1 ()))} & \Rightarrow 1 \\
\text{(size (cons 1 . 2))} & \Rightarrow 1 \\
\text{(size (cons 1 (cons 2 . 3)))} & \Rightarrow 2 \\
\text{(size '(1 2 3))} & \Rightarrow 3 \\
\text{(size #(a b c))} & \Rightarrow 3 \\
\text{(size (make <table> 'entries '((0 . a)))))} & \Rightarrow 1
\end{align*}

A.2.17 (converter <list>)  

**method**

A.2.17.1 Specialized Arguments

\((\text{collection} <\text{object}>):\) A collection to be converted into a list.

A.2.17.2 Result

If \(\text{collection}\) is a list, the result is the argument. Otherwise a list is constructed and returned whose elements are the elements of \(\text{collection}\). If \(\text{collection}\) has natural order, then the elements will appear in the result in the same order as in \(\text{collection}\). If \(\text{collection}\) does not have natural order, the order in the resulting list is undefined.

A.2.17.3 See also: Conversion (A.4).

A.2.18 (converter <string>)  

**method**

A.2.18.1 Specialized Arguments

\((\text{collection} <\text{object}>):\) A collection to be converted into a string.

A.2.18.2 Result

If \(\text{collection}\) is a string, the result is the argument. Otherwise a string is constructed and returned whose elements are the elements of \(\text{collection}\) as long as all the elements of \(\text{collection}\) are characters. An error is signalled (condition class: conversion-condition) if any element of \(\text{collection}\) is not a character. If \(\text{collection}\) has natural order, then the elements will appear in the result in the same order as in \(\text{collection}\). If \(\text{collection}\) does not have natural order, the order in the resulting string is undefined.

A.2.18.3 See also: Conversion (A.4).

A.2.19 (converter <table>)  

**method**

A.2.19.1 Specialized Arguments

\((\text{collection} <\text{object}>):\) A collection to be converted into a table.

A.2.19.2 Result

If \(\text{collection}\) is a table, the result is the argument. Otherwise a table is constructed and returned whose elements are the elements of \(\text{collection}\). If \(\text{collection}\) has natural order, then the elements will be stored under integer keys in the range \([0 .. \text{size})\), otherwise the keys used will be the keys associated with the elements of \(\text{collection}\).

A.2.19.3 See also: Conversion (A.4).

A.2.20 (converter <vector>)  

**method**

A.2.20.1 Specialized Arguments

\((\text{collection} <\text{object}>):\) A collection to be converted into a vector.

A.2.20.2 Result

If \(\text{collection}\) is a vector, the result is the argument. Otherwise a vector is constructed and returned whose elements are the elements of \(\text{collection}\). If \(\text{collection}\) has natural order, then the elements will appear in the result in the same order as in \(\text{collection}\). If \(\text{collection}\) does not have natural order, the order in the resulting vector is undefined.

A.2.20.3 See also: Conversion (A.4).
A.3 Comparison

The defined name of this module is `compare`. There are four functions for comparing objects for equality, of which `=` is specifically for comparing numeric values and `eq`, `eql` and `equal` are for all objects. The latter three are related in the following way:

\[
\begin{align*}
(eq \ a \ b) & \Rightarrow (eql \ a \ b) \Rightarrow (equal \ a \ b) \\
(eq \ a \ b) & \nRightarrow (eql \ a \ b) \nRightarrow (equal \ a \ b)
\end{align*}
\]

There is one function for comparing objects by order, which is called `<`, and which is implemented by the generic function `binary<?`. A summary of the comparison functions and the classes for which they have defined behaviour is given in Table A.2.

### A.3.1 eq

#### A.3.1.1 Arguments

- `object1`: An object.
- `object2`: An object.

#### A.3.1.2 Result

Compares `object1` and `object2` and returns `t` if they are the same object, otherwise `()`. *Same* in this context means "identifies the same memory location".

#### A.3.1.3 Remarks

In the case of numbers and characters the behaviour of `eq` might differ between processors because of implementation choices about internal representations. Therefore, `eq` might return `t` or `()` for numbers which are `=` and similarly for characters which are `eql` depending on the implementation.

#### A.3.1.4 Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(eq 'a 'a)</code></td>
<td><code>t</code></td>
</tr>
<tr>
<td><code>(eq 'a 'b)</code></td>
<td><code>()</code></td>
</tr>
<tr>
<td><code>(eq 3 3)</code></td>
<td><code>t</code></td>
</tr>
<tr>
<td><code>(eq 3 3.0)</code></td>
<td><code>()</code></td>
</tr>
<tr>
<td><code>(eq (cons 'a 'b) (cons 'a 'c))</code></td>
<td><code>()</code></td>
</tr>
<tr>
<td><code>(eq (cons 'a 'b) (cons 'a 'b))</code></td>
<td><code>t</code></td>
</tr>
<tr>
<td><code>(eq '(a . b) '(a . b))</code></td>
<td><code>t</code></td>
</tr>
<tr>
<td><code>(let ((x (cons 'a 'b))) (eq x x))</code></td>
<td><code>t</code></td>
</tr>
<tr>
<td><code>(let ((x (cons 'a 'b))) (eq x x))</code></td>
<td><code>()</code></td>
</tr>
<tr>
<td><code>(eq #\a #\a)</code></td>
<td><code>t</code></td>
</tr>
<tr>
<td><code>(eq &quot;string&quot; &quot;string&quot;)</code></td>
<td><code>t</code></td>
</tr>
<tr>
<td><code>(eq #\'b #\'b)</code></td>
<td><code>()</code></td>
</tr>
<tr>
<td><code>(let ((x #\(a 'b))) (eq x x))</code></td>
<td><code>t</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### A.3.2 eql

#### A.3.2.1 Arguments

- `object1`: An object.
- `object2`: An object.

### A.3.3 equal

#### A.3.3.1 Arguments

- `object1`: An object.
- `object2`: An object.

#### A.3.3.2 Result

Returns true or false according to the method for the class(es) of `object1` and `object2`. It is an error if either or both of the arguments is self-referential.

#### A.3.3.3 Remarks

See also: Class specific methods on `equal` are defined for characters (A.1), lists (A.12), numbers (A.13), strings (A.15) and vectors (A.18). All other cases are handled by the default method.

### A.3.4 equal

#### A.3.4.1 Specialized Arguments

- `(object1 <object2>)`: An object.
- `(object2 <object2>)`: An object.

#### A.3.4.2 Result

The result is as if `eql` had been called with the arguments supplied.

#### A.3.4.3 Remarks

Note that in the case of this method being invoked from `equal`, the arguments cannot be characters or numbers.

### A.3.5 =

#### A.3.5.1 Arguments

- `number1, ...`: A non-empty sequence of numbers.
Table A.2 — Summary of comparison functions

| eq:          |   <object><object>   |
| eql:         |   <object><object>   |
|             |   <character><character>⇒equal  |
|             |   <fixed-precision-integer><fixed-precision-integer>⇒binary=  |
|             |   <double-float><double-float>⇒binary=  |
| equal:       |   <object><object>   |
|             |   <character><character>  |
|             |   <null><null>   |
|             |   <number><number>⇒eql  |
|             |   <cons><cons>  |
|             |   <string><string>  |
|             |   <vector><vector>  |
| =:           |   <number><number>⇒binary=  |
| binary=      |   <fixed-precision-integer><fixed-precision-integer>  |
|             |   <double-float><double-float>  |
| <=           |   <object><object>⇒binary<  |
| binary<      |   <character><character>  |
|             |   <fixed-precision-integer><fixed-precision-integer>  |
|             |   <double-float><double-float>  |
|             |   <string><string>  |

A.3.5.2 Result
Given one argument the result is true. Given more than one argument the result is determined by binary=, returning true if all the arguments are the same, otherwise ( ).

A.3.6 binary= generic function

A.3.6.1 Generic Arguments

(number1 number2): A number.

A.3.6.2 Result
One of the arguments, or ( ).

A.3.6.3 Remarks
The result is either a number or ( ). This is determined by whichever class specific method is most applicable for the supplied arguments.

A.3.6.4 See also: Class specific methods on binary= are defined for fixed precision integer (A.9) and double float (A.6).

A.3.7 < function

A.3.7.1 Arguments

object1 . . . : A non-empty sequence of objects.

A.3.7.2 Result
Given one argument the result is true. Given more than one argument the result is true if the sequence of objects object1 up to objectn is strictly increasing according to the generic function binary<. Otherwise, the result is ( ).

A.3.8 binary< generic function

A.3.8.1 Generic Arguments

(object1 object2): An object.

A.3.8.2 Result
The first argument if it is less than the second, according to the method for the class of the arguments, otherwise ( )

A.3.8.3 See also: Class specific methods on binary< are defined for characters (A.1), strings (A.15), fixed precision integers (A.9) and double floats (A.6).

A.3.9 max function

A.3.9.1 Arguments

object1 . . . : A non-empty sequence of objects.

A.3.9.2 Result
The maximal element of the sequence of objects object1 up to objectn using the generic function binary<. Zero arguments is an error. One argument returns object1.

A.3.10 min function

A.3.10.1 Arguments

object1 . . . : A non-empty sequence of objects.
A.3.10.2 Result
The minimal element of the sequence of objects \( \text{object}_1 \) up to \( \text{object}_n \), using the generic function \( \text{binary}< \). Zero arguments is an error. One argument returns \( \text{object}_1 \).

A.4 Conversion

The defined name of this module is convert.

The mechanism for the conversion of an instance of one class to an instance of another is defined by a user-extensible framework which has some similarity to the setter mechanism.

To the user, the interface to conversion is via the function convert, which takes an object and some class to which the object is to be converted. The target class is used to access an associated converter function, in fact, a generic function, which is applied to the source instance, dispatching on its class to select the method which implements the appropriate conversion. Thus, having defined a new class to which it may be desirable to convert instances of other classes, the programmer defines a generic function:

\[
\begin{align*}
&\text{(defgeneric (converter new-class) (instance))} \\
&\text{(defmethod (converter new-class)} } \\
&\quad \text{(instance other-class))}
\end{align*}
\]

Hereafter, new converter methods may be defined for new-class using a similar extended syntax for defmethod:

\[
\begin{align*}
&\text{(defmethod (converter new-class)} } \\
&\quad \text{(instance other-class))}
\end{align*}
\]

The conversion is implemented by defining methods on the converter for new-class which specialize on the source class. This is also how methods are documented in this text: by an entry for a method on the converter function for the target class. In general, the method for a given source class is defined in the section about that class, for example, converters from one kind of collection to another are defined in section A.2, converters from string in section A.15, etc..

A.4.1 convert function

A.4.1.1 Arguments

- \( \text{object} \): An instance of some class to be converted to an instance of \( \text{class} \).
- \( \text{class} \): The class to which \( \text{object} \) is to be converted.

A.4.1.2 Result

Returns an instance of \( \text{class} \) which is equivalent in some class-specific sense to \( \text{object} \), which may be an instance of any type. Calls the converter function associated with \( \text{class} \) to carry out the conversion operation. An error is signalled (condition: \( \text{no-converter} \)) if there is no associated function. An error is signalled (condition: \( \text{no-applicable-method} \)) if there is no method to convert an instance of the class of \( \text{object} \) to an instance of \( \text{class} \).

A.4.2 conversion-condition condition

This is the general condition class for all conditions arising from conversion operations.
A.4.2.1 Initialization Options

source <object>: The object to be converted into an instance of target-class.

target-class <class>: The target class for the conversion operation.

A.4.2.2 Remarks
Should be signalled by convert or a converter method.

A.4.3 converter function

A.4.3.1 Arguments

target-class: The class whose set of conversion methods is required.

generic-function: The new converter function.

A.4.3.2 Result
The accessor returns the converter function for the class target-class. The converter is a generic-function with methods specialized on the class of the object to be converted.

A.4.4 (setter converter) setter

A.4.4.1 Arguments

target-class: The class whose converter function is to be replaced.

generic-function: The new converter function.

A.4.4.2 Result
The new converter function. The setter function replaces the converter function for the class target-class by generic-function. The new converter function must be an instance of generic-function.

A.4.4.3 Remarks
Converter methods from one class to another are defined in the section pertaining to the source class.

A.4.4.4 See also: Converter methods are defined for collections (??), double float (A.6), fixed precision integer (A.9), string (A.15), symbol (A.16), vector (A.18).

A.5 Copying

The defined name of this module is copy.

A.5.1 deep-copy generic function

A.5.1.1 Generic Arguments

object: An object to be copied.

A.5.1.2 Result
Constructs and returns a copy of the source which is the same (under some class specific predicate) as the source and whose slots contain copies of the objects stored in the corresponding slots of the source, and so on. The exact behaviour for each class of object is defined by the most applicable method for object.

A.5.1.3 See also: Class specific sections which define methods on deep-copy: list (A.12), string (A.15), table (A.17) and vector (A.18).

A.5.2 deep-copy method

A.5.2.1 Specialized Arguments

(object <object>): An object.

A.5.2.2 Result
Returns object.

A.5.3 deep-copy method

A.5.3.1 Specialized Arguments

(struct <structure-class>): A structure.

A.5.3.2 Result
Constructs and returns a new structure whose slots are initialized with copies (using deep-copy) of the contents of the slots of struct.

A.5.4 shallow-copy generic function

A.5.4.1 Generic Arguments

object: An object to be copied.

A.5.4.2 Result
Constructs and returns a copy of the source which is the same (under some class specific predicate) as the source. The exact behaviour for each class of object is defined by the most applicable method for object.
A.5.4.3 See also: Class specific sections which define methods on shallow-copy: pair (A.12), string (A.15), table (A.17) and vector (A.18).

A.5.5 shallow-copy method

A.5.5.1 Specialized Arguments

(object <object>): An object.

A.5.5.2 Result

Returns object.

A.5.6 shallow-copy method

A.5.6.1 Specialized Arguments

(struct <structure-class>): A structure.

A.5.6.2 Result

Constructs and returns a new structure whose slots are initialized with the contents of the corresponding slots of struct.

A.6 Double Precision Floats

The defined name of this module is double. Arithmetic operations for <double-float> are defined by methods on the generic functions defined in the number module (A.13):

- binary+, binary-, binary*, binary/, binary<, binary=, mod, negate, zero

the float module (A.8):

- ceiling, floor, round, truncate

and the elementary functions module (A.7):

- acos, asin, atan, atan2, cos, sin, tan, cosh, sinh, tanh

- exp, log, log10, pow, sqrt

The behaviour of these functions is defined in the modules noted above.

A.6.1 <double-float> class

The class of all double precision floating point numbers.

The syntax for the exponent of a double precision floating point is given below. The general syntax for floating point numbers is given in section A.8.

```
  double exponent = ('D' | 'd'), [sign], decimal integer; (* A.11 *)
```

A.6.2 double-float-p function

A.6.2.1 Arguments

object: Object to examine.

A.6.2.2 Result

Returns object if it is a double float, otherwise ().

A.6.2.3 See also: floatp (A.13).

A.6.3 most-positive-double-float double-float

A.6.3.1 Remarks

The value of most-positive-double-float is that positive double precision floating point number closest in value to (but not equal to) positive infinity that the processor provides.

A.6.4 least-positive-double-float double-float

A.6.4.1 Remarks

The value of least-positive-double-float is that positive
double precision floating point number closest in value to (but not equal to) zero that the processor provides.

A.6.5 least-negative-double-float double-float

A.6.5.1 Remarks
The value of least-negative-double-float is that negative double precision floating point number closest in value to (but not equal to) zero that the processor provides. Even if the processor provide negative zero, this value must not be negative zero.

A.6.6 most-negative-double-float double-float

A.6.6.1 Remarks
The value of most-negative-double-float is that negative double precision floating point number closest in value to (but not equal to) negative infinity that the processor provides.

A.6.7 equal method

A.6.7.1 Specialized Arguments
(double1 <double-float>): A double precision float.
(double2 <double-float>): A double precision float.

A.6.7.2 Result
The result of calling binary= on double1 and double2.

A.6.8 (coereter <string>) method

A.6.8.1 Specialized Arguments
(x <double-float>): A double precision float.

A.6.8.2 Result
Constructs and returns a string, the characters of which correspond to the external representation of x as produced by generic-prin, namely that specified in the syntax as [sign]float format 3.

A.6.9 (coereter <fixed-precision-integer>) method

A.6.9.1 Specialized Arguments
(x <double-float>): A double precision float.

A.6.9.2 Result
A fixed precision integer.

A.6.9.3 Remarks
This function is the same as round. It is defined for the sake of symmetry.

A.6.10 generic-prin method

A.6.10.1 Specialized Arguments
(double <double-float>): The double float to be output on stream.
(stream <stream>): The stream on which the representation is to be output.

A.6.10.2 Result
The double float supplied as the first argument.

A.6.11 generic-write method

A.6.11.1 Specialized Arguments
(double <double-float>): The double float to be output on stream.
(stream <stream>): The stream on which the representation is to be output.

A.6.11.2 Result
The double float supplied as the first argument.

A.6.11.3 Remarks
Outputs the external representation of double on stream, as an optional sign preceding the syntax defined by float format 3. Finer control over the format of the output of floating point numbers is provided by some of the formatting specifications of format (see section A.10).
A.7 Elementary Functions

The defined name of this module is elementary-functions. The functionality defined for this module is intentionally precisely that of the trigonometric functions, hyperbolic functions, exponential and logarithmic functions and power functions defined for <math.h> in ISO/IEC 9899:1990 with the exceptions of frexp, ldexp and modf.

A.7.1 pi double-float

A.7.1.1 Remarks

The value of pi is the ratio the circumference of a circle to its diameter stored to double precision floating point accuracy.

A.7.2 acos generic function

A.7.2.1 Generic Arguments

(float <float>): A floating point number.

A.7.2.2 Result

Computes the principal value of the arc cosine of float which is a value in the range [0, π] radians. An error is signalled (condition-class: domain-condition) if float is not in the range [-1, +1].

A.7.3 asin generic function

A.7.3.1 Generic Arguments

(float <float>): A floating point number.

A.7.3.2 Result

Computes the principal value of the arc sine of float which is a value in the range [-π/2, +π/2] radians. An error is signalled (condition-class: domain-condition) if float is not in the range [-1, +1].

A.7.4 atan generic function

A.7.4.1 Generic Arguments

(float <float>): A floating point number.

A.7.4.2 Result

Computes the principal value of the arc tangent of float which is a value in the range [-π/2, +π/2] radians.

A.7.5 atan2 generic function

A.7.5.1 Generic Arguments

(float1 <float>): A floating point number.

A.7.6 cos generic function

A.7.6.1 Generic Arguments

(float <float>): A floating point number.

A.7.6.2 Result

Computes the cosine of float (measured in radians).

A.7.7 sin generic function

A.7.7.1 Generic Arguments

(float <float>): A floating point number.

A.7.7.2 Result

Computes the sine of float (measured in radians).

A.7.8 tan generic function

A.7.8.1 Generic Arguments

(float <float>): A floating point number.

A.7.8.2 Result

Computes the tangent of float (measured in radians).

A.7.9 cosh generic function

A.7.9.1 Generic Arguments

(float <float>): A floating point number.

A.7.9.2 Result

Computes the hyperbolic cosine of float. An error might be signalled (condition class: range-condition) if the magnitude of float is too large.

A.7.10 sinh generic function

A.7.10.1 Generic Arguments

(float <float>): A floating point number.
A.7.10.2 Result
Computes the hyperbolic sine of float. An error might be signalled (condition class: range-condition) if the magnitude of float is too large.

A.7.11 tanh

A.7.11.1 Generic Arguments
(float <float>): A floating point number.

A.7.11.2 Result
Computes the hyperbolic tangent of float.

A.7.12 exp

A.7.12.1 Generic Arguments
(float <float>): A floating point number.

A.7.12.2 Result
Computes the exponential function of float. An error might be signalled (condition class: range-condition) if the magnitude of float is too large.

A.7.13 log

A.7.13.1 Generic Arguments
(float <float>): A floating point number.

A.7.13.2 Result
Computes the natural logarithm of float. An error is signalled (condition class: domain-condition) if float is negative. An error might be signalled (condition class: range-condition) if float is zero.

A.7.14 log10

A.7.14.1 Generic Arguments
(float <float>): A floating point number.

A.7.14.2 Result
Computes the base-ten logarithm of float. An error is signalled (condition class: domain-condition) if float is negative. An error might be signalled (condition class: range-condition) if float is zero.

A.7.15 pow

A.7.15.1 Generic Arguments
(float1 <float>), (float2 <float>): A floating point number.

A.7.16 sqrt

A.7.16.1 Generic Arguments
(float <float>): A floating point number.

A.7.16.2 Result
Computes float; raised to the power float2. An error is signalled (condition class: domain-condition) if float1 is negative and float2 is not integral. An error is signalled (condition class: domain-condition) if the result cannot be represented when float1 is zero and float2 is less than or equal to zero. An error might be signalled (condition class: range-condition) if the result cannot be represented.
A.8 Floating Point Numbers

The defined name of this module is float. This module defines the abstract class <float> and the behaviour of some generic functions on floating point numbers. Further operations on numbers are defined in the numbers module (A.13) and further operations on floating point numbers are defined in the elementary functions module (A.7). A concrete float class is defined in the double float module (A.6).

A.8.1 float syntax

The syntax for the external representation of floating point literals is defined in Table A.3. The representation used by write and prin is that of a sign, a whole part and a fractional part without an exponent, namely that defined by float format 3. Finer control over the format of the output of floating point numbers is provided by some of the formatting specifications of format (section A.10).

To be continued...
A.9 Fixed Precision Integers

The defined name of this module is fpi. Arithmetic operations for <fixed-precision-integer> are defined by methods on the generic functions defined in the number module:

binary+, binary-, binary*, binary/, binary%, binary<, binary>, binary-gcd, binary-lcm, mod, negate, zero

and in the integer module:

evenp

The behaviour of these functions is defined in the modules noted above.

A.9.1 <fixed-precision-integer> class

The class of all instances of fixed precision integers.

A.9.2 fpi class

A.9.2.1 Remarks

A constant binding whose value is <fixed-precision-integer>.

A.9.3 fixed-precision-integer-p function

A.9.3.1 Arguments

object: Object to examine.

A.9.3.2 Result

Returns object if it is fixed precision integer, otherwise 0.

A.9.4 most-positive-fixed-precision-integer function

A.9.4.1 Remarks

This is an implementation-defined constant. A conforming processor must support a value greater than or equal to 32767 and greater than or equal to the value of maximum-vector-index.

A.9.5 most-negative-fixed-precision-integer function

A.9.5.1 Remarks

This is an implementation-defined constant. A conforming processor must support a value less than or equal to -32768.

A.9.6 equal method

A.9.6.1 Specialized Arguments

(integer, <fixed-precision-integer>): A fixed precision integer.

(integer, <fixed-precision-integer>): A fixed precision integer.

A.9.6.2 Result

The result of calling binary= on integer1 and integer2.

A.9.6.3 Remarks

The difference between equal and = is that the former only compares numbers of the same class, whereas the latter coerces the numbers to be of the same class before comparison.

A.9.7 (converter <string>) method

A.9.7.1 Specialized Arguments

(integer <fixed-precision-integer>): An integer.

A.9.7.2 Result

Constructs and returns a string, the characters of which correspond to the external representation of integer in decimal notation.

A.9.8 (converter <double-float>) method

A.9.8.1 Specialized Arguments

(integer <fixed-precision-integer>): An integer.

A.9.8.2 Result

Returns a double float whose value is the floating point approximation to integer.

A.9.9 generic-prin method

A.9.9.1 Specialized Arguments

(integer <fixed-precision-integer>): An integer to be output on stream.

(stream <stream>): The stream on which the representation is to be output.

A.9.9.2 Result

The fixed precision integer supplied as the first argument.

A.9.9.3 Remarks

Outputs external representation of integer on stream in decimal as defined by decimal integer at the beginning of this section.
A.9.10 generic-write

method

A.9.10.1 Specialized Arguments

\(\text{integer} \langle\text{fixed-precision-integer}\rangle\): An integer to be output on \text{stream}.

\(\langle\text{stream}\ \langle\text{stream}\rangle\rangle\): The stream on which the representation is to be output.

A.9.10.2 Result

The fixed precision integer supplied as the first argument.

A.9.10.3 Remarks

Outputs external representation of \text{integer} on \text{stream} in decimal as defined by \text{decimal integer} at the beginning of this section.

A.10 Formatted-IO

The defined name of this module is \text{formatted-io}.

A.10.1 scan

function

A.10.1.1 Arguments

\text{format-string}: A string containing format directives.

\(\langle\text{stream}\rangle\): A stream from which input is to be taken.

A.10.1.2 Result

Returns a list of the objects read from \text{stream}.

A.10.1.3 Remarks

This function provides support for formatted input. The \text{format-string} specifies reading directives, and inputs are matched according to these directives. An error is signaled (condition: \text{scan-mismatch}) if the class of the object read is not compatible with the specified directive. The second (optional) argument specifies a stream from which to take input. If \text{stream} is not supplied, input is taken from the result of calling \text{standard-input-stream}. Scan returns a list of the objects read in.

- \(\sim\text{a}\) any: any object
- \(\sim\text{b}\) binary: an integer in binary format.
- \(\sim\text{c}\) character: a single character
- \(\sim\text{d}\) decimal: an integer decimal format.
- \(\sim[n]\text{e}\) a exponential-format floating-point number.
- \(\sim[n]\text{f}\) a fixed-format floating-point number.
- \(\sim\text{o}\) octal: an integer in octal format.
- \(\sim\text{r}\) radix: an integer in specified radix format.
- \(\sim\text{x}\) hexadecimal: an integer in hexadecimal format.
- \(\sim\%\) newline: matches a \#\newline character in the input.

A.10.2 scan-mismatch

stream-condition

A.10.2.1 Initialization Options

\text{format-string string}: The value of this option is the format string that was passed to \text{scan}.

\text{input list}: The value of this option is a list of the items read by \text{scan} up to and including the object that caused the condition to be signaled.

A.10.2.2 Remarks

This condition is signalled by \text{scan} if the format string does not match the data input from \text{stream}.
A.10.3 format function

A.10.3.1 Arguments

- stream: One of O, t or a stream.

- format-string: A string containing format directives.

- [object, ...]: A sequence of objects to be output on stream.

A.10.3.2 Result

Returns a list comprising those objects left in the sequence after the last format directive was processed.

A.10.3.3 Remarks

Has side-effect of outputting objects according the formats specified in format-string. If stream is t, the output is to the current output stream. If stream is O, a formatted string is returned as the result of the call. Otherwise stream must be a valid output stream. Characters are output as if the string were output by the prin function with the exception of those prefixed by tilde—graphic representation—which are treated specially as detailed in the following list. These formatting directives are intentionally compatible with the facilities defined for the function fprintf in ISO/IEC 9899 : 1990.

- ~a any: uses prin to output the argument.

- ~b binary: the argument must be an integer and is output in binary notation (Table A.4).

- ~c character: the argument must be a character and is output using write (Table A.1).

- ~d decimal: the argument must be an integer and is output using write (Table A.4).

- ~[m].[n]e exponential-format floating-point: the argument must be a floating point number. It is output in the style [-]d.ddde±dd, in a field of width m characters, where there are n precision digits after the decimal point, or 6 digits, if n is not specified (Table A.3). If the value to be output has fewer characters than m it is padded on the left with spaces.

- ~[m].[n]f fixed-format floating-point: the argument must be a floating point number. It is output in the style [-]d.dddd, in a field of width m characters, where the are n precision digits after the decimal point, or 6 digits, if n is not specified (Table A.3). The value is rounded to the appropriate number of digits. If the value to be output has fewer characters than m it is padded on the left with spaces.

- ~[m].[n]g generalized floating-point: the argument must be a floating point number. It is output in either fixed-format or exponential notation as appropriate (Table A.3).

- ~o octal: the argument must be an integer and is output in octal notation (Table A.4).
A.11 Integers

module The defined name of this module is integer. This module defines the abstract class <integer> and the behavior of some generic functions on integers. Further operations on numbers are defined in the numbers module (A.13). A concrete integer class is defined in the fixed precision integer module (A.9).

A.11.1 integer syntax

A positive integer is has its external representation as a sequence of digits optionally preceded by a plus sign. A negative integer is written as a sequence of digits preceded by a minus sign. For example, 1234567890, -456, +1959.

Integer literals have an external representation in any base up to base 36. For convenience, base 2, base 8 and base 16 have distinguished notations—#b, #o and #x, respectively. For example: 1234, #b10011010010, #o23222 and #x4d2 all denote the same value.

The general notation for an arbitrary base is #b<base>, where <base> is an unsigned decimal number. Thus, the above examples may also be written: #10r1234, #2r10011010010, #8r23222, #16r4d2 or #36rya. The reading of any number is terminated on encountering a character which cannot be a constituent of that number. The syntax for the external representation of integer literals is defined in Table A.4.

NOTE — At present this text does not define a class integer with variable precision. It is planned this should appear in a future version at level 1 of the language. The class will be named <variable-precision-integer>. The syntax given here is applicable to both fixed and variable precision integers.

A.11.2 <integer> class

The abstract class which is the superclass of all integer numbers.

A.11.3 integerp function

A.11.3.1 Arguments

object: Object to examine.

A.11.3.2 Result

Returns object if it is an integer, otherwise ()

A.11.4 evenp generic function

A.11.4.1 Arguments

integer, <integer>: An integer.

A.11.4.2 Result

Returns t if two divides integer, otherwise ()

A.11.5 oddp function

A.11.5.1 Arguments

integer: An integer.

A.11.5.2 Result

Returns the equivalent of the logical negation of evenp applied to integer.

A.11.5.3 Table A.4 — Integer Syntax

| integer | = | [sign] unsigned integer |
| sign | = | + | - |
| unsigned integer | = | [binary integer] | [octal integer] | [decimal integer] | [hexadecimal integer] | [specified base integer] |
| binary integer | = | #b<digit>, binary digit, {binary digit} |
| octal integer | = | #o<digit>, octal digit, {octal digit} |
| decimal integer | = | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| hexadecimal integer | = | #x<digit>, hexadecimal digit, {hexadecimal digit} |
| specified base integer | = | #<base> | specified base digit | {specified base digit} |
| base specification | = | decimal digit | {decimal digit} |
| specified base digit | = | decimal digit | {decimal digit} |

Table A.4 — Integer Syntax

| integer | [sign] unsigned integer |
| sign | = | + | - |
| unsigned integer | = | binary integer | octal integer | decimal integer | hexadecimal integer | specified base integer |
| binary integer | = | #b<digit>, binary digit, {binary digit} |
| octal integer | = | #o<digit>, octal digit, {octal digit} |
| decimal integer | = | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| hexadecimal integer | = | #x<digit>, hexadecimal digit, {hexadecimal digit} |
| specified base integer | = | #<base> | specified base digit | {specified base digit} |
| base specification | = | decimal digit | {decimal digit} |
| specified base digit | = | decimal digit | {decimal digit} |
A.12 Lists

The name of this module is list. The class <list> is an abstract class and has two subclasses: <null> and <cons>. The only instance of <null> is the empty list. The combination of these two classes allows the creation of proper lists, since a proper list is one whose last pair contains the empty list in its cdr field. See also section A.2 (collections) for further operations on lists.

A.12.1 () syntax

A.12.1.1 Remarks
The empty list, which is the only instance of the class <null>, has as its external representation an open parenthesis followed by a close parenthesis. The empty list is also used to denote the boolean value false.

A.12.2 <null> class

The class whose only instance is the empty list, denoted ()

A.12.3 null function

A.12.3.1 Arguments
  object: Object to examine.

A.12.3.2 Result
  Returns t if object is the empty list, otherwise ()

A.12.4 generic-prin method

A.12.4.1 Specialized Arguments
  null: The empty list.
  stream: The stream on which the representation is to be output.

A.12.4.2 Result
  The empty list.

A.12.4.3 Remarks
  Output the external representation of the empty list on stream as described above.

A.12.5 generic-write method

A.12.5.1 Specialized Arguments
  null: The empty list.
  stream: The stream on which the representation is to be output.

A.12.5.2 Result
  The empty list.

A.12.5.3 Remarks
  Output the external representation of the empty list on stream as described above.

A.12.6 pair syntax

A pair is written as (object1, object2), where object1 is called the car and object2 is called the cdr. There are two special cases in the external representation of pair. If object2 is the empty list, then the pair is written as (object1). If object2 is an instance of pair, then the pair is written as (object1, object2, object3), where object2 is the car of object2 and object3 is the cdr with the above rule for the empty list applying. By induction, a list of length n is written as (object1, ..., object_{n-1}, object_n), with the above rule for the empty list applying. The representations of object1 and object2 are determined by the external representations defined in other sections of this definition (see <character> (A.1), <double-float> (A.6), <fixed-precision-integer> (A.9), <string> (A.15), <symbol> (A.16) and <vector> (A.18), as well as instances of <cons> itself. The syntax for the external representation of pairs and lists is defined in Table A.5.

Table A.5 — Pair and List Syntax

| null | '() | |
| pair | '(, object, , , object, ) | |
| list | empty list | proper list | improper list |
|      | = '() | |
|      | = '(, object, ) | |
|      | = '(, object, , object, ) | |

A.12.6.1 Examples

|      | the empty list | |
|      | (1) a list whose car is 1 and cdr is () | |
|      | (1 2) a pair whose car is 1 and cdr is 2 | |
|      | (1 2) a list whose car is 1 and cdr is (2) | |

A.12.7 <cons> class

The class of all instances of <cons>. An instance of the class <cons> (also known informally as a dotted pair or a pair) is a 2-tuple, whose slots are called, for historical reasons, car and cdr. Pairs are created by the function cons and the slots are accessed by the functions car and cdr. The major use of pairs is in the construction of (proper) lists. A (proper) list is defined as either the empty list (denoted by ()) or a pair whose cdr is a proper list. An improper list is one containing a cdr which is not a list (see Table A.5).
It is an error to apply \texttt{car} or \texttt{cdr} or their \texttt{setter} functions

\textbf{A.12.8 cons} \hspace{1cm} \textit{function}

\textbf{A.12.8.1 Arguments}

\texttt{object}: Object to examine.

\textbf{A.12.8.2 Result}

Returns \texttt{object} if it is a pair, otherwise \texttt{()}.\n
\textbf{A.12.9 atom} \hspace{1cm} \textit{function}

\textbf{A.12.9.1 Arguments}

\texttt{object}: Object to examine.

\textbf{A.12.9.2 Result}

Returns \texttt{object} if it is not a pair, otherwise \texttt{()}.\n
\textbf{A.12.10 cons} \hspace{1cm} \textit{function}

\textbf{A.12.10.1 Arguments}

\texttt{object}_1: An object, pair.

\texttt{object}_2: An object, pair.

\textbf{A.12.10.2 Result}

Allocates a new pair whose slots are initialized with \texttt{object}_1
in the car and \texttt{object}_2 in the cdr.

\textbf{A.12.11 car} \hspace{1cm} \textit{function}

\textbf{A.12.11.1 Arguments}

\texttt{pair}: A pair.

\textbf{A.12.11.2 Result}

Given a pair, such as the result of \texttt{(cons \textit{object}_1 \textit{object}_2)},
then the function \texttt{car} returns \texttt{object}_1.

\textbf{A.12.12 cdr} \hspace{1cm} \textit{function}

\textbf{A.12.12.1 Arguments}

\texttt{pair}: A pair.

\textbf{A.12.12.2 Result}

Given a pair, such as the result of \texttt{(cons \textit{object}_1 \textit{object}_2)},
then the function \texttt{cdr} returns \texttt{object}_2.

\textbf{A.12.13 (setter car)} \hspace{1cm} \textit{setter}

\textbf{A.12.13.1 Arguments}

\texttt{pair}: A pair.

\texttt{object}: An object.

\textbf{A.12.13.2 Result}

Given a pair, such as the result of \texttt{(cons \textit{object}_1 \textit{object}_2)},
then the function \texttt{(setter car)} replaces \texttt{object}_1 with \texttt{object}.
The result is \texttt{object}.

\textbf{A.12.14 (setter cdr)} \hspace{1cm} \textit{setter}

\textbf{A.12.14.1 Arguments}

\texttt{pair}: A pair.

\texttt{object}: An object.

\textbf{A.12.14.2 Result}

Given a pair, such as the result of \texttt{(cons \textit{object}_1 \textit{object}_2)},
then the function \texttt{(setter cdr)} replaces \texttt{object}_2 with \texttt{object}.
The result is \texttt{object}.

\textbf{A.12.14.3 Remarks}

Note that if \texttt{object} is not a proper list, then
the use of \texttt{(setter cdr)} might change \texttt{pair} into an improper list.

\textbf{A.12.15 equal} \hspace{1cm} \textit{method}

\textbf{A.12.15.1 Specialized Arguments}

\texttt{pair}_1: A pair.

\texttt{pair}_2: A pair.

\textbf{A.12.15.2 Result}

The result is the conjunction of the pairwise application of \texttt{equal}
to the car fields and the cdr fields of the arguments.

\textbf{A.12.16 deep-copy} \hspace{1cm} \textit{method}

\textbf{A.12.16.1 Specialized Arguments}

\texttt{(pair <cons>)}: A pair.

\textbf{A.12.16.2 Result}

Constructs and returns a copy of the list starting at \texttt{pair}
copying both the car and the cdr slots of the list. The list
can be proper or improper. Treatment of the objects stored
in the car slot (and the cdr slot in the case of the final pair
of an improper list) is determined by the \texttt{deep-copy} method
for the class of the object.
A.12.17 shallow-copy method

A.12.17.1 Specialized Arguments

(pair <cons>): A pair.

A.12.17.2 Result

Constructs and returns a copy of the list starting at pair but copying only the cdr slots of the list, terminating when a pair is encountered whose cdr slot is not a pair. The list beginning at pair can be proper or improper.

A.12.18 list function

A.12.18.1 Arguments

[object₁ ... objectₙ]: A sequence of objects.

A.12.18.2 Result

Allocates a set of pairs each of which has been initialized with object in the car field and the pair whose car field contains objectᵢ₊₁ in the cdr field. Returns the pair whose car field contains object₁.

A.12.18.3 Examples

(list)  ⇒  ()
(list 1 2 3)  ⇒  (1 2 3)

A.12.19 generic-prin method

A.12.19.1 Specialized Arguments

(pair <cons>): The pair to be output on stream.

(stream <stream>): The stream on which the representation is to be output.

A.12.19.2 Result

The pair supplied as the first argument.

A.12.19.3 Remarks

Output the external representation of pair on stream as described at the beginning of this section. Uses generic-write to produce the external representation of the contents of the car and cdr slots of pair.

A.12.20 generic-write method

A.12.20.1 Specialized Arguments

(pair <cons>): The pair to be output on stream.

(stream <stream>): The stream on which the representation is to be output.


A.13 Numbers

The defined name of this module is number. Numbers can take on many forms with unusual properties, specialized for different tasks, but two classes of number suffice for the majority of needs, namely integers (A.11, A.9) and floating point numbers (A.8, A.6). Thus, these only are defined at level-0.

In Figure A.1 shows the initial number class hierarchy at level-0. The inheritance relationships by this diagram are part of this definition, but it is not defined whether they are direct or not. For example, <integer> and <float> are not necessarily direct subclasses of <number>, while the class of each number class might be a subclass of number-class. Since there are only two concrete number classes at level-0, coercion is simple, namely from <fixed-precision-integer> to <double-float>. Any level-0 version of a library module, for example, elementary functions, need only define methods for these two classes. Mathematically, the reals are regarded as a superset of the integers and for the purposes of this definition we regard <float> as a superset of <integer> (even though this will cause representation problems when variable precision integers are introduced). Hence, <float> is referred to as being higher that <integer> and arithmetic involving instances of both classes will cause integers to be converted to an equivalent floating point value, before the calculation proceeds1) (see in particular binary/, binary% and binary-mod).

Figure A.1 — Level-0 number class hierarchy

<number> [ <number-class> ]
<float> [ <number-class> ]
<double-float> [ <number-class> ]
<integer> [ <number-class> ]
<fixed-precision-integer> [ <number-class> ]

A.13.1 <number> class

The abstract class which is the superclass of all number classes.

A.13.2 number function

A.13.2.1 Arguments

object: Object to examine.

A.13.2.2 Result

Returns object if it is a number, otherwise ()

A.13.3 arithmetic-condition condition

A.13.3.1 Initialization Options

operator object: The operator which signalled the condition.

1) This behaviour is popularly referred to as floating point contagion

operator-list list: The operands passed to the operator.

A.13.3.2 Remarks

This is the general condition class for conditions arising from arithmetic operations.

A.13.4 division-by-zero arithmetic-condition

Signalled by any of binary/, binary% and binary-mod if their second argument is zero.

A.13.5 + function

A.13.5.1 Arguments

[number; number2 ... ]: A sequence of numbers.

A.13.5.2 Result

Computes the sum of the arguments using the generic function binary+. Given zero arguments, + returns 0 of class <integer>. One argument returns that argument. The arguments are combined left-associatively.

A.13.6 - function

A.13.6.1 Arguments

number; [ number2 ... ]: A non-empty sequence of numbers.

A.13.6.2 Result

Computes the result of subtracting successive arguments—from the second to the last—from the first using the generic function binary-. Zero arguments is an error. One argument returns the negation of the argument, using the generic function negate. The arguments are combined left-associatively.

A.13.7 * function

A.13.7.1 Arguments

[number; number2 ... ]: A sequence of numbers.

A.13.7.2 Result

Computes the product of the arguments using the generic function binary*. Given zero arguments, * returns 1 of class integer. One argument returns that argument. The arguments are combined left-associatively.

A.13.8 / function

A.13.8.1 Arguments

number; [ number2 ... ]: A non-empty sequence of numbers.
A.13.8.2 Result
Computes the result of dividing the first argument by its succeeding arguments using the generic function binary/. Zero arguments is an error. One argument computes the reciprocal of the argument. It is an error in the single argument case, if the argument is zero.

A.13.9 % function

A.13.9.1 Arguments

number1 [number2 ...]: A non-empty sequence of numbers.

A.13.9.2 Result
Computes the result of taking the remainder of dividing the first argument by its succeeding arguments using the generic function binary%. Zero arguments is an error. One argument returns that argument.

A.13.10 gcd function

A.13.10.1 Arguments

number1 [number2 ...]: A non-empty sequence of numbers.

A.13.10.2 Result
Computes the greatest common divisor of number1 up to numbern using the generic function binary-gcd. Zero arguments is an error. One argument returns number1.

A.13.11 lcm function

A.13.11.1 Arguments

number1 [number2 ...]: A non-empty sequence of numbers.

A.13.11.2 Result
Computes the least common multiple of number1 up to numbern using the generic function binary-lcm. Zero arguments is an error. One argument returns number1.

A.13.12 abs function

A.13.12.1 Arguments

number: A number.

A.13.12.2 Result
Computes the absolute value of number.

A.13.13 zero? generic function

A.13.13.1 Generic Arguments
number: A number.

A.13.13.2 Result
Compares number with the zero element of the class of number using the generic function binary=?.

A.13.14 negate generic function

A.13.14.1 Generic Arguments
(number <number>): A number.

A.13.14.2 Result
Computes the additive inverse of number.

A.13.15 signum function

A.13.15.1 Arguments
number: A number.

A.13.15.2 Result
Returns number if zero? applied to number is true. Otherwise returns the result of converting ±1 to the class of number with the sign of number.

A.13.16 positivep function

A.13.16.1 Arguments
number: A number.

A.13.16.2 Result
Compares number against the zero element of the class of number using the generic function binary<.

A.13.17 negativep function

A.13.17.1 Arguments
number: A number.

A.13.17.2 Result
Compares number against the zero element of the class of number using the generic function binary<.
A.13.18 binary+ generic function

A.13.18.1 Generic Arguments

(number1 <number>): A number.
(number2 <number>): A number.

A.13.18.2 Result
Computes the sum of number1 and number2.

A.13.19 binary- generic function

A.13.19.1 Generic Arguments

(number1 <number>): A number.
(number2 <number>): A number.

A.13.19.2 Result
Computes the difference of number1 and number2.

A.13.20 binary* generic function

A.13.20.1 Generic Arguments

(number1 <number>): A number.
(number2 <number>): A number.

A.13.20.2 Result
Computes the product of number1 and number2.

A.13.21 binary/ generic function

A.13.21.1 Generic Arguments

(number1 <number>): A number.
(number2 <number>): A number.

A.13.21.2 Result
Computes the division of number1 by number2 expressed as a number of the class of the higher of the classes of the two arguments. The sign of the result is positive if the signs of the arguments are the same. If the signs are different, the sign of the result is negative. If the second argument is zero, the result might be zero or an error might be signalled (condition class: division-by-zero).

A.13.22 binary% generic function

A.13.22.1 Generic Arguments

(number1 <number>): A number.
(number2 <number>): A number.

A.13.23 binary-mod generic function

A.13.23.1 Generic Arguments

(number1 <number>): A number.
(number2 <number>): A number.

A.13.23.2 Result
Computes the largest integral value not greater than number1 number2 expressed as a number of the class of the higher of the classes of the two arguments, such that if number2 is non-zero, the result has the same sign as number2 and magnitude less than the magnitude of number2. If the second argument is zero, the result might be zero or an error might be signalled (condition class: division-by-zero).

A.13.24 binary-gcd generic function

A.13.24.1 Generic Arguments

(number1 <number>): A number.
(number2 <number>): A number.

A.13.24.2 Result
Computes the greatest common divisor of number1 and number2.

A.13.25 binary-lcm generic function

A.13.25.1 Generic Arguments

(number1 <number>): A number.
(number2 <number>): A number.

A.13.25.2 Result
Computes the lowest common multiple of number1 and number2.
A.14 Streams

The defined name of this module is stream.

Level-0 streams only offer the most basic functionality, supporting character input and output on files via <char-file-stream> and character input and output on strings via <string-stream>. This functionality is implemented by the two generic functions input and output.

The behaviour at end-of-stream is governed by an option to open. This option takes a cos-action, which is a continuation, or some function of one argument, as its value. Upon attempting to read beyond the end of stream the value of the stream’s cos-action is called with the stream as its argument, and the result of the call of cos-action is returned as the value of the input operation. This allows special actions to be performed, special values to be returned, or transfers of control via continuations. The default action is signal an error (condition class: end-of-stream).

A.14.1 <stream> class

The abstract class of streams.

A.14.2 streamp generic function

A.14.2.1 Generic Arguments

(object <object>); The object to be examined.

A.14.2.2 Result

Returns object if it is a stream, otherwise ()

A.14.3 file-streamp generic function

A.14.3.1 Generic Arguments

(object <object>); The object to be examined.

A.14.3.2 Result

Returns object if it is a stream held on some external medium, otherwise ()

A.14.4 character-streamp generic function

A.14.4.1 Generic Arguments

(object <object>); The object to be examined.

A.14.4.2 Result

Returns object if it is a character stream, otherwise ()

A.14.5 <char-file-stream> class

The class of the default character file stream. Often instances of this stream type are positionable. It is an error to make an instance of this class by any other means than by the use of open.

A.14.5.1 See also: input, output methods on <char-file-stream>.

A.14.6 <string-stream> class

The class of the default string stream.

A.14.6.1 See also: The converter method (A.14.27) for <string-stream> to <string>.

A.14.7 standard-input-stream function

This function takes no arguments.

A.14.7.1 Result

A stream, which is the standard input stream.

A.14.7.2 Remarks

This stream is often the default for input functions.

A.14.8 standard-output-stream function

This function takes no arguments.

A.14.8.1 Result

A stream, which is the standard output stream.

A.14.8.2 Remarks

This stream is often the default for output functions.

A.14.9 standard-error-stream function

This function takes no arguments.

A.14.9.1 Result

A stream, which is the standard error stream.

A.14.10 open function

A.14.10.1 Arguments

name: A string.
[options]: A sequence of options.

A.14.10.2 Result
A stream.

A.14.10.3 Remarks
Options are input, output, update, append, and eos-action.

If no options are specified input is assumed and the eos-action will signal an error (condition class: end-of-stream).

The eos-action option takes a stream action (a continuation or a function of one argument) as an argument. Upon an attempt to read beyond the end of the stream this stream action is called with the stream as argument, and the value returned by the stream action is the value returned by the reading operation. If the eos-action is omitted, the default stream action is to signal an error: (condition class: end-of-stream).

It is an error to combine the input option with output, update or append. It is an error to combine the output option with input.

A.14.10.4 Examples
(open "foo")
Open the file foo for input.

(open "bar" 'output 'append)
Open bar for output, appending to its end.

(open "fik-c"
  'eos-action
  (lambda (s) ;
    ((setter stream-position) s 0)
    (read s)))
A self-resetting stream

A.14.11 end-of-stream stream-condition

A.14.11.1 Initialization Options
(stream <stream>): A stream.

A.14.11.2 Remarks
Signalled by the default end of stream action, as a consequence of an input operation on stream, when it is at end of stream.

A.14.12 close generic function

A.14.12.1 Generic Arguments
(stream <stream>): A stream.

A.14.12.2 Result
Returns t if the close operation was successful, otherwise ()

A.14.13 flush generic function

A.14.13.1 Generic Arguments
(stream <stream>): A stream.

A.14.13.2 Result
Returns t if the flush operation was successful, otherwise ()

A.14.13.3 Remarks
flush causes any buffered data for the stream to be written to the stream. The stream remains open.

A.14.14 stream-position generic function

A.14.14.1 Generic Arguments
(stream <stream>): A stream.

A.14.14.2 Result
Returns a non-negative integer indicating the current position within the stream, or () if it cannot be determined.

A.14.14.3 Remarks
The value returned by stream-position increases monotonically as input/output operations are performed. Generally, this value may be used in a call of (setter stream-position) on the same stream.

A.14.15 (setter stream-position) generic function

A.14.15.1 Generic Arguments
(stream <stream>): A stream.

(position <object>): An integer, or the symbol stream-end.

A.14.15.2 Result
Returns t if the positioning operation was successful, otherwise ()

A.14.15.3 Remarks
If position is an integer, this sets the position of the stream to be that value. If position is stream-end, this sets the position of the stream to be the end of the stream. The value () is returned if the stream is not positionable. If the value of position is too large, or is otherwise inappropriate, an error is signalled (condition class: inappropriate-stream-position).
Programming Language EuLisp, version 0.99

A.14.16 end-of-stream-p function

A.14.16.1 Arguments

stream: A stream.

A.14.16.2 Result

Returns t if stream is exhausted, otherwise t.

A.14.17 input generic function

A.14.17.1 Generic Arguments

(stream <stream>): A stream.

A.14.17.2 Result

Reads and returns a single object from the stream. The type of the object returned depends on the type of the stream.

A.14.17.3 Remarks

This module defines two methods on input, one for character file streams and one for string streams. Both methods return characters.

A.14.18 uninput generic function

A.14.18.1 Generic Arguments

(stream <stream>): A stream.

A.14.18.2 Result

Returns t.

A.14.18.3 Remarks

Replaces object back onto the input stream. It is an error if the object was not the last thing to be returned by input.

This module defines two methods on uninput, one for character file streams and one for string streams. Both methods specialize on characters for the second argument.

A.14.19 output generic function

A.14.19.1 Generic Arguments

(stream <stream>): A stream.

A.14.19.2 Result

Returns object.

A.14.19.3 Remarks

Outputs the object to stream.

This module defines four methods on output, two for character file streams and two for string streams. In each case the methods specialize on characters or sequences.

A.14.20 read-line generic function

A.14.20.1 Generic Arguments

(stream <stream>): A stream.

A.14.20.2 Result

A string.

A.14.20.3 Remarks

Reads a line (terminated by a newline character or the end of the stream) from the stream of characters which is stream. Returns the line as a string, discarding the terminating newline, if any. If the stream is already at end of stream, then the stream action is called: the default stream action is to signal an error: (condition class: end-of-stream).

A.14.21 prin function

A.14.21.1 Arguments

object: An object to be output on stream.

(stream): A character stream on which object is to be output.

A.14.21.2 Result

Returns object.

A.14.21.3 Remarks

Outputs the external representation of object on the output stream stream using generic-prin. If stream is not specified the standard output stream is used.

Also: standard-output-stream, generic-prin.

A.14.22 print function

A.14.22.1 Arguments

object: An object to be output on stream.

(stream): A character stream on which object is to be output.

A.14.22.2 Result

Returns object.
A.14.22.3 Remarks
Outputs the external representation of object on the output stream stream using generic-prin, followed by a newline character. If stream is not specified the standard output stream is used.

A.14.25.4 also: standard-output-stream, generic-prin.

A.14.23 write function

A.14.23.1 Arguments

object: An object to be output on stream.

[stream]: A character stream on which object is to be output.

A.14.23.2 Result
Returns object.

A.14.23.3 Remarks
Outputs the external representation of object on the output stream stream using generic-write. If stream is not specified the standard output stream is used.

A.14.25.4 also: standard-output-stream, generic-write.

A.14.24 newline function

A.14.24.1 Arguments

[stream]: A stream on which the newline is to be output.

A.14.24.2 Result
Returns \#\n.

A.14.24.3 Remarks
Outputs a newline character on stream. If stream is not specified the standard output stream is used.

A.14.24.4 See also: standard-output-stream.

A.14.25 generic-prin generic function

A.14.25.1 Generic Arguments

(object <object>): An object to be output on stream.

(stream <stream>): A stream on which object is to be output.

A.14.25.2 Result
Returns object.

A.14.25.3 Remarks
Outputs the external representation of object on the output stream stream.

A.14.25.4 See also: methods on generic-prin are defined for <character> (A.1), <double-float> (A.6), <fixed-precision-integer> (A.9), <list> (A.12), <string> (A.15), <symbol> (A.16) and <vector> (A.18).

A.14.26 generic-write generic function

A.14.26.1 Generic Arguments

(object <object>): An object to be output on stream.

(stream <stream>): A stream on which object is to be output.

A.14.26.2 Result
Returns object.

A.14.26.3 Remarks
Outputs the external representation of object on the output stream stream.

A.14.26.4 See also: Methods on generic-write are defined for <character> (A.1), <double-float> (A.6), <fixed-precision-integer> (A.9), <list> (A.12), <string> (A.15), <symbol> (A.16) and <vector> (A.18).

A.14.27 (converter <string>) method

A.14.27.1 Specialized Arguments

(stream <string-stream>): A string stream.

A.14.27.2 Result
The current value of the string associated with stream.

A.14.27.3 Remarks
The current value is reset to the empty string after the call to converter.
A.15 Strings

The defined name of this module is string. See also section A.2 (collections) for further operations on strings.

### A.15.1 string syntax

String literals are delimited by the glyph called "quotation mark ("). For example, "abcd".

Sometimes it might be desirable to include string delimiter characters in strings. The aim of escaping in strings is to fulfill this need. The *string-escape* glyph is defined as reverse solidus (\). String escaping can also be used to include certain other characters that would otherwise be difficult to denote. The set of named special characters (see sections 8 and A.1) have a particular syntax to allow their inclusion in strings. To allow arbitrary characters to appear in strings, the hex-insertion diagram is followed by an integer denoting the position of the character in the current character set. Some examples of string literals appear in Table A.6. The syntax for the external representation of strings is defined in Table A.7.

NOTE — At present this document refers to the "current character set" but defines no means of selecting alternative character sets. This is to allow for future extensions and implementations defined extensions which support more than one character set.

#### Table A.6 — Examples of string literals

<table>
<thead>
<tr>
<th>Example</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;a\nb&quot;</td>
<td>\a, \newline and \b</td>
</tr>
<tr>
<td>&quot;c\n&quot;</td>
<td>\c and \n</td>
</tr>
<tr>
<td>&quot;\x1 &quot;</td>
<td>\x1 followed by \space</td>
</tr>
<tr>
<td>&quot;\xabcde&quot;</td>
<td>\xabcde followed by \e</td>
</tr>
<tr>
<td>&quot;\x1x2&quot;</td>
<td>\x1x2 followed by \x2</td>
</tr>
<tr>
<td>&quot;\xabcg&quot;</td>
<td>\xabc followed by \g</td>
</tr>
<tr>
<td>&quot;\x00abc&quot;</td>
<td>\x00abc followed by \c</td>
</tr>
</tbody>
</table>

The function write outputs a re-readable form of any escaped characters in the string. For example, "a\n\b" (input notation) is the string containing the characters \newline, \a, \b and \b. The function write produces "a\n\b", whilst prin produces

```
  a
  \b
```

The function write outputs characters which do not have a glyph associated with their position in the character set as a hex insertion in which all four hex digits are specified, even if there are leading zeros, as in the last example in Table A.6. The function prin outputs the interpretation of the characters according to the definitions in section A.1 without the delimiting quotation marks.

### A.15.2 <string> class

The class of all instances of <string>.

#### A.15.2.1 Initialization Options

#### A.15.3 string function

**Arguments**

- **object**: Object to examine.

**Returns**

Returns *object* if it is a string, otherwise ( ).

#### A.15.4 (converter <symbol>) method

**Specialized Arguments**

- **(string <string>)**: A string to be converted to a symbol.
A.15.4.2 Result
If the result of symbol-exists-p when applied to string is a symbol, that symbol is returned. If the result is (), then a new symbol is constructed whose name is string. This new symbol is returned.

A.15.5 equal method

A.15.5.1 Specialized Arguments

(string1 <string2>): A string.

(string2 <string>): A string.

A.15.5.2 Result
If the size of string1 is the same (under =) as that of string2, then the result is the conjunction of the pairwise application of equal to the elements of the arguments. If not the result is ().

A.15.6 deep-copy method

A.15.6.1 Specialized Arguments

(string <string>): A string.

A.15.6.2 Result
Constructs and returns a copy of string in which each element is eq1 to the corresponding element in string.

A.15.7 shallow-copy method

A.15.7.1 Specialized Arguments

(string <string>): A string.

A.15.7.2 Result
Constructs and returns a copy of string in which each element is eq1 to the corresponding element in string.

A.15.8 binary< method

A.15.8.1 Specialized Arguments

(string1 <string2>): A string.

(string2 <string>): A string.

A.15.8.2 Result
If the second argument is longer than the first, the result is O. Otherwise, if the sequence of characters in string2 is pairwise less than that in string1 according to binary< the result is t. Otherwise the result is (). Since it is an error to compare lower case, upper case and digit characters with any other kind than themselves, so it is an error to compare two strings which require such comparisons and the results are undefined.

A.15.8.3 Examples

(< "a" "b") \quil t
(< "b" "a") \quil ()
(< "a" "a") \quil ()
(< "a" "ab") \quil t
(< "ab" "a") \quil ()
(< "a" "B") \quil t
(< "0" "1") \quil t
(< "a1" "a2") \quil t
(< "a1" "bb") \quil t
(< "a1" "ab") \quil undefined

A.15.8.4 See also: Method on binary< (A.3) for characters (A.1).

A.15.9 as-lowercase method

A.15.9.1 Specialized Arguments

(string <string>): A string.

A.15.9.2 Result
Returns a copy of string in which each character denoting an upper case character, is replaced by a character denoting its lower case counterpart. The result must not be eq to string.

A.15.10 as-uppercase method

A.15.10.1 Specialized Arguments

(string <string>): A string.

A.15.10.2 Result
Returns a copy of string in which each character denoting an lower case character, is replaced by a character denoting its upper case counterpart. The result must not be eq to string.

A.15.11 generic-prin method

A.15.11.1 Specialized Arguments

(string <string>): String to be output on stream.

(stream <stream>): Stream on which string is to be output.

A.15.11.2 Result
The string string.

Output external representation of string on stream as described in the introduction to this section, interpreting each of the characters in the string. The opening and closing quotation marks are not output.
A.15.12 generic-write method

A.15.12.1 Specialized Arguments

\[ \text{(string <string>)}: \text{String to be output on stream.} \]

\[ \text{(stream <stream>)}: \text{Stream on which string is to be output.} \]

A.15.12.2 Result

The string \text{string}.

Output external representation of \text{string} on \text{stream} as described in the introduction to this section, replacing single characters with escape sequences if necessary. Opening and closing quotation marks are output.

A.16 Symbols

The defined name of this module is symbol.

A.16.1 symbol syntax

The syntax of symbols also serves as the syntax for identifiers. Identifiers in EuLisp are very similar lexically to identifiers in other Lisps and in other programming languages. Informally, an identifier is a sequence of alphabetic, digit and other characters starting with a character that is not a digit. Characters which are special (see section 8) must be escaped if they are to be used in the names of identifiers. However, because the common notations for arithmetic operations are the glyphs for plus (+) and minus (-), which are also used to indicate the sign of a number, these glyphs are classified as identifiers in their own right as well as being part of the syntax of a number.

Sometimes, it might be desirable to incorporate characters in an identifier that are normally not legal constituents. The aim of escaping in identifiers is to change the meaning of particular characters so that they can appear where they are not otherwise acceptable. Identifiers containing characters that are not ordinarily legal constituents can be written by delimiting the sequence of characters by multiple-escape, the glyph for which is called vertical bar (|). The multiple-escape denotes the beginning of an escaped part of an identifier and the next multiple-escape denotes the end of an escaped part of an identifier. A single character that would otherwise not be a legal constituent can be written by preceding it with single-escape, the glyph for which is called reverse solidus (\). Therefore, single-escape can be used to incorporate the multiple-escape or the single-escape character in an identifier, delimited (or not) by multiple-escapes. For example, \text{1/(1/2/3)\(a/b\) is the identifier whose name contains the three characters \#1, \#1, and \#1, and \#a and \#b is the identifier whose name contains the characters \#1a and \#1b. The sequence \#1 is the identifier with no name, and so is \#11, but \#11 is the identifier whose name contains the single character \#1, which can also be written \#1 without delimiting multiple-escapes.

Any identifier can be used as a literal, in which cases it denotes a symbol. Because there are two escaping mechanisms and because the first character of a token affects the interpretation of the remainder, there are many ways in which to input the same identifier. If this same identifier is used as a literal the results of processing each token denoting the identifier will be eq to one another. For example, the following tokens all denote the same symbol:

\[ \text{123, 1|2, 1|23, 1|23, |1|123} \]

which will be output by the function write as \text{123}. If output by write, the representation of the symbol will permit reconstruction by read—escape characters are preserved—so that equivalence is maintained between read and write for symbols. For example: \text{1a(b) and abc.def} are two symbols as output by write such that read can read them as two symbols. If output by print, the escapes necessary to re-read the symbol will not be included. Thus, taking the same examples, print outputs \text{a(b) and abc.def}, which read interprets as the symbol a followed by the start of a list, the symbol b and the symbol abc.def.

The syntax of the external representation of identifiers and
Table A.8 — Identifier/Symbol Syntax

```
symbol
  = identifier;

identifier
  = normal identifier | peculiar identifier;

normal identifier
  = normal initial, (normal constituent);

normal initial
  = letter (S * ) | other character (S * )

normal constituent
  = letter (S * ) | digit (S * ) | other character (S * )

peculiar identifier
  = ('+' | '-'), [peculiar constituent, (normal constituent)];

peculiar constituent
  = letter (S * ) | other character (S * )
```
symbol is defined in Table A.8.

Computationally, the most important aspect of symbols is that each is unique, or, stated the other way around: the result of processing every syntactic token comprising the same sequence of characters which denote an identifier is the same object. Or, more briefly, every identifier with the same name denotes the same symbol.

A.16.2 <symbol> class

The class of all instances of <symbol>.

A.16.2.1 Initialization Options

string string: The string containing the characters to be used to name the symbol. The default value for string is the empty string, thus resulting in the symbol with no name, written NIL.

A.16.3 symbolp function

A.16.3.1 Arguments

object: Object to examine.

A.16.3.2 Result

Returns object if it is a symbol.

A.16.4 gensym function

A.16.4.1 Arguments

[string]: A string containing characters to be prepended to the name of the new symbol.

A.16.5 symbol-name function

A.16.5.1 Arguments

symbol: A symbol.

A.16.5.2 Result

Returns a string which is equal to that given as the argument to the call to make which created symbol. It is an error to modify this string.

A.16.6 symbol-exists-p function

A.16.6.1 Arguments

string: A string containing the characters to be used to determine the existence of a symbol with that name.

A.16.6.2 Result

Returns the symbol whose name is string if that symbol has already been constructed by make. Otherwise, returns NIL.

A.16.7 generic-prin method

A.16.7.1 Specialized Arguments

(symbol <symbol>): The symbol to be output on stream.

(stream <stream>): The stream on which the representation is to be output.

A.16.7.2 Result

The symbol supplied as the first argument.

A.16.7.3 Remarks

Outputs the external representation of symbol on stream as described in the introduction to this section, interpreting each of the characters in the name.

A.16.8 generic-write method

A.16.8.1 Specialized Arguments

(symbol <symbol>): The symbol to be output on stream.

(stream <stream>): The stream on which the representation is to be output.
A.16.8.2 Result
The symbol supplied as the first argument.

A.16.8.3 Remarks
Outputs the external representation of symbol on stream as described in the introduction to this section. If any characters in the name would not normally be legal constituents of an identifier or symbol, the output is preceded and succeeded by multiple-escape characters.

A.16.8.4 Examples

```
(write (make <symbol> 'string "abc") )  ⇒  abc
(write (make <symbol> 'string "a c")  )  ⇒  [a c]
(write (make <symbol> 'string ""." )  )  ⇒  [ ].
```

A.16.9 (converter <string>)  method

A.16.9.1 Specialized Arguments

```
(s symbol <symbol>): A symbol to be converted to a string.
```

A.16.9.2 Result
A string.

A.16.9.3 Remarks
This function is the same as symbol-name. It is defined for the sake of symmetry.

A.17 Tables
The defined name of this module is table. See also section A.2 (collections) for further operations on tables.

A.17.1 <table>  class

The class of all instances of <table>.

A.17.1.1 Initialization Options

```
comparator <function>: The function to be used to compare keys. The default comparison function is eql.
fill-value <object>: An object which will be returned as the default value for any key which does not have an associated value. The default fill value is ()
hash-function <function>: The function to be used to compute an unique key for each object stored in the table. This function must return a fixed precision integer. The hash function must also satisfy the constraint that if the comparison function returns true for any two objects, then the hash function must return the same key when applied to these two objects. The default is an implementation defined function which satisfies these conditions.
```

A.17.2 tablep  function

A.17.2.1 Arguments

```
object: Object to examine.
```

A.17.2.2 Result
Returns object if it is a table, otherwise 0.

A.17.3 clear-table  function

A.17.3.1 Arguments

```
table: A table.
```

A.17.3.2 Result
An empty table.

A.17.3.3 Remarks
All entries in table are deleted. The result is eq to the argument, which is to say that the argument is modified.
A.18 Vectors

The defined name of this module is vector. See also section A.2 (collections) for further operations on vectors.

A.18.1 vector syntax

A vector is written as #(obj ... objn). For example: #(1 2 3) is a vector of three elements, the integers 1, 2 and 3. The representations of obji are determined by the external representations defined in other sections of this definition (see <character> (A.1), <fixed-precision-integer> (A.9), <float> (A.8), <list> (A.12), <string> (A.15) and <symbol> (A.16), as well as instances of <vector> itself. The syntax for the external representation of vectors is defined in Table A.9.

Table A.9 — Vector Syntax

| vector = | #(, object) | |

A.18.2 <vector> class

The class of all instances of <vector>.

A.18.2.1 Initialization Options

size <fixed-precision-integer>: The number of elements in the vector. Vectors are zero-based and thus the maximum index is size-1. If not supplied the size is zero.

fill-value <object>: An object with which to initialize the vector. The default fill value is ()

A.18.2.2 Examples

(make <vector>) ⇒ ()
(make <vector> 'size 2) ⇒ #(())
(make <vector> 'size 3) ⇒ #(\a \a \a)
'fill-value \a)

A.18.3 vectorp function

A.18.3.1 Arguments

object: Object to examine.

A.18.3.2 Result

Returns object if it is a vector, otherwise ()

A.18.4 maximum-vector-index integer

A.18.4.1 Remarks

This is an implementation-defined constant. A conforming processor must support a maximum vector index of at least 32767.

A.18.5 equal method

A.18.5.1 Specialized Arguments

(vector <vector>): A vector.

(vector2 <vector>): A vector.

A.18.5.2 Result

If the size of vector1 is the same (under =) as that of vector2, then the result is the conjunction of the pairwise application of equal to the elements of the arguments. If not the result is ()

A.18.6 deep-copy method

A.18.6.1 Specialized Arguments

(vector <vector>): A vector.

A.18.6.2 Result

Constructs and returns a copy of vector, in which each element is the result of calling deep-copy on the corresponding element of vector.

A.18.7 shallow-copy method

A.18.7.1 Specialized Arguments

(vector <vector>): A vector.

A.18.7.2 Result

Constructs and returns a copy of vector in which each element is eql to the corresponding element in vector.

A.18.8 generic-prin method

A.18.8.1 Specialized Arguments

(vector <vector>): A vector to be output on stream.

(stream <stream>): A stream on which the representation is to be output.

A.18.8.2 Result

The vector supplied as the first argument.

A.18.8.3 Remarks

Output the external representation of vector on stream as described in the introduction to this section. Calls the generic function again to produce the external representation of the elements stored in the vector.
A.18.0 generic-write

A.18.0.1 Specialized Arguments

(vector <vector>): A vector to be output on stream.

(stream <stream>): A stream on which the representation is to be output.

A.18.0.2 Remarks

Output the external representation of vector on stream as described in the introduction to this section. Calls the generic function again to produce the external representation of the elements stored in the vector.

A.19 Syntax of Level-0 objects

This section repeats the syntax for reading and writing of the various classes defined in Annex A.

```
(object
  - character (* A.1 *)
  | float (* A.8 *)
  | integer (* A.11 *)
  | list (* A.12 *)
  | string (* A.13 *)
  | symbol (* A.16 *)
  | vector (* A.18 *)

(character token
  - literal character token
  | special character token
  | control character token
  | numeric character token;)

(literal character token
  - '\', letter
  | '\', decimal digit
  | '\', non-alphabetic;

(control character token
  - '\', letter;

(special character token
  - '\', letter
  | '\', backspace
  | '\', delete
  | '\', formfeed
  | '\', linefeed
  | '\', newline
  | '\', return
  | '\', tab
  | '\', space
  | '\', vertical-tab;

(numeric character token
  - '\', hex digit
  | '\', hex digit, hex digit
  | '\', hex digit, hex digit, hex digit;

(float
  - [sign] unsigned float [exponent];

(sign
  = :+ | :-;

(unsigned float
  = float format 1
  | float format 2
  | float format 3;

(float format 1
  = decimal integer, \?, (* A.11 *)

(float format 2
  = \?, decimal integer; (* A.11 *)

(float format 3
  = float format 1, decimal integer; (* A.11 *)

(exponent
  = double exponent; (* A.6 *)

(double exponent
  = (\d | \D), [sign], decimal integer; (* A.11 *)
```
Programming Language EuLisp, version 0.99

integer = [sign] unsigned integer;

sign = '+\#| '-\#| ;

unsigned integer
  = binary integer
    | octal integer
    | decimal integer
    | hexadecimal integer
    | specified base integer

binary integer
  = '#b', binary digit, {binary digit};

binary digit
  = '0'|'1';

octal integer
  = '#o', octal digit, {octal digit};

octal digit
  = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7';

decimal integer
  = decimal digit, {decimal digit};

hexadecimal integer
  = '#x', hexadecimal digit, {hexadecimal digit};

hexadecimal digit
  = decimal digit
    | hex lower letter
    | hex upper letter;

hex lower letter
  = 'a'|'b'|'c'|'d'|'e'|'f';

hex upper letter
  = 'A'|'B'|'C'|'D'|'E'|'F';

specified base integer
  = '#', base specification, 'r', specified base digit, {specified base digit};

base specification
  = decimal digit - ('0'|'1')
    | ('2'|'3'), decimal digit
    | ('4'|'5'|'6'), decimal digit
    | ('7'|'8'), decimal digit
    | specified base digit

specified base digit
  = decimal digit | letter;

null = '(, )';

pair = '(, object, ?, object, ?)';

list = empty list | proper list | improper list;

empty list = '()';

proper list = '(?, [object], ?)';

improper list = '(?, [object], ?, ?, object, ?)';

string token = "'(string constituent) '";

string constituent
  = normal string constituent
    | digram string constituent
    | numeric string constituent;

normal string constituent
  = level 0 character - '(', '/\'\', )

digram string constituent
  = '\a' (* alert *)
    | '\b' (* backspace *)
    | '\d' (* delete *)
    | '\f' (* formfeed *)
    | '\n' (* newline *)
    | '\r' (* return *)
    | '\t' (* tab *)
    | '\v' (* vertical tab *)
    | '\"' (* string delimiter *)
    | '\\' : (* string escape *)

numeric string constituent
  = '\x' hex digit
    | '\x' hex digit, hex digit
    | '\x' hex digit, hex digit, hex digit
    | '\x' hex digit, hex digit, hex digit, hex digit;

symbol = identifier;

identifier
  = normal identifier | peculiar identifier;

normal identifier
  = normal initial, {normal constituent};

normal initial
  = letter (+ S *)
    | other character - ' '|�-'�; (+ S *)

normal constituent
  = letter (+ S *)
    | digit (+ S *)
    | other character; (+ S *)

peculiar identifier
  = '(', [peculiar constituent], [normal constituent]);

peculiar constituent
  = letter (+ S *)
    | other character; (+ S *)

vector
  = '#(' [object], ?)';
Annex B
(normative)

Programming Language EuLisp, Level-1

B.1 Classes and Objects

B.1.1 defclass defining form

<table>
<thead>
<tr>
<th>defclass form</th>
</tr>
</thead>
<tbody>
<tr>
<td>= (', 'defclass',</td>
</tr>
<tr>
<td>class name, (* 10.3.1 *)</td>
</tr>
<tr>
<td>superclass list,</td>
</tr>
<tr>
<td>defclass slot description list,</td>
</tr>
<tr>
<td>(defclass class option), ')')</td>
</tr>
<tr>
<td>superclass list</td>
</tr>
<tr>
<td>= (', (superclass name), ')')</td>
</tr>
<tr>
<td>defclass slot option</td>
</tr>
<tr>
<td>= (', initform, level 1 expression</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>defclass class option</td>
</tr>
<tr>
<td>= (', class name</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

B.1.2 Arguments

class-name: A symbol naming a binding to be initialised with the new class.

(superclass*): A list of symbols naming bindings of classes to be used as the superclasses of the new class. This is different from defstruct at level-0, where only one superclass may be specified.

(slot-description*): A list of slot specifications (see below), comprising either a slot-name or a list of slot-name followed by some slot-options. One of the class options (see below) allows the specification of the class of the slot description.

class-option*: A sequence of keys and values (see below). One of the class options (class) allows the specification of the class of the class being defined.

B.1.3 Remarks

This defining form defines a new class. The resulting class will be bound to class-name. The second argument is a list of superclasses. If this list is empty, the superclass is <object>. The third argument is a list of slot-descriptions. The remaining arguments are class options. All the slot options and class options are exactly the same way as for defstruct (10.3.1).

The slot-options are interpreted as follows:

initarg symbol: The value of this option is a identifier naming a symbol, which is the name of an argument to be supplied in the init-options of a call to make on the new class. The value of this argument in the call to make is the initial value of the slot. This option must only be specified once for a particular slot. The same initarg name may be used for several slots, in which case they will share the same initial value if the initarg is given to make.

initform form: The value of this option is a form, which is evaluated as the default value of the slot, to be used if no initarg is defined for the slot or given to a call to make. The form is evaluated in the lexical environment of the call to defstruct and the dynamic environment of the call to make. The form is evaluated each time make is called and the default value is called for. The order of evaluation of the initforms in all the slots is determined by initialize. This option must only be specified once for a particular slot.

reader symbol: The value is the identifier of the variable to which the reader function will be bound. The reader function is a means to access the slot. The reader function is a function of one argument, which should be an instance of the new class. No writer function is automatically bound with this option. This option can be specified more than once for a slot, creating several readers. It is an error to specify the same reader, writer, or accessor name for two different slots.

writer symbol: The value is the identifier of the variable to which the writer function will be bound. The writer function is a means to change the slot value. The creation of the writer is analogous to that of the reader function. This option can be specified more than once for a slot. It is an error to specify the same reader, writer, or accessor name for two different slots.

accessor symbol: The value is the identifier of the variable to which the reader function will be bound. In addition, the use of this slot-option causes that the writer function is associated to the reader via the setter mechanism. This option can be specified more than once for a slot. It is an error to specify the same reader, writer, or accessor name for two different slots.

identifier expression: The symbol named by identifier and the value of expression are passed to make of the slot description class along with other slot options. The values are evaluated in the lexical and dynamic environment of the defclass. For the language defined slot description classes, no slot initargs are defined which are not specified by particular defclass slot options.

The class-options are interpreted as follows:

initargs list: The value of this option is a list of identifiers naming symbols, which extend the inherited names of arguments to be supplied in the init-options of a call to make on the new class. Initargs are inherited by union. The values of all legal arguments in the call to make are the initial values of corresponding slots if they name a slot or are ignored by the default initialize method, otherwise. This option must only be specified once for a class.

constructor constructor-spec: Creates a constructor
function for the new class. The constructor specification gives the name to which the constructor function will be bound, followed by a sequence of legal initargs for the class. The new function creates an instance of the class and fills in the slots according to the match between the specified initargs and the given arguments to the constructor function. This option may be specified more than once.

**Predicate symbol**: Creates a predicate function for the new class. The predicate specification gives the name to which the predicate function will be bound. This option may be specified any number of times for a class.

**Class**: The value of this option is the class of the new class. By default, this is `<class>`. This option must only be specified once for the new class.

**Identifier expression**: The symbol named by `identifier` and the value of `expression` are passed to `make` on the class of the new class. This list is appended to the end of the list that `defclass` constructs. The values are evaluated in the lexical and dynamic environment of the `defclass`. This option is used for metaclasses which need extra information not provided by the standard options.

### B.2 Generic Functions

#### B.2.1 Generic Lambda

**Macro**

**Syntax**

```
generic lambda form
  = '(? (\generic-lambda\), gf lambda list, (level 1 init option), ?)?

level 1 init option
  = '\class\', class name
  | '\method-class\', class name
  | '\method\' level 1 method description
  | identifier, level 1 expression;
  | level 0 init option;
level 1 method description
  = '(?, (method init option), specialized lambda list, (* 10.4.1 *)
(fom), ?)

method init option
  = '\class\', class name
  | identifier level 1 expression
```

**Arguments**

- **gf-lambda-list**: As level-0. See section 10.4.1.
- **level-1-init-option**: Format as level-0, but with additional options, which are defined below.

**Result**

A generic function.

**Remarks**

The syntax of `generic-lambda` is an extension of the level-0 syntax allowing additional init-options. These allow the specification of the class of the new generic function, which defaults to `<generic-function>`, the class of all methods, which defaults to `<method>`, and non-standard options. The latter are evaluated in the lexical and dynamic environment of `generic-lambda` and passed to `make` of the generic function as additional initialization arguments. The additional init-options over level-0 are interpreted as follows:

- **class gf-class**: The class of the new generic function. This must be a subclass of `<generic-function>`. The default is `<generic-function>`.
- **method-class method-class**: The class of all methods to be defined on this generic function. All methods of a generic function must be instances of this class. The `<method-class>` must be a subclass of `<method>` and defaults to `<method>`.
- **identifier expression**: The symbol named by `identifier` and the value of `expression` are passed to `make` as initargs. The values are evaluated in the lexical and dynamic environment of the `defgeneric`. This option is used for classes which need extra information not provided by the standard options.

In addition, method init options can be specified for the individual methods on a generic function. These are interpreted as follows:

- **class method-class**: The class of the method to be defined. The method class must be a subclass of `<method>` and, by default, `<method>`. The value is passed to `make` as the first argument. The symbol and the value are not passed as initargs to `make`.
- **identifier expression**: The symbol named by `identifier` and the value of `expression` are passed to `make` creating a new method as initargs. The values are evaluated in the lexical and dynamic environment of the `generic-lambda`. This option is used for classes which need extra information not provided by the standard options.

#### B.2.1.5 Examples

In the following example an anonymous version of `gf-1` (see `defgeneric`) is defined. All other respects the resulting object is the same as `gf-1`.

```
generic lambda (arg1 (arg2 <class-a>))
class <another-gf-class>
class-key a class-value a
class-key b class-value b
method-class <another-method-class-a>
method (class <another-method-class-b>)
  method-class-b-key a method-class-b-value a
  ((m1-arg1 <class-b>) (m1-arg2 <class-c>))
  ...
method (method-class-s-key a method-class-s-value a
  ((m2-arg1 <class-d>) (m2-arg2 <class-e>)))
  ...
method (class <another-method-class-c>)
  method-class-c-key a method-class-c-value a
  ((m3-arg1 <class-f>) (m3-arg2 <class-g>))
  ...
```

#### B.2.1.6 See also: defgeneric.
B.2.2.2 Arguments

gf name: As level-0. See section 10.4.1.

gf lambda list: As level-0. See section 10.4.1.

init option*: As for generic-lambda, defined above, below.

B.2.2.3 Remarks

This defining form defines a new generic function. The resulting generic function will be bound to gf-name. The second argument is the formal parameter list. An error is signalled (condition: non-congruent-lambda-lists) if any of the methods defined on this generic function do not have lambda lists congruent to that of the generic function. This applies both to methods defined at the same time as the generic function and to any methods added subsequently by defmethod or add-method. An init-option is a identifier followed by its initial value. The syntax of defgeneric is an extension of the level-0 syntax. The rewrite rules for the defgeneric form are identical to those given in Table 6 except that level 1 init option replaces level 0 init option.

B.2.2.4 Examples

In the following example of the use of defgeneric a generic function named gf-1 is defined. The differences between this function and gf-0 (see 10.4.1) are

a) The class of the generic function is specified (<another-gf-class>) along with some init-options related to the creation of an instance of that class.

b) The default class of the methods to be attached to the generic function is specified (<another-method-class-a>) along with an init-option related to the creation of an instance of that class.

c) In addition, some of the methods to be attached are of a different method class (<another-method-class-b>) and <another-method-class-c>) also with method specific init-options. These method classes are subclasses of <another-method-class-a>.

(defgeneric gf-1 (arg1 (arg2 <class-a>))
  class <another-gf-class>
  class-key a class-value a
  class-key b class-value b
  method-class <another-method-class-a>)

B.3 Methods

B.3.1 method-lambda macro

B.3.1.1 Syntax

method-lambda form
  = '(!, 'method-lambda'.
   (method init option), (* B.2.1 *)
   specialized lambda list,
   (form), '?)';

B.3.1.2 Arguments

method init option: A quoted symbol followed by an expression.

specialized lambda list: As defined under generic-lambda.

form: An expression.

B.3.1.3 Result

This syntax creates and returns an anonymous method with the given lambda list and body. This anonymous method can later be added to a generic function with a congruent lambda list via the generic function add-method. Note that the lambda list can be specialized to specify the method's domain. The value of the special initarg class determines the class to instantiate; the rest of the initlist is passed to make called with this class. The default method class is <method>.

B.3.1.4 Remarks

The additional method-init-options includes class, for specifying the class of the method to be defined, and non-standard options, which are evaluated in the lexical and dynamic environment of method-lambda and passed to initialize of that method.

B.3.2 defmethod macro

B.3.2.1 Syntax
B.3.2.2 Remarks
The `defmethod` form of level-1 extends that of level-0 to accept `method-init-options`. This allows for the specification of the method class by means of the class init option. This class must be a subclass of the method class of the host generic function. The method class otherwise defaults to that of the host generic function. In all other respects, the behaviour is as that defined in level-0.

B.3.3 method-function-lambda  

**B.3.3.1 Arguments**

`lambda-list`: A lambda list

`form`: A sequence of forms.

This macro creates and returns an anonymous method function with the given lambda list and body. This anonymous method function can later be added to a method using `(setter method-function)`, or as the function initialization value in a call of `make` on an instance of `<method>`. A function of this type is also returned by the method accessor `method-function`, and can be called using the special method function calling functions `call-method-function` and `apply-method-function`. Only functions created using this macro can be used as method functions. Note that the lambda list must not be specialized; a method's domain is stored in the method itself.

B.3.4 call-method  

**B.3.4.1 Arguments**

`method`: A method.

`next-methods`: A list of methods.

`arg`: A sequence of expressions.

This function calls the method `method` with arguments `arg`. The argument `next-methods` is a list of methods which are used as the applicable method list for `arg`; it is an error if this list is different from the methods which would be produced by the method lookup function of the generic function of `method`. If `method` is not attached to a generic function, its behavior is unspecified. The `next-methods` are used to determine the next method to call when `call-next-method` is called within `method-fn`.

B.3.5 apply-method  

**B.3.5.1 Arguments**

This function is identical to `call-method` except that its last argument is a list whose elements are the other arguments to pass to the method's method function. The difference is the same as that between normal function application and `apply`.

Programming Language EuLisp, version 0.99

```lisp
(defmethod form
  = (? (? defmethod?, gf name,
    (method init option), (= B.2.1 *)
  specialized lambda list,
  [form], ?)?)

B.3.6 call-method-function  

**B.3.6.1 Arguments**


`next-methods`: A list of method functions

`form`: A sequence of expressions.

This function calls the method function `method-fn` with arguments `arg`. The `method-fn` must have been created by `method-function-lambda`. The argument `next-methods` is a list of method functions which are used as the applicable method list for the `forms`; it is an error if this list is different from the method functions in the list of methods which would be produced by the method lookup function of the generic function of the method to which the method function is attached. These method functions are used to determine the next method to call when `call-next-method` is called within `method-fn`.

B.3.7 apply-method-function  

**B.3.7.1 Arguments**


`next-methods`: A list of method functions

`form`: A sequence of expressions.

This function is identical to `call-method-function` except that its last argument is a list whose elements are the other arguments to pass to the method function. The difference is the same as that between normal function application and `apply`.

77
B.4 Object Introspection

The only reflective capability which every object possesses is the ability to find its class.

B.4.1 class-of

function

B.4.1.1 Arguments

object: An object.

B.4.1.2 Result

The class of the object.

B.4.1.3 Remarks

The function class-of can take any Lisp object as argument and returns an instance of class representing the class of that entity.

B.5 Class Introspection

Standard classes are not redefinable and support single inheritance only. General multiple inheritance or mixin inheritance can be provided by extensions. Nor is it possible to use a class as a superclass which is not defined at the time of class definition. Again, such forward reference facilities can be provided by extensions. The distinction between metaclasses and non-metaclasses is made explicit by a special class, named <metaclass>, which is the class of all metaclasses. This is different from ObjVlisp, where whether a class is a metaclass depends on the superclass list of the class in question. It is implementation-defined whether <metaclass> itself is specializable or not. This implies that implementations are free to restrict the instantiation tree (excluding the selfinstantiation loop of <metaclass>) to a depth of three levels.

The minimum information associated with a class metaobject is:

a) The class precedence list, ordered most specific first, beginning with the class itself.

b) The list of (effective) slot descriptions.

c) The list of (effective) initargs.

Standard classes support local slots only. Shared slots can be provided by extensions. The minimal information associated with a slot description metaobject is:

a) The name, which is required to perform inheritance computations.

b) The initfunction, called by default to compute the initial slot value when creating a new instance.

c) The reader, which is a function to read the corresponding slot value of an instance.

d) The writer, which is a function to write the corresponding slot of an instance.

e) The initarg, which is a symbol to access the value which can be supplied to a make call in order to initialize the corresponding slot in a newly created object.

The metaobject classes defined for slot descriptions at level-1 are shown in Table B.1.

B.5.1 class-precedence-list

function

B.5.1.1 Arguments

class: A class.

B.5.1.2 Result

A list of classes, starting with class itself, succeeded by the superclasses of class and ending with <object>. This list is equivalent to the result of calling compute-class-precedence-list.

B.5.1.3 Remarks

The class precedence list is used to control the inheritance of slots and methods.

B.5.2 class-slot-descriptions

function

B.5.2.1 Arguments

class: A class.

B.5.2.2 Result

A list of slot-descriptions, one for each of the slots of an instance of class.

B.5.2.3 Remarks

The slot-descriptions determine the instance size (number of slots) and the slot access.

B.5.3 class-initargs

function

B.5.3.1 Arguments

class: A class.
B.5.3.2 Result
A list of symbols, which can be used as legal keywords to initialize instances of the class.

B.5.3.3 Remarks
The initargs correspond to the keywords specified in the initargs slot-option or the initargs class-option when the class and its superclasses were defined.

B.6 Slot Introspection

B.6.1 <slot-description> class
The abstract class of all slot descriptions.

B.6.2 <local-slot-description> class
The class of all local slot descriptions.

B.6.2.1 Initialization Options
name string: The name of the slot.
reader function: The function to access the slot.
writer function: The function to update the slot.
initfunction function: The function to compute the initial value in the absence of a supplied value.
initarg symbol: The key to access a supplied initial value.
The default value for all initoptions is false.

B.6.3 slot-description-name function

B.6.3.1 Arguments
slot-description: A slot description.

B.6.3.2 Result
The symbol which was used to name the slot when the class, of which the slot-description is part, was defined.

B.6.3.3 Remarks
The slot description name is used to identify a slot description in a class. It has no effect on bindings.

B.6.4 slot-description-initfunction function

B.6.4.1 Arguments
slot-description: A slot description.

B.6.5 slot-description-slot-reader function

B.6.5.1 Arguments
slot-description: A slot description.

B.6.5.2 Result
A function of one argument that returns the value of the slot in that argument.

B.6.6 slot-description-slot-writer function

B.6.6.1 Arguments
slot-description: A slot description.

B.6.6.2 Result
A function of two arguments that installs the second argument as the value of the slot in the first argument.

B.7 Generic Function Introspection
The default generic dispatch scheme is class-based; that is, methods are class specific. The default argument precedence order is left-to-right.
The minimum information associated with a generic function metaobject is:
a) The domain, restricting the domain of each added method to a subdomain.
b) The method class, restricting each added method to be an instance of that class.
c) The list of all added methods.
d) The method look-up function used to collect and sort the applicable methods for a given domain.
e) The discriminating function used to perform the generic dispatch.

B.7.1 generic-function-domain function

B.7.1.1 Arguments
generic-function: A generic function.

B.7.1.2 Result
A list of classes.
B.7.3 Remarks
This function returns the domain of a generic function. The domains of all methods attached to a generic function are constrained to be within this domain. In other words, the domain classes of each method must be subclasses of the corresponding generic function domain class. It is an error to modify this list.

B.7.2 \texttt{generic-function-method-class} \hspace{1em} \texttt{function}

\textbf{B.7.2.1 Arguments}
\begin{enumerate}
\item \texttt{generic-function}: A generic function.
\end{enumerate}

\textbf{B.7.2.2 Result}
This function returns the class which is the class of all methods of the generic function. Each method attached to a generic function must be an instance of this class. When a method is created using \texttt{defmethod}, \texttt{method-lambda}, or by using the \texttt{method} generic function option in a \texttt{defgeneric} or \texttt{generic-lambda}, it will be an instance of this class by default.

B.7.3 \texttt{generic-function-methods} \hspace{1em} \texttt{function}

\textbf{B.7.3.1 Arguments}
\begin{enumerate}
\item \texttt{generic-function}: A generic function.
\end{enumerate}

\textbf{B.7.3.2 Result}
This function returns a list of the methods attached to the generic function. The order of the methods in this list is undefined. It is an error to modify this list.

B.7.4 \texttt{generic-function-method-lookup-function} \hspace{1em} \texttt{function}

\textbf{B.7.4.1 Arguments}
\begin{enumerate}
\item \texttt{generic-function}: A generic function.
\end{enumerate}

\textbf{B.7.4.2 Result}
A function.

\textbf{B.7.4.3 Remarks}
This function returns a function which, when applied to the arguments given to the generic function, returns a sorted list of applicable methods. The order of the methods in this list is determined by \texttt{compute-method-lookup-function}.

B.7.5 \texttt{generic-function-discriminating-function} \hspace{1em} \texttt{function}

\textbf{B.7.5.1 Arguments}
\begin{enumerate}
\item \texttt{generic-function}: A generic function.
\end{enumerate}

B.7.5.2 Result
A function.

B.7.5.3 Remarks
This function returns a function which may be applied to the same arguments as the generic function. This function is called to perform the generic dispatch operation to determine the applicable methods whenever the generic function is called, and call the most specific applicable method function. This function is created by \texttt{compute-discriminating-function}.

B.8 Method Introspection

The minimal information associated with a method metaobject is:

\begin{enumerate}
\item The domain, which is a list of classes.
\item The function comprising the code of the method.
\item The generic function to which the method has been added, or \texttt{nil} if it is attached to no generic function.
\end{enumerate}

The metaobject classes for generic functions defined at level 1 are shown in Table B.1.

B.8.1 \texttt{method-domain} \hspace{1em} \texttt{function}

\textbf{B.8.1.1 Arguments}
\begin{enumerate}
\item \texttt{method}: A method.
\end{enumerate}

\textbf{B.8.1.2 Result}
A list of classes defining the domain of a method.

B.8.2 \texttt{method-function} \hspace{1em} \texttt{function}

\textbf{B.8.2.1 Arguments}
\begin{enumerate}
\item \texttt{method}: A method.
\end{enumerate}

\textbf{B.8.2.2 Result}
This function returns a function which implements the method. The returned function which is called when \texttt{method} is called, either by calling the generic function with appropriate arguments, through a \texttt{call-next-method}, or by using \texttt{call-method}. A method metaobject itself cannot be applied or called as a function.

B.8.3 \texttt{method-generic-function} \hspace{1em} \texttt{function}

\textbf{B.8.3.1 Arguments}
\begin{enumerate}
\item \texttt{method}: A method.
B.8.3.2 Result
This function returns the generic function to which method
is attached; if method is not attached to a generic function, it returns ()

B.9 Class Initialization

B.9.1 initialize method

B.9.1.1 Specialized Arguments
(class <class>): A class.
(initlist <list>): A list of initialization options as follows:
  name symbol: Name of the class being initialized.
  direct-superclasses list: List of direct superclasses.
  direct-slot-descriptions list: List of direct slot
    specifications.
  direct-initargs list: List of direct initargs.

B.9.1.2 Result
The initialized class.

B.9.1.3 Remarks
The initialization of a class takes place as follows:
  a) Check compatibility of direct superclasses
  b) Perform the logical inheritance computations of:
     1) class precedence list
     2) initargs
     3) slot descriptions
  c) Compute new slot accessors and ensure all (new and
    inherited) accessors to work correctly on instances of the
    new class.
  d) Make the results accessible by class readers.

The basic call structure is laid out in Figure B.2.

Note that compute-initargs is called by the default initialize method with all direct initargs as the second argument: those specified as slot option and those specified as class option.

B.9.2 compute-predicate generic function

B.9.2.1 Generic Arguments
(class <class>): A class.

B.9.2.2 Result
Computes and returns a function of one argument, which is a predicate function for class.

B.9.3 compute-predicate method

B.9.3.1 Specialized Arguments
(class <class>): A class.

B.9.3.2 Result
Computes and returns a function of one argument, which
returns true when applied to direct or indirect instances of
class and false otherwise.

B.9.4 compute-constructor generic function

B.9.4.1 Generic Arguments
(class <class>): A class.
(parameters <list>): The argument list of the function being created.

B.9.4.2 Result
Computes and returns a constructor function for class.

B.9.5 compute-constructor method

B.9.5.1 Specialized Arguments
(class <class>): A class.
(parameters <list>): The argument list of the function being created.

B.9.5.2 Result
Computes and returns a constructor function, which returns
a new instance of class.

B.9.6 allocate generic function

B.9.6.1 Generic Arguments
(class <class>): A class.
(initlist <list>): A list of initialization arguments.

B.9.6.2 Result
An instance of the first argument.

B.9.6.3 Remarks
Creates an instance of the first argument. Users may define
new methods for new metaclasses.
Programming Language EuLisp, version 0.99

```
compatible-supertypes p cl direct-supertypes — boolean
compatible-supertype p cl supertype — boolean
compute-supervariant p cl direct-supertypes — (cl*)
compute-inherits-cl class direct-supertypes — (cl*)
compute-inherits-slots class direct-supertypes — (cl*)
inherited-slots class direct-supertypes — (cl*)
compute-inherits-slots class direct-supertypes — (cl*)
inherited-slots class direct-supertypes — (cl*)
either
compute-defined-slot-description cl slot-spec — sd
compute-defined-slot-description-class cl slot-spec — sd-class
or
compute-specialized-slot-description cl inherited-slots slot-spec — sd
compute-specialized-slot-description-class cl inherited-slots slot-spec — sd-class
compute-instance-size cl effective-slots — integer
calculate-and-ensure-slot-accessors cl effective-slots inherited-slots — (sd*)
calculate-slot-reader cl sd effective-slots — function
calculate-slot-reader cl sd effective-slots — function
calculate-slot-reader cl sd effective-slots reader — function
calculate-primitive-reader-using-slot-description sd cl effective-slots — function
calculate-primitive-reader-using-class cl sd effective-slots — function
calculate-primitive-reader-using-slot-description sd cl effective-slots — function
calculate-primitive-reader-using-class cl sd effective-slots — function
```

Figure B.2 — Initialization Call Structure

### B.9.7 allocate

**method**

#### B.9.7.1 Specialized Arguments

- (class <class>): A class.
- (initlist <list>): A list of initialization arguments.

#### B.9.7.2 Result

An instance of the first argument.

#### B.9.7.3 Remarks

The default method creates a new uninitialized instance of the first argument. The initlist is not used by this allocate method.

### B.10 Slot Description Initialization

**method**

#### B.10.1 initialize

**method**

#### B.10.1.1 Specialized Arguments

- (slot-description <slot-description>): A slot description.
- (initlist <list>): A list of initialization options as follows:
  - name symbol: The name of the slot.
  - initfunction function: A function.
  - initarg symbol: A symbol.

reader function: A slot reader function.
writer function: A slot writer function.

#### B.10.1.2 Result

The initialized slot description.

### B.11 Generic Function Initialization

**method**

#### B.11.1 initialize

**method**

#### B.11.1.1 Specialized Arguments

- (gf <generic-function>): A generic function.
- (initlist <list>): A list of initialization options as follows:
  - name symbol: The name of the generic function.
  - domain list: List of argument classes.
  - method-class class: Class of attached methods.
  - method method-description: A method to be attached. This option may be specified more than once.

#### B.11.1.2 Result

The initialized generic function.

#### B.11.1.3 Remarks

This method initializes and returns the generic-function. The specified methods are attached to the generic function by add-method, and its slots are initialized from
the information passed in initlist and from the results of calling compute-method-lookup-function and compute-discriminating-function on the generic function. Note that these two functions may not be called during the call to initialize, and that they may be called several times for the generic function.

The basic call structure is:
- add-method gf method -> gf
- compute-method-lookup-function gf domain -> function
- compute-discriminating-function gf domain lookup fn methods -> function

B.12 Method Initialization

B.12.1 initialize method

B.12.1.1 Specialized Arguments

(method <method>): A method.

(initlist <list>): A list of initialization options as follows:
  - domain list: The list of argument classes.
  - function fn: A function, created with method-function-lambda.
  - generic-function gf: A generic function.

B.12.1.2 Result

This method returns the initialized method metaobject method. If the generic-function option is supplied, add-method is called to install the new method in the generic-function.

B.13 Inheritance Protocol

B.13.1 compatible-superclasses-p generic function

B.13.1.1 Generic Arguments

(class <class>): A class.

(direct-superclasses <list>): A list of potential direct superclasses of class.

B.13.1.2 Result

Returns t if class is compatible with direct-superclasses, otherwise ()

B.13.2 compatible-superclasses-p method

B.13.2.1 Specialized Arguments

(class <class>): A class.

(direct-superclasses <list>): A list of potential direct superclasses.

B.13.3 compatible-superclass-p generic function

B.13.3.1 Generic Arguments

(subclass <class>): A class.

(superclass <class>): A potential direct superclass.

B.13.3.2 Result

Returns t if subclass is compatible with superclass, otherwise ()

B.13.4 compatible-superclass-p method

B.13.4.1 Specialized Arguments

(subclass <class>): A class.

(superclass <class>): A potential direct superclass.

B.13.4.2 Result

Returns t if the class of the first argument is a subclass of the class of the second argument, otherwise ()

If the implementation wishes to restrict the instantiation tree (see introduction to B.4), this method should return false if superclass is <metaclass>.

B.13.5 compatible-superclass-p method

B.13.5.1 Specialized Arguments

(subclass <class>): A class.

(superclass <abstract-class>): A potential direct superclass.

B.13.5.2 Result

Always returns t.

B.13.6 compatible-superclass-p method

B.13.6.1 Specialized Arguments

(subclass <abstract-class>): A class.

(superclass <class>): A potential direct superclass.

B.13.6.2 Result

Always returns ().
B.13.7 compatible-superclass-p method

B.13.7.1 Specialized Arguments

(subclass <abstract-class>): A class.

(superclass <abstract-class>): A potential direct superclass.

B.13.7.2 Result
Always returns t.

B.13.8 compute-class-precedence-list generic function

B.13.8.1 Generic Arguments

(class <class>): Class being defined.

(direct-superclasses <list>): List of direct superclasses.

B.13.8.2 Result
Computes and returns a list of classes which represents the linearized inheritance hierarchy of class and the given list of direct superclasses, beginning with class and ending with <object>.

B.13.9 compute-class-precedence-list method

B.13.9.1 Specialized Arguments

(class <class>): Class being defined.

(direct-superclasses <list>): List of direct superclasses.

B.13.9.2 Result
A list of classes.

B.13.9.3 Remarks
This method can be considered to return a cons of class and the class precedence list of the first element of direct-superclasses (single inheritance assumption). If no direct-superclasses has been supplied, the result is the list of two elements: class and <object>.

B.13.10 compute-slot-descriptions generic function

B.13.10.1 Generic Arguments

(class <class>): Class being defined.

(direct-slot-specifications <list>): A list of direct slot specification.

(inherited-slot-descriptions <list>): A list of lists of inherited slot descriptions.

B.13.10.2 Result
Computes and returns the list of effective slot descriptions of class.

B.13.11 compute-initargs method

B.13.11.1 Specialized Arguments

(class <class>): Class being defined.

(slot-specs <list>): List of (direct) slot specifications.

(inherited-slot-description-lists <list>): A list of lists (in fact one list in single inheritance) of inherited slot descriptions.

B.13.11.2 Result
A list of effective slot descriptions.

B.13.11.3 Remarks
The default method computes two sublists:

a) Calling compute-specialized-slot-description with the three arguments (i) class, (ii) each inherited-slot-description as a singleton list, (iii) the slot-spec corresponding (by having the same name) to the slot description, if it exists, otherwise , giving a list of the specialized slot descriptions.

b) Calling compute-defined-slot-description with the three arguments (i) class, (ii) each slot-specification which does not have a corresponding (by having the same name) inherited-slot-description.

The method returns the concatenation of these two lists as its result. The order of elements in the list is significant. All specialized slot descriptions have the same position as in the effective slot descriptions list of the direct superclass (due to the single inheritance). The slot accessors (computed later) may rely on this assumption minimizing the number of methods to one for all subclasses and minimizing the access time to an indexed reference.

B.13.11.4 See also: compute-specialized-slot-description, compute-defined-slot-description, compute-and-ensure-slot-accessors.

B.13.12 compute-initargs generic function

B.13.12.1 Generic Arguments

(class <class>): Class being defined.

(initargs <list>): List of direct initargs.

(inherited-initargs-lists <list>): A list of lists of inherited initargs.
**B.13.12.2 Result**
List of symbols.

**B.13.12.3 Remarks**
Computes and returns all legal initargs for `class`.

**B.13.12.4 See also:** compute-inherited-initargs.

---

**B.13.13 compute-initargs method**

**B.13.13.1 Specialized Arguments**

- `(class <class>): Class being defined.
- `(initargs <list>): List of direct initargs.
- `(inherited-initarg-lists <list>): A list of lists of inherited initargs.

---

**B.13.13.2 Result**
List of symbols.

**B.13.13.3 Remarks**
This method appends the second argument with the third argument of the third argument (single inheritance assumption), removes duplicates and returns the result. Note that `compute-initargs` is called by the default initialize method with all direct initargs as the second argument: those specified as slot option and those specified as class option.

---

**B.13.14 compute-inherited-slot-descriptions generic function**

**B.13.14.1 Generic Arguments**

- `(class <class>): Class being defined.
- `(direct-superclasses <list>): List of direct superclasses.

**B.13.14.2 Result**
List of lists of inherited slot descriptions.

**B.13.14.3 Remarks**
Computes and returns a list of lists of effective slot descriptions.

**B.13.14.4 See also:** compute-slot-descriptions.

---

**B.13.15 compute-inherited-slot-descriptions method**

**B.13.15.1 Specialized Arguments**

- `(class <class>): Class being defined.
- `(direct-superclasses <list>): List of direct superclasses.

---

**B.13.15.2 Result**
List of lists of inherited slot descriptions.

**B.13.15.3 Remarks**
The result of the default method is a list of one element: a list of effective slot descriptions of the first element of the second argument (single inheritance assumption). Its result is used by `compute-slot-descriptions` as an argument.

---

**B.13.16 compute-inherited-initargs generic function**

**B.13.16.1 Generic Arguments**

- `(class <class>): Class being defined.
- `(direct-superclasses <list>): List of direct superclasses.

**B.13.16.2 Result**
List of lists of symbols.

**B.13.16.3 Remarks**
Computes and returns a list of lists of initargs. Its result is used by `compute-initargs` as an argument.

**B.13.16.4 See also:** compute-initargs.

---

**B.13.17 compute-inherited-initargs method**

**B.13.17.1 Specialized Arguments**

- `(class <class>): Class being defined.
- `(direct-superclasses <list>): List of direct superclasses.

**B.13.17.2 Result**
List of lists of symbols.

**B.13.17.3 Remarks**
The result of the default method contains one list of legal initargs of the first element of the second argument (single inheritance assumption).

---

**B.13.18 compute-defined-slot-description generic function**

**B.13.18.1 Generic Arguments**

- `(class <class>): Class being defined.
- `(slot-spec <list>): Canonicalized slot specification.

**B.13.18.2 Result**
Slot description.
B.13.18.3 Remarks
Computes and returns a new effective slot description. It is called by compute-slot-descriptions on each slot specification which has no corresponding inherited slot descriptions.

B.13.18.4
See also: compute-defined-slot-description-class.

B.13.19 compute-defined-slot-description method

B.13.19.1 Specialized Arguments
(class <class>): Class being defined.
(slot-spec <list>): Canonicalized slot specification.

B.13.19.2 Result
Slot description.

B.13.19.3 Remarks
Computes and returns a new effective slot description. The class of the result is determined by calling compute-defined-slot-description-class.

B.13.19.4
See also: compute-defined-slot-description-class.

B.13.20 compute-defined-slot-description-class generic function

B.13.20.1 Generic Arguments
(class <class>): Class being defined.
(slot-spec <list>): Canonicalized slot specification.

B.13.20.2 Result
Slot description class.

B.13.20.3 Remarks
Determines and returns the slot description class corresponding to class and slot-spec.

B.13.20.4 See also: compute-defined-slot-description.

B.13.21 compute-defined-slot-description-class method

B.13.21.1 Specialized Arguments
(class <class>): Class being defined.
(slot-spec <list>): Canonicalized slot specification.

B.13.21.2 Result
The class <local-slot-description>.

B.13.21.3 Remarks
This method just returns the class <local-slot-description>.

B.13.22 compute-specialized-slot-description generic function

B.13.22.1 Generic Arguments
(class <class>): Class being defined.
(inherited-slot-descriptions <list>): List of inherited slot descriptions (each of the same name as the slot being defined).
(slot-spec <list>): Canonicalized slot specification or () if no one is specified with the same name.

B.13.22.2 Result
Slot description.

B.13.22.3 Remarks
Computes and returns a new effective slot description. It is called by compute-slot-descriptions on the class, each list of inherited slots with the same name and with the specialising slot specification list or () if no one is specified with the same name.

B.13.22.4
See also: compute-specialized-slot-description-class.

B.13.23 compute-specialized-slot-description method

B.13.23.1 Specialized Arguments
(class <class>): Class being defined.
(inherited-slot-descriptions <list>): List of inherited slot descriptions.
(slot-spec <list>): Canonicalized slot specification or ()

B.13.23.2 Result
Slot description.

B.13.23.3 Remarks
Determines and returns the slot description class corresponding to class and slot-spec.

B.13.23.4
See also: compute-specialized-slot-description-class.
B.13.24 `compute-specialized-slot-description-class`

**Generic function**

B.13.24.1 Generic Arguments

`(class <class>)`: Class being defined.

`(inherited-slot-descriptions <list>)`: List of inherited slot descriptions.

`(slot-spec <list>)`: Canonicalized slot specification or `()`. 

B.13.24.2 Result
Slot description class.

B.13.24.3 Remarks
Determines and returns the slot description class corresponding to (i) the class being defined, (ii) the inherited slot descriptions being specialized (iii) the specializing information in `slot-spec`.

B.13.25 `compute-specialized-slot-description-class`

**Method**

B.13.25.1 Specialized Arguments

`(class <class>)`: Class being defined.

`(inherited-slot-descriptions <list>)`: List of inherited slot descriptions.

`(slot-spec <list>)`: Canonicalized slot specification or `()`. 

B.13.25.2 Result
The class `<local-slot-description>`. 

B.13.25.3 Remarks
This method just returns the class `<local-slot-description>`. 

B.14 Slot Access Protocol

The slot access protocol is defined via accessor (readers and writers) only. There is no primitive like CLOS’s `slot-value`. The accessors are generic for standard classes, since they have to work on subclasses and should do the applicability check anyway. The key idea is that the discrimination on slot-descriptions and classes is performed once at class definition time rather than again and again at slot access time.

Each slot-description has exactly one reader and one writer as anonymous objects. If a reader/writer slot-option is specified in a class definition, the anonymous reader/writer of that slot-description is bound to the specified identifier. Thus, if a reader/writer option is specified more than once, the same object is bound to all the identifiers. If the accessor slot-option is specified the anonymous writer will be installed as the setter of the reader. Specialized slot-descriptions refer to the same objects as those in the superclasses (single inheritance makes that possible). Since the readers/writers are generic, it is possible for a subclass (at the meta-level) to add new methods for inherited slot-descriptions in order to make the readers/writers applicable on instances of the subclass. A new method might be necessary if the subclasses have a different instance allocation or if the slot positions cannot be kept the same as in the superclass (in multiple inheritance extensions). This can be done during the initialization computations.

B.14.1 `compute-and-ensure-slot-accessors`

**Generic function**

B.14.1.1 Generic Arguments

`(class <class>)`: Class being defined.

`(slot-descriptions <list>)`: List of effective slot descriptions.

`(inherited-slot-descriptions <list>)`: List of lists of inherited slot descriptions.

B.14.1.2 Result
List of effective slot descriptions.

B.14.1.3 Remarks
Computes new accessors or ensures that inherited accessors work correctly for each effective slot description.

B.14.2 `compute-and-ensure-slot-accessors`

**Method**

B.14.2.1 Specialized Arguments

`(class <class>)`: Class being defined.

`(slot-descriptions <list>)`: List of effective slot descriptions.

`(inherited-slot-descriptions <list>)`: List of lists of inherited slot descriptions.

B.14.2.2 Result
List of effective slot descriptions.

B.14.2.3 Remarks
For each slot description in `slot-descriptions` the default method checks if it is a new slot description and not an inherited one. If the slot description is new,

a) calls `compute-slot-reader` to compute a new slot reader and stores the result in the slot description;
b) calls compute-slot-writer to compute a new slot writer and stores the result in the slot description;

Otherwise, it assumes that the inherited values remain valid.

Finally, for every slot description (new or inherited) it ensures the reader and writer work correctly on instances of class by means of ensure-slot-reader and ensure-slot-writer.

---

B.14.3 compute-slot-reader  
generic function

B.14.3.1 Generic Arguments

(class <class>): Class.

(slot-description <slot-description>): Slot description.

(slot-description-list <list>): List of effective slot descriptions.

B.14.3.2 Result
Function.

B.14.3.3 Remarks
Computes and returns a new slot reader applicable to instances of class returning the slot value corresponding to slot-description. The third argument can be used in order to compute the logical slot position.

---

B.14.4 compute-slot-reader  
method

B.14.4.1 Specialized Arguments

(class <class>): Class.

(slot-description <slot-description>): Slot description.

(slot-descriptions <list>): List of effective slot descriptions.

B.14.4.2 Result
Generic function.

B.14.4.3 Remarks
The default method returns a new generic function of one argument without any methods. Its domain is class x <object>.

---

B.14.5 compute-slot-writer  
generic function

B.14.5.1 Generic Arguments

(class <class>): Class.

(slot-description <slot-description>): Slot description.

(slot-descriptions <list>): List of effective slot descriptions.

B.14.5.2 Result
Function.

B.14.5.3 Remarks
Computes and returns a new slot writer applicable to instances of class and any value to be stored as the new slot value corresponding to slot-description. The third argument can be used in order to compute the logical slot position.

---

B.14.6 compute-slot-writer  
method

B.14.6.1 Specialized Arguments

(class <class>): Class.

(slot-description <slot-description>): Slot description.

(slot-descriptions <list>): List of effective slot descriptions.

B.14.6.2 Result
Generic function.

B.14.6.3 Remarks
The default method returns a new generic function of two arguments without any methods. Its domain is class x <object>.

---

B.14.7 ensure-slot-reader  
generic function

B.14.7.1 Generic Arguments

(class <class>): Class.

(slot-description <slot-description>): Slot description.

(slot-descriptions <list>): List of effective slot descriptions.

(reader <function>): The slot reader.

B.14.7.2 Result
Function.

B.14.7.3 Remarks
Ensures function correctly fetches the value of the slot from instances of class.
B.14.8 ensure-slot-reader  
method

B.14.8.1 Specialized Arguments

(class <class>): Class.

(slot-description <slot-description>): Slot description.

(slot-descriptions <list>): List of effective slot descriptions.

(reader <generic-function>): The slot reader.

B.14.8.2 Result
Generic function.

B.14.8.3 Remarks
The default method checks if there is a method in the generic-function. If not, it creates and adds a new one, otherwise it assumes that the existing method works correctly. The domain of the new method is class and the function is

(method-function-lambda ((object class))
  (primitive-reader object))

compute-primitive-reader-using-slot-description is called by ensure-slot-reader method to compute the primitive reader used in the function of the new created reader method.

B.14.9 ensure-slot-writer  
generic function

B.14.9.1 Generic Arguments

(class <class>): Class.

(slot-description <slot-description>): Slot description.

(slot-descriptions <list>): List of effective slot descriptions.

(writer <function>): The slot writer.

B.14.9.2 Result
Function.

B.14.9.3 Remarks
Ensures function correctly updates the value of the slot in instances of class.

B.14.10 ensure-slot-writer  
method

B.14.10.1 Specialized Arguments

(class <class>): Class.

B.14.11 compute-primitive-reader-using-slot-description

generic function

B.14.11.1 Generic Arguments

(slot-description <slot-description>): Slot description.

(class <class>): Class.

(slot-descriptions <list>): List of effective slot descriptions.

B.14.11.2 Result
Function.

B.14.11.3 Remarks
Computes and returns a function which returns a slot value when applied to an instance of class.

B.14.12 compute-primitive-reader-using-slot-description

method

B.14.12.1 Specialized Arguments

(slot-description <slot-description>): Slot description.

(class <class>): Class.

(slot-descriptions <list>): List of effective slot descriptions.
B.14.12.2 Result
Function.

B.14.12.3 Remarks
Calls `compute-primitive-reader-using-class`. This is the default method.

B.14.13 `compute-primitive-reader-using-class`

  generic function

B.14.13.1 Generic Arguments

  (class <class>): Class.

  (slot-description <slot-description>): Slot description.

  (slot-descriptions <list>): List of effective slot descriptions.

B.14.13.2 Result
Function.

B.14.13.3 Remarks
Computes and returns a function which returns the slot value when applied to an instance of `class`.

B.14.14 `compute-primitive-reader-using-class`

  method

B.14.14.1 Specialized Arguments

  (class <class>): Class.

  (slot-description <slot-description>): Slot description.

  (slot-descriptions <list>): List of effective slot descriptions.

B.14.14.2 Result
Function.

B.14.14.3 Remarks
The default method returns a function of one argument.

B.14.15 `compute-primitive-reader-using-slot-description`

  generic function

B.14.15.1 Generic Arguments

  (slot-description <slot-description>): Slot description.

  (class <class>): Class.

  (slot-descriptions <list>): List of effective slot descriptions.

B.14.15.2 Result
Function.

B.14.15.3 Remarks
Computes and returns a function which stores a new slot value when applied on an instance of `class` and a new value.

B.14.16 `compute-primitive-writer-using-slot-description`

  method

B.14.16.1 Specialized Arguments

  (slot-description <slot-description>): Slot description.

  (class <class>): Class.

  (slot-descriptions <list>): List of effective slot descriptions.

B.14.16.2 Result
Function.

B.14.16.3 Remarks
Calls `compute-primitive-writer-using-class`. This is the default method.

B.14.17 `compute-primitive-writer-using-class`

  generic function

B.14.17.1 Generic Arguments

  (class <class>): Class.

  (slot-description <slot-description>): Slot description.

  (slot-descriptions <list>): List of effective slot descriptions.

B.14.17.2 Result
Function.

B.14.17.3 Remarks
Computes and returns a function which stores the new slot value when applied on an instance of `class` and new value.

B.14.18 `compute-primitive-writer-using-class`

  method

B.14.18.1 Specialized Arguments

  (class <class>): Class.

  (slot-description <slot-description>): Slot description.

  (slot-descriptions <list>): List of effective slot descriptions.
B.14.18.2 Result
Function.

B.14.18.3 Remarks
The default method returns a function of two arguments.

B.15 Method Lookup and Generic Dispatch

B.15.1 compute-method-lookup-function

generic function

B.15.1.1 Generic Arguments

(domain <list>): A list of classes which cover the domain.

B.15.1.2 Result
A function.

B.15.1.3 Remarks
Computes and returns a function which will be called at least once for each domain to select and sort the applicable methods by the default dispatch mechanism. New methods may be defined for this function to implement different method lookup strategies. Although only one method lookup function generating method is provided by the system, each generic function has its own specific lookup function which may vary from generic function to generic function.

B.15.2 compute-method-lookup-function

method

B.15.2.1 Specialized Arguments

(domain <list>): A list of classes which cover the domain.

B.15.2.2 Result
A function.

B.15.2.3 Remarks
Computes and returns a function which will be called at least once for each domain to select and sort the applicable methods by the default dispatch mechanism. It is not defined, whether each generic function may have its own lookup function.

B.15.3 compute-discriminating-function

generic function

B.15.3.1 Generic Arguments

(gf <generic-function>): A generic function.

B.15.3.2 Result
A function.

B.15.3.3 Remarks
This generic function computes and returns a function which is called whenever the generic function is called. The returned function controls the generic dispatch. Users may define methods on this function for new generic function classes to implement non-default dispatch strategies.

B.15.4 compute-discriminating-function

method

B.15.4.1 Specialized Arguments

(domain <list>): A list of classes which span the domain.

B.15.4.2 Result
A function.

B.15.4.3 Remarks
This method computes and returns a function which is called whenever the generic function is called. This default method implements the standard dispatch strategy: The generic function’s methods are sorted using the function returned by compute-method-lookup-function, and the first is called as if by call-method, passing the others as the list of next methods. Note that call-method need not be directly called.

B.15.5 add-method

generic function

B.15.5.1 Generic Arguments

(gf <generic-function>): A generic function.

B.15.5.2 Result
This generic function adds method to the generic function gf. This method will then be taken into account when gf is called with appropriate arguments. It returns the generic function gf. New methods may be defined on this generic function for new generic function and method classes.
B.1.5.5.3 Remarks
In contrast to CLOS, add-method does not remove a method with the same domain as the method being added. Instead, a noncontinuable error is signalled.

B.1.5.6 add-method

B.1.5.6.1 Specialized Arguments

\( (\text{gf} \text{<generic-function>}) \): A generic function.

\( (\text{method} \text{<method>}) \): A method to be attached.

B.1.5.6.2 Result
The generic function.

B.1.5.6.3 Remarks
This method checks that the domain classes of the method are subclasses of those of the generic function, and that the method is an instance of the generic function's method class. If not, signals an error (condition: incompatible-method-and-gf). It also checks if there is a method with the same domain already attached to the generic function. If so, a noncontinuable error is signalled (condition: methods-exists). If no error occurs, the method is added to the generic function. Depending on particular optimizations of the generic dispatch, adding a method may cause some updating computations, e.g., by calling compute-method-lookup-function and compute-discriminating-function.

B.16 Low Level Allocation Primitives

This module provides primitives which are necessary to implement new allocation methods portably. However, they should be defined in such a way that objects cannot be destroyed unintentionally. In consequence it is an error to use primitive-class-of, primitive-ref and their setters on objects not created by primitive-allocate.

B.16.1 primitive-allocate

B.16.1.1 Arguments

\( \text{class} \): A class.

\( \text{size} \): An integer.

B.16.1.2 Result
An instance of the first argument.

B.16.1.3 Remarks
This function returns a new instance of the first argument which has a vector-like structure of length size. The components of the new instance can be accessed using primitive-ref and updated using (setter primitive-ref). It is intended to be used in new allocate methods defined for new metaclasses.

B.16.2 primitive-class-of

B.16.2.1 Arguments

\( \text{object} \): An object created by primitive-allocate.

B.16.2.2 Result
A class.

B.16.2.3 Remarks
This function returns the class of an object. It is similar to class-of, which has a defined behaviour on any object. It is an error to use primitive-class-of on objects which were not created by primitive-allocate.

B.16.3 (setter primitive-class-of)

B.16.3.1 Arguments

\( \text{object} \): An object created by primitive-allocate.

\( \text{class} \): A class.

B.16.3.2 Result
The class.
B.16.3.3 Remarks
This function supports portable implementations of
a) dynamic classification like change-class in CLOS.
b) automatic instance updating of redefined classes.

B.16.4 primitive-ref function

B.16.4.1 Arguments
object: An object created by primitive_allocate.
index: The index of a component.

B.16.4.2 Result
An object.

B.16.4.3 Remarks
Returns the value of the objects component corresponding to the supplied index. It is an error if index is outside the index range of object. This function is intended to be used when defining new kinds of accessors for new metaclasses.

B.17 Dynamic Binding

B.17.1 dynamic special form

B.17.1.1 Syntax
dynamic form

B.17.1.2 Arguments
identifier: A symbol naming a dynamic binding.

B.17.1.3 Result
The value of closest dynamic binding of identifier is returned. If no visible binding exists, an error is signaled (condition: unbound-dynamic-variable).

B.17.2 dynamic-setq special form

B.17.2.1 Syntax
dynamic-setq form

B.17.2.2 Arguments
identifier: A symbol naming a dynamic binding to be updated.
form: An expression whose value will be stored in the dynamic binding of identifier.

B.17.2.3 Result
The value of form.

B.17.2.4 Remarks
The form is evaluated and the result is stored in the closest dynamic binding of identifier. If no visible binding exists, an error is signaled (condition: unbound-dynamic-variable).

B.17.3 unbound-dynamic-variable execution-condition

B.17.3.1 Initialization Options
symbol symbol: A symbol naming the (unbound) dynamic variable.

B.17.3.2 Remarks
Signalled by dynamic or dynamic-setq if the given dynamic variable has no visible dynamic binding.
B.17.4 dynamic-let

special form

B.17.4.1 Syntax

dynamic-let form
  = (\(\cdot\) dynamic-let?,
    (binding), (* 13 *)
    (form), ?);?

B.17.4.2 Arguments

  binding*: A list of binding specifiers.
  body: A sequence of forms.

B.17.4.3 Result

The sequence of forms is evaluated in order, returning the value of the last one as the result of the dynamic-let form.

B.17.4.4 Remarks

A binding specifier is either an identifier or a two element list of an identifier and an initializing form. All the initializing forms are evaluated from left to right in the current environment and the new bindings for the symbols named by the identifiers are created in the dynamic environment to hold the results. These bindings have dynamic scope and dynamic extent. Each form in body is evaluated in order in the environment extended by the above bindings. The result of evaluating the last form in body is returned as the result of dynamic-let.

B.17.5 defvar

defining form

B.17.5.1 Syntax

defvar form
  = (\(\cdot\) defvar?, identifier, expression, ?);

B.17.5.2 Arguments

  identifier: A symbol naming a top dynamic binding containing the value of form.
  form: The form whose value will be stored in the top dynamic binding of identifier.

B.17.5.3 Remarks

The value of form is stored as the top dynamic value of the symbol named by identifier. The binding created by defvar is mutable. An error is signaled (condition: dynamic-multiply-defined), on processing this form more than once for the same identifier.

NOTE — The problems engendered by cross-module reference necessitated by a single top-dynamic environment are leading to a reconsideration of the defined model. Another unpleasant aspect of the current model is that it is not clear how to address the issue of importing (or hiding) dynamic variables—they are in every sense global, which conflicts with the principle of module abstraction. A model in which a separate top-dynamic environment is associated with each module is under consideration for a later version of the definition.

B.17.6 dynamic-multiply-defined

execution-condition

B.17.6.1 Initialization Options

symbol symbol: A symbol naming the dynamic variable which has already been defined.

B.17.6.2 Remarks

Signalled by defvar if the named dynamic variable already exists.

B.18 Exit Extensions

B.18.1 catch

macro

B.18.1.1 Syntax

catch macro
  = (\(\cdot\) catch?, tag, (form), ?);
  tag
  = symbol; (* A.16 *)

B.18.1.2 Remarks

The catch operator is similar to block except that the scope of the name (tag) of the exit function is dynamic. The catch tag must be a symbol because it is used as a dynamic variable to create a dynamically scoped binding of tag to the continuation of the catch form. The continuation can be invoked anywhere within the dynamic extent of the catch form by using throw. The forms are evaluated in sequence and the value of the last one is returned as the value of the catch form. The rewrite rules for catch are:

\[
\begin{align*}
\text{(catch)} & \equiv \text{Is an error} \\
\text{(catch tag)} & \equiv (\text{progm tag ()}) \\
\text{(catch tag form*)} & \equiv \text{(let/cc tmp} \\
& \quad \text{(dynamic-let ((tag tmp)) form*)}
\end{align*}
\]

Exiting from a catch, by whatever means, causes the restoration of the lexical environment and dynamic environment that existed before the catch was entered. The above rewrite for catch, causes the variable tmp to be shadowed. This is an artifact of the above presentation only and a conforming processor must not shadow any variables that could occur in the body of catch in this way.

B.18.1.3 See also: throw.
B.18.2.1 Syntax

throw macro = '(' 'throw' tag form ')';;

B.18.2.2 Remarks

In throw, the tag names the continuation of the catch from which to return. throw is the invocation of the continuation of the catch named tag. The form is evaluated and the value are returned as the value of the catch named by variable. The tag is a symbol because it used to access the current dynamic binding of the symbol, which is where the continuation is bound. The rewrite rules for throw are:

| (throw) | ≡ In an error |
| (throw tag) | ≡ ((dynamic tag) ()) |
| (throw tag form) | ≡ ((dynamic tag) form) |

B.18.2.3 See also: catch.

B.19 Summary of Level-1 Expressions and Definitions

This section gives the syntax of all level-1 expressions and definitions together. Any productions undefined here appear elsewhere in the definition, specifically: the syntax of most expressions and definitions is given in the section defining level-0.

B.19.1 Syntax of Level-1 defining forms

level 1 module form
= level 1 expression
| level 0 module form;
defclass form
= '(' 'defclass'
 class name, (* 10.3.1 *)
 superclass list
 defclass slot description list
 (defclass class option), ')');
superclass list
= '(' (superclass name), ')'; (* 10.3.1 *)
defclass slot description list
= '(' (defclass slot description), ')');
defclass slot option
= 'initform', level 1 expression
| identifier, level 1 expression
| defstruct slot option; (* 10.3.1 *)
defclass class option
= 'class', class name
| identifier, level 1 expression
| defstruct class option; (* 10.3.1 *)
defgeneric form
= '(' 'defgeneric'
 gf name, (* 10.4.1 *)
 gf lambda list, (* 10.4.1 *)
 (level 1 init option), ')');
defmethod form
= '(' 'defmethod'
 gf name,
 (method init option), (* B.2.1 *)
specialized lambda list,
 (form), ')');
defvar form
= '(' 'defvar'
 identifier, expression, ')');
defgeneric form
= '(' 'defgeneric'
 gf name, (* 10.4.1 *)
 gf lambda list, (* 10.4.1 *)
 (level 1 init option), ')');
defmethod form
= '(' 'defmethod'
 gf name,
 (method init option), (* B.2.1 *)
specialized lambda list,
 (form), ')');
defvar form
= '(' 'defvar'
 identifier, expression, ')');

dynamic form
= '(' 'dynamic'
 identifier, *)
dynamic setq form
= '(' 'dynamic-setq'
 identifier, form, *)
dynamic let form
= '(' 'dynamic-let'
 (binding), (* 13 *)
 (form), *)
generic lambda form
= '(' 'generic-lambda'
 gf lambda list,
 (level 1 init option), *)
level 1 init option
= 'class'
| class name
| 'method-class'
| class name
| 'method' level 1 method description
| identifier, level 1 expression;
level 0 init option
= '(' (method init option),
 specialized lambda list, (* 10.4.1 *)
 (form), *)
method init option
= 'class'
| class name
| identifier level 1 expression
method lambda form
= '(' (method init option),
 specialized lambda list, (* B.2.1 *)
specialized lambda list,
 (form), *)
catch macro
= '(' 'catch'
 tag, form, *)
tag
= symbol; (* A.16 *)
throw macro
= '(' 'throw'
 tag, form, *)
Programming Language EuLisp, version 0.99

References


Class Index

<character> (character), 37
<char-file-stream> (stream), 62
<class-name> (1, 5
<condition> (condition), 22
<cons> (list), 56
<double-float> (double), 47
<fixed-precision-integer> (fprint), 52
<float> (float), 51
<integer> (integer), 55
<local-slot-description> (level-1), 79
<lock> (lock), 21
.null> (list), 56
<number> (number), 59
<object> (level-0), 13
<slot-description> (level-1), 79
<stream> (stream), 62
<string> (string), 66
<string-stream> (stream), 62
<structure> (level-0), 13
<symbol> (symbol), 69
<table> (table), 70
<thread> (thread), 20
<vector> (vector), 71
Condition Index

a-condition(), 5
arithmetic-condition (number), 59
<cannot-update-setter> (level-0), 29
cannot-update-setter (level-0), 29
collection-condition (collection), 40, 39
conversion-condition (collection), 42, 45
division-by-zero (number), 61, 61, 61, 59
domain-condition (elementary-functions), 49, 49, 49, 50, 50, 50, 23
dynamic-multiply-defined (level-1), 94, 94
end-of-stream (stream), 62, 63, 63, 64, 63
environment-condition (condition), 23
execution-condition (condition), 22
inappropriate-stream-position (stream), 63
incompatible-method-and-gf (level-1), 92
incompatible-method-domain (level-0), 15, 16, 17
invalid-operator (level-0), 28, 28, 28
method-domain-clash (level-0), 15, 16, 17
method-exists (level-1), 92
no-applicable-method (convert), 45, 16, 17
no-converter (convert), 45
non-congruent-lambda-lists (level-0), 15, 16, 76, 17
no-next-method (level-0), 17, 17
no-setter (level-0), 29, 29
range-condition (elementary-functions), 49, 50, 50, 50, 50, 23
scan-mismatch (formatted-io), 53, 53
telos-condition (level-0), 14
thread-already-started (thread), 21, 21
thread-condition (thread), 21
unbound-dynamic-variable (level-1), 93, 93, 93
wrong-condition-class (condition), 24, 24
wrong-thread-continuation (threads), 20, 21
Constant Index
<fpi> (fpint), 52
least-negative-double-float (double), 48
least-positive-double-float (double), 47
maximum-vector-index (vector), 71
most-negative-double-float (double), 48
most-negative-fixed-precision-integer (fpint), 52
most-positive-double-float (double), 47
most-positive-fixed-precision-integer (fpint), 52
nil (level-0), 26
pi (elementary-functions), 49
t (level-0), 26
ticks-per-second (event), 33
## Function Index

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>* (number), + (number), - (number), / (number)</td>
<td>Arithmetic Functions</td>
<td>0</td>
</tr>
<tr>
<td>&lt; (compare), = (compare)</td>
<td>Comparison Functions</td>
<td>0</td>
</tr>
<tr>
<td>abs (number)</td>
<td>Absolute Value Function</td>
<td>0</td>
</tr>
<tr>
<td>a-function ()</td>
<td>Function</td>
<td>0</td>
</tr>
<tr>
<td>allocate (level-0)</td>
<td>Allocate Function</td>
<td>0</td>
</tr>
<tr>
<td>apply (level-0)</td>
<td>Apply Method</td>
<td>0</td>
</tr>
<tr>
<td>apply-method (level-1), apply-method-function (level-1)</td>
<td>Apply Method Function</td>
<td>1</td>
</tr>
<tr>
<td>atom (list)</td>
<td>Atom</td>
<td>0</td>
</tr>
<tr>
<td>cdr (list)</td>
<td>Cdr</td>
<td>0</td>
</tr>
<tr>
<td>error (condition)</td>
<td>Error</td>
<td>0</td>
</tr>
<tr>
<td>characterp (character)</td>
<td>Character Function</td>
<td>0</td>
</tr>
<tr>
<td>class-initargs (level-1)</td>
<td>Class Initialization Function</td>
<td>1</td>
</tr>
<tr>
<td>class-of (level-0)</td>
<td>Class</td>
<td>1</td>
</tr>
<tr>
<td>class-precedence-list (level-1)</td>
<td>Class Precedence List Function</td>
<td>1</td>
</tr>
<tr>
<td>class-slot-descriptions (level-1)</td>
<td>Class Slot Descriptions</td>
<td>1</td>
</tr>
<tr>
<td>conditionp (condition)</td>
<td>Condition Function</td>
<td>0</td>
</tr>
<tr>
<td>cons (list)</td>
<td>Cons</td>
<td>0</td>
</tr>
<tr>
<td>convert (convert)</td>
<td>Convert Function</td>
<td>0</td>
</tr>
<tr>
<td>converter (convert)</td>
<td>Converter Function</td>
<td>0</td>
</tr>
<tr>
<td>current-thread (thread)</td>
<td>Current Thread Function</td>
<td>0</td>
</tr>
<tr>
<td>double-float-p (double)</td>
<td>Double Float Function</td>
<td>0</td>
</tr>
<tr>
<td>end-of-stream-p (stream)</td>
<td>End of Stream Function</td>
<td>0</td>
</tr>
<tr>
<td>eq (compare), eql (compare)</td>
<td>Equality Functions</td>
<td>0</td>
</tr>
<tr>
<td>error (condition)</td>
<td>Error</td>
<td>0</td>
</tr>
<tr>
<td>float (float)</td>
<td>Float</td>
<td>0</td>
</tr>
<tr>
<td>format (formatted-io)</td>
<td>Format Function</td>
<td>0</td>
</tr>
<tr>
<td>gcd (number)</td>
<td>Gcd</td>
<td>0</td>
</tr>
<tr>
<td>generic-function-discriminating-function (level-1)</td>
<td>Generic Function Discriminating Function</td>
<td>1</td>
</tr>
<tr>
<td>generic-function-domain (level-1)</td>
<td>Generic Function Domain</td>
<td>1</td>
</tr>
<tr>
<td>generic-function-method (level-0)</td>
<td>Generic Function Method</td>
<td>0</td>
</tr>
<tr>
<td>generic-function-method-domain (level-1)</td>
<td>Generic Function Method Domain</td>
<td>1</td>
</tr>
<tr>
<td>generic-function-method-lookup-function (level-1)</td>
<td>Generic Function Method Lookup Function</td>
<td>1</td>
</tr>
<tr>
<td>generic-function-methods (level-1)</td>
<td>Generic Function Methods</td>
<td>1</td>
</tr>
<tr>
<td>gensym (symbol)</td>
<td>Gensym</td>
<td>0</td>
</tr>
<tr>
<td>integerp (integer)</td>
<td>Integer Function</td>
<td>0</td>
</tr>
<tr>
<td>lcm (number)</td>
<td>Lcm</td>
<td>0</td>
</tr>
<tr>
<td>list (list)</td>
<td>List</td>
<td>0</td>
</tr>
<tr>
<td>lock (lock)</td>
<td>Lock</td>
<td>0</td>
</tr>
<tr>
<td>lockp (lock)</td>
<td>Lock Function</td>
<td>0</td>
</tr>
<tr>
<td>make (level-0)</td>
<td>Make</td>
<td>0</td>
</tr>
<tr>
<td>max (compare)</td>
<td>Max</td>
<td>0</td>
</tr>
<tr>
<td>method (level-1)</td>
<td>Method Function</td>
<td>1</td>
</tr>
<tr>
<td>method-domain (level-1)</td>
<td>Method Domain Function</td>
<td>1</td>
</tr>
<tr>
<td>method-function (level-1)</td>
<td>Method Function Function</td>
<td>1</td>
</tr>
<tr>
<td>min (compare)</td>
<td>Min</td>
<td>0</td>
</tr>
<tr>
<td>negativep (number)</td>
<td>Negative Function</td>
<td>0</td>
</tr>
<tr>
<td>newline (stream)</td>
<td>Newline Function</td>
<td>0</td>
</tr>
<tr>
<td>null (list)</td>
<td>Null</td>
<td>0</td>
</tr>
<tr>
<td>numberp (number)</td>
<td>Number Function</td>
<td>0</td>
</tr>
<tr>
<td>oddp (integer)</td>
<td>Odd</td>
<td>0</td>
</tr>
<tr>
<td>open (stream)</td>
<td>Open Stream Function</td>
<td>0</td>
</tr>
<tr>
<td>positivep (number)</td>
<td>Positive Function</td>
<td>0</td>
</tr>
<tr>
<td>primitive-allocate (level-1)</td>
<td>Primitive Allocate Function</td>
<td>1</td>
</tr>
<tr>
<td>primitive-class-of (level-1)</td>
<td>Primitive Class Of Function</td>
<td>1</td>
</tr>
</tbody>
</table>

*1/0/0
Macro Index

and (level-0), 30
block (level-0), 30
catch (level-1), 94
cond (level-0), 29
defmethod (level-0), 16
defmethod (level-1), 76
generic-lambda (level-0), 16
generic-lambda (level-1), 75
let* (level-0), 32
let (level-0), 31
method-function-lambda (level-1), 77
method-lambda (level-1), 76
or (level-0), 30
quasiquote (syntax-0), 33
return-from (level-0), 31
throw (level-1), 94
Generic Function Index

accumulate (collection), 39
accumulated (collection), 39
acos (elementary-functions), 49
add-method (level-1), 91
a-generic (), 5
allocate (level-1), 81
any? (collection), 39
asin (elementary-functions), 49
as-lowercase (character), 38
as-uppercase (character), 38
atan (elementary-functions), 49
atan2 (elementary-functions), 49
binary* (number), 61
binary+ (number), 60
binary- (number), 61
binary/ (number), 61
binary< (compare), 44
binary> (compare), 44
binary% (number), 61
binary-gcd (number), 61
binary-lcm (number), 61
binary-mod (number), 61
ceiling (float), 51
close (stream), 63
concatenate (collection), 40
cos (elementary-functions), 49
cosh (elementary-functions), 49
cosine (elementary-functions), 50
initialize (level-0), 18
input (stream), 64
log (elementary-functions), 50
log10 (elementary-functions), 50
map (collection), 41
member (collection), 41
negate (number), 60
output (stream), 64
pow (elementary-functions), 50
reverse (collection), 41
round (float), 51
sequencep (collection), 41
(setter stream-position) (stream), 63
shallow-copy (copy), 46
sin (elementary-functions), 49
sinh (elementary-functions), 49
size (collection), 42
sqrt (elementary-functions), 50
stream (stream), 62
stream-position (stream), 63
tan (elementary-functions), 49
tanh (elementary-functions), 50
truncate (float), 51
uninput (stream), 64
wait (event), 33
zerop (number), 60
generic-prin (stream), 65
generic-write (stream), 65
input (stream), 64
log (elementary-functions), 50
log10 (elementary-functions), 50
map (collection), 41
member (collection), 41
negate (number), 60
output (stream), 64
pow (elementary-functions), 50
reverse (collection), 41
round (float), 51
sequencep (collection), 41
(setter stream-position) (stream), 63
shallow-copy (copy), 46
sin (elementary-functions), 49
sinh (elementary-functions), 49
size (collection), 42
sqrt (elementary-functions), 50
stream (stream), 62
stream-position (stream), 63
tan (elementary-functions), 49
tanh (elementary-functions), 50
truncate (float), 51
uninput (stream), 64
wait (event), 33
zerop (number), 60
generic-prin (stream), 65
generic-write (stream), 65
initialize (level-0), 18
input (stream), 64
log (elementary-functions), 50
log10 (elementary-functions), 50
map (collection), 41
member (collection), 41
negate (number), 60
output (stream), 64
pow (elementary-functions), 50
reverse (collection), 41
round (float), 51
sequencep (collection), 41
(setter stream-position) (stream), 63
shallow-copy (copy), 46
sin (elementary-functions), 49
sinh (elementary-functions), 49
size (collection), 42
sqrt (elementary-functions), 50
stream (stream), 62
stream-position (stream), 63
tan (elementary-functions), 49
tanh (elementary-functions), 50
truncate (float), 51
uninput (stream), 64
wait (event), 33
zerop (number), 60

Method Index
acos (double), 47
add-method (level-1), 92
a-generic (), 5
allocate (level-1), 81
asin (double), 47
as-lowercase (character), 38
as-lowercase (string), 67
as-uppercase (character), 38
as-uppercase (string), 67
atan (double), 47
atan2 (double), 47
binary* (double), 52
binary* (fpint), 52
binary* (fpint), 52
binary* (double), 47
binary (fpint), 52
binary (double), 52
binary/ (fpint), 52
binary< (character), 37
binary< (double), 47
binary< (fpint), 52
binary< (string), 67
cos (double), 47
deprecated (copy), 46
deprecated (copy), 46
deprecated (list), 57
deprecated (string), 67
deprecated (vector), 71
deprecated-slot-reader (level-1), 88
deprecated-slot-writer (level-1), 89
equal (character), 37
equal (compare), 43
equal (double), 48
equal (fpint), 52
equal (list), 57
equal (string), 67
ensure-slot-reader (level-1), 71
ensure-slot-writer (level-1), 89
evenp (fpint), 52
exp (double), 47
floor (double), 47
generic-prin (character), 38
generic-prin (double), 48
generic-prin (fpint), 52
generic-prin (list), 56
generic-prin (string), 67
generic-prin (symbol), 69
generic-prin (vector), 71
generic-write (character), 38
generic-write (double), 48
generic-write (fpint), 52
generic-write (list), 56
generic-write (string), 67
generic-write (symbol), 69
generic-write (vector), 71
initialize (condition), 23
initialize (level-0), 19
initialize (level-1), 81
initialize (level-2), 82
initialize (level-3), 82
initialize (level-4), 83
input (stream), 64
log (double), 47
log10 (double), 47
mod (double), 47
mod (fpint), 52
negate (double), 47
negate (fpint), 52
output (stream), 64
pow (double), 47
round (double), 47
shallow-copy (copy), 47
shallow-copy (copy), 47
shallow-copy (list), 57
shallow-copy (string), 67
shallow-copy (vector), 71
sin (double), 47
sinh (double), 47
sqrt (double), 47
tan (double), 47
tanh (double), 47
truncate (double), 47
uninput (stream), 64
wait (thread), 21
zerop (double), 47
zerop (fpint), 52
(converter <vector>) (collection), 42
(converter <double-float>) (fpint), 52
(converter <fixed-precision-integer>) (double), 48
(converter <list>) (collection), 42
(converter <string>) (collection), 42
(converter <string>) (double), 48
(converter <string>) (fpint), 52
(converter <string>) (stream), 65
(converter <string>) (symbol), 70
(converter <symbol>) (string), 66
(converter <table>) (collection), 42
(converter <vector>) (collection), 42
(converter <double-float>) (fpint), 52
(converter <fixed-precision-integer>) (double), 48
(converter <list>) (collection), 42
(converter <string>) (collection), 42
(converter <string>) (double), 48
(converter <string>) (fpint), 52
(converter <string>) (stream), 65
(converter <string>) (symbol), 70
(converter <symbol>) (string), 66
(converter <table>) (collection), 42
General Index
21
(, 56
(converter <double-float>), 52
(converter <fixed-precision-integer>), 48
(converter <list>), 42
(converter <string>), 42, 48, 52, 65, 70
(converter <symbol>), 66
(converter <table>), 42
(converter <vector>), 42
(setter car), 57
(setter cdr), 57
(setter converter), 46
(setter element), 40
(setter primitive-class-of), 92
(setter primitive-ref), 93
(setter stream-position), 63
*, 59
+, 59
-, 59
/, 59
<, 44
<char-file-stream>, 62
<character>., 37
<class-name>, 5
<condition>, 22
<cons>, 56
<double-float>, 47
<fixed-precision-integer>, 52
<float>, 51
<integer>, 55
<local-slot-description>, 79
<lock>, 21
<mult>, 56
<number>, 59
<object>, 13
<slot-description>, 79
<string-stream>, 62
<string>., 66
<structure>, 13
<symbol>, 69
<table>, 70
<thread>, 20
<vector>, 71
=, 43
a-function, 4
a-generic, 5
a-special-form, 4
abs, 60
accumulate, 39
accumulate1, 39
acos, 47, 49
add-method, 91, 92
allocate, 18, 81
and, 30
anyp, 39
apply, 28
apply-method, 77
apply-method-function, 77
as-lowercase, 38, 67
as-uppercase, 38, 67
asin, 47, 49
assignment, 28
atan, 47, 49
atan2, 47, 49
atom, 57
backquoting, 33
base, 55
arbitrary base literals, 55
limitation on input, 55
binary literals, 55
binary*, 47, 52, 61
binary+, 47, 52, 60
binary-, 47, 52, 61
binary-gcd, 52, 61
binary-lcm, 52, 61
binary-mod, 61
binary/, 47, 52, 61
binary<, 37, 47, 52, 67
binary=, 44, 47, 52
binary%, 52, 61
dynamically scoped, 2
module, 27
of module names, 10, 12
top dynamic, 94
block, 30
see also let/cc, 31
boolean, 6
call-method, 77
call-method-function, 77
call-next-method, 17
car, 57
case sensitivity, 9
catch, 94
cdr, 57
ceiling, 47, 51
cerror, 25
character, 37
character-extension glyph, 37
module, 37
character-stream-p, 62
characterp, 37
class, 2, 6
primitive, 2
self-instantiated, 7
class-initargs, 78
class-of, 78
class-precedence-list, 78
class-slot-descriptions, 78
clear-table, 70
CLOS, 2
close, 63
coercion, 59
collection, 39
collection-alignment, 39
collection-condition, 41
collectionp, 40
comment, 9
collection-comment-begin glyph, 9
Common Lisp Error System, 22
module, 43
compatible-superclass-p, 83
compatible-superclasses-p, 83
compliance, 4
compute-and-ensure-slot-accessors, 87
compute-class-precedence-list, 84
compute-constructor, 81
compute-defined-slot-description, 85, 86
compute-defined-slot-description-class, 86
compute-discriminating-function, 91
compute-inherited-initargs, 85
compute-inherited-slot-descriptions, 85
compute-init-args, 84, 85
compute-method-lookup-function, 91
compute-predicate, 81
compute-primitive-reader-using-class, 90
compute-primitive-reader-using-slot-description, 89
compute-primitive-writer-using-class, 90
compute-primitive-writer-using-slot-description, 90
compute-slot-descriptions, 84
compute-slot-reader, 88
compute-slot-writer, 88
compute-specialized-slot-description, 86
compute-specialized-slot-description-class, 86, 87
concatenate, 40
cond, 29
condition, 22
continuous, 22
conditionp, 23
configuration, 3
conformance, 3
level-0, 4
level-1, 4
conforming processor, 3
conforming program, 3
conformity clause, 3
least negative double precision float, 48
least positive double precision float, 47
maximum vector index, 71
most negative double precision float, 48
most negative fixed precision integer, 52
most positive double precision float, 47
most positive fixed precision integer, 52
cons, 57
consp, 57
constant, 26
definition, 26
list, 26
continuation, 31, 95
conventions, 4
convert, 45
module, 45
converter, 46, 46
cos, 47, 49
cosh, 47, 49
current-thread, 20
deep-copy, 46, 57, 67, 71
defclass, 74
defcondition, 23
defconstant, 26
defgeneric, 13, 15, 75
defining form, 6
defclass, 74
defcondition, 23
defconstant, 26
defgeneric, 15, 75
deflocal, 26
defstruct, 14
defvar, 94
deflocal, 26
defmacro, 27
defmethod, 16, 76
defmodule, 12
defstruct, 14
defun, 27
defvar, 94
direct instance, 6
direct subclass, 6
direct superclass, 6
do, 40
double float, 47
  module, 47
double-float-p, 47
dynamic, 93
dynamic environment, 7
dynamic error, 3
dynamic-let, 93
dynamic-seq, 93
element, 40
elementary functions, 49, 49
emptyp, 40
end-of-stream-p, 63
ensure-slot-reader, 88
ensure-slot-writer, 89
environmental error, 4
eq, 43
  implementation-defined behaviour, 43
eq, 43
equal, 37, 43, 48, 52, 57, 67, 71
error, 3, 4
error, 25
can be signalled, 4
dynamic, 3
environmental, 4
handler, 24
signalled, 3
static, 4
evenp, 52, 55
exp, 47, 50
extension, 3
  floating point, 51
  integer, 55
  list, 56
  null (empty list), 56
  pair, 56
  string, 66
  vector, 71
external representation (see also prin and write), 27
false, 6
file-stream-p, 62
fill, 41
  fixed precision integer, 52
  module, 52
fixed-precision-integer-p, 52
float, 51
floor, 47, 51
flush, 63
format, 53
formatted-io, 53, 53
function, 7
  calling, 28
  reader, 29
  standard function, 6
  writer, 29
function call, 28
gcd, 60
generic arithmetic, 59
generic function, 7
  lambda-list, 15
generic-function-discriminating-function, 80
generic-function-domain, 79
generic-function-method-class, 80
generic-function-method-lookup-function, 80
generic-function-methods, 80
generic-lambda, 13, 16, 75
Programming Language EuLisp, version 0.99

generic-print, 38, 48, 52, 56, 58, 65, 67, 69, 71
generic-write, 38, 48, 52, 56, 58, 65, 67, 69, 71
gensym, 69
Haskell, 1, 10
hexadecimal literals, 55
identifier, 68
  definition of, 68
  peculiar identifiers, 68
  syntax, 9
if, 29
implementation-defined, 3
  behaviour of eq, 43
  least negative double precision float, 48
  least positive double precision float, 47
  maximum vector index, 71
  module directives, 10
  most negative double precision float, 48
  most positive double precision float, 47
  most positive fixed precision integer, 52
  time units per second, 33
unhandled conditions, 25
indirect instance, 7
indirect subclass, 7
  single, 14
inheritance graph, 7
inherited skt description, 7
init-list, 7
initarg, 7
initform, 7
initialization, 18
initialize, 18, 19, 23, 81, 82, 83
inner dynamic, 7
inner lexical, 7
input, 64
instance, 7
  direct, 6
  indirect, 7
instantiation graph, 7
integer, 55
integerp, 55
labels, 31
lambda, 27
lcm, 60
LeLisp, 10
let, 31
let*, 32
let/cc, 30
see also block and return-from, 30
level-0, 26
  <character>, 37
  <char-file-stream>, 62
  <double-float>, 47
  <float>, 51
  <fixed-precision-integer>, 52
  <integer>, 55
  <lock>, 22
  <null>, 56
  <object>, 13
  <cons>, 56
  <stream>, 62
  <string>, 66
  <string-stream>, 62
  <structure>, 14
  <symbol>, 69
  <table>, 70
  <thread>, 20
  <vector>, 71
character, 37
collection, 39
compare, 43
condition, 22
convert, 45
copy, 46
double-float, 47
elementary-functions, 49
eulisp-level-0, 2
float, 51
formatted-io, 53
fpi, 52
integer, 55
lock, 19
number, 59
stream, 62
string, 66
symbol, 68
table, 70
thread, 19
vector, 71
lexical environment, 7
list, 56
list, 58
module, 56
literal, 26
  arbitrary base, 55
  binary, 55
  character, 37
  hexadecimal, 55
  modification of, 27
  octal, 55
  quotation, 27
lock, 19, 21
lockp, 22
log, 47, 50
log10, 47, 50
macro, 27
definition by defmacro, 27
macro expansion—see also syntax, 12
macros—see also syntax, 12
make, 18
map, 41
max, 44
MCS, 2
member, 41
metaclass, 7
method, 7
  bindings, 17
  specificity, 7
method specificity, 7
method-domain, 80
method-function, 80
method-function-lambda, 77
method-generic-function, 80
method-lambda, 76
MicroCeyx, 2
min, 44
mod, 47, 52
module, 2, 10
directives, 10
environments, 2
export, 10
expose, 11
import, 10

106
name bindings, 10, 12
syntax, 11
multi-method, 7
negate, 47, 52, 60
negativep, 60
new instance, 7
newline, 65
next-method-p, 17
null, 56
number, 59
coercion, 59
module, 59
numberp, 59
Oaklisp, 2
object, 18
syntax, 72
ObjVlisp, 2
octal literals, 55
oddp, 55
open, 62
or, 30
output, 64
pair, 56
pair, 56
positivep, 60
pow, 47, 50
primitive-allocate, 92
primitive-class-of, 92
primitive-ref, 93
print, 64
print, 64
constants, 26
symbols, 26
processor, 3
gensym names, 69
progn, 32
quasiquote, 33
quasiquotation, 33
abbreviation with `, 34
quote, 26
abbreviation with `#, 27
range-condition, 50
read-line, 64
reflective, 7
return-from, 31
see also let/cc, 31
reverse, 41
round, 47, 51
scan, 53
in labels expressions, 31
of dynamic-let bindings, 94
of lambdaboundings, 27
of let/cc binding, 30
self-instantiated class, 7
sequencep, 41
setq, 28
setter, 29
setter function, 7
shallow-copy, 46, 47, 57, 67, 71
signal, 24
signum, 60
simple function, 8
sin, 47, 49
single inheritance, 14
sinh, 47, 49
size, 42
slot, 8
slot description, 8
slot option, 8
slot specification, 8
slot-description-initfunction, 79
slot-description-name, 79
slot-description-slot-reader, 79
slot-description-slot-writer, 79
special form, 2, 8
a-special-form, 4
call-next-method, 17
dynamic, 93
dynamic-let, 93
dynamic-setq, 93
if, 29
labels, 31
lambda, 27
let/cc, 30
next-method-p, 17
progn, 32
quote, 26
setq, 28
unwind-protect, 32
with-handler, 24
specialize, 8
specialize on, 8
sqrt, 47, 50
Standard ML, 1, 10, 22
standard module, 2
standard-error-stream, 62
standard-input-stream, 62
standard-output-stream, 62
static error, 4
stream, 62
module, 62
stream-position, 63
streamop, 62
string, 66
string, 66
escaping in, 66
module, 66
string-escape glyph, 66
strings, 66
subclass, 8
direct, 6
indirect, 7
superclass, 8
direct, 6
symbol, 26
symbol, 68, 68
module, 68
symbol-exists-p, 69
symbol-name, 69
symbolp, 69
syntax, 9
(), 56, 12, 72
caller, 37
class, 26
defmacro, 27
defmodule, 12
defun, 27
def, 27
float, 51
function call, 28
generic function lambda-list, 15
identifier, 9
integer, 55
pair, 56
string, 66
symbol, 26, 68
Programming Language EuLisp, version 0.99

unquote, 34
unquote-splicing, 34
vector, 71

syntax expansion, 12
T, 1, 2
table, 70

mode, 70
tablep, 70
tan, 47, 49
tanh, 47, 50
thread, 19, 20
thread-reschedule, 20
thread-start, 21
thread-value, 21
threadp, 20
throw, 94
top dynamic, 8
top lexical, 8
ttrue, 6
trunc, 47, 51
uninput, 64
unlock, 22

unquote, 34
abbreviation with ., 34
unquote-splicing, 34
abbreviation to ., 34

unwind-protect, 32
vector, 71
vector, 71, 71
vectorp, 71

wait, 21, 33
definition of, 9

with-handler, 24
write, 65
zerop, 47, 52, 60

%, 60

Common Lisp, 1, 2
level-0, 2
level-1, 2
libraries, 2

InterLISP, 1, 2
LE-LISP, 1, 2
LISP/VM, 1, 2