Programming Language EuLisp

Version 0.991

WORKING DRAFT

Contents

Page

1 Language Structure . . . 3

2 Scope . . . 3

3 Normative References . . . 3

4 Conformance Definitions . . . 4

5 Error Definitions . . . 4

6 Compliance . . . 4

7 Conventions . . . 5
- 7.1 Layout and Typography . . . 5
- 7.2 Naming . . . 6

8 Definitions . . . 7

9 Lexical Syntax . . . 9
- 9.1 Character Set . . . 9
- 9.2 Whitespace and Comments . . . 9
- 9.3 Identifiers . . . 10
- 9.4 Objects . . . 10
- 9.5 Boolean . . . 10

10 Modules . . . 11
- 10.1 Module Definition . . . 11
- 10.2 Directives . . . 12
  - 10.2.1 `export` Directive . . . 12
  - 10.2.2 `import` Directive . . . 12
  - 10.2.3 `expose` Directive . . . 13
  - 10.2.4 `syntax` Directive . . . 13
- 10.3 Definitions and Expressions . . . 13
- 10.4 Special Forms . . . 14
- 10.5 Module Processing . . . 14

11 Objects . . . 15
- 11.1 System Defined Classes . . . 15
- 11.2 Single Inheritance . . . 15
- 11.3 Defining Classes . . . 16
- 11.4 Defining Generic Functions and Methods . . . 17
- 11.5 Specializing Methods . . . 18
- 11.6 Method Lookup and Generic Dispatch . . . 19
- 11.7 Creating and Initializing Objects . . . 19
- 11.8 Accessing Slots . . . 20
- 11.9 Other Abstract Classes . . . 20

12 Level-0 Defining, Special and Function-call Forms . . . 21
- 12.1 Simple Expressions . . . 21
- 12.2 Functions: creation, definition and application . . . 21
- 12.3 Destructive Operations . . . 23
- 12.4 Conditional Expressions . . . 24
- 12.5 Variable Binding and Sequences . . . 25
- 12.6 Quasiquotation Expressions . . . 27
- 12.7 Summary of Level-0 Defining, Special and Function-call Forms . . . 28
  - 12.7.1 Syntax of Level-0 modules . . . 29
  - 12.7.2 Syntax of Level-0 defining forms . . . 29
  - 12.7.3 Syntax of Level-0 special forms . . . 30
  - 12.7.4 Syntax of Level-0 function calls . . . 31
- 12.8 Conditions . . . 31

13 Condition Classes . . . 31

14 Condition Signalling and Handling . . . 33

15 Concurrency . . . 35
- 15.1 Threads . . . 36
- 15.2 Locks . . . 38

16 Level-0 Module Library . . . 38

16.1 Characters . . . 38
16.2 Collections . . . 40
16.3 Comparison . . . 46
16.4 Conversion . . . 48
16.5 Copying . . . 49
16.6 Double Precision Floats . . . 50
16.7 Floating Point Numbers . . . 51
16.8 Formatted-IO . . . 52
16.9 Fixed Precision Integers . . . 54
16.10 Integers . . . 55
16.11 Keywords . . . 56
16.12 Lists . . . 58
16.13 Elementary Functions . . . 60
16.14 Numbers . . . 62
16.15 Streams . . . 65
16.15.1 Stream classes . . . 65
16.15.2 Stream operators . . . 66
16.15.3 Stream objects . . . 67
16.15.4 Buffer management . . . 67
16.15.5 Reading from streams . . . 68
16.15.6 Writing to streams . . . 69
16.15.7 Additional functions . . . 70
16.15.8 Convenience forms . . . 71
16.16 Strings . . . 72
16.17 Symbols . . . 74
16.18 Tables . . . 75
16.19 Vectors . . . 76
16.20 Syntax of Level-0 objects . . . 77

17 Programming Language EuLisp, Level-1 . . . 78
17.1 Modules . . . 78
17.2 Classes and Objects . . . 78
17.3 Generic Functions . . . 79
17.4 Methods . . . 80
17.5 Object Introspection . . . 81
17.6 Class Introspection . . . 81
17.7 Slot Introspection . . . 82
17.8 Generic Function Introspection . . . 83
17.9 Method Introspection . . . 84
17.10 Class Initialization . . . 84
17.11 Slot Description Initialization . . . 85
17.12 Generic Function Initialization . . . 86
17.13 Method Initialization . . . 86
17.14 Inheritance Protocol . . . 86
17.15 Slot Access Protocol . . . 90
17.16 Method Lookup and Generic Dispatch . . . 93
17.17 Low Level Allocation Primitives . . . 94
17.18 Dynamic Binding . . . 95
17.19 Exit Extensions . . . 96
17.20 Syntax of Level-1 objects . . . 97
17.20.1 Syntax of Level-1 modules . . . 97
17.20.2 Syntax of Level-1 defining forms . . . 97
17.20.3 Syntax of Level-1 special forms . . . 97

Bibliography . . . 98

Module Index . . . 99

Class Index . . . 100

Special Forms Index . . . 101

Function Index . . . 102

Generic Function Index . . . 103

Condition Index . . . 104

Constant Index . . . 105

Index . . . 106

Figures

Example 1 — module directives 13
Example 1 — module using `expose` 13
Example 1 — using `let/cc` 25
Example 2 — Interaction of `unwind-protect` with non-local exits 27
Example 1 — handler actions 34
Example 1 — Examples of string literals 72

Tables

Table 1 — Level-0 class hierarchy 15
Table 2 — Condition class hierarchy 32
Table 3 — Character digrams 39
Table 4 — Level-0 number class hierarchy 62
Table 5 — Level-1 metaclass hierarchy 81
Table 6 — Level-1 class hierarchy 82
Table 7 — Initialization Call Structure 85

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 [15] and at the European Conference on Artificial Intelligence (ECAI-86) held in Brighton, Sussex [22]. 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.

Work on this version started in 2010 from the material archived by Julian Padget in 1993 with the aim of finalising an Eu Lisp-1.0 definition as close to the plans of the original contributors as is possible to ascertain from the remaining documents. If there is interest from any of the original contributors or others parties to participate in the process of finalising EuLisp-1.0 your input would be greatly appreciated.

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

The ancestors of EuLisp (in alphabetical order) are Common Lisp [20], InterLISP [23], LE-LISP [4], LISP/VM [1], Scheme [6], and T [17] [18]. 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 [14] and the draft report on Haskell [10] 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 Council in Germany (Academic Research Collaboration programme), British Standards Institute, Deutscher Akademischer Austauschdienst (DAAD), Departament de Llenguatges 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 Coordination Acoustique Musique (IRCAM), Ministerio de Educacion y Ciencia (MEC), Rank Xerox France, Science and Engineering Research Council (UK), Siemens AG, University of Bath, University of Technology, Delft, University of Edinburgh, Universität Erlangen and Université Paris VI (LITP).

The following people (in alphabetical order) have contributed in various ways to the evolution of the language: Giuseppe Attardi, Javier Béjar, Neil Berrington, Russell Bradford, Harry Bretthauer, Peter Broadbery, Christopher Burdorf, Jérôme Chailloux, Odile Chenetier, Thomas Christaller, Jeff Dalton, Klaus Däßler, Harley Davis, David DeRoure, John Fitch, Richard Gabriel, Brigitte Glas, Nicolas Graube, Dieter Kolb, Jürgen Kopp, Antonio Moreno, Eugen Neidl, Greg Nuyens, Pierre Parquier, Keith Playford, Willem van der Poel, Christian Queinnec, Nitsan Seniak, Enric Sesa, Herbert Stoyan, and Richard Tobin.

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

Harry Bretthauer (`harry.bretthauer@gmd.de`) GMD mbH
Postfach 1316
53737 Sankt Augustin
Germany

original editors.

Programming Language EuLisp — Version 0.991

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 forty 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 supports both propagation of conditions and continuable 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, LE-LISP, LISP/VM, Scheme, and T. EuLisp both introduces new ideas and takes from these Lisps. It also extends or simplifies their ideas as seen fit. 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 TELOS. TELOS incorporates elements of the Common Lisp Object System (CLOS) [3], ObjVLisp [7], Oaklisp [12], MicroCeyx [5], and MCS [24].

1 Language Structure

The EuLisp definition comprises the following items:

Level-0: comprises all the level-0 classes, functions, defining forms and special forms , which is this text minus §17. The object system can be extended by user-defined structure classes, and generic functions.

Level-1: extends level-0 with the classes, functions, defining forms and special forms defined in §17. The object system can be extended by user-defined classes and metaclasses. The implementation of level-1 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 `level-0`. This module imports and re-exports the modules specified in table 1.

Modules comprising `eulisp0`:

Module	Section(s)
`character`	16.1
`collection`	16.2
`compare`	16.3
`condition`	12.8
`convert`	16.4
`copy`	16.5
`double`	16.6
`fpi`	16.9
`formatted-io`	16.8
`function`	12.2
`keyword`	16.11
`list`	16.12
`lock`	15.2
`mathlib`	??
`number`	16.14
`telos0`	11
`stream`	16.15
`string`	16.16
`symbol`	16.17
`table`	16.18
`thread`	15.1
`vector`	16.19

This definition is organized in three parts:

Sections 9–12: describes 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.

Section 16: describes the classes required at level-0 and the operations defined on instances of those classes. The section is organized by module 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.

Section 17: describes the reflective aspects of the object system and how to program the metaobject protocol and some additional control forms.

Prior to these, sections 2–8 define the scope of the text, cite normative references, conformance definitions, error definitions, typographical and layout conventions and terminology definitions used in this text.

2 Scope

This text specifies the syntax and semantics of the computer programming language EuLisp by defining the requirements for a conforming EuLisp processor and a conforming EuLisp program (the textual representation of data and algorithms). This text does not specify:

a) The size or complexity of an EuLisp 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 EuLisp processor.

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

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

e) The typographical representation of a EuLisp 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 Normative References

The following standards contain provisions, which through references in this text constitute provisions of this definition. At the time of writing, the editions indicated were valid. All standards are subject to revision and parties making use of this definition are encouraged to apply the most recent edition of the standard listed below.

[ISO 646 : 1991] Information processing — ISO 7-bit coded character set for information interchange, 1991.

[ISO 2382] Data processing — vocabulary.

[ISO TR 10034 : 1990] Information technology — Guidelines for the preparation of conformity clauses in programming language standards.

[ISO TR 10176 : 1991] Information technology — Guidelines for the preparation of programming language standards. Note: this is currently a draft technical report.

[ISO/IEC 9899:1999] Programming Languages — C.

4 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.

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

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

4.3 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.

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

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

4.6 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.

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

4.8 processor
Compiler, translator or interpreter working in combination with a *configuration*.

5 Error Definitions

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

5.1 static error
An error which is detected during the execution of a EuLisp program or which is a violation of the defined behaviour of Eu Lisp. Static 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 EuLisp 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.

If the result of preparation is a runnable program, then that program must signal any static error.

5.2 environmental error
An error which is detected by the configuration supporting the EuLisp processor. The processor must signal the corresponding static error which is identified and handled as described above.

5.3 violation
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 in the program under preparation. A *conforming processor* is required to issue a diagnostic if a violation is detected.

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

6 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-0 and at level-1.

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.

7 Conventions

This section defines the conventions employed in this text, how definitions will be laid out, the typefaces to be used, the meta-language used in descriptions and the naming conventions. A later section (8) contains definitions of the terms used in this text.

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 1 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.

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 `()`—representing the *boolean* value false, or any value other than `()`, representing true.

7.1 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. The syntax notation used is based on that described in [ISO/IEC 9899:1999] with modifications to support the specification of a return type and to improve clarity. Syntactic categories (non-terminals) are indicated by *italic* type, and literal words and characters (terminals) by `constant width` type. A colon (:) following a non-terminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words "one of". An optional symbol is indicated by the subscript "*opt*", a list of zero or more occurrences of a symbol are indicated by the superscript "*", and a list of one or more occurrences of a symbol are indicated by the superscript "+". Examples of several kinds of entry are now given. Some subsections of entries are optional and are only given where it is felt necessary.

7.1.1 `a-special-form` *special operator*

7.1.1.1 Syntax

a-special-form-form: → *result-class*
  (`a-special-form` *form-1* ... *form-n*$_{opt}$)

Arguments

form-1 : description of structure and rôle of *form-1*.

⋮

form-n$_{opt}$: description of structure and rôle of optional argument *form-n*$_{opt}$.

Result

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

Remarks

Any additional information defining the behaviour of `a-special-form` or the syntax category *a-special-form-form*.

Examples

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

See also

Cross references to related entries.

7.1.2 `a-function` *function*

7.1.2.1 Signature

(`a-function` *argument-1* ... *argument-n*$_{opt}$)
  → *result-class*

Arguments

argument-1 : information about the class or classes of *argument-1*.

⋮

argument-n$_{opt}$: information about the class or classes of the optional argument *argument-n*.

Result

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

Remarks

Any additional information about the actions of `a-function`.

Examples

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

See also

Cross references to related entries.

7.1.3 `a-generic` *generic function*

Generic Arguments

argument-a `<class-a>` : means that *argument-a* of `a-generic` must be an instance of `<class-a>` and that *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 `<class-a>` for *argument-a*.

⋮

argument-n: means that (i) *argument-n* is an instance of `<object>`, *i.e.* it is unconstrained, (ii) `a-generic` does not specialize on *argument-n*, (iii) no method on `a-generic` can specialize on *argument-n*.

Result

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

Remarks

Any additional information about the actions of `a-generic`. This can take two forms depending on the function. This will probably be in general terms, since the actual behaviour will be determined by the methods.

See also

Cross references to related entries.

7.1.4 `a-generic <class-a>` *method*

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

Specialized Arguments

argument-a `<class-a>`: means that *argument-a* of the method must be an instance of `<class-a>`. Of course, this argument must be one which was defined in `a-generic` as being open to specialization and `<class-a>` must be a subclass of the class.

⋮

argument-n: means that (i) *argument-n* is an instance of `<object>`, *i.e.* it is unconstrained, because `a-generic` does not specialize on *argument-n*.

Result

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

Remarks

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

See also

Cross references to related entries.

7.1.5 `<a-condition>` `<condition>` *condition*

Initialization Options

`keyword-a` *value-a*: means that an instance of `<a-condition>` has a slot called `keyword-a` which should be initialized to *value-a*, where *value-a* is often the name of a class, indicating that *value-a* should be an instance of that class and a description of the information that *value-a* is supposed to provide about the exceptional situation that has arisen.

⋮

`keyword-n` *value-n*: As for `keyword-a`.

Remarks

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

7.1.6 `<a-class>` `<class>` *class*

Initialization Options

`keyword-a` *value-a*: means that `<a-class>` has an keyword whose name is `keyword-a` and the description will usually say of what class (or classes) *value-a* should be an instance. This keyword is required.

⋮

[`keyword-n` *value-n*] : The enclosing square brackets denote that this keyword is optional. Otherwise the interpretation of the definition is as for `keyword-a`.

Remarks

A description of the rôle of `<a-class>`.

7.1.7 `a-constant <a-class>` *constant*

Remarks

A description of the constant of type `<a-class>`.

7.2 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.

7.1 "`<name>`" wrapping: By convention, classes have names which begin with "`<`" and end with "`>`".

7.2 "`binary`" prefix: The two argument version of a n-ary argument function. For example `binary+` is the two argument (generic) function corresponding to the n-ary argument `+` function.

7.3 "`-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. 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.

7.4 "`generic-`" prefix: The generic version of the function named by the stem.

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

7.6 "`make-`" prefix: For most primitive or defined classes there is constructor function, which is usually named `make-`*class-name*.

7.7 "!" suffix: A destructive function is named by a "!" suffix, for example the destructive version of `reverse` is named `reverse!`.

7.8 "?" suffix: A predicate function is named by a "?" suffix, for example `cons?`.

8 Definitions

This set of definitions, which are be used throughout this document, 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*. Names in `courier` font refer to entities defined in the language.

8.1 abstract class
A Class that by definition has no direct instances.

8.2 applicable method
A *method* is applicable for a particular set of arguments if each element in its *domain* is a *superclass* of the *class* of the corresponding argument.

8.3 binding
A location containing a value.

8.4 class
A class is an *object* which describes the structure and behaviour of a set of *object*s 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*.

8.5 class precedence list
Each *class* has a linearised list of all its *super-classes*, *direct* and *indirect*, beginning with the *class* itself and ending with the root of the *inheritance graph*, the *class* `<object>`. This list determines the specificity of slot and method *inheritance*. A class's class precedence list may be accessed through the *function* `class-precedence-list`. The rules used to compute this list are determined by the *class* of the *class* through *methods* of the *generic function* `compute-class-precedence-list`.

8.6 class option
A keyword and its associated value applying to a *class* appearing in a class definition form, for example: the `predicate` keyword and its value, which defines a predicate *function* for the *class* being defined.

8.7 closure
A first-class function with *free variables* that are bound in the *lexical environment*. Such a function is said to be "closed over" its free variables. Example: the function returned by the expression `(let ((x 1)) #'(lambda () x))` is a closure since it closes over the free variable `x`.

8.8 congruent
A constraint on the form of the *lambda-list* of a method, which requires it to have the same number of elements as the generic function to which it is to be attached.

8.9 continuation
A continuation is a function of one parameter which represents the rest of the program. For every point in a program there is the remainder of the program coming after that point; this can be viewed as a function of one argument awaiting the result of that point. The *current continuation* is the continuation that would be derived from the current point in a program's execution, see `let/cc`.

8.10 converter function
The generic function associated with a class (the target) that is used to project an instance of another class (the source) to an instance of the target.

8.11 defining form
Any form or *syntax expression* expanding into a form whose operator is a *defining operator*.

8.12 defining operator
One of `defclass`, `defcondition`, `defconstant`, `defgeneric`, `deflocal`, `defsyntax`, `defun`, or `defglobal`.

8.13 direct instance
A direct instance of a class $class_1$ is any *object* whose most specific *class* is $class_1$.

8.14 direct subclass
A $class_1$ is a direct *subclass* of $class_2$ if $class_1$ is a *subclass* of $class_2$, $class_1$ is not identical to $class_2$, and there is no other $class_3$ which is a *superclass* of $class_1$ and a *subclass* of $class_2$.

8.15 direct superclass
A *direct superclass* of a class $class_1$ is any *class* for which $class_1$ is a direct *subclass*.

8.16 dynamic environment
The *inner* and *top dynamic* environment, taken together, are referred to as the dynamic environment.

8.17 dynamic extent
A lifetime constraint, such that the entity is created on control entering an expression and destroyed when control exits the expression. Thus the entity only exists for the time between control entering and exiting the expression.

8.18 dynamic scope
An access constraint, such that the *scope* of the entity is limited to the *dynamic extent* of the expression that created the entity.

8.19 extent
That lifetime for which an entity exists. Extent is constrained to be either *dynamic* or *indefinite*.

8.20 first-class
First-class entities are those which can be passed as parameters, returned from functions, or assigned into a variables.

8.21 function
A function is either a *continuation*, a *simple function* or a *generic function*.

8.22 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*.

8.23 identifier
An identifier is the syntactic representation of a *variable*.

8.24 indefinite extent
A lifetime constraint, such that the entity exists for ever. In practice, this means for as long as the entity is accessible.

8.25 indirect instance

An indirect instance of a class $class_1$ is any *object* whose *class* is an *indirect subclass* of $class_1$.

8.26 indirect subclass

A $class_1$ is an indirect subclass of $class_2$ if $class_1$ is a *subclass* of $class_2$, $class_1$ is not identical to $class_2$, and there is at least one other $class_3$ which is a *superclass* of $class_1$ and a *subclass* of $class_2$.

8.27 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*.

8.28 inherited slot description

A *slot description* is inherited for a $class_1$ if the *slot description* is defined for another $class_2$ which is a direct or indirect *superclass* of $class_1$.

8.29 keyword

A *keyword* used 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 *keyword* may be declared for a *slot* in a class definition form using the `keyword` *slot-option* or the `keywords` *class-option*.

8.30 default

A form which is evaluated to produce a default initial *slot* value. Defaults are closed in their *lexical* environments and the resulting *closure* is called a *default-function*. A *default* may be declared for a *slot* in a class definition form using the `default` *slot-option*.

8.31 default-function

A *function* of no arguments whose result is used as the default value of a *slot*. *default-functions* capture the *lexical* environment of a *default* declaration in a class definition form.

8.32 initlist

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

8.33 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*.

8.34 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*.

8.35 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*.

8.36 instantiation graph

A directed graph whose nodes are *object*s and whose edges are defined by the *instance* relations between the *object*s. This graph has only one cycle, an edge from `<class>` to itself. The instantiation graph is a tree and `<class>` is the root.

8.37 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.

8.38 lexical scope

An access constraint, such that the *scope* of the entity is limited to the textual region of the form creating the entity. See also *lexically closer* and *shadow*.

8.39 syntax operator

A syntax operator is distinguished by when it is used: syntax operators are only used during the syntax expansion of modules to transform expressions.

8.40 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.

8.41 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 *object*s which contain, among other information, a *function* representing the method's behaviour.

8.42 method function

A *function* which implements the behaviour of a particular *method*. Method functions have special restrictions which do not apply to all *functions*: their formal parameter bindings are immutable, and the special operators `call-next-method` and `next-method?` are only valid within the lexical scope of a method function.

8.43 method specificity

A domain $domain_1$ is more specific than another $domain_2$ if the first *class* in $domain_1$ is a *subclass* of the first *class* in $domain_2$, or, if they are the same, the rest of $domain_1$ is more specific than the rest of $domain_2$.

8.44 multi-method

A *method* which specializes on more than one argument.

8.45 new instance

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

8.46 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-extensible operations upon them.

8.47 scope

That part of the extent in which a given *variable* is accessible. Scope is constrained to be *lexical*, *dynamic* or *indefinite*.

8.48 self-instantiated class

A *class* which is an *instance* of itself. In EuLisp, `<class>` is the only example of a self-instantiated class.

8.49 setter function

The function associated with the function that accesses a place in an entity, which changes the value stored in that place.

8.50 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>`.

8.51 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*.

8.52 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-slots`. A slot description can be either *direct* or *inherited*.

8.53 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.

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

8.55 special form
Any form or *syntax expression* expanding into a form whose operator is a *special operator*. They are semantic primitives of the language and 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.

8.56 special operator
One of `a-special-form`, `call-next-handler`, `call-next-method`, `dynamic`, `dynamic-let`, `dynamic-setq`, `if`, `letfuns`, `lambda`, `let/cc`, `next-method?`, `progn`, `quote`, `setq`, `unwind-protect`, or `with-handler`.

8.57 specialize
A verbal form used to describe the creation of a more specific version of some entity. Normally applied to classes, slots and methods.

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

8.59 subclass
The behaviour and structure defined by a class *class*$_1$ are inherited by a set of *classes* which are termed *subclasses* of *class*$_1$. A *subclass* can be either *direct* or *indirect* or itself.

8.60 superclass
A *class*$_1$ is a superclass of *class*$_2$ iff *class*$_2$ is a subclass of *class*$_1$. A *superclass* can be either *direct* or *indirect* or itself.

8.61 top dynamic
Top dynamic bindings are created by `defglobal`, referenced by `dynamic` and modified by `dynamic-setq`. There is only one *top dynamic* environment.

8.62 top lexical
Top lexical bindings are created in the *top lexical* environment of a module by a defining form. All these bindings are immutable except those created by `deflocal` which creates a mutable top-lexical binding. All such bindings are referenced by *variables* and those made by `deflocal` are modified by `setq`. Each module defines its own distinct *top lexical* environment.

9 Lexical Syntax

9.1 Character Set

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.* `#\x3Ad`) 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 646 : 1991].

The minimal character set to support EuLisp is defined in syntax table 9.1. 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*. Any character not specified here is classified under *other-character*, which permits its use as an initial or a constituent character of an *identifier* (see § 9.3.0.3).

9.1.0.1 Syntax

```
decimal-digit: one of
      0 1 2 3 4 5 6 7 8 9
upper-letter: one of
      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: one of
      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
normal-other-character: one of
      * / < = > + .
other-character:
      normal-other-character
special-character: one of
      ; ' , \ " # ( ) ` | @
level-0-character:
      decimal-digit
      letter
      other-character
      special-character
```

9.2 Whitespace and Comments

Whitespace characters are spaces, newlines, line feeds, carriage returns, character tabulations, line tabulations and form feeds. 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. Whitespace separates tokens and is only significant in a string or when it occurs escaped within an *identifier*.

A line comment is introduced by a *semicolon* (`;`) and continues up to, but does not include, the end of the line. Hence, a line comment cannot occur in the middle of a token because of the whitespace in the form of the newline which is to *whitespace*. An *object* comment is introduced by the `#;` sequence optionally followed by *whitespace* and an *object* to be "commented out".

9.2.0.2 Syntax

whitespace:
 space
 newline
 line-feed
 return
 tab
 vertical-tab
 form-feed
comment:
 ; *all subsequent characters up to the end of the line*
 `#;` *whitespace** *object*

NOTE 1 There is no notation in EuLisp for block comments.

9.3 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 *letter*, *decimal-digit* and *other-characters* starting with a character that is not a *decimal-digit*. *special-characters* 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, `|).(|` is the *identifier* whose name contains the three characters `#\)`, `#\.` and `#\(`, and `a|b|` is the *identifier* whose name contains the characters `#\a` and `#\b`. The sequence `||` is the *identifier* with no name, and so is `||||`, but `|\||` is the *identifier* whose name contains the single character `|`, which can also be written `\|`, without delimiting *multiple-escapes*.

9.3.0.3 Syntax

identifier:
 normal-identifier
 peculiar-identifier
 escaped-identifier
normal-identifier:
 *normal-initial normal-constituent**
normal-initial:
 letter
 normal-other-character
normal-constituent:
 letter
 decimal-digit
 other-character
peculiar-identifier:
 `{+ | -}`
 {*peculiar-constituent normal-constituent**}$_{opt}$
 . *peculiar-constituent normal-constituent**
peculiar-constituent:
 letter
 other-character
escaped-identifier:
 *escaped-sequence escaped-sequences**
 *normal-initial escaped-sequences**
 *level-0-character escaped-sequences**
escaped-sequences:
 escaped-sequence
 *escaped-or-normal-constituent**
escaped-sequence:
 `|` *escaped-sequence-constituent** `|`
escaped-or-normal-constituent:
 level-0-character
 normal-constituent
escaped-sequence-constituent:
 level-0-character
 level-0-character `other than |`

9.4 Objects

An object is either a *literal*, a *symbol* or a *list*. 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 below:

9.4.0.4 Syntax

object:
 literal
 list §16.12
 symbol §16.17
literal:
 boolean
 character §16.1
 float §16.7
 integer §16.10
 string §16.16
 vector §16.19

9.5 Boolean

A boolean value is either *false* , which is represented by the empty list—written `()` and is also the value of `nil`—or *true* , which is represented by any other value than `()` or if specified as `t`:

9.5.0.5 Syntax

```
boolean:
        true
        false
true:
        t
        object not ()
false:
        ()
        nil
```

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 definition, to mean that set of values.

10 Modules

The EuLisp module scheme has several influences: LeLisp's module system and module compiler (complice), Haskell, ML [13], 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 level-0 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. 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.

> **top-lexical:** The top-lexical environment comprises all the locally defined bindings and all the imported bindings.
>
> **external:** 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.
>
> **syntax:** The syntax environment comprises all the bindings available for the syntax expansion of the module.

Each binding is a pair of a *local-name* and a *module-name*. It is a violation 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.

10.1 Module Definition

10.1.1 `defmodule` *syntax*

10.1.1.1 Syntax

```
defmodule-0-form:
    ( defmodule module-name
        module-directives
        level-0-module-form* )
module-name:
    identifier
module-directives:
    ( module-directive* )
module-directive:
    export ( identifier* )
    expose ( module-descriptor* )
    import ( module-descriptor* )
    syntax ( module-descriptor* )
level-0-module-form:
    ( export identifier* )
    level-0-form
    defining-0-form
    ( progn level-0-module-form* )
module-descriptor:
    module-name
    module-filter
module-filter:
    ( except ( identifier* ) module-descriptor )
    ( only ( identifier* ) module-descriptor )
    ( rename ( rename-pair* ) module-descriptor )
rename-pair:
    ( identifier identifier )
level-0-form:
    identifier
    literal
    special-0-form
    function-call-form
form:
    level-0-form
special-form:
    special-0-form
```

Arguments

module name: A symbol used to name the module.

module directives: A form specifying the exported names, exposed modules, imported modules and syntax modules used by this module.

module form: One of a defining form, an expression or an export directive.

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

Examples

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

10.2 Directives

The list of module directives is a sequence of keywords and forms, where the keywords indicate the interpretation of the forms (see syntax table 10.1.1.1). 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 syntax table 10.1.1.1 and an example of their use appears in example 1.

10.2.1 `export` Directive

This is denoted by the keyword `export` followed by a sequence of names of top-lexical bindings—these could be either locally-defined or imported—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.

10.2.2 `import` Directive

This is denoted by the keyword `import` followed by a sequence of *module-descriptor*s (see syntax table 10.1.1.1), being module names or the filters `except`, `only` and `rename`. This sequence 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.

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.

In processing import directives, every name should be thought of as a pair of a *module-0-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:

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

`except` : Filters the names from each *module-descriptor* discarding those 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.

`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 names exported by a module are required, since it is only necessary to name those that are wanted to include them.

`rename` : Filters the names from each *module-descriptor* replacing those with *old-name* by *new-name*. Any name not mentioned in the replacement list is passed unchanged. Note that once a name has been replaced the new-name is not compared against the replacement list again. Thus, a binding can only be renamed once by a single `rename` directive. In consequence, name exchanges are possible.

Example 1 – module directives

```
(defmodule a-module
  (import
    (module-1                                         ;; import everything from module-1
     (except (binding-a) module-2)                    ;; all but binding-a from module-2
     (only (binding-b) module-3)                      ;; only binding-b from module-3
     (rename
      ((binding-c binding-d) (binding-d binding-c))   ;; all of module-4, but exchange
      module-4))                                      ;; the names of binding-c and binding-d

   syntax
    (syntax-module-1                                  ;; all of the module syntax-module-1
     (rename ((syntax-a syntax-b))                    ;; rename the binding of syntax-a
      syntax-module-2)                                ;; of syntax-module-2 as syntax-b
     (rename ((syntax-c syntax-a))                    ;; rename the binding of syntax-c
      syntax-module-3))                               ;; of syntax-module-3 as syntax-a

   expose
    ((except (binding-e) module-5)                    ;; all but binding-e from module-5
     module-6)                                        ;; export all of module-6

   export
    (binding-1 binding-2 binding-3))                  ;; and three bindings from this module
  ...
  (export local-binding-4)                            ;; a fourth binding from this module
  ...
  (export binding-c)                                  ;; the imported binding binding-c
  ...)
```

10.2.3 `expose` Directive

This is denoted by the keyword `expose` followed by a list of *module-directive*s (see syntax table 10.1.1.1). The purpose of this directive is to allow a module to export subsets of the external environments of various modules without importing them itself. Processing an expose directive employs the same model as for imports, namely, a pair of a *module-name* and a *local-name* with the same filtering operations. When the namelist has been filtered, the names are added to the external environment of the module begin processed. A name clash can arise in the external environment from interaction with exports or between different exposed modules. As an example of the use of `expose`, a possible implementation of the `level-0` module is shown in example 1.

Example 1 – module using `expose`

```
(defmodule level-0
  (expose
    (character collection compare condition convert
     copy double-float elementary-functions event
     formatted-io int function
     keyword list lock number object-0 stream string
     symbol syntax-0 table thread vector)))
```

It is also meaningful for a module to include itself in an expose directive. In this way, it is possible to refer to all the bindings in the module being defined. This is convenient, in combination with `except` (see § 10.2.2), as a way of exporting all but a few bindings in a module, especially if syntax expansion creates additional bindings whose names are not known, but should be exported.

10.2.4 `syntax` Directive

This directive is processed in the same way as an import directive, except that the bindings are added to the syntax environment. This environment is used in the second phase of module processing (syntax expansion). These constitute the dependencies for the syntax expansion of the definitions and expressions in the body of the module. A name clash can arise in the syntax environment from interaction between different syntax modules.

It is important to note that special forms are considered part of the syntax and they may also be renamed.

10.3 Definitions and Expressions

Definitions in a module only contain unqualified names—that is, *local-names*, using the above terminology and are created by defining forms:

defining-0-form: *defclass-form* *defcondition-form* *defconstant-form* *deflocal-form* *defgeneric-form* *defsyntax-form* *defun-form*

A top-lexical binding is created exactly once and shared with all modules that import its exported name from the module that created the binding. A name clash can arise in the top-lexical environment from interaction between local definitions and between local definitions and imported modules. Only top-lexical bindings created by `deflocal` are mutable—both in the defining module and in any importing module. It is a violation to modify an immutable binding. Expressions, that is non-defining forms, are collected and evaluated in order of appearance at the end of the module definition process when the top-lexical environment is complete—that is after the creation and initialization of the top-lexical bindings. The exception to this is the `progn` form, which is descended and the forms within it are treated as if the `progn` were not present. Definitions may only appear either at top-level within a module definition or inside any number of `progn` forms. This is specified more precisely in the grammar for a module in syntax table 10.1.1.1.

10.4 Special Forms

***HGW Say something!

special-0-form:
 defmethod-form
 generic-lambda-form
 quote-form
 lambda-form
 setq-form
 if-form
 let/cc-form
 letfuns-form
 progn-form
 unwind-protect-form
 quasiquote-form
 unquote-form
 unquote-splicing-form
 call-next-handler-form
 with-handler-form
 cond-form
 and-form
 or-form
 block-form
 return-from-form
 let-form
 let-star-form
 with-input-file-form
 with-output-file-form
 with-source-form
 with-sink-form

10.5 Module Processing

The following steps summarize the module definition process:

directive processing: This is described in detail in § 10.2–10.2.4. 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 1 The semantics of syntax expansion are still under discussion and will be described fully in a future version of the EU LISP definition. In outline, however, it is intended that the mechanism should provide for hygenic 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. Currently syntax expansion is unhygienic to allow a simple syntax for syntax operator definition.

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 violation, if a free identifier in an expression or defining form does not have a binding in the top-lexical environment.

NOTE 2 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 defining forms in the body of the module in the order they appear.

NOTE 3 In this sense, a module can be regarded as a generalization of the `letfuns` form of this and other Lisp dialects.

expression evaluation: The expressions in the body of the module are evaluated in the order in which they appear.

11 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 μ Ceyx. 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. The defined name of this module is `telos0`.

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 `defclass` (11.3) and `defcondition` (13) defining forms, both of which create top-lexical bindings.

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` syntax operator, which adds a new method to an existing generic function. This syntax operator is described in detail later in this section.

11.1 System Defined Classes

The basic classes of EuLisp are elements of the object system class hierarchy, which is shown in table 1. 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 § 12.8. 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.

Table 1 – Level-0 class hierarchy

- $\mathcal{A}$ `<object>`
  - $\mathcal{C}$ `<character>`
  - $\mathcal{A}$ `<condition>` See table 2
  - $\mathcal{A}$ `<function>`
    - $\mathcal{C}$ `<continuation>`
    - $\mathcal{C}$ `<simple-function>`
    - $\mathcal{P}$ `<generic-function>`
      - $\mathcal{C}$ `<simple-generic-function>`
  - $\mathcal{A}$ `<collection>`
    - $\mathcal{A}$ `<sequence>`
      - $\mathcal{P}$ `<list>`
        - $\mathcal{C}$ `<cons>`
        - $\mathcal{C}$ `<null>`
      - $\mathcal{A}$ `<character-sequence>`
        - $\mathcal{C}$ `<string>`
      - $\mathcal{C}$ `<vector>`
    - $\mathcal{A}$ `<table>`
      - $\mathcal{C}$ `<hash-table>`
  - $\mathcal{C}$ `<lock>`
  - $\mathcal{A}$ `<number>`
    - $\mathcal{A}$ `<integer>`
      - $\mathcal{C}$ `<fpi>`
    - $\mathcal{A}$ `<float>`
      - $\mathcal{C}$ `<double-float>`
  - $\mathcal{A}$ `<stream>`
    - $\mathcal{A}$ `<buffered-stream>`
      - $\mathcal{C}$ `<string-stream>`
      - $\mathcal{C}$ `<file-stream>`
  - $\mathcal{A}$ `<name>`
    - $\mathcal{C}$ `<symbol>`
    - $\mathcal{C}$ `<keyword>`
  - $\mathcal{A}$ `<thread>`
    - $\mathcal{C}$ `<simple-thread>`

11.1.1 `<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>`.

11.1.2 `<class>` *class*

The default super-class including for itself. All classes defined using the `defclass` form are direct or indirect subclasses of `<class>`. Thus, this class is specializable by user-defined classes at level-0.

11.2 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.

11.3 Defining Classes

11.3.1 `defclass` *defining operator*

11.3.1.1 Syntax

defclass-form:
  (`defclass` *class-name* *superclass-name*
   (*slot**) *class-option**)
class-name:
  identifier
superclass-name:
  identifier
slot:
  slot-name
  (*slot-name* *slot-option**)
slot-name:
  identifier
slot-option:
  `keyword`: *identifier*
  `default`: *level-0-form*
  `reader`: *identifier*
  `writer`: *identifier*
  `accessor`: *identifier*
  `required?`: *boolean*
class-option:
  `keywords`: (*identifier**)
  `constructor`: *constructor-specification*
  `predicate`: *identifier*
  `abstract?`: *boolean*
constructor-specification:
  (*identifier* *identifier**)
initlist:
  {*identifier object*}*

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.

slot: Either a *slot-name* or a list of *slot-name* followed by some *slot-option*s.

class-option: A key and a value (see below) which, taken together, apply to the class as a whole.

Remarks

`defclass` defines 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-option*s are interpreted as follows:

`keyword` : *identifier*: The value of this option is an identifier naming a symbol, which is the name of an argument to be supplied in the initialization 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 *keyword* name may be used for several slots, in which case they will share the same initial value if the *keyword* is given to `make`. Subclasses inherit the *keyword*. Each slot must have at most one *keyword* including the inherited one. That means, a subclass can not shadow or add a new *keyword*, if a superclass has already defined one.

`default` : *level-0-form*: The value of this option is a form, which is evaluated as the default value of the slot, to be used if no *keyword* is defined for the slot or given to a call to `make`. The expression is evaluated in the lexical environment of the call to `defclass` and the dynamic environment of the call to `make`. The expression is evaluated each time `make` is called and the default value is called for. The order of evaluation of the defaults in all the slots is determined by `initialize`. This option must only be specified once for a particular slot. Subclasses inherit the default. 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 variable to which the reader function will be bound. The binding is immutable. 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 bindings for the same reader function. It is a violation to specify the same reader, writer, or accessor name for two different slots.

`writer` : *identifier*: The value is the identifier of the variable to which the writer function will be bound. The binding 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 function 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 violation 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 imbues level-0 programs with static semantics.

specify the same reader, writer, or accessor name for two different slots.

`accessor` : *identifier*: 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 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 violation to specify the same reader, writer, or accessor name for two different slots.

`required?` : *boolean*: The value is either `t` or `()`. `t` indicates that an initialization argument must be supplied for this slot.

The class-options are interpreted as follows:

`keywords` : (*identifier**): The value of this option is a list of identifiers naming symbols, which extend the inherited names of arguments to be supplied to `make` on the new class. Keywords 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 *keyword* or are ignored by the default `initialize` `<object>` method, otherwise. This option must only be specified once for a class.

`constructor` : *constructor-specification*: 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 *keyword*s for the class. The new function creates an instance of the class and fills in the slots according to the match between the specified *keyword*s and the given arguments to the constructor function. 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 predicate function will be bound. This option may be specified any number of times for a class.

`abstract?` : *boolean*: The value is either `t` or `()`. `t` indicates that the class being defined is abstract.

11.3.2 `abstract-class?` *function*

11.3.2.1 Signature

(`abstract-class?` *object*) → `<object>`

Arguments

object: Returns *object*, if it is an abstract class, otherwise `()`.

NOTE 1 `abstract-class?` is not currently implemented in Youtoo.

11.4 Defining Generic Functions and Methods

11.4.1 `<function>` `<object>` *class*

The class of all functions.

11.4.2 `<simple-function>` `<function>` *class*

Place holder for `<simple-function>` class.

11.4.3 `<generic-function>` `<function>` *class*

The class of all generic functions.

11.4.4 `<simple-generic-function>` `<generic-function>` *class*

Place holder for `<simple-generic-function>` class..

11.4.5 `defgeneric` *defining operator*

11.4.5.1 Syntax

defgeneric-form:
    (`defgeneric` *gf-name* *gf-lambda-list*
      level-0-init-option)
gf-name:
    identifier
gf-lambda-list:
    specialized-lambda-list
level-0-init-option:
    `method` *method-description*
method-description:
    (*specialized-lambda-list* *form**)
specialized-lambda-list:
    (*specialized-parameter*$^+$ {. *identifier*}$_{opt}$)
specialized-parameter:
    (*identifier* *class-name*)
    identifier

Arguments

gf-name: One of a symbol, or a form denoting a setter function or a converter function.

gf-lambda-list: The parameter list of the generic function, which may be specialized to restrict the domain of methods to be attached to the generic function.

level-0-init-option: is `method` *method-description* where *method-description* is a list comprising 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 lexical environment in which the generic function definition appears. This option may be specified more than once.

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. The method's specialized lamba list must be congruent to that of the generic function. Two lambda lists are said to be *congruent* 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 apply both to methods defined at the same time as the generic function and to any methods added subsequently by `defmethod`. An *level-0-init-option* is an identifier followed by a corresponding value.

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

11.4.5.2 Rewrite Rules

```
(defgeneric identifier
  gf-lambda-list level-0-init-option*)
            ≡ (defconstant identifier
                (generic-lambda
                  gf-lambda-list level-0-init-option*))
(defgeneric
  (setter identifier)
  gf-lambda-list level-0-init-option*)
            ≡ ((setter setter) identifier
                (generic-lambda
                  gf-lambda-list level-0-init-option*))
(defgeneric
  (converter identifier)
  gf-lambda-list level-0-init-option*)
            ≡ ((setter converter)
                identifier
                (generic-lambda
                  gf-lambda-list level-0-init-option*))
```

Examples

In the following example of the use of `defgeneric` a generic 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 generic 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>)) ...))
```

See also

`defmethod`, `generic-lambda`.

11.4.6 `defmethod` *defining operator*

11.4.6.1 Syntax

```
defmethod-form:
    ( defmethod gf-locator
        specialized-lambda-list
        body )
gf-locator:
    identifier
    ( setter identifier )
    ( converter identifier )
```

Remarks

This syntax operator is used for defining new methods on generic functions. A new method object is defined with the specified body and with the domain given by the specialized-lambda-list. This method is added to the generic function 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 gneric function, an error is signalled (condition class: <method-domain-clash>).

11.4.7 `generic-lambda` *special operator*

11.4.7.1 Syntax

```
generic-lambda-form:
    ( generic-lambda gf-lambda-list
        level-0-init-option* )
```

Remarks

`generic-lambda` creates and returns an anonymous generic function that can be applied immediately, much like the normal `lambda`. The *gf-lambda-list* and the *level-0-init-option*s are interpreted exactly as for the level-0 definition of `defgeneric`.

Examples

In the following example an anonymous version of `gf-0` (see `defgeneric` above) is defined. In all other respects the resulting 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>)) ...))
```

See also

`defgeneric`.

11.5 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.

11.5.1 `call-next-method` *special operator*

11.5.1.1 Signature

(`call-next-method`) → `<object>`

Result

The result of calling the next most specific applicable method.

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.

11.5.2 `next-method?` *special operator*

11.5.2.1 Signature

(`next-method?`) → *boolean*

Result

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

11.6 Method Lookup and Generic Dispatch

The system defined method lookup and generic function dispatch is purely class based.

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 iff C_1 appears before C_2 in the class precedence list (CPL) of C_3 [2)].

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 $D_1 \times \ldots \times D_m[\times$ `<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_1 with the domain $D_{11} \times \ldots \times D_{1m}[\times$ `<list>`$]$ is *more specific* than a method M_2 with the domain $D_{21} \times \ldots \times D_{2m}[\times$ `<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_{1i} is more specific than D_{2i} with respect to C_i, the class of a_i, and for all $j = 1 \ldots i-1$, D_{2j} is *not* more specific than D_{1j} with respect to C_j, the class of a_j.

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

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 $M_1 \ldots M_k$ 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*.

11.7 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.

11.7.1 `make` *function*

Arguments

class : The class of the object to create.

`key`$_1$ *obj*$_1$ ... `key`$_n$ *obj*$_n$: Initialization arguments.

Result

An instance of *class*.

Remarks

The general constructor `make` creates a new object calling `allocate` and initializes it by calling `initialize`. `make` returns whatever `allocate` returns as its result.

11.7.2 `allocate` *function*

Arguments

class : The class to allocate.

initlist : The list of initialization arguments.

Result

A new uninitialized direct instance of the first argument.

[2)] This definition is required when multiple inheritance comes into play. Then, two classes have to be compared with respect to a third class even if they are not related to each other via the subclass relationship. Although, multiple inheritance is not provided at level-0, it is provided at level-1 the method lookup protocol is independent of the inheritance strategy defined on classes. It depends on the class precedence lists of the domains of methods attached to the generic function and the argument classes involved.

Remarks

The *class* must be a structure class, the *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.

11.7.3 `initialize` *generic function*

Generic Arguments

object `<object>`: The object to initialize.

initlist: The list of initialization arguments.

Result

The initialized object.

Remarks

Initializes an object and returns the initialized object as the result. It is called by `make` on a new uninitialized object created by calling `allocate`.

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

11.7.4 `initialize <object>` *method*

Specialized Arguments

object `<object>`: The object to initialize.

initlist: The list of initialization arguments.

Result

The initialized object.

Remarks

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

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

b) Initializes the slots of the object according to the *keyword*, if supplied, or according to the most specific `default`, if specified. Otherwise, the slot remains “unbound”.

Legal *keyword*s which do not initialize a slot are ignored by the default `initialize <object>` method. More specific methods may handle these *keyword*s and call the default method by calling `call-next-method`.

11.8 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 `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.

11.9 Other Abstract Classes

11.9.1 `<name>` `<object>` *class*

The class of all “names”.

See also

`<symbol>` and `<keyword>`.

12 Level-0 Defining, Special and Function-call Forms

This section gives the syntax of well-formed expressions and describes the semantics of the special-forms, functions and syntax forms of the level-0 language. In the case of level-0 syntax forms, 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.

12.1 Simple Expressions

12.1.1 `constant` *syntax*

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.

Examples

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

12.1.2 `defconstant` *defining operator*

12.1.2.1 Syntax

defconstant-form: → `<object>`
    (`defconstant` *constant-name* *form*)
constant-name:
    identifier

Arguments

identifier: A symbol naming an immutable top-lexical binding to be initialized with the value of *form*.

form: The *form* whose value will be stored in the binding of *identifier*.

Remarks

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

12.1.3 `t` `<symbol>` *constant*

Remarks

This may be used to denote the abstract boolean value *true*, but so may any other value than `()`.

12.1.4 `symbol` *syntax*

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

12.1.5 `deflocal` *defining operator*

12.1.5.1 Syntax

deflocal-form: → `<object>`
    (`deflocal` *local-name* *form*)
local-name:
    identifier

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*.

Remarks

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

See also

`setq`.

12.1.6 `quote` *special operator*

12.1.6.1 Syntax

quote-form: → *object*
    (`quote` *object*)
    `'`*object*

Arguments

object: the *object* to be quoted.

Result

The result is *object*.

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 .

12.1.7 `'` *syntax*

Remarks

See `quote`.

12.2 Functions: creation, definition and application

12.2.1 `lambda` *special operator*

12.2.1.1 Syntax

lambda-form: → <function>
    `(` `lambda` *lambda-list* *body* `)`
lambda-list:
    identifier
    simple-list
    rest-list
simple-list:
    `(` *identifier** `)`
rest-list:
    `(` *identifier** `.` *identifier* `)`
body:
    *form**

Arguments

lambda-list: The parameter list of the function conforming to the syntax 12.2.1.1.

form: An expression.

Result

A function with the specified *lambda-list* and sequence of *forms*.

Remarks

The function construction operator is `lambda`. Access to the lexical environment of definition is guaranteed. The syntax of *lambda-list* is defined in reflambda-syntax-table.

If *lambda-list* is an *identifier*, it is bound to a newly allocated list of the actual parameters. This binding has lexical scope and indefinite extent. If *lambda-list* is a *simple-list*, the arguments are bound to the corresponding *identifier*. Otherwise, *lambda-list* must be a *rest-list*. In this case, each *identifier* preceding the dot is bound to the corresponding argument and the *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 violation if the same identifier appears more than once in a *lambda-list*. It is an error to modify *rest-list*.

12.2.2 `defsyntax` *defining operator*

12.2.2.1 Syntax

defsyntax-form: → <function>
    `(` `defsyntax` *syntax-operator-name* *lambda-list* *body* `)`
syntax-operator-name:
    identifier

Arguments

syntax-operator-name: A symbol naming an immutable top-lexical binding to be initialized with a function having the specified *lambda-list* and *body*.

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

body: A sequence of forms.

Remarks

The `defsyntax` form defines a syntax operator named by *syntax-operator-name* and stores the definition as the top-lexical binding of *syntax-operator-name* . The interpretation of the *lambda-list* is as defined for `lambda` (see 12.2.1.1).

NOTE 1 A syntax operator is automatically exported from the the module which defines because it cannot be used in the module which defines it.

See also

`lambda`.

12.2.3 `defun` *defining operator*

12.2.3.1 Syntax

defun-form: → <function>
    simple-defun
    setter-defun
simple-defun:
    `(` `defun` *function-name* *lambda-list*
      body `)`
setter-defun:
    `(` `defun` `(` `setter` *function-name* `)` *lambda-list*
      body `)`
function-name:
    identifier

Arguments

function-name: A symbol naming an immutable top-lexical binding to be initialized with a function having the specified *lambda-list* and *body*.

`(setter` *function-name*`)`: An expression denoting the setter function to correspond to *function-name*.

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

body: A sequence of forms.

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`.

12.2.3.2 Rewrite Rules

(`defun` *identifier* *lambda-list* *body*)	≡	(`defconstant` *identifier* (`lambda` *lambda-list* *body*))
(`defun` (`setter` *identifier*) *lambda-list* *body*)	≡	((`setter setter`) *identifier* (`lambda` *lambda-list* *body*))

12.2.4 `function call` *syntax*

12.2.4.1 Syntax

function-call-form: → <object>
    `(` *operator* *operand** `)`
operator:
    identifier
operand:
    identifier
    literal
    special-form
    function-call-form

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 *identifier*, a *literal*, a *special-form* or a *function-call-form*.

Result

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

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 2 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.

See also

`constant`, `symbol`, `quote`.

12.2.5 `<invalid-operator>` `<general-condition>` *condition*

Initialization Options

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

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

Remarks

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

12.2.6 `apply` *function*

12.2.6.1 Syntax

apply-form: → `<object>`
  (`apply` *function body*)
function:
  level-0-form

Arguments

function: A form which must evaluate to an instance of `<function>`.

$form_1$ ... $form_{n-1}$: A sequence of expressions, which will be evaluated according to the rules given in *function-call-form*.

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

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 $form_1$ through $form_{n-1}$. An error is signalled (condition class: `<invalid-operator>`) if the first argument is not an instance of `<function>`.

See also

function-call-form, `<invalid-operator>`.

12.3 Destructive Operations

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

12.3.1 `setq` *special operator*

12.3.1.1 Syntax

setq-form: → `<object>`
  (`setq` *identifier form*)

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*.

Result

The result is the value of *form*.

Remarks

The *form* is evaluated and the result is stored in the closest lexical binding named by *identifier*. It is a violation to modify an immutable binding.

12.3.2 `setter` *function*

Arguments

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

Result

The *writer* corresponding to *reader*.

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 signalled (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>`)

See also
defgeneric, defmethod, defclass, defun.

12.3.3 <no-setter> <general-condition> *condition*

Initialization Options

object *object*: The object given to setter.

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

12.3.4 <cannot-update-setter> <general-condition> *condition*

Initialization Options

accessor $object_1$: The given accessor object.

updater $object_2$: The given updater object.

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

See also
setter.

12.4 Conditional Expressions

12.4.1 if *special operator*

12.4.1.1 Syntax

```
if-form:    → <object>
      ( if antecedent
           consequent
         alternative )
antecedent:
      form
consequent:
      form
alternative:
      form
```

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

Remarks
The *antecedent* is evaluated. If the result is t the *consequent* is evaluated, otherwise the *alternative* is evaluated. Both *consequent* and *alternative* must be specified. The result of if is the result of the evaluation of whichever of *consequent* or *alternative* is chosen.

12.4.2 cond *special operator*

12.4.2.1 Syntax

```
cond-form:    → <object>
      ( cond
        {( antecedent consequent* )}* )
```

Remarks
The cond syntax operator provides a convenient syntax for collections of *if-then-elseif...else* expressions.

12.4.2.2 Rewrite Rules

```
(cond)                           ≡ ()
(cond (antecedent)
  ...)                           ≡ (or antecedent (cond ...))
(cond                            ≡ (or antecedent1
  (antecedent1)                       (cond
  (antecedent2 consequent*)             (antecedent2
  ...)                                    consequent*)
                                        ...))
(cond                            ≡ (if antecedent1
  (antecedent1 consequent*)           (progn consequent*)
  (antecedent2 consequent*)           (cond
  ...)                                  (antecedent2
                                          consequent*)
                                        ...))
```

12.4.3 else <symbol> *constant*

Remarks
This may be used to denote the default clause in cond and case forms and has the value t, *i.e.* it is an alias for t introduced to improve readability of the cond and case forms.

12.4.4 when *special operator*

12.4.4.1 Syntax

```
when-form:    → <object>
      ( when antecedent
          consequent )
```

Result
The *antecedent* is evaluated and if the result is t the *consequent* is evaluated and returned otherwise () is returned.

12.4.4.2 Rewrite Rules

```
(when antecedent   ≡  (if antecedent
  consequent)               consequent
                          ())
```

12.4.5 unless *special operator*

12.4.5.1 Syntax

```
unless-form:    → <object>
      ( unless antecedent
          consequent )
```

Result

The *antecedent* is evaluated and if the result is `()` the *consequent* is evaluated and returned otherwise `()` is returned.

12.4.5.2 Rewrite Rules

```
(unless antecedent    ≡  (if antecedent
   consequent)               ()
                             consequent)
```

12.4.6 `and` *special operator*

12.4.6.1 Syntax

and-form: → `<object>`
  `( and consequent* )`

Remarks

The expansion of an `and` form leads to the evaluation of the sequence of *form*s from left to right. The first *form* in the sequence that evaluates to `()` stops evaluation and none of the *form*s to its right will be evaluated—that is to say, it is non-strict. The result of `(and)` is `t`. If none of the *form*s evaluate to `()`, the value of the last *form* is returned.

12.4.6.2 Rewrite Rules

```
(and)                    ≡  t
(and form)               ≡  form
(and form1 form2 ...)    ≡  (if form1
                                (and form2 ...)
                                ())
```

12.4.7 `or` *special operator*

12.4.7.1 Syntax

or-form: → `<object>`
  `( or form* )`

Remarks

The expansion of an `or` form leads to the evaluation of the sequence of *form*s from left to right. The value of the first *form* that evaluates to `t` is the result of the `or` form and none of the *form*s to its right will be evaluated—that is to say, it is non-strict. If none of the forms evaluate to `t`, the value of the last *form* is returned.

12.4.7.2 Rewrite Rules

```
(or)                     ≡  ()
(or form)                ≡  form
(or form1 form2 ...)     ≡  (let ((x form1))
                                (if x
                                    x
                                    (or form2 ...)))
```

Note that `x` does not occur free in any of $form_2 \ldots form_n$.

12.5 Variable Binding and Sequences

12.5.1 `let/cc` *special operator*

12.5.1.1 Syntax

let/cc-form: → `<object>`
  `( let/cc identifier body )`

Arguments

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

body: A sequence of forms to evaluate.

Result

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

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.

Examples

An example of the use of `let/cc` is given in example 1. 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 16.2) terminates and an error (condition class: `<cannot-open-path>`) is signalled.

Example 1 – using `let/cc`

```
(defun path-open (pathlist name . options)
  (let/cc succeed
    (map
      (lambda (path)
        (let/cc fail
          (with-handler
            (lambda (condition resume) (fail ()))
            (succeed
              (apply open
                (format () "~a/~a" path name)
                options)))))
      pathlist)
    (error
      (format ()
        "Cannot open stream for (~a) ~a"
        pathlist name)
      <cannot-open-path>)))
```

See also

`block`, `return-from`.

12.5.2 `block` *special operator*

12.5.2.1 Syntax

block-form: → `<object>`
  (`block` *identifier* *body*)

Remarks

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

12.5.2.2 Rewrite Rules

(`block` *identifier*)	≡	()
(`block` *identifier* *body*)	≡	(`let/cc` *identifier* *body*)

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.

See also

`return-from`.

12.5.3 `return-from` *special operator*

12.5.3.1 Syntax

return-from-form: → `<object>`
  (`return-from` *identifier* $form_{opt}$)

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*.

12.5.3.2 Rewrite Rules

(`return-from` *identifier*)	≡	(*identifier* ())
(`return-from` *identifier* *form*)	≡	(*identifier* *form*)

See also

`block`.

12.5.4 `letfuns` *special operator*

12.5.4.1 Syntax

letfuns-form: → `<object>`
  (`letfuns`
    (*function-definition**)
    letfuns-body)
function-definition:
  (*identifier* *lambda-list* *body*)
letfuns-body:
  *form**

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.

letfuns-body: A sequence of forms.

Result

The `letfuns` 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 *identifier*s is the entire `letfuns` form. The *lambda-list* is either a single variable or a list of variables—see `lambda`. Each form in *letfuns-body* is evaluated in order in the lexical environment extended with the bindings of the *identifier*s. The result of evaluating the last form in *letfuns-body* is returned as the result of the `letfuns` form.

12.5.5 `let` *special operator*

12.5.5.1 Syntax

let-form: → `<object>`
  (`let` $identifier_{opt}$ (*binding**)
    body)
binding:
  variable
  (*variable* *form*)
variable:
  identifier *var*:
  variable

Remarks

The optional *identifier* denotes that the let form can be called from within its *body*. This is an abbreviation for `letfuns` form in which *identifier* is bound to a function whose parameters are the identifiers of the *binding*s of the `let`, whose body is that of the `let` and whose initial call passes the values of the initializing form of the *binding*s. 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 *binding*s 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.

12.5.5.2 Rewrite Rules

(`let` () *body*)	≡	(`progn` *body*)
(`let` ((var_1 $form_1$) (var_2 $form_2$) var_3 ...) *body*)	≡	((`lambda` (var_1 var_2 var_3 ...) *body*) $form_1$ $form_2$ () ...)
(`let` var_0 ((var_1 $form_1$) var_2 ...) *body*)	≡	(`letfuns` ((var_0 (var_1 var_2 ...) *body*)) (var_0 $form_1$ () ...))

12.5.6 let* *special operator*

12.5.6.1 Syntax

let-star-form: → <object>
    (let* (*binding**)
      body)

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.

12.5.6.2 Rewrite Rules

```
(let* () body)           ≡  (progn body)
(let* ((var1 form1)      ≡  (let ((var1 form1))
       (var2 form2)            (let* ((var2 form2)
       var3                           var3
       ...)                           ...)
  body)                        body))
```

12.5.7 progn *special operator*

12.5.7.1 Syntax

progn-form: → <object>
    (progn *body*)

Arguments

*form** : A sequence of forms and in certain circumstances, defining forms.

Result

The sequence of *form*s is evaluated from left to right, returning the value of the last one as the result of the progn form. If the sequence of forms is empty, progn returns ().

Remarks

If the progn form occurs enclosed only by progn forms and a defmodule form, then the *form*s within the progn can be defining forms, since they appear in the top-lexical environment. It is a violation for defining forms to appear in inner-lexical environments.

12.5.8 unwind-protect *special operator*

12.5.8.1 Syntax

unwind-protect-form: → <object>
    (unwind-protect *protected-form*
      *after-form**)
protected-form:
    form
after-form:
    form

Arguments

protected-form: A form.

*after-form** : A sequence of forms.

Result

The value of *protected-form*.

Remarks

The normal action of unwind-protect is to process *protected-form* and then each of *after-form*s in order, returning the value of *protected-form* as the result of unwind-protect. A non-local exit from the dynamic extent of *protected-form*, which can be caused by processing a non-local exit form, will cause each of *after-form*s to be processed before control goes to the continuation specified in the non-local exit form. The *after-form*s are not protected in any way by the current unwind-protect. Should any kind of non-local exit occur during the processing of the *after-form*s, the *after-form*s being processed are not reentered. Instead, control is transferred to wherever specified by the new non-local exit but the *after-form*s of any intervening unwind-protects between the dynamic extent of the target of control transfer and the current unwind-protect are evaluated in increasing order of dynamic extent.

Examples

Example 2 – Interaction of unwind-protect with non-local exits

```
(progn
  (let/cc k1
    (letfuns
      ((loop
         (let/cc k2
           (unwind-protect (k1 10) (k2 99))
         ;; continuation bound to k2
         (loop))))
      (loop)))
  ;; continuation bound to k1
  ...)
```

The code fragment in example 2 illustrates both the use of unwind-protect and of a difference between the semantics of EuLisp and some other Lisps. Stepping through the evaluation of this form: k1 is bound to the continuation of its let/cc form; a recursive function named loop is constructed, loop is called from the body of the letfuns form; k2 is bound to the continuation of its let/cc form; unwind-protect calls k1; the after forms of unwind-protect are evaluated in order; k2 is called; loop is called; etc.. This program loops indefinitely.

12.6 Quasiquotation Expressions

12.6.1 quasiquote *special operator*

12.6.1.1 Syntax

quasiquote-form: → <object>
    (quasiquote *skeleton*)
    ` *skeleton*
skeleton:
    form

Remarks

Quasiquotation 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` syntax operator can be abbreviated by using the glyph called *grave accent* (`) , so that (`quasiquote` *skeleton*) can be written ` *skeleton*.

12.6.2 ` *syntax*

Remarks

See `quasiquote`.

12.6.3 `unquote` *special operator*

12.6.3.1 Syntax

unquote-form: → `<object>`
  (`unquote` *form*)
  ,*form*

Remarks

See `unquote-splicing`.

12.6.4 , *syntax*

Remarks

See `unquote`.

12.6.5 `unquote-splicing` *special operator*

12.6.5.1 Syntax

unquote-splicing-form: → `<object>`
  (`unquote-splicing` *form*)
  ,@*form*

Remarks

The holes in a `quasiquote`d 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-splicing` 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`.

Examples

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

12.6.6 ,@ *syntax*

Remarks

See `unquote-splicing`.

12.7 Summary of Level-0 Defining, Special and Function-call Forms

This section gives the syntax of the character-set, comments and all level-0 forms starting with modules. The syntax of data objects is given in the section pertaining to the class and is summarized in section 16.20.

decimal-digit: one of
  `0 1 2 3 4 5 6 7 8 9`
upper-letter: one of
  `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: one of
  `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
normal-other-character: one of
  `* / < = > + .`
other-character:
  normal-other-character
special-character: one of
  `; ' , \ " # ( ) ` | @`
level-0-character:
  decimal-digit
  letter
  other-character
  special-character
whitespace:
  space
  newline
  line-feed
  return
  tab
  vertical-tab
  form-feed
comment:
  ; *all subsequent characters up to the end of the line*
  #; *whitespace** *object*

12.7.1 Syntax of Level-0 modules

defmodule-0-form:
    (`defmodule` *module-name*
      module-directives
      level-0-module-form*)
module-name:
    identifier
module-directives:
    (*module-directive**)
module-directive:
    `export` (*identifier**)
    `expose` (*module-descriptor**)
    `import` (*module-descriptor**)
    `syntax` (*module-descriptor**)
level-0-module-form:
    (`export` *identifier**)
    level-0-form
    defining-0-form
    (`progn` *level-0-module-form**)
module-descriptor:
    module-name
    module-filter
module-filter:
    (`except` (*identifier**) *module-descriptor*)
    (`only` (*identifier**) *module-descriptor*)
    (`rename` (*rename-pair**) *module-descriptor*)
rename-pair:
    (*identifier* *identifier*)
level-0-form:
    identifier
    literal
    special-0-form
    function-call-form
form:
    level-0-form
special-form:
    special-0-form

12.7.2 Syntax of Level-0 defining forms

defining-0-form:
    defclass-form
    defcondition-form
    defconstant-form
    deflocal-form
    defgeneric-form
    defsyntax-form
    defun-form
defclass-form:
    (`defclass` *class-name* *superclass-name*
      (*slot**) *class-option**)
class-name:
    identifier
superclass-name:
    identifier
slot:
    slot-name
    (*slot-name* *slot-option**)
slot-name:
    identifier
slot-option:
    `keyword:` *identifier*
    `default:` *level-0-form*
    `reader:` *identifier*
    `writer:` *identifier*
    `accessor:` *identifier*
    `required?:` *boolean*
class-option:
    `keywords:` (*identifier**)
    `constructor:` *constructor-specification*
    `predicate:` *identifier*
    `abstract?:` *boolean*
constructor-specification:
    (*identifier* *identifier**)
initlist:
    $\{$*identifier object*$\}^{*}$
defgeneric-form:
    (`defgeneric` *gf-name* *gf-lambda-list*
      level-0-init-option)
gf-name:
    identifier
gf-lambda-list:
    specialized-lambda-list
level-0-init-option:
    `method` *method-description*
method-description:
    (*specialized-lambda-list* *form**)
specialized-lambda-list:
    (*specialized-parameter*$^{+}$ $\{$. *identifier*$\}_{opt}$)
specialized-parameter:
    (*identifier* *class-name*)
    identifier
defmethod-form:
    (`defmethod` *gf-locator*
      specialized-lambda-list
      body)
gf-locator:
    identifier
    (`setter` *identifier*)
    (`converter` *identifier*)
defconstant-form: → `<object>`
    (`defconstant` *constant-name* *form*)
constant-name:
    identifier
deflocal-form: → `<object>`
    (`deflocal` *local-name* *form*)
local-name:
    identifier

defsyntax-form: → `<function>`
(`defsyntax` *syntax-operator-name lambda-list body*)
syntax-operator-name:
identifier
defun-form: → `<function>`
simple-defun
setter-defun
simple-defun:
(`defun` *function-name lambda-list*
body)
setter-defun:
(`defun` (`setter` *function-name*) *lambda-list*
body)
function-name:
identifier

12.7.3 Syntax of Level-0 special forms

special-0-form:
defmethod-form
generic-lambda-form
quote-form
lambda-form
setq-form
if-form
let/cc-form
letfuns-form
progn-form
unwind-protect-form
quasiquote-form
unquote-form
unquote-splicing-form
call-next-handler-form
with-handler-form
cond-form
and-form
or-form
block-form
return-from-form
let-form
let-star-form
with-input-file-form
with-output-file-form
with-source-form
with-sink-form
generic-lambda-form:
(`generic-lambda` *gf-lambda-list*
level-0-init-option*)
lambda-form: → `<function>`
(`lambda` *lambda-list body*)
lambda-list:
identifier
simple-list
rest-list
simple-list:
(*identifier**)
rest-list:
(*identifier** . *identifier*)
body:
form*
quote-form: → *object*
(`quote` *object*)
'*object*
setq-form: → `<object>`
(`setq` *identifier form*)

if-form: → `<object>`
(`if` *antecedent*
consequent
alternative)
antecedent:
form
consequent:
form
alternative:
form

cond-form: → `<object>`
(`cond`
{(*antecedent consequent**)}*)
when-form: → `<object>`
(`when` *antecedent*
consequent)

unless-form: → `<object>`
(`unless` *antecedent*
consequent)

and-form: → `<object>`
(`and` *consequent**)
or-form: → `<object>`
(`or` *form**)
let/cc-form: → `<object>`
(`let/cc` *identifier body*)
letfuns-form: → `<object>`
(`letfuns`
(*function-definition**)
letfuns-body)
function-definition:
(*identifier lambda-list body*)
letfuns-body:
form*
progn-form: → `<object>`
(`progn` *body*)
unwind-protect-form: → `<object>`
(`unwind-protect` *protected-form*
after-form*)
protected-form:
form
after-form:
form
apply-form: → `<object>`
(`apply` *function body*)
function:
level-0-form

block-form: → `<object>`
(`block` *identifier body*)
return-from-form: → `<object>`
(`return-from` *identifier form*$_{opt}$)
let-form: → `<object>`
(`let` *identifier*$_{opt}$ (*binding**)
body)
binding:
variable
(*variable form*)
variable:
identifier var:
variable
let-star-form: → `<object>`
(`let*` (*binding**)
body)

quasiquote-form: → `<object>`
    (`quasiquote` *skeleton*)
    ‘*skeleton*
skeleton:
    form
unquote-form: → `<object>`
    (`unquote` *form*)
    ,*form*
unquote-splicing-form: → `<object>`
    (`unquote-splicing` *form*)
    ,@*form*

12.7.4 Syntax of Level-0 function calls

function-call-form: → `<object>`
    (*operator* *operand**)
operator:
    identifier
operand:
    identifier
    literal
    special-form
    function-call-form

12.8 Conditions

The defined name of this module is `condition`.

The condition system was influenced by the Common Lisp error system [16] 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 actions of functions in terms of their normal behaviour. 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 represent exceptional situations. A condition can be signalled continuably 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 recovered 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 continuably 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 allocated. A condition should be signalled non-continuably when there is no reasonable way to resume processing.

A condition class is defined using `defcondition` (see § 13). 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 resume 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 table 2.

13 Condition Classes

13.0.1 `defcondition` *defining operator*

13.0.1.1 Syntax

defcondition-form:
    (`defcondition` *condition-class-name*
      condition-superclass-name
      (*slot**) *class-option**)
condition-class-name:
    identifier
condition-superclass-name:
    identifier

Arguments

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

Table 2 – Condition class hierarchy

```
<condition>
  <general-condition>
    <invalid-operator>
    <cannot-update-setter>
    <no-setter>
  <environment-condition>
  <arithmetic-condition>
    <division-by-zero>
  <conversion-condition>
    <no-converter>
  <stream-condition>
    <end-of-stream>
  <read-error>
  <thread-condition>
    <thread-already-started>
    <wrong-thread-continuation>
    <wrong-condition-class>
  <telos-condition>
    <no-next-method>
    <generic-function-condition>
      <non-congruent-lambda-lists>
      <incompatible-method-domain>
      <no-applicable-method>
      <method-domain-clash>
```

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

slot: Either a *slot-name* or a list of *slot-name* followed by some *slot-option*s.

class-option: A key and a value which, taken together, apply to the condition as a whole.

Remarks

This defining form defines a new condition class, it is analogous to `defclass` except in the in the specification of and default superclass. 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. 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.

13.0.2 `<condition>` `<object>` *class*

Initialization Options

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

Remarks

The class which is the superclass of all condition classes.

13.0.3 `condition?` *function*

Arguments

object: An object to examine.

Result

Returns *object* if it is a condition, otherwise `()`.

13.0.4 `initialize <condition>` *method*

Specialized Arguments

condition `<condition>`: a condition.

initlist: 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.

`message-arguments` `<list>`: A list of objects to be used in processing the message format string.

Result

A new, initialized condition.

Remarks

First calls `call-next-method` to carry out initialization specified by superclasses then does the `<condition>` specific initialization.

13.0.5 `<general-condition>` `<condition>` *condition*

This is the general condition class for conditions arising from the execution of programs by the processor.

13.0.6 `<domain-condition>` `<general-condition>` *condition*

Initialization Options

`argument` `<object>`: An argument, which was not of the expected class, or outside a defined range and therefore lead to the signalling of this condition.

13.0.7 `<range-condition>` `<general-condition>` *condition*

Initialization Options

`result` `<object>`: A result, which was not of the expected class, or outside a defined range and therefore lead to the signalling of this condition.

13.0.8 `<environment-condition>` `<condition>` *condition*

This is the general condition class for conditions arising from the environment of the processor.

13.0.9 `<wrong-condition-class>` `<thread-condition>` *condition*

Initialization Options

`condition` *condition*: A condition.

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

13.0.10 <generic-function-condition>

<condition> *condition*

This is the general condition class for conditions arising from operations in the object system at level 0. The following direct subclasses of <generic-function-condition> are defined at level 0:

<no-applicable-method>: signalled by a generic function when there is no method which is applicable to the arguments.

<incompatible-method-domain>: signalled by if the domain of the method being added to a generic function is not a subdomain of the generic function's domain.

<non-congruent-lambda-lists>: signalled if the lambda list of the method being added to a generic function is not congruent to that of the generic function.

<method-domain-clash>: signalled if the method being added to a generic function has the same domain as a method already attached to the generic function.

<no-next-method>: signalled by call-next-method if there is no next most specific method.

13.0.11 <no-applicable-method>

<generic-function-condition> *condition*

Initialization Options

generic *function*: The generic function which was applied.

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

Remarks

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

13.0.12 <incompatible-method-domain>

<generic-function-condition> *condition*

Initialization Options

generic *function*: The generic function to be extended.

method *method*: The method to be added.

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.

13.0.13 <non-congruent-lambda-lists>

<generic-function-condition> *condition*

Initialization Options

generic *function*: The generic function to be extended.

method *method*: The method to be added.

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.

13.0.14 <method-domain-clash>

<generic-function-condition> *condition*

Initialization Options

generic *function*: The generic function to be extended.

methods *list*: The methods with the same domain.

Remarks

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

13.0.15 <no-next-method>

<generic-function-condition> *condition*

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.

Remarks

Signalled by call-next-method if there is no next most specific method.

14 Condition Signalling and 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.

14.0.16 signal

function

Arguments

condition: The condition to be signalled.

function: The function to be called if the condition is handled and resumed, that is to say, the condition is continuable, or () otherwise.

thread$_{opt}$: 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.

Result

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

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 continuable condition. If the second argument is `()`, it indicates that the condition was signalled as a non-continuable 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.

See also

`thread-reschedule`, `thread-value`, `with-handler`.

14.0.17 `call-next-handler` *special operator*

14.0.17.1 Syntax

call-next-handler-form:
    (`call-next-handler`)

Remarks

The `call-next-handler` special form calls the next enclosing handler. It is an error to evaluate this form other than within an established handler function. The `call-next-handler` special form is normally used when a handler function does not know how to deal with the class of condition. However, it may also be used to combine handler function behaviour in a similar but orthogonal way to `call-next-method` (assuming a generic handler function).

14.0.18 `with-handler` *special operator*

14.0.18.1 Syntax

with-handler-form:
    (`with-handler` *handler-function*
      form*)
handler-function:
    level-0-form

Arguments

handler-function : The result of evaluating the handler function expression must be either a function or a generic function. This function will be called if a condition is signalled during the dynamic extent of *protected-forms* and there are no intervening handler functions which accept or resume the condition. 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.

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

Result

The value of the last form in the sequence of *forms*.

Remarks

A `with-handler` form is evaluated in three steps:

a) The new *handler-function* is evaluated. This now identifies the nearest enclosing handler and shadows the previous nearest enclosing handler.

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

c) the *handler-function* is disestablished as the nearest enclosing handler, and the previous handler function is restored as the nearest enclosing.

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 nearest closing *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 `signal`s occurring during the execution of *handler-function* are dealt with by the nearest enclosing handler outside the extent of the form which established *handler-function*. It is an error if there is no 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.

Examples

There are three ways in which a *handler-function* can respond: actions:

a) The error is very serious and the computation must be abandoned; this is likely to be characterised by a non-local exit from the handler function.

b) The situation can be corrected by the handler, so it does and then returns. Thus the call to `signal` returns with the result passed back from the handler function.

c) The handler function does not know how to deal with the class of condition signalled; control is passed explicitly to the next enclosing handler via the `call-next-handler` special form.

An illustration of the use of all three cases is given here:

Example 1 – handler actions

```
(defgeneric error-handler (condition)
  method: (((c <serious-error>))
```

```
            ... abort thread ...)
    method: (((c <fixable-situation>))
            ... apply fix and return ... )
    method: (((c <condition>) (call-next-handler))))

  (with-handler error-handler
      ;; the protected expression
      (something-which-might-signal-an-error))
```

See also

signal.

14.0.19 error *function*

Arguments

condition-class : the class of condition to be signalled.

error-message : a string containing relevant information.

*init-option** : a sequence of options to be passed to `initialize-instance` when making the instance of condition.

Result

The result is ().

Remarks

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

14.0.20 cerror *function*

Arguments

condition-class : the class of condition to be signalled.

error-message : a string containing relevant information.

*init-option** : a sequence of options to be passed to `initialize-instance` when making the instance of condition.

Result

The result is ().

Remarks

The cerror function signals a continuable error. It calls signal with an instance of a condition of *condition-class* initialized from the *error-message*, *init-options* and a *resume* continuation value which is the continuation of the cerror expression. A non-() resume continuation signifies that the condition has been signalled continuably.

15 Concurrency

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

A thread is allocated and initialized, by calling make. The keyword 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: *new*, *running*, *aborted* and *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 *determined*. Although a program can find out whether a thread is determined or not by means of `wait` with a timeout of 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 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 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 thread-value will signal the condition that aborted the thread on the thread accessing the value. Note that 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 thread-value on an undetermined thread. In each of these cases, 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 thread-reschedule returns. The condition must be an instance of <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 <thread-condition>, in which case an error is signalled on the signalling thread. See also § 12.8.

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 lock and 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 lock or one of the threads which has already called lock on the lock can gain exclusive access.

NOTE 1 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 multi-processor 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 (c.f. 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 i/o 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:

```
(while (= data 0) (thread-reschedule))
```

15.1 Threads

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

15.1.1 `<thread>` — `<object>` *class*

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

Initialization Options

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

15.1.2 `thread?` — *function*

Arguments

object: An object to examine.

Result

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

15.1.3 `thread-reschedule` — *function*

This function takes no arguments.

Result

The result is `()`.

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`.

See also

`thread-value`, `signal` and § 12.8 for details of conditions and signalling.

15.1.4 `current-thread` — *function*

This function takes no arguments.

Result

The thread on which `current-thread` was executed.

15.1.5 `thread-start` — *function*

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>`).

$obj_1 \ldots obj_n$: values to be passed as the arguments to the initial function of *thread*.

Result

The thread which was supplied as the first argument.

Remarks

The state of thread is changed to running. The values obj_1 to obj_n will be passed as arguments to the initial function of *thread*.

15.1.6 `thread-value` *function*

Arguments

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

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.

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.

See also

`thread-reschedule`, `signal`.

15.1.7 `wait` *generic function*

Generic Arguments

obj: An object.

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

Result

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

Remarks

`wait` provides a generic interface to operations which may block. Execution of the current thread will continue beyond the `wait` form only when one of the following happened:

a) A condition associated with *obj* returns `t`;

b) *timeout* time units elapse;

c) A condition is raised by another thread on this thread.

`wait` returns `()` if timeout occurs, else it returns a non-nil value.

A *timeout* argument of `()` or zero denotes a polling operation. A *timeout* argument of `t` denotes indefinite blocking (cases a or c above). A *timeout* argument of a non-negative integer denotes the minimum number of time units before timeout. The number of time units in a second is given by the implementation-defined constant `ticks-per-second`.

Examples

This code fragment copies characters from stream `s` to the current output stream until no data is received on the stream for a period of at least 1 second.

```
(letfuns
  ((loop ()
     (when (wait s (round ticks-per-second))
           (print (read-char s))
           (loop))))
  (loop))
```

See also

threads (section 15.1), streams (section 16.15).

15.1.8 `wait <thread>` *method*

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.

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.

See also

`wait` and `ticks-per-second` (§ 12.8).

15.1.9 `ticks-per-second` `<double-float>` *constant*

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

15.1.10 `<thread-condition>` `<condition>` *condition*

Initialization Options

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

Remarks

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

15.1.11 `<wrong-thread-continuation>` `<thread-condition>` *condition*

Initialization Options

`continuation` *continuation*: A continuation.

`thread` *thread*: The thread on which *continuation* was created.

Remarks

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

15.1.12 `<thread-already-started>` `<thread-condition>` *condition*

Initialization Options

`thread` *thread*: A thread.

Remarks

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

15.2 Locks

The defined name of this module is `lock`.

15.2.1 `<lock>` `<object>` *class*

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

15.2.2 `lock?` *function*

Arguments

object: An object to examine.

Result

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

15.2.3 `lock` *function*

Arguments

lock: the lock to be acquired.

Result

The lock supplied as argument.

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 signalled 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 condition will be signalled continuably so that the process of acquiring the lock may continue after the condition has been handled.

See also

`unlock` and § 12.8 for details of conditions and signalling.

15.2.4 `unlock` *function*

Arguments

lock: the lock to be released.

Result

The lock supplied as argument.

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.

See also

`<lock>`.

15.2.5 `<simple-thread>` `<thread>` *class*

Place holder for `<simple-thread>` class.

16 Level-0 Module Library

This section describes the classes required at level-0 and the operations defined on instances of those classes. The section is organized by module in alphabetical order. These sub-sections contain information about the predefined classes in EuLisp that are necessary to make the language usable.

16.1 Characters

The defined name of this module is `character`.

16.1.1 `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 syntax table 16.1.1.1. 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 9.1 and also syntax table 16.1.1.1) form the syntax category *special-character-token* and are referred to using the digrams defined in table 16.1. Any character which does not have an external representation dealt with by cases described so far is represented by the digram `#\x` (see table 16.1) followed 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 `#\x0000` and `#\xabcd`, which denote, respectively, the characters at position 0 and at position 43981 in the character set current at the time of reading or writing. The syntax for the external representation of characters is defined in syntax table 16.1.1.1 below:

Table 3 – Character digrams

Operation	Digram
alert	`\a`
backspace	`\b`
delete	`\d`
formfeed	`\f`
linefeed	`\l`
newline	`\n`
return	`\r`
tab	`\t`
vertical tab	`\v`
hex-insertion	`\x`
string delimiter	`\"`
string escape	`\\`

16.1.1.1 Syntax

character:
  literal-character-token
  special-character-token
  numeric-character-token
literal-character-token:
  #*letter*
  #*decimal-digit*
  #*other-character*
  #*special-character*
special-character-token:
  #\\\a
  #\\\b
  #\\\d
  #\\\f
  #\\\l
  #\\\n
  #\\\r
  #\\\t
  #\\\v
  #\\\"
  #\\\\
numeric-character-token:
  #\\\x *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit*

NOTE 1 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.

16.1.2 `<character>` `<object>` *class*

The class of all characters.

16.1.3 `character?` *function*

Arguments

object: Object to examine.

Result

Returns *object* if it is a character, otherwise `()`.

16.1.4 `binary= <character>` *method*

Specialized Arguments

$character_1$ `<character>`: A character.

$character_2$ `<character>`: A character.

Result

If $character_1$ is the same character as $character_2$ the result is $character_1$, otherwise the result is `()`.

16.1.5 `binary< <character>` *method*

Specialized Arguments

$character_1$ `<character>`: A character.

$character_2$ `<character>`: A character.

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 `t`, the result is $character_1$, otherwise it is `()`. Any other comparison is an error and the result of such comparisons is undefined.

Examples

```
(binary< #\A #\Z)  ⇒  #\A
(binary< #\a #\z)  ⇒  #\a
(binary< #\0 #\9)  ⇒  #\0
(binary< #\A #\a)  ⇒  undefined
(binary< #\A #\0)  ⇒  undefined
(binary< #\a #\0)  ⇒  undefined
```

See also

Method `binary< <string>` for

16.1.6 `<string>` *class*

.

16.1.7 `as-lowercase` *generic function*

Generic Arguments

object `<object>`: An object to convert to lower case.

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*.

See also

Another method `as-lowercase <string>` for `<string>`.

16.1.8 `as-lowercase <character>` *method*

Specialized Arguments

character `<character>`: A character.

Result

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

16.1.9 as-uppercase *generic function*

Generic Arguments

object `<object>`: An object to convert to upper case.

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*.

See also

Another method is defined on `as-uppercase <string>` for `<string>`.

16.1.10 as-uppercase <character> *method*

Specialized Arguments

character `<character>`: A character.

Result

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

16.1.11 generic-print <character> *method*

Specialized Arguments

character `<character>`: Character to be ouptut on *stream*.

stream `<stream>`: Stream on which *character* is to be ouptut.

Result

The character *character*.

Remarks

Output the interpretation of *character* on *stream*.

16.1.12 generic-write <character> *method*

Specialized Arguments

character `<character>`: Character to be ouptut on *stream*.

stream `<stream>`: Stream on which *character* is to be ouptut.

Result

The character *character*.

Remarks

Output external representation of *character* on *stream* in the format `#\`*name* as described at the beginning of this section.

16.2 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 operators 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 `sequence?`, while collections in general can be identified by `collection?`.

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 $[0 \ldots size)$ of the collection with natural order.

A conforming level-0 implementation must define methods on these functions to support operations on lists (16.12), strings (16.16), tables (16.18), vector (16.19) and any combination of these.

16.2.1 <collection> `<object>` *class*

The class of all collections.

16.2.2 <sequence> `<collection>` *class*

The class of all sequences, the subset of collections which have natural order.

16.2.3 <character-sequence> `<sequence>` *class*

The class of all sequences of characters *e.g.* `<string>`.

16.2.4 <collection-condition> `<condition>` *condition*

This is the condition class for all collection processing conditions.

16.2.5 accumulate *generic function*

Generic Arguments

function `<function>`: A function of two arguments.

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

collection `<collection>`: The collection which is the subject of the accumulation operation.

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*.

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.

```
(accumulate * 1 #(1 2 3 4 5))           ⇒  120
(accumulate                             ⇒  (c b a)
  (lambda (a v) (cons v a))
  ()
  (make <table>
    'entries
    '((1 .  b) (0 .  a) (2 .  c))))
```

16.2.6 `accumulate1` *generic function*

Generic Arguments

function `<function>`: A function of two arguments.

collection `<collection>`: The collection which is the subject of the accumulation operation.

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 `()`.

Examples

```
(accumulate1                         ⇒  (4 2)
  (lambda (a v)
    (if (evenp v) (cons v a) a))
  '(1 2 3 4 5))
```

16.2.7 `all?` *generic function*

Generic Arguments

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

collection `<collection>`: A collection.

more-collections$_{opt}$: More collections.

Result

The *function* is applied to argument lists constructed from corresponding successive elements of *collection* and *more-collections*. If the result is `t` for all elements of *collection* and *more-collections* the result of `all?` is `t` otherwise `()`.

Examples

```
(all? even? #(1 2 3 4))  ⇒  ()
(all? even? #(2 4 6 8))  ⇒  t
```

See also

`any?`.

16.2.8 `any?` *generic function*

Generic Arguments

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

collection `<collection>`: A collection.

more-collections$_{opt}$: More collections.

Result

The *function* is applied to argument lists constructed from corresponding successive elements of *collection* and *more-collections*. If the result is `t`, the result of `any?` is `t` 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 `()`.

Examples

```
(any? even? #(1 2 3 4))          ⇒  t
(let ((x (list 1 2 3 4)))        ⇒  ()
  (any? > x (cdr x)))
(any?                            ⇒  t
  (lambda (a b) (= (% a b) 0))
  #(3 2) '(1 0))
```

See also

`all?`.

16.2.9 `collection?` *generic function*

Generic Arguments

object `<object>`: An object to examine.

Result

Returns `t` if *object* is a collection, otherwise `()`.

Remarks

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

16.2.10 `concatenate` *generic function*

Generic Arguments

collection `<collection>`: A collection.

more-collections$_{opt}$: More collections.

Result

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

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.

Examples

```
(concatenate #(1) '(2) "bc")  ⇒  #(1 2 #\b #\c))
(concatenate "a" '(#\b))      ⇒  "ab"
(concatenate                  ⇒  #<table:
  (make <table>)                    0 -> "c",
  '(a b)                            1 -> b>
  "c")
```

16.2.11 `delete` *generic function*

Generic Arguments

object `<object>`: Object to be removed.

collection `<collection>`: A collection.

test$_{opt}$: The function to be used to compare *object*—/ and the elements of *collection*.

Result

If there is an element of *collection* such that test returns `t` when applied to *object*—/ and that element, then the result is the modified *collection*, less that element. Otherwise, the result is *collection*.

Remarks

`delete` is destructive. The *test* function defaults to `eql`.

16.2.12 `do` *generic function*

Generic Arguments

function `<function>`: A function.

collection `<collection>`: A collection.

more-collections$_{opt}$: More collections.

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.

Examples

```
(do prin '(1 b #\c))   ⇒  1bc
(do write '(1 b #\c))  ⇒  1b#\c
```

16.2.13 `element` *generic function*

Generic Arguments

collection `<collection>`: The object to be accessed or updated.

key `<object>`: The object identifying the key of the element in *collection*.

Result

The value associated with *key* in *collection*.

Examples

```
(element "abc" 1)                    ⇒  #\b
(element '(a b c) 1)                 ⇒  b
(element #(a b c) 1)                 ⇒  b
(element                             ⇒  b
  (make <table> fill-value:  'b)
  1)
```

16.2.14 `(setter element)` *setter*

Generic Arguments

collection `<collection>`: 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).

Result

The argument supplied as *value*, having updated the association of *key* in *collection* to refer to *value*.

16.2.15 `empty?` *generic function*

Generic Arguments

collection `<collection>`: The object to be examined.

Result

Returns `t` if *collection* is the object identified with the empty object for that class of collection.

Examples

```
(emptyp "")                ⇒  t
(emptyp ())                ⇒  t
(emptyp #())               ⇒  t
(emptyp (make <table>))    ⇒  t
```

16.2.16 `fill` *generic function*

Generic Arguments

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

object <object>: The object with which to fill *collection*.

keys$_{opt}$: The keys with which *object* is to be associated.

Result

The result is ().

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*.

16.2.17 `find-key` *generic function*

Generic Arguments

collection <collection>: A collection.

test <function>: A function.

skip$_{opt}$: An integer.

failure$_{opt}$: An integer.

Result

The function *test* is applied to successive elements of *collection*. If *test* returns t when applied to an element, then the result of find-key is the key associated with that element.

Remarks

The value *skip*, which defaults to zero, indicates how many successful tests are to be made before returning a result. The value *failure*, which defaults to (), is returned if no key satisfying the test was found. Note that *skip* and *failure* are positional arguments and that *skip* must be specified if *failure* is specified.

16.2.18 `first` *generic function*

Generic Arguments

sequence <sequence>: A sequence.

Result

The result is contents of index position 0 of *sequence*.

16.2.19 `last` *generic function*

Generic Arguments

sequence <sequence>: A sequence.

Result

The result is last element of *sequence*.

16.2.20 `key-sequence` *generic function*

Generic Arguments

collection <collection>: A collection.

Result

The result is a collection comprising the keys of *collection*.

16.2.21 `map` *generic function*

Generic Arguments

function <function>: A function.

collection <collection>: A collection.

more-collections$_{opt}$: More collections.

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.

Examples

```
(map cons #(1 2) '(3))  ⇒  #((1 . 3))
(map                    ⇒  #(3 -1 2 1)
  (lambda (f) (f 1 2))
  #(+ - * %))
```

16.2.22 `member` *generic function*

Generic Arguments

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

collection <collection>: The collection to be searched.

test$_{opt}$: The function to be used to compare *object* and the elements of *collection*.

Result

Returns t if there is an element of *collection* such that the result of the application of *test* to *object* and that element is t. If *test* is not supplied, eql is used by default. Note that t 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.

Examples

```
(member #\b "abc")                    ⇒ t
(member 'b '(a b c))                  ⇒ (b c)
(member 'b #(a b c))                  ⇒ t
(member                               ⇒ t
  'b
  (make <table>
    'entries
    '((1 . b) (0 . a) (2 . c))))
```

16.2.23 `remove` *generic function*

Generic Arguments

object `<object>`: Object to be removed.

collection `<collection>`: A collection.

test$_{opt}$: The function to be used to compare *object* and the elements of *collection*.

Result

If there is an element of *collection* such that test returns `t` when applied to *object* and that element, then the result is a shallow copy of *collection* less that element. Otherwise, the result is *collection*.

Remarks

The test function defaults to `eql`.

16.2.24 `reverse` *generic function*

Generic Arguments

collection `<collection>`: A collection.

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.

Examples

```
(reverse "abc")      ⇒ "cba"
(reverse '(1 2 3))   ⇒ (3 2 1)
(reverse #(a b c))   ⇒ #(c b a)
```

16.2.25 `reverse!` *generic function*

Generic Arguments

collection `<collection>`: A collection.

Result

Destructively reverses the order of the elements in *collection* (see `reverse`) and returns it.

16.2.26 `sequence?` *generic function*

Generic Arguments

object `<object>`: An object to examine.

Result

Returns `t` if *object* is a sequence (has natural order), otherwise `()`.

Remarks

This predicate does not return *object* because `()` is a sequence.

16.2.27 `size` *generic function*

Generic Arguments

collection `<collection>`: The object to be examined.

Result

An integer which denotes the size of *collection* according to the method for the class of *collection*.

Examples

```
(size "")                                   ⇒ 0
(size ())                                   ⇒ 0
(size #())                                  ⇒ 0
(size (make <table>))                       ⇒ 0
(size "abc")                                ⇒ 3
(size (cons 1 ()))                          ⇒ 1
(size (cons 1 . 2))                         ⇒ 1
(size (cons 1 (cons 2 . 3)))                ⇒ 2
(size '(1 2 3))                             ⇒ 3
(size #(a b c))                             ⇒ 3
(size (make <table> 'entries '((0 . a)))    ⇒ 1
```

16.2.28 `slice` *generic function*

Generic Arguments

sequence `<sequence>`: A sequence.

start `<fpi>`: The index of the first element of the slice.

end `<fpi>`: The index of the last element of the slice.

Result

The result is new sequence of the same class as *sequence* containing the elements of *sequence* from *start* up to but not including *end*.

Examples

```
(slice '(a b c d) 1 3)  ⇒  (b c)
```

16.2.29 `sort` *generic function*

Generic Arguments

sequence `<sequence>`: A sequence.

comparator `<function>`: A function.

Result

The result of sort is a new *sequence* comprising the elements of *sequence* ordered according to *comparator*.

Remarks

Methods on this function are only defined for `<list>` and `<vector>`.

16.2.30 `sort!` *generic function*

Generic Arguments

sequence `<sequence>`: A sequence.

comparator `<function>`: A function.

Result

Destructively sorts the elements of *sequence* (see `sort`) and returns it.

Remarks

Methods on this function are only defined for `<list>` and `<vector>`.

16.2.31 `(converter <list>)` *converter*

Specialized Arguments

collection `<collection>`: A collection to be converted into a list.

Result

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

See also

Conversion (16.4).

16.2.32 `(converter <string>)` *converter*

Specialized Arguments

collection `<collection>`: A collection to be converted into a string.

Result

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

See also

Conversion (16.4).

16.2.33 `(converter <table>)` *converter*

Specialized Arguments

collection `<collection>`: A collection to be converted into a table.

Result

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

See also

Conversion (16.4).

16.2.34 `(converter <vector>)` *converter*

Specialized Arguments

collection `<collection>`: A collection to be converted into a vector.

Result

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

See also

Conversion (16.4).

16.3 Comparison

The defined name of this module is compare. There are three binary functions for comparing objects for equality, eq, eql, binary= and the n-ary = which uses binary=. The three binary functions are related in the following way:

$$(\texttt{eq}\ a\ b) \Rightarrow (\texttt{eql}\ a\ b) \Rightarrow (\texttt{binary=}\ a\ b)$$
$$(\texttt{eq}\ a\ b) \not\Leftarrow (\texttt{eql}\ a\ b) \not\Leftarrow (\texttt{binary=}\ a\ b)$$

There are four n-ary function for comparing objects by order, < and > which are implemented by the generic function binary<, <= and >= which are implemented by the generic functions binary< and binary=. There is also one binary function for comparing objects for inequality, !=. A summary of the comparison functions and the classes for which they have defined behaviour is given below:

eq:	<object>×<object>
eql:	<object>×<object> ⇒eq
	<character>×<character>
	<fpi>×<fpi>
	<double-float>×<double-float> ⇒binary=
binary=:	<object>×<object>
	<character>×<character>
	<null>×<null>
	<number>×<number> ⇒eql
	<fpi>×<fpi>
	<double-float>×<double-float>
	<double-float>×<fpi>
	<fpi>×<double-float>
	<cons>×<cons>
	<string>×<string>
	<vector>×<vector>
binary<:	<character>×<character>
	<symbol>×<symbol>
	<fpi>×<fpi>
	<double-float>×<double-float>
	<string>×<string>
=:	<object>×<object> ⇒binary=
!=:	<object>×<object>
<:	<object>×<object> ⇒binary<
>:	<object>×<object>
<=:	<object>×<object>
>=:	<object>×<object>

16.3.1 eq *function*

Arguments

object$_1$: An object.

object$_2$: An object.

Result

Compares *object*$_1$ and *object*$_2$ and returns t if they are the *same* object, otherwise (). *Same* in this context means "identifies the same memory location".

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 .

Examples

```
(eq 'a 'a)                          ⇒  t
(eq 'a 'b)                          ⇒  ()
(eq #\a #\a)                        ⇒  t or ()
(eq 3 3)                            ⇒  t or ()
(eq 3 3.0)                          ⇒  ()
(eq 3.0 3.0)                        ⇒  t or ()
(eq (cons 'a 'b) (cons 'a 'c))      ⇒  ()
(eq (cons 'a 'b) (cons 'a 'b))      ⇒  ()
(eq '(a . b) '(a . b))              ⇒  t or ()
(let ((x (cons 'a 'b))) (eq x x))   ⇒  t
(let ((x '(a . b))) (eq x x))       ⇒  t
(eq "string" "string")              ⇒  t or ()
(eq #('a 'b) #('a 'b))              ⇒  t or ()
(let ((x #('a 'b))) (eq x x))       ⇒  t
```

16.3.2 eql *function*

Arguments

object$_1$: An object.

object$_2$: An object.

Result

If the class of *object*$_1$ and of *object*$_2$ is the same and is a subclass of <character> or <number>, the result is that of comparing them under binary= <character> or binary= <number> respectively. Otherwise the result is that of comparing them under eq.

Examples

Given the same set of examples as for eq, the same result is obtained except in the following cases:

```
(eql #\a #\a)  ⇒  t
(eql 3 3)      ⇒  t
(eql 3.0 3.0)  ⇒  t
```

16.3.3 binary= *generic function*

Arguments

object$_1$, <object>: An object.

object$_2$, <object>: An object.

Result

Returns t or () according to the method for the class(es) of *object*$_1$ and *object*$_2$. It is an error if either or both of the arguments is self-referential.

See also

Class specific methods on binary= are defined for <character>, <list>, <number> (with specialisations for <fpi> and <double-float>), <string>, <vectors>. All other cases are handled by the default method defined for <object>:

16.3.4 binary= <object> *method*

Specialized Arguments

object$_1$ <object>: An object.

object$_2$ <object>: An object.

Result

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

16.3.5 `binary<` *generic function*

Generic Arguments

object$_1$ `<object>` : An object.

object$_2$ `<object>` : An object.

Result

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

See also

Class specific methods on `binary<` are defined for `<character>`, `<string>`, `<fpi>` and `<double-float>`.

16.3.6 `=` *function*

Arguments

number$_1$ ... : A non-empty sequence of numbers.

Result

Given one argument the result is `t`. Given more than one argument the result is determined by `binary=`, returning `t` if all the arguments are the same, otherwise `()`.

16.3.7 `!=` *function*

Arguments

number$_1$ ... : A non-empty sequence of numbers.

Result

Given one argument the result is `()`. Given more than one argument the result is determined by `binary=`, returning `()` if all the arguments are the same, otherwise `t`.

16.3.8 `<` *function*

Arguments

object$_1$ ... : A non-empty sequence of objects.

Result

Given one argument the result is `t`. Given more than one argument the result is `t` if the sequence of objects *object*$_1$ up to *object*$_n$ is strictly increasing according to the generic function `binary<`. Otherwise, the result is `()`.

16.3.9 `>` *function*

Arguments

object$_1$ ... : A non-empty sequence of objects.

Result

Given one argument the result is `t`. Given more than one argument the result is `t` if the sequence of objects *object*$_1$ up to *object*$_n$ is strictly decreasing according to the generic function `binary<` applied to the arguments in reverse order. Otherwise, the result is `()`.

16.3.10 `<=` *function*

Arguments

object$_1$ ... : A non-empty sequence of objects.

Result

Given one argument the result is `t`. Given more than one argument the result is `t` if the sequence of objects *object*$_1$ up to *object*$_n$ is strictly increasing according to the generic function `binary<` and `binary=`. Otherwise, the result is `()`.

16.3.11 `>=` *function*

Arguments

object$_1$ ... : A non-empty sequence of objects.

Result

Given one argument the result is `t`. Given more than one argument the result is `t` if the sequence of objects *object*$_1$ up to *object*$_n$ is strictly decreasing according to the generic function `binary<` and `binary=` applied to the arguments in reverse order. Otherwise, the result is `()`.

16.3.12 `max` *function*

Arguments

object$_1$ ... : A non-empty sequence of objects.

Result

The maximal element of the sequence of objects *object*$_1$ up to *object*$_n$ using the generic function `binary<`. Zero arguments is an error. One argument returns *object*$_1$.

16.3.13 `min` *function*

Arguments

object$_1$ ... : A non-empty sequence of objects.

Result

The minimal element of the sequence of objects *object*$_1$ up to *object*$_n$ using the generic function `binary<`. Zero arguments is an error. One argument returns *object*$_1$.

16.4 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:

```
(defgeneric (converter new-class) (instance))
```

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

```
(defmethod (converter new-class)
           ((instance other-class)))
```

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 16.2, converters from string in section 16.16, etc..

16.4.1 `convert` *function*

Arguments

object: An instance of some class to be converted to an instance of *class*.

class: The class to which *object* is to be converted.

Result

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

16.4.2 `<conversion-condition>` `<condition>` *condition*

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

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.

Remarks

Should be signalled by `convert` or a converter method.

16.4.3 `<no-converter>` `<conversion-condition>` *condition*

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.

Remarks

Should be signalled by `convert` if there is no associated function.

16.4.4 `converter` *function*

Arguments

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

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.

16.4.5 `(setter converter)` *setter*

Arguments

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

generic-function: The new converter function.

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>`.

Remarks

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

See also

Converter methods are defined for collections (16.2), double float (16.6), fixed precision integer (16.9), string (16.16), symbol (16.17), vector (16.19).

16.5 Copying

The defined name of this module is `copy`.

16.5.1 `deep-copy` *generic function*

Generic Arguments

object: An object to be copied.

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*.

See also

Class specific sections which define methods on `deep-copy`: list (16.12), string (16.16), table (16.18) and vector (16.19).

16.5.2 `deep-copy <object>` *method*

Specialized Arguments

object `<object>`: An object.

Result

Returns *object*.

16.5.3 `deep-copy <class>` *method*

Specialized Arguments

class `<class>`: A class.

Result

Constructs and returns a new structure whose slots are initialized with copies (using `deep-copy`) of the contents of the slots of *class*.

16.5.4 `shallow-copy` *generic function*

Generic Arguments

object: An object to be copied.

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*.

See also

Class specific sections which define methods on `shallow-copy`: pair (16.12), string (16.16), table (16.18) and vector (16.19).

16.5.5 `shallow-copy <object>` *method*

Specialized Arguments

object `<object>`: An object.

Result

Returns *object*.

16.5.6 `shallow-copy <class>` *method*

Specialized Arguments

class `<class>`: A class.

Result

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

16.6 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 compare module (16.3):

`binary=`, `binary<`,

the number module (16.14):

`binary+`, `binary-`, `binary*`, `binary/`, `binary-mod`, `negate`, `zero?`

the float module (16.7):

`ceiling`, `floor`, `round`, `truncate`

and the elementary functions module (**??**):

`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.

16.6.1 `<double-float>` — `<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:

double-exponent:
  `d` *sign*$_{opt}$ *decimal-integer*
  `D` *sign*$_{opt}$ *decimal-integer*

The general syntax for floating point numbers is given in syntax table 16.7.1.1.

16.6.2 `double-float?` *function*

Arguments

object: Object to examine.

Result

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

See also

`float?` (16.14).

16.6.3 `most-positive-double-float` `<double-float>` *constant*

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.

16.6.4 `least-positive-double-float` `<double-float>` *constant*

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.

16.6.5 `least-negative-double-float` `<double-float>` *constant*

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.

16.6.6 `most-negative-double-float` `<double-float>` *constant*

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.

16.6.7 `(converter <string>)` *converter*

Specialized Arguments

x `<double-float>`: A double precision float.

Result

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

16.6.8 `(converter <fpi>)` *converter*

Specialized Arguments

x `<double-float>`: A double precision float.

Result

A fixed precision integer.

Remarks

This function is the same as the `<double-float>` method of `round`. It is defined for the sake of symmetry.

16.6.9 `generic-print <double-float>` *method*

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.

Result

The double float supplied as the first argument.

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 16.8).

16.6.10 `generic-write <double-float>` *method*

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.

Result

The double float supplied as the first argument.

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 16.8).

16.7 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 (16.14) and further operations on floating point numbers are defined in the elementary functions module (**??**). A concrete float class is defined in the double float module (16.6).

16.7.1 `float` *syntax*

The syntax for the external representation of floating point literals is defined in syntax table 16.7.1.1. 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 16.8).

16.7.1.1 Syntax

float:
  sign$_{opt}$ *unsigned-float* *exponent*$_{opt}$
unsigned-float:
  float-format-1
  float-format-2
  float-format-3
float-format-1:
  decimal-integer `.`
float-format-2:
  `.` *decimal-integer*
float-format-3:
  float-format-1 *decimal-integer*
exponent:
  double-exponent

A floating point number has six forms of external representation depending on whether either or both the whole and the fractional part are specified and on whether an exponent is specified. In addition, a positive floating point number is optionally preceded by a plus sign and a negative floating point number is preceded by a minus sign. For example: `+123.` (*float format 1*), `-.456` (*float format 2*), `123.456` (*float format 3*); and with exponents: `+123456.D-3`, `1.23455D2`, `-.123456D3`.

16.7.2 `<float>` `<number>` *class*

The abstract class which is the superclass of all floating point numbers.

16.7.3 `float?` *function*

Arguments

object: Object to examine.

Result

Returns *object* if it is a floating point number, otherwise `()`.

16.7.4 `ceiling` *generic function*

Generic Arguments

float `<float>`: A floating point number.

Result
Returns the smallest integral value not less than *float* expressed as a float of the same class as the argument.

16.7.5 `floor` *generic function*

Generic Arguments

float `<float>`: A floating point number.

Result
Returns the largest integral value not greater than *float* expressed as a float of the same class as the argument.

16.7.6 `round` *generic function*

Arguments

float: A floating point number.

Result
Returns the integer whose value is closest to *float*, except in the case when *float* is exactly half-way between two integers, when it is rounded to the one that is even.

16.7.7 `truncate` *generic function*

Arguments

float: A floating point number.

Result
Returns the greatest integer value whose magnitude is less than or equal to *float*.

16.8 Formatted-IO

The defined name of this module is `formatted-io`.

16.8.1 `scan` *function*

Arguments

format-string: A string containing format directives.

stream$_{opt}$: A stream from which input is to be taken.

Result
Returns a list of the objects read from *stream*.

Remarks
This function provides support for formatted input. The *format-string* specifies reading directives, and inputs are matched according to these directives. An error is signaled (condition: `<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 *stream* is not supplied, input is taken from `stdin`. Scan returns a list of the objects read in.

`~a` **any:** any object.

`~b` **binary:** an integer in binary format.

`~c` **character:** a single character

`~d` **decimal:** an integer decimal format.

`~`n_{opt}`e`: a exponential-format floating-point number.

`~`n_{opt}`f`: a fixed-format floating-point number.

`~o` **octal:** an integer in octal format.

`~r` **radix:** an integer in specified radix format.

`~x` **hexadecimal:** an integer in hexadecimal format.

`~%` **newline:** matches a newline character in the input.

16.8.2 `<scan-mismatch>` `<stream-condition>` *condition*

Initialization Options

`format-string` *string*: The value of this option is the format string that was passed to `scan`.

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

Remarks
This condition is signalled by `scan` if the format string does not match the data input from *stream*.

16.8.3 `sformat` *function*

Arguments

stream: A stream.

format-string: A string containing format directives.

object$_1$ ...$_{opt}$: A sequence of objects to be output on *stream*.

Result

Returns *stream* and has the side-effect of outputting *objects* according the formats specified in *format-string* to *stream*. Characters are output as if the string were output by the `sprin` 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 except for the prefix ~ rather than `%`.

`~a` **any:** uses `sprin` to output the argument.

`~b` **binary:** the argument must be an integer and is output in binary notation (syntax table 16.10.1.1).

`~c` **character:** the argument must be a character and is output using `write` (syntax table 16.1.1.1).

`~d` **decimal:** the argument must be an integer and is output using `write` (syntax table 16.10.1.1).

`~`m_{opt}`.`n_{opt}`e` **exponential-format floating-point:** the argument must be a floating point number. It is output in the style $-_{opt}$*d.ddd*`e`±*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 (syntax table 16.7.1.1). If the value to be output has fewer characters than m it is padded on the left with spaces.

`~`m_{opt}`.`n_{opt}`f` **fixed-format floating-point:** the argument must be a floating point number. It is output in the style $-_{opt}$*ddd.ddd*, 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 (syntax table 16.7.1.1). 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_{opt}`.`n_{opt}`g` **generalized floating-point:** the argument must be a floating point number. It is output in either fixed-format or exponential notation as appropriate (syntax table 16.7.1.1).

`~o` **octal:** the argument must be an integer and is output in octal notation (syntax table 16.10.1.1).

`~`n`r` **radix:** the argument must be an integer and is output in radix n notation (syntax table 16.10.1.1).

`~s` **s-expression:** uses `write` to output the argument (syntax table 9.5.0.5).

`~`n_{opt}`t` **tab:** output sufficient spaces to reach the next tab-stop, if n is not specified, or the n^{th} tab stop if it is.

`~x` **hexadecimal:** the argument must be an integer and is output in hexadecimal notation (syntax table 16.10.1.1).

`~%` **newline:** output a newline character.

`~&` **conditional newline:** output a newline character using, if it cannot be determined that the output stream is at the beginning of a fresh line.

`~~` **tilde:** output a tilde character using `sprin`.

16.8.4 `format` *function*

Arguments

format-string: A string containing format directives.

object$_1$ ...$_{opt}$: A sequence of objects to be output on `stdout`.

Result

Returns `stdout` and has the side-effect of outputting *objects* according the formats specified in *format-string* (see `sformat`) to `stdout`.

16.8.5 `fmt` *function*

Arguments

format-string: A string containing format directives.

object$_1$ ...$_{opt}$: A sequence of objects to be formatted into a string.

Remarks

Return the string created by formatting the *objects* according the formats specified in *format-string* (see `sformat`).

16.9 Fixed Precision Integers

The defined name of this module is `fpi`. Arithmetic operations for `<fpi>` are defined by methods on the generic functions defined in the compare module (16.3):

`binary=`, `binary<`,

the number module:

`binary+`, `binary-`, `binary*`, `binary/`, `binary%`, `binary-gcd`, `binary-lcm`, `binary-mod`, `negate`, `zero?`

and in the integer module:

`even?`

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

16.9.1 `<fpi>` `<integer>` *class*

The class of all instances of fixed precision integers.

16.9.2 `int?` *function*

Arguments

object: Object to examine.

Result

Returns *object* if it is fixed precision integer, otherwise `()`.

16.9.3 `most-positive-int` `<fpi>` *constant*

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`.

16.9.4 `most-negative-int` `<fpi>` *constant*

Remarks

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

16.9.5 `(converter <string>)` *converter*

Specialized Arguments

integer `<fpi>`: An integer.

Result

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

16.9.6 `(converter <double-float>)` *converter*

Specialized Arguments

integer `<fpi>`: An integer.

Result

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

16.9.7 `generic-print <fpi>` *method*

Specialized Arguments

integer `<fpi>`: An integer to be output on *stream*.

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

Result

The fixed precision integer supplied as the first argument.

Remarks

Outputs external representation of *integer* on *stream* in decimal as defined by *decimal integer* at the beginning of this section.

16.9.8 `generic-write <fpi>` *method*

Specialized Arguments

integer `<fpi>`: An integer to be output on *stream*.

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

Result

The fixed precision integer supplied as the first argument.

Remarks

Outputs external representation of *integer* on *stream* in decimal as defined by *decimal integer* at the beginning of this section.

16.10 Integers

The defined name of this module is `integer`. This module defines the abstract class `<integer>` and the behaviour of some generic functions on integers. Further operations on numbers are defined in the numbers module (16.14). A concrete integer class is defined in the fixed precision integer module (16.9).

16.10.1 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`, `#o2322` and `#x4d2` all denote the same value.

The general notation for an arbitrary base is `#`*base*`r`, where *base* is an unsigned decimal number. Thus, the above examples may also be written: `#10r1234`, `#2r10011010010`, `#8r2322`, `#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 below.

16.10.1.1 Syntax

```
integer:
        sign_opt unsigned-integer
sign:  one of
        + -
unsigned-integer:
        binary-integer
        octal-integer
        decimal-integer
        hexadecimal-integer
        specified-base-integer
binary-integer:
        #b binary-digit+
binary-digit:  one of
        0 1
octal-integer:
        #o octal-digit+
octal-digit:  one of
        0 1 2 3 4 5 6 7
decimal-integer:
        decimal-digit+
hexadecimal-integer:
        #x hexadecimal-digit+
hexadecimal-digit:
        decimal-digit
        hex-lower-letter
        hex-upper-letter
hex-lower-letter:  one of
        a b c d e f
hex-upper-letter:  one of
        A B C D E F
specified-base-integer:
        # base-specification r
        specified-base-digit
        specified-base-digit*
base-specification:
        { 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
        { 1 | 2 } decimal-digit
        3 { 0 | 1 | 2 | 3 | 4 | 5 | 6 }
specified-base-digit:
        decimal-digit
        letter
```

NOTE 1 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.

16.10.2 `<integer>` `<number>` *class*

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

16.10.3 integer? *function*

Arguments

object: Object to examine.

Result

Returns *object* if it is an integer, otherwise `()`.

16.10.4 even? *generic function*

Arguments

integer, `<integer>`: An integer.

Result

Returns `t` if two divides *integer*, otherwise `()`.

16.10.5 `odd?` *function*

Arguments

integer: An integer.

Result

Returns the equivalent of the logical negation of `even?` applied to *integer*.

16.11 Keywords

The defined name of this module is `keyword`.

16.11.1 `keyword` *syntax*

The syntax of keywords is very similar to that of identifiers and of symbols, including all the escape conventions, but are distinguished by a colon (`:`) suffix:

16.11.1.1 Syntax

keyword:
    identifier:

It is an error to use a keyword where an identifier is expected, such as, for example, in lambda parameter lists or in let binding forms.

The matter of keywords appering in lambda parameter lists, for example, `rest:`*, instead of the dot notation, is currently an open issue.*

Operationally, the most important aspect of keywords 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 a keyword is the same object. Or, more briefly, every keyword with the same name denotes the same keyword. A consequence of this guarantee is that keywords may be compared using `eq`.

16.11.2 `<keyword>` `<name>` *class*

The class of all instance of `<keyword>`.

Initialization Options

`string` *string*: The string containing the characters to be used to name the keyword. The default value for string is the empty string, thus resulting in the keyword with no name, written `|:|`.

What is the defined behaviour if the last character of string is colon?

16.11.3 `keyword?` *function*

Arguments

object: Object to examine.

Result

Returns *object* if it is a keyword.

16.11.4 `keyword-name` *function*

Arguments

keyword: A keyword.

Result

Returns a *string* which is `binary=` `<string>` to that given as the argument to the call to `make` which created *keyword*. It is an error to modify this string.

16.11.5 `keyword-exists?` *function*

Arguments

string: A string containing the characters to be used to determine the existence of a keyword with that name.

Result

Returns the keyword whose name is *string* if that keyword has already been constructed by `make`. Otherwise, returns `()`.

16.11.6 `generic-print <keyword>` *method*

Specialized Arguments

keyword `<keyword>`: The keyword to be output on *stream*.

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

Result

The keyword supplied as the first argument.

Remarks

Outputs the external representation of *keyword* on *stream* as described in the section on symbols, interpreting each of the characters in the name.

16.11.7 `generic-write <keyword>` *method*

Specialized Arguments

keyword `<keyword>`: The keyword to be output on *stream*.

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

Result

The keyword supplied as the first argument.

Remarks

Outputs the external representation of *keyword* on *stream* as described in the section on symbols. If any characters in the name would not normally be legal constituents of a keyword, the output is preceded and succeeded by multiple-escape characters.

Examples

```
(write (make <keyword> 'string "abc"))  ⇒  abc:
(write (make <keyword> 'string "a c"))  ⇒  |a c:|
(write (make <keyword> 'string ").("))  ⇒  |).(:|
```

16.11.8 `(converter <string>)` *converter*

Specialized Arguments

keyword `<keyword>`: A keyword to be converted to a string.

Result

A string.

Remarks

This function is the same as `keyword-name`. It is defined for the sake of symmetry.

16.12 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 16.2 (collections) for further operations on lists.

16.12.1 `<list>` — `<collection>` *class*

The class of all lists.

16.12.2 `()` — *syntax*

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*.

16.12.3 `<null>` — `<list>` *class*

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

16.12.4 `null?` — *function*

Arguments

object: Object to examine.

Result

Returns `t` if *object* is the empty list, otherwise `()`.

16.12.5 `generic-print <null>` — *method*

Specialized Arguments

null: The empty list.

stream: The stream on which the representation is to be output.

Result

The empty list.

Remarks

Output the external representation of the empty list on *stream* as described above.

16.12.6 `generic-write <null>` — *method*

Specialized Arguments

null: The empty list.

stream: The stream on which the representation is to be output.

Result

The empty list.

Remarks

Output the external representation of the empty list on *stream* as described above.

16.12.7 `pair` — *syntax*

A pair is written as `(`$object_1$ `.` $object_2$`)`, where $object_1$ is called the `car` and $object_2$ is called the `cdr`. There are two special cases in the external representation of pair. If $object_2$ is the empty list, then the pair is written as `(`$object_1$`)`. If $object_2$ is an instance of `pair`, then the pair is written as `(`$object_1$ $object_3$ `.` $object_4$`)`, where $object_3$ is the `car` of $object_2$ and $object_4$ is the `cdr` with the above rule for the empty list applying. By induction, a list of length n is written as `(`$object_1 \ldots object_{n-1}$ `.` $object_n$`)`, with the above rule for the empty list applying. The representations of $object_1$ and $object_2$ are determined by the external representations defined in other sections of this definition (see `<character>` (16.1), `<double-float>` (16.6), `<fpi>` (16.9), `<string>` (16.16), `<symbol>` (16.17) and `<vector>` (16.19), as well as instances of `<cons>` itself. The syntax for the external representation of pairs and lists is defined in syntax table 16.12.7.1.

16.12.7.1 Syntax

```
null:
      ()
pair:
      ( object . object )
list:
      empty-list
      proper-list
      improper-list
empty-list:
      ()
proper-list:
      ( object+ )
improper-list:
      ( object+ . object )
```

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)`

16.12.8 `<cons>` — `<list>` *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 syntax table 16.12.7.1).

It is an error to apply `car` or `cdr` or their `setter` functions to anything other than a pair. The empty list is not a pair and `(car ())` or `(cdr ())` is an error.

16.12.9 `cons?` *function*

Arguments

object: Object to examine.

Result

Returns *object* if it is a pair, otherwise `()`.

16.12.10 `atom?` *function*

Arguments

object: Object to examine.

Result

Returns *object* if it is not a pair, otherwise `()`.

16.12.11 `cons` *function*

Arguments

$object_1$: An object. pair.

$object_2$: An object. pair.

Result

Allocates a new pair whose slots are initialized with $object_1$ in the `car` and $object_2$ in the `cdr`.

16.12.12 `car` *function*

Arguments

pair: A pair.

Result

Given a pair, such as the result of `(cons` $object_1$ $object_2$`)`, then the function `car` returns $object_1$.

16.12.13 `cdr` *function*

Arguments

pair: A pair.

Result

Given a pair, such as the result of `(cons` $object_1$ $object_2$`)`, then the function `cdr` returns $object_2$.

16.12.14 `(setter car)` *setter*

Arguments

pair: A pair.

object: An object.

Result

Given a pair, such as the result of `(cons` $object_1$ $object_2$`)`, then the function `(setter car)` replaces $object_1$ with *object*. The result is *object*.

16.12.15 `(setter cdr)` *setter*

Arguments

pair: A pair.

object: An object.

Result

Given a pair, such as the result of `(cons` $object_1$ $object_2$`)`, then the function `(setter cdr)` replaces $object_2$ with *object*. The result is *object*.

Remarks

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

16.12.16 `binary= <cons>` *method*

Specialized Arguments

$pair_1$ `<cons>`: A pair.

$pair_2$ `<cons>`: A pair.

Result

If the result of the conjunction of the pairwise application of `binary=` to the `car` fields and the `cdr` fields of the arguments is `t` the result is $pair_1$ otherwise the result is `()`.

16.12.17 `deep-copy <cons>` *method*

Specialized Arguments

pair `<cons>`: A pair.

Result

Constructs and returns a copy of the list starting at *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 `deep-copy` method for the class of the object.

16.12.18 `shallow-copy <cons>` *method*

Specialized Arguments

pair `<cons>`: A pair.

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.

16.12.19 `list` *function*

Arguments

$object_1$ ... $object_{n\,opt}$: A sequence of objects.

Result

Allocates a set of pairs each of which has been initialized with $object_i$ in the car field and the pair whose car field contains $object_{i+1}$ in the cdr field. Returns the pair whose car field contains $object_1$.

Examples

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

16.12.20 generic-print <cons> *method*

Specialized Arguments

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

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

Result

The pair supplied as the first argument.

Remarks

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

16.12.21 generic-write <cons> *method*

Specialized Arguments

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

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

Result

The pair supplied as the first argument.

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*.

16.13 Elementary Functions

The defined name of this module is `mathlib`. The functionality defined for this module is intentionally precisely that of the trigonmetric 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`.

16.13.1 pi <double-float> *constant*

Remarks

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

16.13.2 acos *generic function*

Generic Arguments

float <float>: A floating point number.

Result

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

16.13.3 asin *generic function*

Generic Arguments

float <float>: A floating point number.

Result

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

16.13.4 atan *generic function*

Generic Arguments

float <float>: A floating point number.

Result

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

16.13.5 atan2 *generic function*

Generic Arguments

$float_1$ <float>: A floating point number.

$float_2$ <float>: A floating point number.

Result

Computes the principal value of the arc tangent of $float_1/float_2$, which is a value in the range $[-\pi, +\pi]$ radians, using the signs of both arguments to determine the quadrant of the result. An error might be signalled (condition-class: <domain-condition>) if either $float_1$ or $float_2$ is zero.

16.13.6 `cos` *generic function*

Generic Arguments

float `<float>`: A floating point number.

Result

Computes the cosine of *float* (measured in radians).

16.13.7 `sin` *generic function*

Generic Arguments

float `<float>`: A floating point number.

Result

Computes the sine of *float* (measured in radians).

16.13.8 `tan` *generic function*

Generic Arguments

float `<float>`: A floating point number.

Result

Computes the tangent of *float* (measured in radians).

16.13.9 `cosh` *generic function*

Generic Arguments

float `<float>`: A floating point number.

Result

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

16.13.10 `sinh` *generic function*

Generic Arguments

float `<float>`: A floating point number.

Result

Computes the hyperbolic sine of *float*. An error might be signalled (condition class: `<range-condition>`) if the magnitude of *float* is too large.

16.13.11 `tanh` *generic function*

Generic Arguments

float `<float>`: A floating point number.

Result

Computes the hyperbolic tangent of *float*.

16.13.12 `exp` *generic function*

Generic Arguments

float `<float>`: A floating point number.

Result

Computes the exponential function of *float*. An error might be signalled (condition class: `<range-condition>`) if the magnitude of *float* is too large.

16.13.13 `log` *generic function*

Generic Arguments

float `<float>`: A floating point number.

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.

16.13.14 `log10` *generic function*

Generic Arguments

float `<float>`: A floating point number.

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.

16.13.15 `pow` *generic function*

Generic Arguments

$float_1$ `<float>`: A floating point number.

$float_2$ `<float>`: A floating point number.

Result

Computes $float_1$ raised to the power $float_2$. An error is signalled (condition class: `<domain-condition>`) if $float_1$ is negative and $float_2$ is not integral. An error is signalled (condition class: `<domain-condition>`) if the result cannot be represented when $float_1$ is zero and $float_2$ is less than or equal to zero. An error might be signalled (condition class: `<range-condition>`) if the result cannot be represented.

16.13.16 `sqrt` *generic function*

Generic Arguments

float `<float>`: A floating point number.

Result

Computes the non-negative square root of *float*. An error is signalled (condition class: `<domain-condition>`) if *float* is negative.

16.14 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 (`<integer>`, `<fpi>`) and floating point numbers (`<float>`, `<double-float>`). Thus, these only are defined at level-0.

Table 4 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>`. Since there are only two concrete number classes at level-0, coercion is simple, namely from `<fpi>` 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 proceeds[3)] (see in particular `binary/`, `binary%` and `binary-mod`).

Table 4 – Level-0 number class hierarchy

```
<number>
    <float>
        <double-float>
    <integer>
        <fpi>
```

16.14.1 `<number>` `<object>` *class*

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

16.14.2 `number?` *function*

Arguments

object: Object to examine.

Result

Returns *object* if it is a number, otherwise `()`.

16.14.3 `<arithmetic-condition>` `<condition>` *condition*

Initialization Options

`operator` *object*: The operator which signalled the condition.

`operand-list` *list*: The operands passed to the operator.

Remarks

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

16.14.4 `<division-by-zero>` `<arithmetic-condition>` *condition*

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

16.14.5 `+` *function*

Arguments

number$_1$ *number*$_2$ ...$_{opt}$: A sequence of numbers.

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.

16.14.6 `-` *function*

Arguments

number$_1$ *number*$_2$ ...$_{opt}$: A non-empty sequence of numbers.

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.

16.14.7 `*` *function*

Arguments

number$_1$ *number*$_2$ ...$_{opt}$: A sequence of numbers.

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.

16.14.8 `/` *function*

Arguments

number$_1$ *number*$_2$ ...$_{opt}$: A non-empty sequence of numbers.

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.

[3)] This behaviour is popularly referred to as *floating point contagion*

16.14.9 `%` *function*

Arguments

number$_1$ *number*$_2$ $\ldots_{opt}$: A non-empty sequence of numbers.

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.

16.14.10 `mod` *function*

Arguments

number$_1$ *number*$_2$ $\ldots_{opt}$: A non-empty sequence of numbers.

Result

Computes the largest integral value not greater than the result of dividing the first argument by its succeeding arguments using the generic function `binary-mod`. Zero arguments is an error. One argument returns *number*$_1$.

16.14.11 `gcd` *function*

Arguments

number$_1$ *number*$_2$ $\ldots_{opt}$: A non-empty sequence of numbers.

Result

Computes the greatest common divisor of *number*$_1$ up to *number*$_n$ using the generic function `binary-gcd`. Zero arguments is an error. One argument returns *number*$_1$.

16.14.12 `lcm` *function*

Arguments

number$_1$ *number*$_2$ $\ldots_{opt}$: A non-empty sequence of numbers.

Result

Computes the least common multiple of *number*$_1$ up to *number*$_n$ using the generic function `binary-lcm`. Zero arguments is an error. One argument returns *number*$_1$.

16.14.13 `abs` *function*

Arguments

number: A number.

Result

Computes the absolute value of *number*.

16.14.14 `zero?` *generic function*

Generic Arguments

number: A number.

Result

Compares *number* with the zero element of the class of *number* using the generic function `binary=`.

16.14.15 `negate` *generic function*

Generic Arguments

number `<number>`: A number.

Result

Computes the additive inverse of *number*.

16.14.16 `signum` *function*

Arguments

number: A number.

Result

Returns *number* if `zero?` applied to *number* is `t`. Otherwise returns the result of converting ± 1 to the class of *number* with the sign of *number*.

16.14.17 `positive?` *function*

Arguments

number: A number.

Result

Compares *number* against the zero element of the class of *number* using the generic function `binary<`.

16.14.18 `negative?` *function*

Arguments

number: A number.

Result

Compares *number* against the zero element of the class of *number* using the generic function `binary<`.

16.14.19 `binary= <number>` *method*

Generic Arguments

number$_1$ `<number>`: A number.

number$_2$ `<number>`: A number.

Result

Returns `t` if *number*$_1$ and *number*$_2$ are numerically equal otherwise `()`;

16.14.20 `binary+` *generic function*

Generic Arguments

number$_1$ <number> : A number.

number$_2$ <number> : A number.

Result

Computes the sum of *number*$_1$ and *number*$_2$.

16.14.21 `binary-` *generic function*

Generic Arguments

number$_1$ <number> : A number.

number$_2$ <number> : A number.

Result

Computes the difference of *number*$_1$ and *number*$_2$.

16.14.22 `binary*` *generic function*

Generic Arguments

number$_1$ <number> : A number.

number$_2$ <number> : A number.

Result

Computes the product of *number*$_1$ and *number*$_2$.

16.14.23 `binary/` *generic function*

Generic Arguments

number$_1$ <number> : A number.

number$_2$ <number> : A number.

Result

Computes the division of *number*$_1$ by *number*$_2$ 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 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>).

16.14.24 `binary%` *generic function*

Generic Arguments

number$_1$ <number> : A number.

number$_2$ <number> : A number.

Result

Computes the value of $number_1 - i * number_2$ expressed as a number of the class of the higher of the classes of the two arguments, for some integer i such that, if *number*$_2$ is non-zero, the result has the same sign as *number*$_1$ and magnitude less then the magnitude of *number*$_2$. If the second argument is zero, the result might be zero or an error might be signalled (condition class: <division-by-zero>).

16.14.25 `binary-mod` *generic function*

Generic Arguments

number$_1$ <number> : A number.

number$_2$ <number> : A number.

Result

Computes the largest integral value not greater than *number*$_1$ *number*$_2$ expressed as a number of the class of the higher of the classes of the two arguments, such that if *number*$_2$ is non-zero, the result has the same sign as *number*$_2$ and magnitude less than *number*$_2$. If the second argument is zero, the result might be zero or an error might be signalled (condition class: <division-by-zero>).

16.14.26 `binary-gcd` *generic function*

Generic Arguments

number$_1$ <number> : A number.

number$_2$ <number> : A number.

Result

Computes the greatest common divisor of *number*$_1$ and *number*$_2$.

16.14.27 `binary-lcm` *generic function*

Generic Arguments

number$_1$ <number> : A number.

number$_2$ <number> : A number.

Result

Computes the lowest common multiple of *number*$_1$ and *number*$_2$.

16.15 Streams

The defined name of this module is `stream`.

The aim of the stream design presented here is an open architecture for programming with streams, which should be applicable when the interface to some object can be characterized by either serial access to, or delivery of, objects.

The two specific objectives are: (i) transfer of objects between a process and disk storage; (ii) transfer of objects between one process and another.

The fundamental purpose of a stream object in the scheme presented here is to provide an interface between two objects through the two functions `read`, for streams from which objects are received, and `write`, for streams to which objects are sent.

16.15.1 Stream classes

16.15.1 `<stream>` `<object>` *class*

This is the root of the stream class hierarchy and also defines the basic stream class.

Initialization Options

`read-action` `<function>`: A function which is called by the `<stream>` `generic-read` `<stream>` method. The accessor for this slot is called `stream-read-action`.

`write-action` `<function>`: A function which is called by the `<stream>` `generic-write` `<stream>` method. The accessor for this slot is called `stream-write-action`.

The following accessor functions are defined for `<stream>`

`stream-lock`: A lock, to be used to allow exclusive access to a stream.

`stream-source`: An object to which the stream is connected and from which input is read.

`stream-sink`: An object to which the stream is connected and to which ouptut is written.

`stream-buffer`: An object which is used to buffer data by some subclasses of `<stream>`. Its default value is `()`.

`stream-buffer-size`: The maximum number of objects that can be stored in *stream-buffer*. Its default value is 0.

The transaction unit of `<stream>` is `<object>`.

16.15.2 `stream?` *function*

Arguments

object, `<object>`: The object to be examined.

Result

Returns *object* if it is a stream, otherwise `()`.

16.15.3 `from-stream` *function*

A constructor function of one argument for `<stream>` which returns a stream whose `stream-read-action` is the given argument.

16.15.4 `to-stream` *function*

A constructor function of one argument for `<stream>` which returns a stream whose `stream-write-action` is the given argument.

16.15.5 `<buffered-stream>` `<stream>` *class*

This class specializes `<stream>` by the use of a buffer which may grow arbitrarily large. The transaction unit of `<buffered-stream>` is `<object>`.

16.15.6 `<fixed-buffered-stream>` `<buffered-stream>` *class*

This class specializes `<buffered-stream>` by placing a bound on the growth of the buffer. The transaction unit of `<fixed-buffered-stream>` is `<object>`.

16.15.7 `<file-stream>` `<fixed-buffered-stream>` *class*

This class specializes `<fixed-buffered-stream>` by providing an interface to data stored in files on disk. The transaction unit of `<file-stream>` is `<character>`. The following additional accessor functions are defined for `<file-stream>`:

`file-stream-filename`: The path identifying the file system object associated with the stream.

`file-stream-mode`: The mode of the connection between the stream and the file system object (usually either read or write).

`file-stream-buffer-position`: A key identifying the current position in the stream's buffer.

16.15.8 `file-stream?` *function*

Arguments

object, `<object>`: The object to be examined.

Result

Returns *object* if it is a `<file-stream>` otherwise `()`.

16.15.9 `<string-stream>` `<buffered-stream>` *class*

The class of the default string stream.

16.15.10 `string-stream?` *function*

Arguments

object, `<object>`: The object to be examined.

Result
Returns *object* if it is a <string-stream> otherwise ().

16.15.2 Stream operators

16.15.11 connect *function*

Arguments

source : The source object from which the stream will read data.

sink : The sink object to which the stream will write data.

options$_{opt}$: An optional argument for specifying implementation-defined options.

Result
The return value is ().

Remarks
Connects *source* to *sink* according to the class-specific behaviours of generic-connect.

16.15.12 generic-connect *generic function*

Generic Arguments

source <object> : The source object from which the stream will read data.

sink <object> : The sink object to which the stream will write data.

options$_{opt}$ <list> : An optional argument for specifying implementation-defined options.

Remarks
Generic form of connect.

16.15.13 generic-connect <stream> *method*

Specialized Arguments

source <stream> : The stream which is to be the source of *sink*.

sink <stream> : The stream which is to be the sink of *source*.

options <list> : A list of implementation-defined options.

Result
The return value is ().

Remarks
Connects the source of *sink* to *source* and the sink of *source* to *sink*.

16.15.14 generic-connect <path> *method*

Specialized Arguments

source <path> : A path name.

sink <file-stream> : The stream via which data will be received from the file named by *path*.

options <list> : A list of implementation-defined options.

Result
The return value is ().

Remarks
Opens the object identified by the path *source* for reading and connects *sink* to it. Hereafter, *sink* may be used for reading data from *sink*, until *sink* is disconnected or reconnected. Implementation-defined options for the opening of files may be specified using the third argument.

See also
open-input-file.

16.15.15 generic-connect <file-stream> *method*

Specialized Arguments

source <file-stream> : The stream via which data will be sent to the file named by *path*.

sink <path> : A path name.

options <list> : A list of implementation-defined options.

Result
The return value is ().

Remarks
Opens the object identifed by the path *sink* for writing and connects *source* to it. Hereafter, *source* may be used for writing data to *sink*, until *source* is disconnected or reconnected. Implementation-defined options for the opening of files may be specified using the third argument.

See also
open-output-file.

16.15.16 reconnect *generic function*

Generic Arguments

s1 <stream> : A stream.

s2 <stream> : A stream.

Result
The return value is ().

Remarks
Transfers the source and sink connections of *s1* to *s2*, leaving *s1* disconnected.

16.15.17 reconnect <stream> *method*

Specialized Arguments

s1 <stream> : A stream.

s2 <stream>: A stream.

Result

The return value is ().

Remarks

Implements the reconnect operation for objects of class <stream>.

16.15.18 disconnect *generic function*

Generic Arguments

s <stream>: A stream.

Result

The return value is ().

Remarks

Disconnects the stream *s* from its source and/or its sink.

16.15.19 disconnect <stream> *method*

Specialized Arguments

s <stream>: A stream.

Result

The return value is ().

Remarks

Implements the diconnect operation for objects of class <stream>.

16.15.20 disconnect <file-stream> *method*

Specialized Arguments

s <file-stream>: A file stream.

Result

The return value is ().

Remarks

Implements the diconnect operation for objects of class <file-stream>. In particular, this involves closing the file associated with the stream *s*.

16.15.3 Stream objects

16.15.21 stdin <file-stream> *instance*

Remarks

The standard input stream, which is a file-stream and whose transaction unit is therefore character. In Posix compliant configurations, this object is initialized from the Posix stdin object. Note that although stdin itself is a constant binding, it may be connected to different files by the reconnect operation.

16.15.22 lispin <stream> *instance*

Remarks

The standard lisp input stream, and its transaction unit is object. This stream is initially connected to stdin (although not necessarily directly), thus a read operation on lispin will case characters to be read from stdin and construct and return an object corresponding to the next lisp expression. Note that although lispin itself is a constant binding, it may be connected to different source streams by the reconnect operation.

16.15.23 stdout <file-stream> *instance*

Remarks

The standard output stream, which is a file-stream and whose transaction unit is therefore character. In Posix compliant configurations, this object is initialized from the Posix stdout object. Note that although stdout itself is a constant binding, it may be connected to different files by the reconnect operation.

16.15.24 stderr <file-stream> *instance*

Remarks

The standard error stream, which is a file-stream and whose transaction unit is therefore character. In Posix compliant configurations, this object is initialized from the Posix stderr object. Note that although stderr itself is a constant binding, it may be connected to different files by the reconnect operation.

16.15.4 Buffer management

16.15.25 fill-buffer *generic function*

Generic Arguments

stream <buffered-stream>: A stream.

Result

The buffer associated with *stream* is refilled from its *source*. Returns a count of the number of items read.

Remarks

This function is guaranteed to be called when an attempt is made to read from a buffered stream whose buffer is either empty, or from which all the items have been read.

16.15.26 fill-buffer <buffered-stream> *method*

Specialized Arguments

stream <buffered-stream>: A stream.

16.15.27 fill-buffer <file-stream> *method*

Specialized Arguments

stream <file-stream>: A stream.

16.15.28 flush-buffer *generic function*

Generic Arguments

stream <buffered-stream>: A stream.

Result

The contents of the buffer associated with *stream* is flushed to its sink. If this operation succeeds, a `t` value is returned, otherwise the result is `()`.

Remarks

This function is guaranteed to be called when an attempt is made to write to a buffered stream whose buffer is full.

16.15.29 `flush-buffer <buffered-stream>` *method*

Specialized Arguments

stream `<buffered-stream>`: A stream.

Result

The contents of the buffer associated with *stream* is flushed to its sink. If this operation succeeds, a `t` value is returned, otherwise the result is `()`.

Remarks

Implements the `flush-buffer` operation for objects of class `<buffered-stream>`.

16.15.30 `flush-buffer <file-stream>` *method*

Specialized Arguments

stream `<file-stream>`: A stream.

Result

The contents of the buffer associated with *stream* is flushed to its sink. If this operation succeeds, a `t` value is returned, otherwise the result is `()`.

Remarks

Implements the `flush-buffer` operation for objects of `<file-stream>`. This method is called both when the buffer is full and after a newline character is written to the buffer.

16.15.31 `<end-of-stream>` `<stream-condition>` *condition*

Initialization Options

`stream` `<stream>`: A stream.

Remarks

Signalled by the default end of stream action, as a consequence of a read operation on *stream*, when it is at end of stream.

See also

`generic-read`.

16.15.32 `end-of-stream` *generic function*

Generic Arguments

stream `<buffered-stream>`: A stream.

Remarks

This function is guaranteed to be called when a read operation encounters the end of *stream* and the `eos-error?` argument to read has a non-`()` value.

16.15.33 `end-of-stream <buffered-stream>` *method*

Specialized Arguments

stream `<buffered-stream>`: A stream.

Remarks

Signals the end of stream condition.

16.15.34 `end-of-stream <file-stream>` *method*

Specialized Arguments

stream `<file-stream>`: A stream.

Remarks

Disconnects *stream* and signals the end of stream condition.

16.15.5 Reading from streams

16.15.35 `<read-error>` `<condition>` *condition*

Generic Arguments

stream `<stream>`: A stream.

Remarks

Signalled by a `read` operation which fails in some manner other than when it is at end of stream.

16.15.36 `read` *function*

Arguments

stream$_{opt}$: A stream.

eos-error?$_{opt}$: A boolean.

eos-value$_{opt}$: Value to be returned to indicate end of stream.

Result

That of calling `generic-read` with the arguments supplied or defaulted as described.

Remarks

The *stream* defaults to `lispin`, *eos-error?* defaults to `()` and *eos-value* defaults to `eos-default-value`.

16.15.37 `generic-read` *generic function*

Generic Arguments

stream `<stream>`: A stream.

eos-error? `<object>`: A boolean.

eos-value `<object>`: Value to be returned to indicate end of stream.

Result

The next transaction unit from *stream*.

Remarks

If the end of *stream* is encountered and the value of *eos-error?* is `()`, the result is *eos-value*. If the end of *stream* is encountered and the value of `eos-error?` is non-`()`, the function `end-of-stream <stream>` is called with the argument *stream*.

16.15.38 `generic-read <stream>` *method*

Specialized Arguments

stream `<stream>`: A stream.

eos-error? `<object>`: A boolean.

eos-value `<object>`: Value to be returned to indicate end of stream.

Result

That of calling the *read-action* of stream with the arguments *stream*, *eos-error?* and *eos-value*. Returns `t`.

Remarks

Implements the `generic-read` operation for objects of class `<stream>`.

16.15.39 `generic-read <buffered-stream>` *method*

Specialized Arguments

stream `<buffered-stream>`: A buffered stream.

eos-error? `<object>`: A boolean.

eos-value `<object>`: Value to be returned to indicate end of stream.

Result

The next object stored in the stream buffer. If the buffer is empty, the function `fill-buffer` is called. If the refilling operation did not succeed, the end of stream action is carried out as described under `generic-read`. Returns `t`.

Remarks

Implements the `generic-read` operation for objects of class `<buffered-stream>`.

16.15.40 `generic-read <file-stream>` *method*

Specialized Arguments

stream `<file-stream>`: A file stream.

eos-error? `<object>`: A boolean.

eos-value `<object>`: Value to be returned to indicate end of stream.

Result

The next object stored in the stream buffer. If the buffer is empty, the function `fill-buffer` is called. If the refilling operation did not succeed, the end of stream action is carried out as described under `generic-read`. Returns `t`.

Remarks

Implements the `generic-read` operation for objects of class `<file-stream>`.

16.15.6 Writing to streams

16.15.41 `generic-write` *generic function*

Generic Arguments

object `<object>`: An object to be written to *stream*.

stream `<stream>`: Stream to which *object* is to be written.

Result

Returns *object*.

Remarks

Outputs the external representation of *object* on the output stream *stream*.

See also

The following `generic-write` methods are defined: `generic-write <character>`, `generic-write <symbol>`, `generic-write <keyword>`, `generic-write <fpi>`, `generic-write <double-float>`, `generic-write <null>`, `generic-write <cons>`, `generic-write <list>`, `generic-write <string>`, `generic-write <vector>`, `generic-write <stream>`, `generic-write <buffered-stream>` and `generic-write <file-stream>`.

16.15.42 `generic-write <stream>` *method*

Specialized Arguments

object `<object>`: An object to be written to *stream*.

stream `<stream>`: Stream to which *object* is to be written.

16.15.43 `generic-write <buffered-stream>` *method*

Specialized Arguments

object `<object>`: An object to be written to *stream*.

stream `<buffered-stream>`: Stream to which *object* is to be written.

16.15.44 `generic-write <file-stream>` *method*

Specialized Arguments

object `<object>`: An object to be written to *stream*.

stream `<file-stream>`: Stream to which *object* is to be written.

16.15.45 `swrite` *function*

Arguments

stream: Stream to which *object* is to be written.

object: An object to be written to *stream*.

Result

Returns *stream*.

Remarks

Outputs the external representation of *object* on the output stream *stream* using `generic-write`.

See also

`generic-write`.

16.15.46 `write` *function*

Arguments

object: An object to be written to *stream*.

Result

Returns `stdout`.

Remarks

Outputs the external representation of *object* on `stdout` using `generic-write`.

See also

`swrite`, `generic-write`.

16.15.7 Additional functions

16.15.47 `read-line` *function*

Arguments

stream: A stream.

eos-error?$_{opt}$: A boolean.

eos-value$_{opt}$: Value to be returned to indicate end of stream.

Result

A string.

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>`).

16.15.48 `generic-print` *generic function*

Generic Arguments

object `<object>`: An object to be output on *stream*.

stream `<stream>`: A character stream on which *object* is to be output.

Result

Returns *object*.

Remarks

Outputs the external representation of *object* on the output stream *stream*.

See also

`prin`. The following `generic-write` methods are defined: `generic-write <character>`, `generic-write <symbol>`, `generic-write <keyword>`, `generic-write <fpi>`, `generic-write <double-float>`, `generic-write <null>`, `generic-write <cons>`, `generic-write <list>`, `generic-write <string>` and `generic-write <vector>`.

16.15.49 `sprint` *function*

Arguments

stream: A character stream on which *object* is to be output.

object$_1$ *object*$_2$ ...$_{opt}$: A sequence of objects to be output on *stream*.

Result

Returns *stream*.

Remarks

Outputs the external representation of *object*$_1$ *object*$_2$ ... on the output stream *stream* using `generic-print` for each object.

See also

`generic-print`.

16.15.50 `print` *function*

Arguments

object$_1$ *object*$_2$ ...$_{opt}$: A sequence of objects to be output on `stdout`.

Result

Returns `stdout`.

Remarks

Outputs the external representation of *object*$_1$ *object*$_2$ ... on the output stream `stdout` using `sprint` for each object.

See also

`sprint` and `generic-print`.

16.15.51 `sflush` *function*

Arguments

stream: A stream to flush.

Result

Returns *stream*.

Remarks

`sflush` causes any buffered data for the stream to be written to the stream. The stream remains open.

16.15.52 `flush` *function*

Result

Returns `stdout`.

Remarks

`flush` causes any buffered data for `stdout` to be written to `stdout`.

See also

`sflush`.

16.15.53 `sprin-char` *function*

Arguments

stream: A stream.

char: Character to be written to *stream*.

$times_{opt}$: Integer count.

Result

Outputs *char* on *stream*. The optional count *times* defaults to 1.

16.15.54 `prin-char` *function*

Arguments

char: Character to be written to `stdout`.

$times_{opt}$: Integer count.

Result

Outputs *char* on `stdout`. The optional count *times* defaults to 1.

See also

`sprin-char`.

16.15.55 `sread` *function*

Arguments

stream: A stream.

$eos\text{-}error?_{opt}$: A boolean.

$eos\text{-}value_{opt}$: Value to be returned to indicate end of stream.

16.15.8 Convenience forms

16.15.56 `open-input-file` *function*

Arguments

path: A path identifying a file system object.

Result

Allocates and returns a new `<file-stream>` object whose source is connected to the file system object identified by *path*.

16.15.57 `open-output-file` *function*

Arguments

path: A path identifying a file system object.

Result

Allocates and returns a new `<file-stream>` object whose sink is connected to the file system object identified by *path*.

16.15.58 `with-input-file` *special operator*

16.15.58.1 Syntax

```
with-input-file-form:  → <object>
      ( with-input-file path
        body )
path:
      string
```

16.15.59 `with-output-file` *special operator*

16.15.59.1 Syntax

```
with-output-file-form:  → <object>
      ( with-output-file path
        body )
```

16.15.60 `with-source` *special operator*

16.15.60.1 Syntax

```
with-source-form:  → <object>
      ( with-source ( identifier form )
        body )
```

16.15.61 `with-sink` *special operator*

16.15.61.1 Syntax

```
with-sink-form:  → <object>
      ( with-sink ( identifier form )
        body )
```

16.16 Strings

The defined name of this module is **string**. See also section 16.2 (collections) for further operations on strings.

16.16.1 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 § 9.1 and § 16.1) are included in strings using the character digrams defined in table 16.1. To allow arbitrary characters to appear in strings, the hex-insertion digram is followed by an integer denoting the position of the character in the current character set as for characters (see § 9.1). The syntax for the external representation of strings is defined in syntax table 16.16.1.1 below:

16.16.1.1 Syntax

```
string:
      " string-constituent* "
string-constituent:
      normal-string-constituent
      digram-string-constituent
      numeric string constituent
normal-string-constituent:
      level-0-character other than " or \
digram-string-constituent: one of
      \a \b \d \f \l \n \r \t \v \" \\
numeric-string-constituent:
      \x hexadecimal-digit
      \x hexadecimal-digit hexadecimal-digit
      \x hexadecimal-digit hexadecimal-digit
         hexadecimal-digit
      \x hexadecimal-digit hexadecimal-digit
         hexadecimal-digit hexadecimal-digit
```

Some examples of string literals appear in table 1.

Example 1 – Examples of string literals

Example	Contents
"a\nb"	#\a, #\n and #\b
"c\\"	#\c and #\\
"\x1 "	#\x1 followed by #\space
"\xabcde"	#\xabcd followed by #\e
"\x1\x2"	#\x1 followed by #\x2
"\x12+"	#\x12 followed by #\+
"\xabcg"	#\xabc followed by #\g
"\x00abc"	#\xab followed by #\c

NOTE 1 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 implementation-defined extensions which support more than one character set.

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 #\n, #\a, #\\ 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 1. The function **prin** outputs the interpretation of the characters according to the definitions in section 16.1 without the delimiting quotation marks.

16.16.2 <string> <character-sequence> *class*

The class of all instances of <string>.

Initialization Options

size <fpi>: The number of characters in the string. Strings are zero-based and thus the maximum index is *size-1*. If not specified the *size* is zero.

fill-value: <character>: A character with which to initialize the string. The default fill character is #\x0.

Examples

```
(make <string>)                ⇒  ""
(make <string> size: 2)        ⇒  "\x0000\x0000"
(make <string> size: 3         ⇒  "aaa"
  fill-value: #\a)
```

16.16.3 string? *function*

Arguments

object: Object to examine.

Result

Returns *object* if it is a string, otherwise ().

16.16.4 (converter <symbol>) *converter*

Specialized Arguments

string <string>: A string to be converted to a symbol.

Result

If the result of **symbol-exists?** 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.

16.16.5 binary= <string> *method*

Specialized Arguments

$string_1$ <string>: A string.

$string_2$ <string>: A string.

Result

If the size of $string_1$ is the same (under =) as that of $string_2$, and the result of the conjunction of the pairwise application of **binary= <character>** to the elements of the arguments is **t** the result is $string_1$. If not the result is ().

16.16.6 deep-copy <string> *method*

Specialized Arguments

string <string>: A string.

Result

Constructs and returns a copy of *string* in which each element is eql to the corresponding element in *string*.

16.16.7 shallow-copy <string> *method*

Specialized Arguments

string <string>: A string.

Result

Constructs and returns a copy of *string* in which each element is eql to the corresponding element in *string*.

16.16.8 binary< <string> *method*

Specialized Arguments

string$_1$ <string>: A string.

string$_2$ <string>: A string.

Result

If the second argument is longer than the first, the result is (). Otherwise, if the sequence of characters in *string*$_1$ is pairwise less than that in *string*$_2$ according to binary< <character> 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.

Examples

```
(< "a" "b")    ⇒ t
(< "b" "a")    ⇒ ()
(< "a" "a")    ⇒ ()
(< "a" "ab")   ⇒ t
(< "ab" "a")   ⇒ ()
(< "A" "B")    ⇒ t
(< "0" "1")    ⇒ t
(< "a1" "a2")  ⇒ t
(< "a1" "bb")  ⇒ t
(< "a1" "ab")  ⇒ undefined
```

See also

Method binary< <character> (16.3) for characters (16.1).

16.16.9 as-lowercase <string> *method*

Specialized Arguments

string <string>: A string.

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*.

16.16.10 as-uppercase <string> *method*

Specialized Arguments

string <string>: A string.

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*.

16.16.11 generic-print <string> *method*

Specialized Arguments

string <string>: String to be ouptut on *stream*.

stream <stream>: Stream on which *string* is to be ouptut.

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.

16.16.12 generic-write <string> *method*

Specialized Arguments

string <string>: String to be ouptut on *stream*.

stream <stream>: Stream on which *string* is to be ouptut.

Result

The string *string*. Output external representation of *string* on *stream* as described in the introduction to this section, replacing single characters with escape sequences if necessary. Opening and closing quotation marks are output.

16.17 Symbols

The defined name of this module is `symbol`.

16.17.1 `symbol` *syntax*

A *symbol* is a literal *identifier* and hence has the same syntax 9.3.0.3:

16.17.1.1 Syntax

symbol:
    identifier

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, *i.e.* a *symbol*, 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*:

`|123|`, `\123`, `|1|23`, `||123`, `||||123`

which will be output by the function `write` as `|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: `|a(b|` and `abc.def` are two *symbols* as output by `write` such that `read` can read them as two *symbols*. If output by `prin`, the escapes necessary to re-`read` the *symbol* will not be included. Thus, taking the same examples, `prin` outputs `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`.

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*.

16.17.2 `<symbol>` `<name>` *class*

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

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 `||`.

16.17.3 `symbol?` *function*

Arguments

object: Object to examine.

Result

Returns *object* if it is a symbol.

16.17.4 `gensym` *function*

Arguments

string$_{opt}$: A string contain characters to be prepended to the name of the new symbol.

Result

Makes a new symbol whose name, by default, begins with the character `#\g` and the remaining characters are generated by an implementation-defined mechanism. Optionally, an alternative prefix string for the name may be specified. It is guaranteed that the resulting symbol did not exist before the call to `gensym`.

16.17.5 `symbol-name` *function*

Arguments

symbol: A symbol.

Result

Returns a *string* which is `binary=` `<string>` to that given as the argument to the call to `make` which created *symbol*. It is an error to modify this string.

16.17.6 `symbol-exists?` *function*

Arguments

string: A string containing the characters to be used to determine the existence of a symbol with that name.

Result

Returns the symbol whose name is *string* if that symbol has already been constructed by `make`. Otherwise, returns `()`.

16.17.7 `generic-print` `<symbol>` *method*

Specialized Arguments

symbol `<symbol>`: The symbol to be output on *stream*.

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

Result

The symbol supplied as the first argument.

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.

16.17.8 `generic-write` `<symbol>` *method*

Specialized Arguments

symbol `<symbol>`: The symbol to be output on *stream*.

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

Result

The symbol supplied as the first argument.

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.

Examples

```
(write (make <symbol> 'string "abc"))  ⇒  abc
(write (make <symbol> 'string "a c"))  ⇒  |a c|
(write (make <symbol> 'string ").("))  ⇒  |).(|
```

16.17.9 `(converter <string>)` *converter*

Specialized Arguments

symbol `<symbol>`: A symbol to be converted to a string.

Result

A string.

Remarks

This function is the same as `symbol-name`. It is defined for the sake of symmetry.

16.18 Tables

The defined name of this module is `table`. See also section 16.2 (collections) for further operations on tables.

16.18.1 `<table>` `<collection>` *class*

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

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 `t` for any two objects, then the hash function must return the same key when applied to those two objects. The default is an implementation defined function which satisfies these conditions.

16.18.2 `table?` *function*

Arguments

object: Object to examine.

Result

Returns *object* if it is a table, otherwise `()`.

16.18.3 `clear-table` *function*

Arguments

table: A table.

Result

An empty table.

Remarks

All entries in *table* are deleted. The result is `eq` to the argument, which is to say that the argument is modified.

16.18.4 `<hash-table>` `<table>` *class*

Place holder for `<hash-table>` class.

16.19 Vectors

The defined name of this module is `vector`. See also section 16.2 (collections) for further operations on vectors.

16.19.1 `vector` *syntax*

A vector is written as `#(`$obj_1 \ldots obj_n$`)`. For example: `#(1 2 3)` is a vector of three elements, the integers `1`, `2` and `3`. The representations of obj_i are determined by the external representations defined in other sections of this definition (see `<character>` (16.1), `<fpi>` (16.9), `<float>` (16.7), `<list>` (16.12), `<string>` (16.16) and `<symbol>` (16.17), as well as instances of `<vector>` itself. The syntax for the external representation of vectors is defined below.

16.19.1.1 Syntax

vector:
    `#(` *object** `)`

16.19.2 `<vector>` `<sequence>` *class*

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

Initialization Options

`size`: `<fpi>`: 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 `()`.

Examples

```
(make <vector>)            ⇒  #()
(make <vector> size: 2)    ⇒  #(() ())
(make <vector> size: 3     ⇒  #(#\a #\a #\a)
  fill-value: #\a)
```

16.19.3 `vector?` *function*

Arguments

object: Object to examine.

Result

Returns *object* if it is a vector, otherwise `()`.

16.19.4 `maximum-vector-index` `<integer>` *constant*

Remarks

This is an implementation-defined constant. A conforming processor must support a maximum vector index of at least 32767.

16.19.5 `binary=` `<vector>` *method*

Specialized Arguments

$vector_1$ `<vector>`: A vector.

$vector_2$ `<vector>`: A vector.

Result

If the size of $vector_1$ is the same (under `=`) as that of $vector_2$, and the result of the conjunction of the pairwise application of `binary=` to the elements of the arguments `t` the result is $vector_1$. If not the result is `()`.

16.19.6 `deep-copy` `<vector>` *method*

Specialized Arguments

vector `<vector>`: A vector.

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*.

16.19.7 `shallow-copy` `<vector>` *method*

Specialized Arguments

vector `<vector>`: A vector.

Result

Constructs and returns a copy of *vector* in which each element is `eql` to the corresponding element in *vector*.

16.19.8 `generic-print` `<vector>` *method*

Specialized Arguments

vector `<vector>`: A vector to be ouptut on stream.

stream `<stream>`: A stream on which the representation is to be output.

Result

The vector supplied as the first argument.

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.

16.19.9 `generic-write` `<vector>` *method*

Specialized Arguments

vector `<vector>`: A vector to be ouptut on stream.

stream `<stream>`: A stream on which the representation is to be output.

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.

16.20 Syntax of Level-0 objects

This section repeats the syntax for reading and writing of the various classes defined in §16.

object:
    literal
    list §16.12
    symbol §16.17
literal:
    boolean
    character §16.1
    float §16.7
    integer §16.10
    string §16.16
    vector §16.19
boolean:
    true
    false
true:
    `t`
    object `not ()`
false:
    `()`
    `nil`
character:
    literal-character-token
    special-character-token
    numeric-character-token
literal-character-token:
    `#\`*letter*
    `#\`*decimal-digit*
    `#\`*other-character*
    `#\`*special-character*
special-character-token:
    `#\\a`
    `#\\b`
    `#\\d`
    `#\\f`
    `#\\l`
    `#\\n`
    `#\\r`
    `#\\t`
    `#\\v`
    `#\\"`
    `#\\\`
numeric-character-token:
    `#\\x` *hexadecimal-digit hexadecimal-digit*
        hexadecimal-digit hexadecimal-digit
float:
    $sign_{opt}$ *unsigned-float* $exponent_{opt}$
unsigned-float:
    float-format-1
    float-format-2
    float-format-3
float-format-1:
    decimal-integer `.`
float-format-2:
    `.` *decimal-integer*
float-format-3:
    float-format-1 decimal-integer
exponent:
    double-exponent
double-exponent:
    `d` $sign_{opt}$ *decimal-integer*
    `D` $sign_{opt}$ *decimal-integer*

integer:
    $sign_{opt}$ *unsigned-integer*
sign: `one of`
    `+ -`
unsigned-integer:
    binary-integer
    octal-integer
    decimal-integer
    hexadecimal-integer
    specified-base-integer
binary-integer:
    `#b` $binary\text{-}digit^{+}$
binary-digit: `one of`
    `0 1`
octal-integer:
    `#o` $octal\text{-}digit^{+}$
octal-digit: `one of`
    `0 1 2 3 4 5 6 7`
decimal-integer:
    $decimal\text{-}digit^{+}$
hexadecimal-integer:
    `#x` $hexadecimal\text{-}digit^{+}$
hexadecimal-digit:
    decimal-digit
    hex-lower-letter
    hex-upper-letter
hex-lower-letter: `one of`
    `a b c d e f`
hex-upper-letter: `one of`
    `A B C D E F`
specified-base-integer:
    `#` *base-specification* `r`
    specified-base-digit
    $specified\text{-}base\text{-}digit^{*}$
base-specification:
    { `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` }
    { `1` | `2` } *decimal-digit*
    `3` { `0` | `1` | `2` | `3` | `4` | `5` | `6` }
specified-base-digit:
    decimal-digit
    letter
keyword:
    identifier:
null:
    `()`
pair:
    `(` *object* `.` *object* `)`
list:
    empty-list
    proper-list
    improper-list
empty-list:
    `()`
proper-list:
    `(` $object^{+}$ `)`
improper-list:
    `(` $object^{+}$ `.` *object* `)`

string:
    " *string-constituent** "
string-constituent:
    normal-string-constituent
    digram-string-constituent
    numeric string constituent
normal-string-constituent:
    level-0-character `other than " or \`
digram-string-constituent: `one of`
    `\a \b \d \f \l \n \r \t \v \" \\`
numeric-string-constituent:
    `\x` *hexadecimal-digit*
    `\x` *hexadecimal-digit hexadecimal-digit*
    `\x` *hexadecimal-digit hexadecimal-digit hexadecimal-digit*
    `\x` *hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit*
symbol:
    identifier

vector:
    `#(` *object** `)`

17 Programming Language EuLisp, Level-1

This section describes the additions features in EuLisp level-1 including the reflective aspects of the object system and how to program the metaobject protocol and support for dynamic variable and related control forms.

17.1 Modules

defmodule-1-form:
    `(` **`defmodule`** *module-name module-directives level-1-module-form** `)`
level-1-module-form:
    level-0-module-form
    level-1-form
    defining-1-form
level-1-form:
    level-0-form
    special-1-form
form:
    level-1-form
special-form:
    special-1-form
defining-1-form:
    defclass-1-form
    defgeneric-1-form
    defglobal-form
special-1-form:
    generic-lambda-form
    method-lambda-form
    defmethod-form
    method-function-lambda-form
    catch-form
    throw-form

17.2 Classes and Objects

17.2.1 `defclass` *defining operator*

17.2.1.1 Syntax

defclass-1-form:
    `(` **`defclass`** *class-name superclass-names*
      `(` *slot-1** `)`
      *class-option-1** `)`
superclass-names:
    `()`
    `superclass-name`
    `(` *superclass-name** `)`
slot-1:
    slot
    `(` *slot-name slot-option-1** `)`
slot-option-1:
    slot-option
    identifier level-1-form
class-option-1
    class-option
    `class``:` *class-name*
    identifier level-1-form

Arguments

superclass-names: Either a symbol or a list of symbols naming bindings of the class/classes to be used as the superclass/superclasses of the new class (multiple inheritance). If this list of superclass names is empty, the superclass is `<object>`. This is different from `defclass` at level-0, where only one superclass may be specified.

slot-1: A list of slot specifications (see below), comprising either a *slot-name* or a list of a *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-1: A key and a value (see below). One of the class options (`<class>`) allows the specification of the class of the class being defined.

Remarks

This defining form defines a new class. The resulting class will be bound to *class-name*. All the slot and class options are exactly the same as for `defclass` (11.3).

The *slot-option-1*s are interpreted as follows:

identifier level-1-form: 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 keywords are defined which are not specified by particular `defclass` slot options.

The *class-option-1*s are interpreted as follows:

`class` *class-name*: 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 level-1-form: 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.

17.3 Generic Functions

17.3.1 generic-lambda *special operator*

17.3.1.1 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-form
    level-0-init-option
level-1-method-description:
    (*method-init-option**
      specialized-lambda-list
      body)
method-init-option:
    class *class-name*
    identifier level-1-form

Arguments

gf-lambda-list: As level-0. See section 11.4.

*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 keywords. 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 is, by default, <method>. The value is passed to make as the first argument. The symbol and the value are not passed as keywords to make.

identifier expression: The symbol named by *identifier* and the value of *expression* are passed to make creating a new method as keywords. 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.

Examples

In the following example an anonymous version of gf-1 (see defgeneric) is defined. In 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-a-key-a method-class-a-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>))0
          ...)
)
```

See also

defgeneric.

17.3.2 defgeneric *defining operator*

17.3.2.1 Syntax

defgeneric-1-form:
    (defgeneric *gf-name gf-lambda-list*
      level-1-init-option)

Arguments

gf name: As level-0. See section 11.4.

gf lambda list: As level-0. See section 11.4.

*init option**: As for generic-lambda, defined above. below.

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 section 11.4.5.2 except that *level 1 init option* replaces *level 0 init option*.

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 11.4) 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>

  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-a-key-a method-class-a-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>))
          ...)
)
```

17.4 Methods

17.4.1 `method-lambda` *special operator*

17.4.1.1 Syntax

method-lambda-form: → <function>
  (method-lambda
    *method-init-option**
    specialized-lambda-list
    body)

Arguments

method init option: A quoted symbol followed by an expression.

specialized lambda list: As defined under generic-lambda.

form: An expression.

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 keywords <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>.

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.

17.4.2 `defmethod` *defining operator*

17.4.2.1 Syntax

defmethod-1-form:
  (defmethod *gf-locator*
    *method-init-option**
    specialized-lambda-list
    body)

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.

17.4.3 `method-function-lambda` *special operator*

Arguments

lambda-list: A lambda list

*form**: A sequence of forms.

This syntax operator 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 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. Only functions created using this syntax operator 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.

17.4.4 `call-method` *function*

Arguments

method: A method.

next-methods: A list of methods.

*arg**: A sequence of expressions.

This function calls the method *method* with arguments *args*. The argument *next-methods* is a list of methods which are used as the applicable method list for *args*; 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*.

17.4.5 `apply-method` *function*

Arguments

method: A method.

next-methods: A list of methods.

$form_1 \ldots form_{n-1}$: A sequence of expressions.

$form_n$: An expression.

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`.

17.5 Object Introspection

The only reflective capability which every object possesses is the ability to find its class.

17.5.1 `class-of` *function*

Arguments

object: An object.

Result

The class of the object.

Remarks

The function `class-of` can take any LISP object as argument and returns an instance of `<class>` representing the class of that entity.

17.6 Class Introspection

Standard classes are not redefinable and support single inheritance only. General multiple 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 metaclasses defined at level-1 are shown in table 5.

Table 5 – Level-1 metaclass hierarchy

$\mathcal{A}$ `<object>` See level-0 table 1
  $\mathcal{A}$ `<class>`
    $\mathcal{C}$ `<simple-class>`
    $\mathcal{C}$ `<function-class>`

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) keywords.

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 default-function, 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 keyword, 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 6.

Table 6 – Level-1 class hierarchy

$\mathcal{A}$ `<object>` See level-0 table 1
  $\mathcal{A}$ `<slot>`
    $\mathcal{C}$ `<local-slot>`
  $\mathcal{A}$ `<function>`
    $\mathcal{A}$ `<generic-function>`
      $\mathcal{C}$ `<simple-generic-function>`
  $\mathcal{A}$ `<method>`
    $\mathcal{C}$ `<simple-method>`

17.6.1 `<metaclass>` *class*

Place holder for `<metaclass>`.

17.6.2 `<simple-class>` `<class>` *class*

Place holder for `<simple-class>`.

17.6.3 `<function-class>` `<class>` *class*

Place holder for `<function-class>`.

17.6.4 `class-name` *function*

Arguments

class: A class.

Result

Returns a *string* which is `binary=` `<string>` to that given as the argument to the call to `defclass` which created *class*. It is an error to modify this string.

17.6.5 `class-precedence-list` *function*

Arguments

class: A class.

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`.

Remarks

The class precedence list is used to control the inheritance of slots and methods.

17.6.6 `class-slots` *function*

Arguments

class: A class.

Result

A list of slots, one for each of the slots of an instance of *class*.

Remarks

The slots determine the instance size (number of slots) and the slot access.

17.6.7 `class-keywords` *function*

Arguments

class: A class.

Result

A list of symbols, which can be used as legal keywords to initialize instances of the class.

Remarks

The keywords correspond to the keywords specified in the `keyword` slot-option or the `keywords` class-option when the class and its superclasses were defined.

17.7 Slot Introspection

17.7.1 `<slot>` *class*

The abstract class of all slot descriptions.

17.7.2 `<local-slot>` `<slot>` *class*

The class of all local slot descriptions.

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.

`default-function` *function*: The function to compute the initial value in the absence of a supplied value.

`keyword` *symbol*: The key to access a supplied initial value.

The default value for all initoptions is `()`.

17.7.3 `slot-name` *function*

Arguments

slot: A slot description.

Result

The symbol which was used to name the slot when the class, of which the slot is part, was defined.

Remarks

The slot description name is used to identify a slot description in a class. It has no effect on bindings.

17.7.4 `slot-default-function` *function*

Arguments

slot: A slot description.

Result

A function of no arguments that is used to compute the initial value of the slot in the absence of a supplied value.

17.7.5 `slot-slot-reader` *function*

Arguments

slot: A slot description.

Result

A function of one argument that returns the value of the slot in that argument.

17.7.6 `slot-slot-writer` *function*

Arguments

slot: A slot description.

Result

A function of two arguments that installs the second argument as the value of the slot in the first argument.

17.8 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 sub-domain.

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.

17.8.1 `generic-function-domain` *function*

Arguments

generic-function: A generic function.

Result

A list of classes.

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.

17.8.2 `generic-function-method-class` *function*

Arguments

generic-function: A generic function.

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 `defmethod`, `method-lambda`, or by using the `method` generic function option in a `defgeneric` or `generic-lambda`, it will be an instance of this class by default.

17.8.3 `generic-function-methods` *function*

Arguments

generic-function: A generic function.

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.

17.8.4 `generic-function-method-lookup-function` *function*

Arguments

generic-function: A generic function.

Result

A function.

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 `compute-method-lookup-function`.

17.8.5 `generic-function-discriminating-function` *function*

Arguments

generic-function: A generic function.

Result

A function.

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 `compute-discriminating-function`.

17.9 Method Introspection

The minimal information associated with a method metaobject is:

a) The domain, which is a list of classes.

b) The function comprising the code of the method.

c) The generic function to which the method has been added, or () if it is attached to no generic function.

The metaobject classes for generic functions defined at level-1 are shown in table 6.

17.9.1 `<method>` *class*

Place holder for `<method>`.

17.9.2 `<simple-method>` `<method>` *class*

Place holder for `<method-class>`.

17.9.3 `method-domain` *function*

Arguments

method: A method.

Result

A list of classes defining the domain of a method.

17.9.4 `method-function` *function*

Arguments

method: A method.

Result

This function returns a function which implements the method. The returned function which is called when *method* is called, either by calling the generic function with appropriate arguments, through a `call-next-method`, or by using `call-method`. A method metaobject itself cannot be applied or called as a function.

17.9.5 `(setter method-function)` *setter*

Arguments

method: A method.

function: A function.

Result

This function sets the function which implements the method.

17.9.6 `method-generic-function` *function*

Arguments

method: A method.

Result

This function returns the generic function to which *method* is attached; if *method* is not attached to a generic function, it returns ().

17.10 Class Initialization

17.10.1 `initialize <class>` *method*

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-slots` *list*: List of direct slot specifications.
- `direct-keywords` *list*: List of direct keywords.

Result

The initialized class.

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) keywords

   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 7 Note that `compute-keywords` is called by the default `initialize` method with all direct keywords as the second argument: those specified as slot option and those specified as class option.

17.10.2 `compute-predicate` *generic function*

Generic Arguments

class `<class>`: A class.

Result

Computes and returns a function of one argument, which is a predicate function for *class*.

17.10.3 `compute-predicate <class>` *method*

Table 7 – Initialization Call Structure

```
compatible-superclasses?
        cl direct-superclasses → boolean
  compatible-superclass?
          cl superclass → boolean
compute-class-precedence-list
        cl direct-superclasses → (cl*)
compute-inherited-keywords
        cl direct-superclasses → ((keyword*)*)
compute-keywords
        cl direct-keywords inherited-keywords
        → (keyword*)
compute-inherited-slots
        cl direct-superclasses → ((sd*)*)
compute-slots
        cl slot-specs inherited-sds → (sd*)
  either
    compute-defined-slot
            cl slot-spec → sd
      compute-defined-slot-description-class
              cl slot-spec → sd-class
  or
    compute-specialized-slot
            cl inherited-sds slot-spec → sd
      compute-specialized-slot-class
              cl inherited-sds slot-spec
              → sd-class
compute-instance-size
        cl effective-sds → integer
compute-and-ensure-slot-accessors
        cl effective-sds inherited-sds → (sd*)
  compute-slot-reader
          cl sd effective-sds → function
  compute-slot-writer
          cl sd effective-sds → function
  ensure-slot-reader
          cl sd effective-sds reader → function
    compute-primitive-reader-using-slot
            sd cl effective-sds → function
      compute-primitive-reader-using-class
              cl sd effective-sds → function
  ensure-slot-writer
          cl sd effective-sds writer → function
    compute-primitive-writer-using-slot
            sd cl effective-sds → function
      compute-primitive-writer-using-class
              cl sd effective-sds → function
```

Specialized Arguments

class `<class>`: A class.

Result

Computes and returns a function of one argument, which returns `t` when applied to direct or indirect instances of *class* and `()` otherwise.

17.10.4 `compute-constructor` *generic function*

Generic Arguments

class `<class>`: A class.

parameters `<list>`: The argument list of the function being created.

Result

Computes and returns a constructor function for *class*.

17.10.5 `compute-constructor <class>` *method*

Specialized Arguments

class `<class>`: A class.

parameters `<list>`: The argument list of the function being created.

Result

Computes and returns a constructor function, which returns a new instance of *class*.

17.10.6 `allocate` *generic function*

Generic Arguments

class `<class>`: A class.

initlist `<list>`: A list of initialization arguments.

Result

An instance of the first argument.

Remarks

Creates an instance of the first argument. Users may define new methods for new metaclasses.

17.10.7 `allocate <class>` *method*

Specialized Arguments

class `<class>`: A class.

initlist `<list>`: A list of initialization arguments.

Result

An instance of the first argument.

Remarks

The default method creates a new uninitialized instance of the first argument. The initlist is not used by this `allocate` method.

17.11 Slot Description Initialization

17.11.1 `initialize <slot>` *method*

Specialized Arguments

slot `<slot>`: A slot description.

initlist `<list>`: A list of initialization options as follows:

- `name` *symbol*: The name of the slot.
- `default-function` *function*: A function.
- `keyword` *symbol*: A symbol.
- `reader` *function*: A slot reader function.

`writer` *function*: A slot writer function.

Result

The initialized slot description.

17.12 Generic Function Initialization

17.12.1 `initialize <generic-function>` *method*

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.

Result

The initialized generic function.

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`*

17.13 Method Initialization

17.13.1 `initialize <method>` *method*

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.

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*.

17.14 Inheritance Protocol

17.14.1 `compatible-superclasses?` *generic function*

Generic Arguments

class `<class>`: A class.

direct-superclasses `<list>`: A list of potential direct superclasses of *class*.

Result

Returns *t* if *class* is compatible with *direct-superclasses*, otherwise `()`.

17.14.2 `compatible-superclasses? <class>` *method*

Specialized Arguments

class `<class>`: A class.

direct-superclasses `<list>`: A list of potential direct superclasses.

Result

Returns the result of calling `compatible-superclass?` on *class* and the first element of the *direct-superclasses* (single inheritance assumption).

17.14.3 `compatible-superclass?` *generic function*

Generic Arguments

subclass `<class>`: A class.

superclass `<class>`: A potential direct superclass.

Result

Returns `t` if *subclass* is compatible with *superclass*, otherwise `()`.

17.14.4 `compatible-superclass? <class>` *method*

Specialized Arguments

subclass `<class>`: A class.

superclass `<class>`: A potential direct superclass.

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 `()` if *superclass* is `<metaclass>`.

17.14.5 `compatible-superclass? <class>` *method*

Specialized Arguments

subclass `<class>`: A class.

superclass `<abstract-class>`: A potential direct superclass.

Result

Always returns `t`.

17.14.6 `compatible-superclass? <abstract-class>` *method*

Specialized Arguments

subclass `<abstract-class>`: A class.

superclass `<class>`: A potential direct superclass.

Result

Always returns `()`.

17.14.7 `compatible-superclass? <abstract-class>` *method*

Specialized Arguments

subclass `<abstract-class>`: A class.

superclass `<abstract-class>`: A potential direct superclass.

Result

Always returns `t`.

17.14.8 `compute-class-precedence-list` *generic function*

Generic Arguments

class `<class>`: Class being defined.

direct-superclasses `<list>`: List of direct superclasses.

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>`.

17.14.9 `compute-class-precedence-list <list>` *method*

Specialized Arguments

class `<class>`: Class being defined.

direct-superclasses `<list>`: List of direct superclasses.

Result

A list of classes.

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>`.

17.14.10 `compute-slots` *generic function*

Generic Arguments

class `<class>`: Class being defined.

direct-slot-specifications `<list>`: A list of direct slot specification.

inherited-slots `<list>`: A list of lists of inherited slot descriptions.

Result

Computes and returns the list of effective slot descriptions of *class*.

See also

`compute-inherited-slots`.

17.14.11 `compute-slots <class>` *method*

Specialized Arguments

class `<class>`: Class being defined.

slot-specs `<list>`: List of (direct) slot specifications.

inherited-slot-lists `<list>`: A list of lists (in fact one list in single inheritance) of inherited slot descriptions.

Result

A list of effective slot descriptions.

Remarks

The default method computes two sublists:

a) Calling `compute-specialized-slot` with the three arguments (i) *class*, (ii) each *inherited-slot* 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` with the three arguments (i) *class*, (ii) each *slot-specification* which does not have a corresponding (by having the same name) *inherited-slot*.

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.

See also

`compute-specialized-slot`, `compute-defined-slot`, `compute-and-ensure-slot-accessors`.

17.14.12 `compute-keywords` *generic function*

Generic Arguments

class `<class>`: Class being defined.

keywords `<list>`: List of direct keywords.

inherited-keyword-lists `<list>`: A list of lists of inherited keywords.

Result

List of symbols.

Remarks

Computes and returns all legal keywords for *class*.

See also

`compute-inherited-keywords`.

17.14.13 `compute-keywords <class>` *method*

Specialized Arguments

class `<class>`: Class being defined.

keywords `<list>`: List of direct keywords.

inherited-keyword-lists `<list>`: A list of lists of inherited keywords.

Result

List of symbols.

Remarks

This method appends the second argument with the first element of the third argument (single inheritance assumption), removes duplicates and returns the result. Note that `compute-keywords` is called by the default `initialize` method with all direct keywords as the second argument: those specified as slot option and those specified as class option.

17.14.14 `compute-inherited-slots` *generic function*

Generic Arguments

class `<class>`: Class being defined.

direct-superclasses `<list>`: List of direct superclasses.

Result

List of lists of inherited slot descriptions.

Remarks

Computes and returns a list of lists of effective slot descriptions.

See also

`compute-slots`.

17.14.15 `compute-inherited-slots <class>` *method*

Specialized Arguments

class `<class>`: Class being defined.

direct-superclasses `<list>`: List of direct superclasses.

Result

List of lists of inherited slot descriptions.

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-slots` as an argument.

17.14.16 `compute-inherited-keywords` *generic function*

Generic Arguments

class `<class>`: Class being defined.

direct-superclasses `<list>`: List of direct superclasses.

Result

List of lists of symbols.

Remarks

Computes and returns a list of lists of keywords. Its result is used by `compute-keywords` as an argument.

See also

`compute-keywords`.

17.14.17 `compute-inherited-keywords <class>` *method*

Specialized Arguments

class `<class>`: Class being defined.

direct-superclasses `<list>`: List of direct superclasses.

Result

List of lists of symbols.

Remarks

The result of the default method contains one list of legal keywords of the first element of the second argument (single inheritance assumption).

17.14.18 `compute-defined-slot` *generic function*

Generic Arguments

class `<class>`: Class being defined.

slot-spec `<list>`: Canonicalized slot specification.

Result

Slot description.

Remarks

Computes and returns a new effective slot description. It is called by `compute-slots` on each slot specification which has no corresponding inherited slot descriptions.

See also

`compute-defined-slot-class`.

17.14.19 `compute-defined-slot <class>` *method*

Specialized Arguments

class `<class>`: Class being defined.

slot-spec `<list>`: Canonicalized slot specification.

Result

Slot description.

Remarks

Computes and returns a new effective slot description. The class of the result is determined by calling `compute-defined-slot-class`.

See also

`compute-defined-slot-class`.

17.14.20 `compute-defined-slot-class` *generic function*

Generic Arguments

class `<class>`: Class being defined.

slot-spec `<list>`: Canonicalized slot specification.

Result

Slot description class.

Remarks

Determines and returns the slot description class corresponding to *class* and *slot-spec* .

See also

`compute-defined-slot`.

17.14.21 `compute-defined-slot-class <class>` *method*

Specialized Arguments

class `<class>`: Class being defined.

slot-spec `<list>`: Canonicalized slot specification.

Result

The class `<local-slot>`.

Remarks

This method just returns the class `<local-slot>`.

17.14.22 `compute-specialized-slot` *generic function*

Generic Arguments

class `<class>`: Class being defined.

inherited-slots `<list>`: List of inherited slot descriptions (each of the same name as the slot being defined).

slot-spec `<list>`: Canonicalized slot specification or `()`.

Result

Slot description.

Remarks

Computes and returns a new effective slot description. It is called by `compute-slots` 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.

See also

`compute-specialized-slot-class`.

17.14.23 `compute-specialized-slot <class>` *method*

Specialized Arguments

class `<class>`: Class being defined.

inherited-slots `<list>`: List of inherited slot descriptions.

slot-spec `<list>`: Canonicalized sdirect-lot specification or `()`.

Result

Slot description.

Remarks

Computes and returns a new effective slot description. The class of the result is determined by calling `compute-specialized-slot-class`.

See also

`compute-specialized-slot-class`.

17.14.24 `compute-specialized-slot-class` *generic function*

Generic Arguments

class `<class>`: Class being defined.

inherited-slots `<list>`: List of inherited slot descriptions.

slot-spec `<list>`: Canonicalized slot specification or `()`.

Result

Slot description class.

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*.

See also

compute-specialized-slot.

17.14.25 `compute-specialized-slot-class <class>` *method*

Specialized Arguments

class `<class>`: Class being defined.

inherited-slots `<list>`: List of inherited slot descriptions.

slot-spec `<list>`: Canonicalized slot specification or `()`.

Result

The class `<local-slot>`.

Remarks

This method just returns the class `<local-slot>`.

17.15 Slot Access Protocol

The slot access protocol is defined via accessors (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 slots and classes is performed once at class definition time rather than again and again at slot access time.

Each slot 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 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 slots 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 slots 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.

17.15.1 `compute-and-ensure-slot-accessors` *generic function*

Generic Arguments

class `<class>`: Class being defined.

slots `<list>`: List of effective slot descriptions.

inherited-slots `<list>`: List of lists of inherited slot descriptions.

Result

List of effective slot descriptions.

Remarks

Computes new accessors or ensures that inherited accessors work correctly for each effective slot description.

17.15.2 `compute-and-ensure-slot-accessors <class>` *method*

Specialized Arguments

class `<class>`: Class being defined.

slots `<list>`: List of effective slot descriptions.

inherited-slots `<list>`: List of lists of inherited slot descriptions.

Result

List of effective slot descriptions.

Remarks

For each slot description in *slots* 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`.

17.15.3 `compute-slot-reader` *generic function*

Generic Arguments

class `<class>`: Class.

slot `<slot>`: Slot description.

slot-list `<list>`: List of effective slot descriptions.

Result

Function.

Remarks

Computes and returns a new slot reader applicable to instances of *class* returning the slot value corresponding to *slot*. The third argument can be used in order to compute the logical slot position.

17.15.4 `compute-slot-reader <class>` *method*

Specialized Arguments

class `<class>`: Class.

slot `<slot>`: Slot description.

slots `<list>`: List of effective slot descriptions.

Result

Generic function.

Remarks

The default method returns a new generic function of one argument without any methods. Its domain is *class*.

17.15.5 `compute-slot-writer` *generic function*

Generic Arguments

class `<class>`: Class.

slot `<slot>`: Slot description.

slots `<list>`: List of effective slot descriptions.

Result

Function.

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*. The third argument can be used in order to compute the logical slot position.

17.15.6 `compute-slot-writer <class>` *method*

Specialized Arguments

class `<class>`: Class.

slot `<slot>`: Slot description.

slots `<list>`: List of effective slot descriptions.

Result

Generic function.

Remarks

The default method returns a new generic function of two arguments without any methods. Its domain is *class* × `<object>`.

17.15.7 `ensure-slot-reader` *generic function*

Generic Arguments

class `<class>`: Class.

slot `<slot>`: Slot description.

slots `<list>`: List of effective slot descriptions.

reader `<function>`: The slot reader.

Result

Function.

Remarks

Ensures *function* correctly fetches the value of the slot from instances of *class*.

17.15.8 `ensure-slot-reader <class>` *method*

Specialized Arguments

class `<class>`: Class.

slot `<slot>`: Slot description.

slots `<list>`: List of effective slot descriptions.

reader `<generic-function>`: The slot reader.

Result

Generic function.

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` is called by `ensure-slot-reader` method to compute the primitive reader used in the function of the new created reader method.

17.15.9 `ensure-slot-writer` *generic function*

Generic Arguments

class `<class>`: Class.

slot `<slot>`: Slot description.

slots `<list>`: List of effective slot descriptions.

writer `<function>`: The slot writer.

Result

Function.

Remarks

Ensures *function* correctly updates the value of the slot in instances of *class*.

17.15.10 `ensure-slot-writer <class>` *method*

Specialized Arguments

class `<class>`: Class.

slot `<slot>`: Slot description.

slot-list `<list>`: List of effective slot descriptions.

writer `<generic-function>`: The slot writer.

Result

Generic function.

Remarks

The default method checks if there is a method in the *generic-function*. If not, creates and adds a new one, otherwise it assumes that the existing method works correctly. The domain of the new method is *class* × `<object>` and the function is:

```
(method-function-lambda ((obj class)
    (new-value <object>))
  (primitive-writer obj new-value))
```

compute-primitive-writer-using-slot is called by ensure-slot-writer method to compute the primitive writer used in the function of the new created writer method.

17.15.11 compute-primitive-reader-using-slot *generic function*

Generic Arguments

slot <slot>: Slot description.

class <class>: Class.

slots <list>: List of effective slot descriptions.

Result

Function.

Remarks

Computes and returns a function which returns a slot value when applied to an instance of *class*.

17.15.12 compute-primitive-reader-using-slot <slot> *method*

Specialized Arguments

slot <slot>: Slot description.

class <class>: Class.

slots <list>: List of effective slot descriptions.

Result

Function.

Remarks

Calls compute-primitive-reader-using-class. This is the default method.

17.15.13 compute-primitive-reader-using-class *generic function*

Generic Arguments

class <class>: Class.

slot <slot>: Slot description.

slots <list>: List of effective slot descriptions.

Result

Function.

Remarks

Computes and returns a function which returns the slot value when applied to an instance of *class*.

17.15.14 compute-primitive-reader-using-class <class> *method*

Specialized Arguments

class <class>: Class.

slot <slot>: Slot description.

slots <list>: List of effective slot descriptions.

Result

Function.

Remarks

The default method returns a function of one argument.

17.15.15 compute-primitive-writer-using-slot *generic function*

Generic Arguments

slot <slot>: Slot description.

class <class>: Class.

slots <list>: List of effective slot descriptions.

Result

Function.

Remarks

Computes and returns a function which stores a new slot value when applied on an instance of *class* and a new value.

17.15.16 compute-primitive-writer-using-slot <slot> *method*

Specialized Arguments

slot <slot>: Slot description.

class <class>: Class.

slots <list>: List of effective slot descriptions.

Result

Function.

Remarks

Calls compute-primitive-writer-using-class. This is the default method.

17.15.17 compute-primitive-writer-using-class *generic function*

Generic Arguments

class <class>: Class.

slot <slot>: Slot description.

slots <list>: List of effective slot descriptions.

Result

Function.

Remarks

Computes and returns a function which stores the new slot value when applied on an instance of *class* and new value.

17.15.18 `compute-primitive-reader-using-class <class>` *method*

Specialized Arguments

class `<class>`: Class.

slot `<slot>`: Slot description.

slots `<list>`: List of effective slot descriptions.

Result

Function.

Remarks

The default method returns a function of two arguments.

17.16 Method Lookup and Generic Dispatch

17.16.1 `compute-method-lookup-function` *generic function*

Generic Arguments

gf `<generic-function>`: A generic function.

domain `<list>`: A list of classes which cover the domain.

Result

A function.

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.

17.16.2 `compute-method-lookup-function <generic-function>` *method*

Specialized Arguments

gf `<generic-function>`: A generic function.

domain `<list>`: A list of classes which cover the domain.

Result

A function.

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.

17.16.3 `compute-discriminating-function` *generic function*

Generic Arguments

gf `<generic-function>`: A generic function.

domain `<list>`: A list of classes which span the domain.

lookup-fn `<function>`: The method lookup function.

methods `<list>`: A list of methods attached to the *generic-function*.

Result

A function.

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.

17.16.4 `compute-discriminating-function <generic-function>` *method*

Specialized Arguments

gf `<generic-function>`: A generic function.

domain `<list>`: A list of classes which span the domain.

lookup-fn `<function>`: The method lookup function.

methods `<list>`: A list of methods attached to the *generic-function*.

Result

A function.

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.

17.16.5 `add-method` *generic function*

Generic Arguments

gf `<generic-function>`: A generic function.

method `<method>`: A method to be attached to the generic function.

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.

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.

17.16.6 add-method <generic-function> *method*

Specialized Arguments

gf <generic-function> : A generic function.

method <method> : A method to be attached.

Result

The generic function.

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 signaled (condition: <method-domain-clash>). 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.

17.17 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.

17.17.1 primitive-allocate *function*

Arguments

class : A class.

size : An integer.

Result

An instance of the first argument.

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 primitive-ref. It is intended to be used in new allocate methods defined for new metaclasses.

17.17.2 primitive-class-of *function*

Arguments

object : An object created by primitive-allocate.

Result

A class.

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.

17.17.3 (setter primitive-class-of) *setter*

Arguments

object : An object created by primitive-allocate.

class : A class.

Result

The *class*.

Remarks

This function supports portable implementations of

a) dynamic classification like change-class in CLOS.

b) automatic instance updating of redefined classes.

17.17.4 primitive-ref *function*

Arguments

object: An object created by `primitive-allocate`.

index: The index of a component.

Result

An object.

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.

17.17.5 `(setter primitive-ref)` *setter*

Arguments

object: An object created by `primitive-allocate`.

index: The index of a component.

value: The new value, which can be any object.

Result

The new value.

Remarks

Stores and returns the new value as 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.

17.18 Dynamic Binding

The name of this module is `dynamic`.

17.18.1 `dynamic` *special operator*

17.18.1.1 Syntax

dynamic-form: → `<object>`
(`dynamic` *identifier*)

Arguments

identifier: A symbol naming a dynamic binding.

Result

The value of closest dynamic binding of *identifier* is returned. If no visible binding exists, an error is signaled (condition: `<unbound-dynamic-variable>`).

17.18.2 `dynamic-setq` *special operator*

17.18.2.1 Syntax

dynamic-setq-form: → `<object>`
(`dynamic-setq` *identifier form*)

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*.

Result

The value of *form*.

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 signalled (condition: `<unbound-dynamic-variable>`).

17.18.3 `<unbound-dynamic-variable>` `<general-condition>` *condition*

Initialization Options

`symbol` *symbol*: A symbol naming the (unbound) dynamic variable.

Remarks

Signalled by `dynamic` or `dynamic-setq` if the given dynamic variable has no visible dynamic binding.

17.18.4 `dynamic-let` *special operator*

17.18.4.1 Syntax

dynamic-let-form: → `<object>`
(`dynamic-let` *binding**
body)

Arguments

*binding** : A list of binding specifiers.

body : A sequence of forms.

Result

The sequence of *form*s is evaluated in order, returning the value of the last one as the result of the `dynamic-let` form.

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`.

17.18.5 `defglobal` *defining operator*

17.18.5.1 Syntax

defglobal-form: → `<object>`
    (`defglobal` *identifier* *level-1-form*)

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*.

Remarks

The value of *form* is stored as the top dynamic value of the symbol named by *identifier* . The binding created by `defglobal` is mutable. An error is signaled (condition: `<dynamic-multiply-defined>`), on processing this form more than once for the same *identifier*.

NOTE 1 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.

17.18.6 `<dynamic-multiply-defined>` `<general-condition>` *condition*

Initialization Options

`symbol` *symbol* : A symbol naming the dynamic variable which has already been defined.

Remarks

Signalled by `defglobal` if the named dynamic variable already exists.

17.19 Exit Extensions

The name of this module is `exit-1`.

17.19.1 `catch` *special operator*

17.19.1.1 Syntax

catch-form: → `<object>`
    (`catch` *tag* *body*)
tag:
    symbol

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 *form*s are evaluated in sequence and the value of the last one is returned as the value of the `catch` form.

17.19.1.2 Rewrite Rules

(`catch`)	≡	Is an error
(`catch` *tag*)	≡	(`progn` *tag* ())
(`catch` *tag* *body*)	≡	(`let/cc` tmp (`dynamic-let` ((*tag* tmp)) *body*))

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.

See also

`throw`.

17.19.2 `throw` *special operator*

17.19.2.1 Syntax

throw-form: → `<object>`
    (`throw` *tag* *body*)

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 *body* is evaluated and the value are returned as the value of the catch named by *tag*. The *tag* is a symbol because it used to access the current dynamic binding of the symbol, which is where the continuation is bound.

17.19.2.2 Rewrite Rules

(`throw`)	≡	Is an error
(`throw` *tag*)	≡	((`dynamic` *tag*) ())
(`throw` *tag* *form*)	≡	((`dynamic` *tag*) *form*)

See also

`catch`.

17.20 Syntax of Level-1 objects

This section gives the syntax of all level-1 forms:

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.

17.20.1 Syntax of Level-1 modules

defmodule-1-form:
  (**defmodule** *module-name*
    module-directives
    *level-1-module-form**)
level-1-module-form:
  level-0-module-form
  level-1-form
  defining-1-form
level-1-form:
  level-0-form
  special-1-form
form:
  level-1-form
special-form:
  special-1-form
defining-1-form:
  defclass-1-form
  defgeneric-1-form
  defglobal-form
special-1-form:
  generic-lambda-form
  method-lambda-form
  defmethod-form
  method-function-lambda-form
  catch-form
  throw-form

17.20.2 Syntax of Level-1 defining forms

defclass-1-form:
  (**defclass** *class-name* *superclass-names*
    (*slot-1**)
    *class-option-1**)
superclass-names:
  ()
  superclass-name
  (*superclass-name**)
slot-1:
  slot
  (*slot-name* *slot-option-1**)
slot-option-1:
  slot-option
  identifier *level-1-form*
class-option-1
  class-option
  class: *class-name*
  identifier *level-1-form*
defgeneric-1-form:
  (**defgeneric** *gf-name* *gf-lambda-list*
    level-1-init-option)
defmethod-1-form:
  (**defmethod** *gf-locator*
    *method-init-option**
    specialized-lambda-list
    body)
defglobal-form: → **<object>**
  (**defglobal** *identifier* *level-1-form*)

17.20.3 Syntax of Level-1 special forms

dynamic-form: → **<object>**
  (**dynamic** *identifier*)
dynamic-setq-form: → **<object>**
  (**dynamic-setq** *identifier* *form*)
dynamic-let-form: → **<object>**
  (**dynamic-let** *binding**
    body)
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-form*
  level-0-init-option
level-1-method-description:
  (*method-init-option**
    specialized-lambda-list
    body)
method-init-option:
  class *class-name*
  identifier *level-1-form*
method-lambda-form: → **<function>**
  (**method-lambda**
    *method-init-option**
    specialized-lambda-list
    body)
catch-form: → **<object>**
  (**catch** *tag* *body*)
tag:
  symbol
throw-form: → **<object>**
  (**throw** *tag* *body*)

Bibliography

[1] Alberga, C.N., Bosman-Clark, C., Mikelsons, M., Van Deusen, M., & Padget, J.A., *Experience with an Uncommon LISP,* Proceedings of 1986 ACM Symposium on LISP and Functional Programming, ACM, New York, 1986 (also available as IBM Research Report RC-11888).

[2] Berrington N., Deroure D. & Padget J.A., *Guaranteeing Unpredictability,* in preparation.

[3] Bobrow D.G., DiMichiel L.G., Gabriel R.P., Keene S.E, Kiczales G. & Moon D.A, *Common Lisp Object System Specification,* SIGPLAN Notices, Vol. 23, September 1988.

[4] Chailloux J., Devin M. & Hullot J-M., *LELISP: A Portable and Efficient Lisp System,* Proceedings of 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, pp113-122, published by ACM Press, New York.

[5] Chailloux J., Devin M., Dupont F., Hullot J-M., Serpette B., & Vuillemin J., *Le-Lisp de l'INRIA, Version 15.2, Manuel de référence,* INRIA, Rocquencourt, May 1987.

[6] Clinger W. & Rees J.A. (eds.), *The Revised^3 Report on Scheme,* SIGPLAN Notices, Vol. 21, No. 12, 1986.

[7] Cointe P., *Metaclasses are First Class: the ObjVlisp model,* Proceedings of OOPSLA '87, published as SIGPLAN Notices, Vol 22, No 12 pp156-167.

[8] Fitch J.P. & Norman A.C., *Implementing Lisp in a High-Level Language,* Software Practice and Experience, Vol 7, pp713-725.

[9] Friedman D. & Haynes C., *Constraining Control,* Proceedings of 11th Annual ACM Symposium on Principles of Programming Languages, pp245-254, published by ACM Press, New York, 1985.

[10] Hudak P. & Wadler P., (eds.) *Report on the Functional Programming Language Haskell,* Yale University, Department of Computer Science, Research Report YALEU/DCS/RR-666, December 1988.

[11] Landin P.J., *The Next 700 Programming Languages,* Communications of the ACM, Vol 9, No 3., 1966, pp156-166.

[12] Lang K.J. & Pearlmutter B.A., *Oaklisp: An Object-Oriented Dialect of Scheme,* Lisp and Symbolic Computation, Vol. 1, No. 1, June 1988, pp39-51, published by Kluwer Academic Publishers, Boston.

[13] MacQueen D., *et al, Modules for Standard ML,* Proceedings of 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, pp198-207, published by ACM Press, New York.

[14] Milner R., *et al, Standard ML,* Laboratory for the Foundations of Computer Science, University of Edinburgh, Technical Report.

[15] Padget J.A., *et al, Desiderata for the Standardisation of Lisp,* Proceedings of 1986 ACM Conference on Lisp and Functional Programming, pp54-66, published by ACM Press, New York, 1986.

[16] Pitman K.M., *An Error System for Common Lisp,* ISO WG16 paper N24.

[17] Rees J.A., *The T Manual,* Yale University Technical Report, 1986.

[18] Slade S., *The T Programming Language, a Dialect of Lisp,* Prentice-Hall 1987.

[19] Shalit A., *Dylan, an object-oriented dynamic language,* Apple Computer Inc., 1992.

[20] Steele G.L. Jr., *Common Lisp the Language,* Digital Press, 1984.

[21] Steele G.L. Jr., *Common Lisp the Language (second edition),* Digital Press, 1990.

[22] Stoyan H. *et al, Towards a Lisp Standard,* published in the Proceedings of the 1986 European Conference on Artificial Intelligence.

[23] Teitelman W., *The Interlisp Reference Manual,* Xerox Palo Alto Research Center, 1978.

[24] Bretthauer, H. and Kopp, J., *The Meta-Class-System MCS. A Portable Object System for Common Lisp. -Documentation-.* Arbeitspapiere der GMD 554, Gesellschaft fur Mathematik und Datenverarbeitung (GMD), Sankt Augustin (July 1991).

Module Index

`character` (level-0) 38
`collection` (level-0) 40
`compare` (level-0) 46
`condition` (level-0) 31
`convert` (level-0) 48
`copy` (level-0) 49
`double-float` (level-0) 50
`dynamic` (level-1) 95
`exit-1` (level-1) 96
`float` (level-0) 51
`formatted-io` (level-0) 52
`fpi` (level-0) 54
`integer` (level-0) 55
`keyword` (level-0) 56
`list` (level-0) 58
`lock` (level-0) 38
`mathlib` (level-0) 60
`number` (level-0) 62
`stream` (level-0) 65
`string` (level-0) 72
`symbol` (level-0) 74
`table` (level-0) 75
`telos0` (level-0) 15
`thread` (level-0) 36
`vector` (level-0) 76

Class Index

<a-class> (undefined-module) 6
<buffered-stream> (stream) 65
<character> (character) 39
<character-sequence> (collection) 40
<class> (telos0) 15
<collection> (collection) 40
<condition> (condition) 32
<cons> (list) 58
<double-float> (double-float) 50
<file-stream> (stream) 65
<fixed-buffered-stream> (stream) 65
<float> (float) 51
<fpi> (fpi) 54
<function> (telos0) 17
<function-class> (level-1) 82
<generic-function> (telos0) 17
<hash-table> (table) 75
<integer> (integer) 55
<keyword> (keyword) 56
<list> (list) 58
<local-slot> (level-1) 82
<lock> (lock) 38
<metaclass> (level-1) 82
<method> (level-1) 84
<name> (telos0) 20
<null> (list) 58
<number> (number) 62
<object> (telos0) 15
<sequence> (collection) 40
<simple-class> (level-1) 82
<simple-function> (telos0) 17
<simple-generic-function> (telos0) 17
<simple-method> (level-1) 84
<simple-thread> (lock) 38
<slot> (level-1) 82
<stream> (stream) 65
<string> (character) 39
<string> (string) 72
<string-stream> (stream) 65
<symbol> (symbol) 74
<table> (table) 75
<thread> (thread) 36
<vector> (vector) 76

Special Forms Index

' (level-0) ... 21
() (list) ... 58
, (syntax-0) ... 28
` (syntax-0) ... 28
a-special-form (undefined-module) ... 5
and (level-0) ... 25
block (level-0) ... 25
call-next-handler (condition) ... 34
call-next-method (telos0) ... 19
catch (exit-1) ... 96
character (character) ... 38
cond (level-0) ... 24
constant (level-0) ... 21
defclass (level-1) ... 78
defclass (telos0) ... 16
defcondition (condition) ... 31
defconstant (level-0) ... 21
defgeneric (level-1) ... 79
defgeneric (telos0) ... 17
defglobal (dynamic) ... 96
deflocal (level-0) ... 21
defmethod (level-1) ... 80
defmethod (telos0) ... 18
defmodule () ... 11
defsyntax (level-0) ... 22
defun (level-0) ... 22
dynamic (dynamic) ... 95
dynamic-let (dynamic) ... 95
dynamic-setq (dynamic) ... 95
float (float) ... 51
function call (level-0) ... 22
generic-lambda (level-1) ... 79
generic-lambda (telos0) ... 18
if (level-0) ... 24
integer (integer) ... 55
keyword (keyword) ... 56
lambda (level-0) ... 21
let (level-0) ... 26
let* (level-0) ... 27
let/cc (level-0) ... 25
letfuns (level-0) ... 26
method-function-lambda (level-1) ... 80
method-lambda (level-1) ... 80
next-method? (telos0) ... 19
or (level-0) ... 25
pair (list) ... 58
progn (level-0) ... 27
quasiquote (syntax-0) ... 27
quote (level-0) ... 21
return-from (level-0) ... 26
setq (level-0) ... 23
string (string) ... 72
symbol (level-0) ... 21
symbol (symbol) ... 74
throw (exit-1) ... 96
unless (level-0) ... 24
unquote (syntax-0) ... 28
unquote-splicing (syntax-0) ... 28
unwind-protect (level-0) ... 27
vector (vector) ... 76
when (level-0) ... 24
with-handler (condition) ... 34
with-input-file (stream) ... 71
with-output-file (stream) ... 71
with-sink (stream) ... 71
with-source (stream) ... 71

Function Index

`*` (number) 62
`+` (number) 62
`-` (number) 62
`/` (number) 62
`<` (compare) 47
`<=` (compare) 47
`=` (compare) 47
`>` (compare) 47
`>=` (compare) 47
`%` (number) 63
`a-function` (undefined-module) 5
`abs` (number) 63
`abstract-class?` (telos0) 17
`allocate` (telos0) 19
`apply` (level-0) 23
`apply-method` (level-1) 81
`atom?` (list) 59
`call-method` (level-1) 80
`car` (list) 59
`cdr` (list) 59
`cerror` (condition) 35
`character?` (character) 39
`class-keywords` (level-1) 82
`class-name` (level-1) 82
`class-of` (level-1) 81
`class-precedence-list` (level-1) 82
`class-slots` (level-1) 82
`clear-table` (table) 75
`condition?` (condition) 32
`connect` (stream) 66
`cons` (list) 59
`cons?` (list) 59
`convert` (convert) 48
`converter` (convert) 48
`current-thread` (thread) 36
`double-float?` (double-float) 50
`eq` (compare) 46
`eql` (compare) 46
`error` (condition) 35
`file-stream-buffer-position` (stream) 65
`file-stream-filename` (stream) 65
`file-stream-mode` (stream) 65
`file-stream?` (stream) 65
`float?` (float) 51
`flush` (stream) 71
`fmt` (formatted-io) 53
`format` (formatted-io) 53
`from-stream` (stream) 65
`gcd` (number) 63
`generic-function-discriminating-function` (level-1) 83
`generic-function-domain` (level-1) 83
`generic-function-method-class` (level-1) 83
`generic-function-method-lookup-function` (level-1) 83
`generic-function-methods` (level-1) 83
`gensym` (symbol) 74
`int?` (fpi) 54
`integer?` (integer) 55
`keyword-exists?` (keyword) 57
`keyword-name` (keyword) 56
`keyword?` (keyword) 56
`lcm` (number) 63
`list` (list) 59
`lock` (lock) 38
`lock?` (lock) 38
`make` (telos0) 19
`max` (compare) 47
`method-domain` (level-1) 84
`method-function` (level-1) 84
`method-generic-function` (level-1) 84
`min` (compare) 47
`mod` (number) 63
`negative?` (number) 63
`null?` (list) 58
`number?` (number) 62
`odd?` (integer) 56
`open-input-file` (stream) 71
`open-output-file` (stream) 71
`positive?` (number) 63
`primitive-allocate` (level-1) 94
`primitive-class-of` (level-1) 94
`primitive-ref` (level-1) 94
`prin-char` (stream) 71
`print` (stream) 70
`read` (stream) 68
`read-line` (stream) 70
`scan` (formatted-io) 52
`setter` (level-0) 23
`(setter car)` (list) 59
`(setter cdr)` (list) 59
`(setter converter)` (convert) 48
`(setter element)` (collection) 42
`(setter method-function)` (level-1) 84
`(setter primitive-class-of)` (level-1) 94
`(setter primitive-ref)` (level-1) 95
`sflush` (stream) 70
`sformat` (formatted-io) 52
`signal` (condition) 33
`signum` (number) 63
`slot-default-function` (level-1) 83
`slot-name` (level-1) 82
`slot-slot-reader` (level-1) 83
`slot-slot-writer` (level-1) 83
`sprin-char` (stream) 71
`sprint` (stream) 70
`sread` (stream) 71
`stream-buffer` (stream) 65
`stream-buffer-size` (stream) 65
`stream-lock` (stream) 65
`stream-sink` (stream) 65
`stream-source` (stream) 65
`stream?` (stream) 65
`string-stream?` (stream) 65
`string?` (string) 72
`swrite` (stream) 69
`symbol-exists?` (symbol) 74
`symbol-name` (symbol) 74
`symbol?` (symbol) 74
`table?` (table) 75
`thread-reschedule` (thread) 36
`thread-start` (thread) 36
`thread-value` (thread) 37
`thread?` (thread) 36
`to-stream` (stream) 65
`unlock` (lock) 38
`vector?` (vector) 76
`write` (stream) 70

Generic Function Index

a-generic . . . 5
    defgeneric (undefined-module) . . . 5
    defmethod class-a (undefined-module) . . . 6
accumulate . . . 40
    defgeneric (collection) . . . 40
accumulate1 . . . 41
    defgeneric (collection) . . . 41
acos . . . 60
    defgeneric (mathlib) . . . 60
add-method . . . 93
    defgeneric (level-1) . . . 93
    defmethod generic-function (level-1) . . . 94
all? . . . 41
    defgeneric (collection) . . . 41
allocate . . . 85
    defgeneric (level-1) . . . 85
    defmethod class (level-1) . . . 85
any? . . . 41
    defgeneric (collection) . . . 41
as-lowercase . . . 39
    defgeneric (character) . . . 39
    defmethod character (character) . . . 39
    defmethod string (string) . . . 73
as-uppercase . . . 40
    defgeneric (character) . . . 40
    defmethod character (character) . . . 40
    defmethod string (string) . . . 73
asin . . . 60
    defgeneric (mathlib) . . . 60
atan . . . 60
    defgeneric (mathlib) . . . 60
atan2 . . . 60
    defgeneric (mathlib) . . . 60
binary* . . . 64
    defgeneric (number) . . . 64
binary+ . . . 63
    defgeneric (number) . . . 63
binary- . . . 64
    defgeneric (number) . . . 64
binary-gcd . . . 64
    defgeneric (number) . . . 64
binary-lcm . . . 64
    defgeneric (number) . . . 64
binary-mod . . . 64
    defgeneric (number) . . . 64
binary/ . . . 64
    defgeneric (number) . . . 64
binary< . . . 47
    defgeneric (compare) . . . 47
    defmethod character (character) . . . 39
    defmethod string (string) . . . 73
binary= . . . 46
    defgeneric (compare) . . . 46
    defmethod character (character) . . . 39
    defmethod cons (list) . . . 59
    defmethod number (number) . . . 63
    defmethod object (compare) . . . 46
    defmethod string (string) . . . 72
    defmethod vector (vector) . . . 76
binary% . . . 64
    defgeneric (number) . . . 64
ceiling . . . 51
    defgeneric (float) . . . 51
collection? . . . 41
    defgeneric (collection) . . . 41
compatible-superclass? . . . 86
    defgeneric (level-1) . . . 86
    defmethod abstract-class (level-1) . . . 87
    defmethod class (level-1) . . . 86
compatible-superclasses? . . . 86
    defgeneric (level-1) . . . 86
    defmethod class (level-1) . . . 86
compute-and-ensure-slot-accessors . . . 90
    defgeneric (level-1) . . . 90
    defmethod class (level-1) . . . 90
compute-class-precedence-list . . . 87
    defgeneric (level-1) . . . 87
    defmethod list (level-1) . . . 87
compute-constructor . . . 85
    defgeneric (level-1) . . . 85
    defmethod class (level-1) . . . 85
compute-defined-slot . . . 88
    defgeneric (level-1) . . . 88
    defmethod class (level-1) . . . 89
compute-defined-slot-class . . . 89
    defgeneric (level-1) . . . 89
    defmethod class (level-1) . . . 89
compute-discriminating-function . . . 93
    defgeneric (level-1) . . . 93
    defmethod generic-function (level-1) . . . 93
compute-inherited-keywords . . . 88
    defgeneric (level-1) . . . 88
    defmethod class (level-1) . . . 88
compute-inherited-slots . . . 88
    defgeneric (level-1) . . . 88
    defmethod class (level-1) . . . 88
compute-keywords . . . 88
    defgeneric (level-1) . . . 88
    defmethod class (level-1) . . . 88
compute-method-lookup-function . . . 93
    defgeneric (level-1) . . . 93
    defmethod generic-function (level-1) . . . 93
compute-predicate . . . 84
    defgeneric (level-1) . . . 84
    defmethod class (level-1) . . . 84
compute-primitive-reader-using-class . . . 92
    defgeneric (level-1) . . . 92
    defmethod class (level-1) . . . 92, 93
compute-primitive-reader-using-slot . . . 92
    defgeneric (level-1) . . . 92
    defmethod slot (level-1) . . . 92
compute-primitive-writer-using-class . . . 92
    defgeneric (level-1) . . . 92
compute-primitive-writer-using-slot . . . 92
    defgeneric (level-1) . . . 92
    defmethod slot (level-1) . . . 92
compute-slot-reader . . . 90
    defgeneric (level-1) . . . 90
    defmethod class (level-1) . . . 90
compute-slot-writer . . . 91
    defgeneric (level-1) . . . 91
    defmethod class (level-1) . . . 91
compute-slots . . . 87
    defgeneric (level-1) . . . 87
    defmethod class (level-1) . . . 87
compute-specialized-slot . . . 89
    defgeneric (level-1) . . . 89
    defmethod class (level-1) . . . 89
compute-specialized-slot-class . . . 89
    defgeneric (level-1) . . . 89
    defmethod class (level-1) . . . 90
concatenate . . . 41
    defgeneric (collection) . . . 41
cos . . . 61
    defgeneric (mathlib) . . . 61
cosh . . . 61

defgeneric (mathlib) 61
deep-copy 49
defgeneric (copy) 49
defmethod class (copy) 49
defmethod cons (list) 59
defmethod object (copy) 49
defmethod string (string) 73
defmethod vector (vector) 76
delete 42
defgeneric (collection) 42
disconnect 67
defgeneric (stream) 67
defmethod file-stream (stream) 67
defmethod stream (stream) 67
do 42
defgeneric (collection) 42
element 42
defgeneric (collection) 42
empty? 42
defgeneric (collection) 42
end-of-stream 68
defgeneric (stream) 68
defmethod buffered-stream (stream) 68
defmethod file-stream (stream) 68
ensure-slot-reader 91
defgeneric (level-1) 91
defmethod class (level-1) 91
ensure-slot-writer 91
defgeneric (level-1) 91
defmethod class (level-1) 91
even? 55
defgeneric (integer) 55
exp 61
defgeneric (mathlib) 61
fill 42
defgeneric (collection) 42
fill-buffer 67
defgeneric (stream) 67
defmethod buffered-stream (stream) 67
defmethod file-stream (stream) 67
find-key 43
defgeneric (collection) 43
first 43
defgeneric (collection) 43
floor 52
defgeneric (float) 52
flush-buffer 67
defgeneric (stream) 67
defmethod buffered-stream (stream) 68
defmethod file-stream (stream) 68
generic-connect 66
defgeneric (stream) 66
defmethod file-stream (stream) 66
defmethod path (stream) 66
defmethod stream (stream) 66
generic-print 70
defgeneric (stream) 70
defmethod character (character) 40
defmethod cons (list) 60
defmethod double-float (double-float) 50
defmethod fpi (fpi) 54
defmethod keyword (keyword) 57
defmethod null (list) 58
defmethod string (string) 73
defmethod symbol (symbol) 74
defmethod vector (vector) 76
generic-read 68
defgeneric (stream) 68
defmethod buffered-stream (stream) 69
defmethod file-stream (stream) 69
defmethod stream (stream) 69
generic-write 69
defgeneric (stream) 69
defmethod buffered-stream (stream) 69
defmethod character (character) 40
defmethod cons (list) 60
defmethod double-float (double-float) 51
defmethod file-stream (stream) 69
defmethod fpi (fpi) 54
defmethod keyword (keyword) 57
defmethod null (list) 58
defmethod stream (stream) 69
defmethod string (string) 73
defmethod symbol (symbol) 74
defmethod vector (vector) 76
initialize 20
defgeneric (telos0) 20
defmethod class (level-1) 84
defmethod condition (condition) 32
defmethod generic-function (level-1) 86
defmethod method (level-1) 86
defmethod object (telos0) 20
defmethod slot (level-1) 85
key-sequence 43
defgeneric (collection) 43
last 43
defgeneric (collection) 43
log 61
defgeneric (mathlib) 61
log10 61
defgeneric (mathlib) 61
map 43
defgeneric (collection) 43
member 43
defgeneric (collection) 43
negate 63
defgeneric (number) 63
pow 61
defgeneric (mathlib) 61
reconnect 66
defgeneric (stream) 66
defmethod stream (stream) 66
remove 44
defgeneric (collection) 44
reverse 44
defgeneric (collection) 44
round 52
defgeneric (float) 52
sequence? 44
defgeneric (collection) 44
shallow-copy 49
defgeneric (copy) 49
defmethod class (copy) 49
defmethod cons (list) 59
defmethod object (copy) 49
defmethod string (string) 73
defmethod vector (vector) 76
sin 61
defgeneric (mathlib) 61
sinh 61
defgeneric (mathlib) 61
size 44
defgeneric (collection) 44
slice 44
defgeneric (collection) 44
sort 44
defgeneric (collection) 44
sqrt 61
defgeneric (mathlib) 61
tan 61
defgeneric (mathlib) 61
tanh 61

`defgeneric` (mathlib) 61
`truncate` 52
`defgeneric` (float) 52
`wait` 37
`defgeneric` (thread) 37
`defmethod thread` (thread) 37
`zero?` 63
`defgeneric` (number) 63

Condition Index

`<cannot-update-setter>` (level-0) 23
`a-condition` (undefined-module) 6
`arithmetic-condition` (number) 62
`cannot-update-setter` (level-0) 24
`collection-condition` (collection) 40, 42
`conversion-condition` (collection) 45
`conversion-condition` (convert) 48
`division-by-zero` (number) 62, 64
`domain-condition` (condition) 32
`domain-condition` (mathlib) 60, 61
`dynamic-multiply-defined` (dynamic) 96
`end-of-stream` (stream) 68, 70
`environment-condition` (condition) 32
`general-condition` (condition) 32
`generic-function-condition` (condition) 33
`incompatible-method-and-gf` (level-1) 94
`incompatible-method-domain` (condition) 33
`incompatible-method-domain` (telos0) 18
`invalid-operator` (level-0) 23
`method-domain-clash` (condition) 33
`method-domain-clash` (level-1) 94
`method-domain-clash` (telos0) 18
`no-applicable-method` (condition) 33
`no-applicable-method` (convert) 48
`no-applicable-method` (telos0) 18
`no-converter` (convert) 48
`no-next-method` (condition) 33
`no-next-method` (telos0) 19
`no-setter` (level-0) 23, 24
`non-congruent-lambda-lists` (condition) 33
`non-congruent-lambda-lists` (level-1) 79
`non-congruent-lambda-lists` (telos0) 18
`range-condition` (condition) 32
`range-condition` (mathlib) 61
`read-error` (stream) 68
`scan-mismatch` (formatted-io) 52
`thread-already-started` (thread) 37, 38
`thread-condition` (thread) 37
`unbound-dynamic-variable` (dynamic) 95
`wrong-condition-class` (condition) 32, 34
`wrong-thread-continuation` (level-0) 36
`wrong-thread-continuation` (thread) 38

Constant Index

`a-constant` (undefined-module) 6
`else` (level-0) 24
`least-negative-double-float` (double-float) 50
`least-positive-double-float` (double-float) 50
`maximum-vector-index` (vector) 76
`most-negative-double-float` (double-float) 50
`most-negative-int` (fpi) 54
`most-positive-double-float` (double-float) 50
`most-positive-int` (fpi) 54
`pi` (mathlib) 60
`t` (level-0) 21
`ticks-per-second` (thread) 37

Index

' ... 21
() ... 58
* ... 62
+ ... 62
, ... 28
- ... 62
/ ... 62
< ... 47
<= ... 47
= ... 47
> ... 47
>= ... 47
% ... 63
` ... 28
<a-class> ... 6
a-constant ... 6
a-function ... 5
  function signature ... 5
a-generic ... 5, 6
a-special-form ... 5
  syntax table ... 5
abs ... 63
abstract class ... 7
abstract-class? ... 17
  function signature ... 17
abstract? ... 17
accessor ... 17
accumulate ... 40
accumulate1 ... 41
acos ... 50, 60
add-method ... 93, 94
all? ... 41
allocate ... 19, 85, 85
and ... 25
  rewrite rules ... 25
  syntax table ... 25
any? ... 41
applicable method ... 7
apply ... 23
  syntax table ... 23
apply-method ... 81
as-lowercase ... 39, 39, 73
as-uppercase ... 40, 40, 73
asin ... 50, 60
assignment ... 23
atan ... 50, 60
atan2 ... 50, 60
atom? ... 59
backquoting ... 27
base ... 55
  arbitrary base literals ... 55
  limitation on input ... 55
binary literals ... 55
binary* ... 50, 54, 64
binary+ ... 50, 54, 63
binary- ... 50, 54, 64
binary-gcd ... 54, 64
binary-lcm ... 54, 64
binary-mod ... 50, 54, 64
binary/ ... 50, 54, 64
binary< ... 39, 47, 50, 54, 73
binary= ... 39, 46, 46, 50, 54, 59, 63, 72, 76
binary% ... 54, 64
binding ... 7
  dynamically scoped ... 2
  module ... 22
  of module names ... 11
  of module names ... 12
  top dynamic ... 96
block ... 25
  rewrite rules ... 26
  see also let/cc ... 26
  syntax table ... 26
boolean ... 10, 11
boolean
  syntax table ... 11
<buffered-stream> ... 65
call-method ... 80
call-next-handler ... 34
  syntax table ... 34
call-next-method ... 19
  function signature ... 19
car ... 59
case sensitivity ... 9
catch ... 96
  rewrite rules ... 96
  syntax table ... 96
cdr ... 59
ceiling ... 50, 51
cerror ... 35
character ... 38
  character-extension glyph ... 38
  minimal character set ... 9
  module ... 38
<character> ... 39
character ... 38
  syntax table ... 39
character set ... 9
<character-sequence> ... 40
character-set
  syntax table ... 9
character? ... 39
Common Lisp ... vi, 2
class ... 2, 7
  primitive ... 2
  self-instantiated ... 8
<class> ... 15
class ... 78
class option ... 7
class precedence list ... 7
class-keywords ... 82
class-name ... 82
class-of ... 81
class-precedence-list ... 82
class-slots ... 82
clear-table ... 75
CLOS ... 2
closure ... 7
coercion ... 62
collection ... 40
  alignment ... 40
  module ... 40
<collection> ... 40
collection-condition ... 43
collection? ... 41
comment
  syntax table ... 10
comments ... 9
  line ... 9
  object ... 9
Common Lisp Error System ... 31
compare ... 46
  module ... 46
compatible-superclass? ... 86, 86, 87
compatible-superclasses? ... 86, 86
compliance ... 4

`compute-and-ensure-slot-accessors` ... 90, 90
`compute-class-precedence-list` ... 87, 87
`compute-constructor` ... 85, 85
`compute-defined-slot` ... 88, 89
`compute-defined-slot-class` ... 89, 89
`compute-discriminating-function` ... 93, 93
`compute-inherited-keywords` ... 88, 88
`compute-inherited-slots` ... 88, 88
`compute-keywords` ... 88, 88
`compute-method-lookup-function` ... 93, 93
`compute-predicate` ... 84, 84
`compute-primitive-reader-using-class` ... 92, 92, 93
`compute-primitive-reader-using-slot` ... 92, 92
`compute-primitive-writer-using-class` ... 92
`compute-primitive-writer-using-slot` ... 92, 92
`compute-slot-reader` ... 90, 90
`compute-slot-writer` ... 91, 91
`compute-slots` ... 87, 87
`compute-specialized-slot` ... 89, 89
`compute-specialized-slot-class` ... 89, 90
`concatenate` ... 41
concurrency ... 35
`cond` ... 24
    rewrite rules ... 24
    syntax table ... 24
condition ... 31
    continuable ... 31
    module ... 31
    non-continuable ... 31
`<condition>` ... 32
`condition?` ... 32
configuration ... 4
conformance ... 4
    level-0 ... 4
    level-1 ... 4
conforming processor ... 4
conforming program ... 4
conformity clause ... 4
conformity-clause
    least negative double precision float ... 50
    least positive double precision float ... 50
    maximum vector index ... 76
    most negative double precision float ... 50
    most negative fixed precision integer ... 54
    most positive double precision float ... 50
    most positive fixed precision integer ... 54
congruent ... 7
`connect` ... 66
`<cons>` ... 58
`cons` ... 59
`cons?` ... 59
constant
    a-constant ... 6
    defined ... 21
    else ... 24
    least-negative-double-float ... 50
    least-positive-double-float ... 50
    literal ... 21
    maximum-vector-index ... 76
    most-negative-double-float ... 50
    most-negative-int ... 54
    most-positive-double-float ... 50
    most-positive-int ... 54
    pi ... 60
    t ... 21
    ticks-per-second ... 37
`constant` ... 21, 21
`constructor` ... 17
continuation ... 7, 26, 96
conventions ... 5
convert ... 48
    module ... 48
`convert` ... 48
`converter` ... 48
converter function ... 7
`(converter <double-float>)` ... 54
`(converter <fpi>)` ... 50
`(converter <list>)` ... 45
`(converter <string>)` ... 45, 50, 54, 57, 75
`(converter <symbol>)` ... 72
`(converter <table>)` ... 45
`(converter <vector>)` ... 45
copy ... 49
    module ... 49
`cos` ... 50, 61
`cosh` ... 50, 61
`current-thread` ... 36
`deep-copy` ... 49, 49, 59, 73, 76
default ... 8
`default` ... 16
default-function ... 8
`defclass` ... 16, 78
    syntax table ... 16
`defclass-1`
    syntax table ... 78
`defcondition` ... 31
    syntax table ... 31
`defconstant` ... 21
    syntax table ... 21
`defgeneric` ... 15, 17, 79
    rewrite rules ... 18
    syntax table ... 17
`defgeneric-1`
    syntax table ... 79
`defglobal` ... 96
    syntax table ... 96
defining form ... 3, 7
defining operator ... 7
`defining-0-forms`
    syntax table ... 13
`deflocal` ... 21
    syntax table ... 21
`defmethod` ... 18, 80
    syntax table ... 18
`defmethod-1`
    syntax table ... 80
`defmodule` ... 11, 11
`defmodule-0`
    syntax table ... 12
`defmodule-1`
    syntax table ... 78
`defsyntax` ... 22
    syntax table ... 22
`defun` ... 22
    rewrite rules ... 22
    syntax table ... 22
`delete` ... 42
direct instance ... 7
direct subclass ... 7
direct superclass ... 7
`disconnect` ... 67, 67
`do` ... 42
double-float ... 50
    module ... 50
`<double-float>` ... 50
`double-float`
    syntax table ... 50
`double-float?` ... 50
dynamic ... 95
    module ... 95
`dynamic` ... 95
    syntax table ... 95

dynamic environment . . . 7
dynamic extent . . . 7
dynamic scope . . . 7
dynamic-let . . . 95
  syntax table . . . 95
dynamic-setq . . . 95
  syntax table . . . 95
element . . . 42
else . . . 24
empty? . . . 42
end-of-stream . . . 68, 68
ensure-slot-reader . . . 91, 91
ensure-slot-writer . . . 91, 91
environmental error . . . 4
eq . . . 46
  implementation-defined behaviour . . . 46
eql . . . 46
error . . . 4
  can be signalled . . . 4
  environmental . . . 4
  handler . . . 33
  signalled . . . 4
  static . . . 4
  violation . . . 4
error . . . 35
errors . . . 4
EuLisp
  level-0 . . . 3
  level-1 . . . 3
  libraries . . . 3
even? . . . 54, 55
except . . . 12
exit-1 . . . 96
  module . . . 96
exp . . . 50, 61
export . . . iii, 12
expose . . . iii, 13
extension . . . 4
extent . . . 7
external representation
  list . . . 58
external representation
  floating point . . . 51
  integer . . . 55
  null (empty list) . . . 58
  pair . . . 58
  string . . . 72
  vector . . . 76
external representation (see also **prin** and **write**) . . . 21
false . . . 5, 10
<file-stream> . . . 65
file-stream-buffer-position . . . 65
file-stream-filename . . . 65
file-stream-mode . . . 65
file-stream? . . . 65
fill . . . 42
fill-buffer . . . 67, 67
find-key . . . 43
first . . . 43
first-class . . . 7
<fixed-buffered-stream> . . . 65
float . . . 51
  module . . . 51
<float> . . . 51
float . . . 51
  syntax table . . . 51
float? . . . 51
floor . . . 50, 52
flush . . . 71
flush-buffer . . . 67, 68
fmt . . . 53
format . . . 53
formatted-io . . . 52
  module . . . 52
fpi . . . 54
  module . . . 54
<fpi> . . . 54
from-stream . . . 65
function . . . 7
  calling . . . 22
  reader . . . 23
  standard function . . . 5
  writer . . . 23
<function> . . . 17
function call . . . 22
function signatures
  a-function . . . 5
  abstract-class? . . . 17
  call-next-method . . . 19
  next-method? . . . 19
function-call
  syntax table . . . 22
<function-class> . . . 82
gcd . . . 63
generic function . . . 7
  lambda-list . . . 17
  rewrite rules . . . 18
generic-1-lambda
  syntax table . . . 79
generic-connect . . . 66, 66
<generic-function> . . . 17
generic-function-discriminating-function . . . 83
generic-function-domain . . . 83
generic-function-method-class . . . 83
generic-function-method-lookup-function . . . 83
generic-function-methods . . . 83
generic-lambda . . . 15, 18, 79
  syntax table . . . 18
generic-print . . . 40, 50, 54, 57, 58, 60, 70, 73, 74, 76
generic-read . . . 68, 69
generic-write . . . 40, 51, 54, 57, 58, 60, 69, 69, 73, 74, 76
gensym . . . 74
<hash-table> . . . 75
Haskell . . . vi, 11
hexadecimal literals . . . 55
identifier . . . 7, 10
  definition of . . . 10
  peculiar identifiers . . . 10
identifier
  syntax table . . . 10
identifiers . . . 10
if . . . 24
  syntax table . . . 24
implementation-defined . . . 4
  behaviour of **eq** . . . 46
  least negative double precision float . . . 50
  least positive double precision float . . . 50
  maximum vector index . . . 76
  module directives . . . 12
  most negative double precision float . . . 50
  most negative fixed precision integer . . . 54
  most positive double precision float . . . 50
  most positive fixed precision integer . . . 54
  time units per second . . . 37
import . . . iii, 12
indefinite extent . . . 7
indirect instance . . . 8
indirect subclass . . . 8
inheritance
  multiple . . . 19, 78
  single . . . 15
inheritance graph . . . 8

inherited slot description . . . 8
init-list . . . 8
initialization . . . 19
`initialize` . . . 20, 20, 32, 84–86
inner dynamic . . . 8
inner lexical . . . 8
instance . . . 8
  direct . . . 7
  indirect . . . 8
instantiation graph . . . 8
`int?` . . . 54
integer . . . 55
  module . . . 55
`<integer>` . . . 55
`integer` . . . 55
  syntax table . . . 55
`integer?` . . . 55
InterLISP . . . vi, 2
`key-sequence` . . . 43
keyword . . . 8, 56
  abstract? . . . 17
  accessor . . . 17
  class . . . 78
  constructor . . . 17
  default . . . 16
  definition of . . . 56
  except . . . 12
  export . . . iii, 12
  expose . . . iii, 13
  import . . . iii, 12
  keyword . . . 16
  keywords . . . 17
  method . . . 17
  module . . . 56
  only . . . 12
  predicate . . . 17
  reader . . . 16
  rename . . . 12
  required? . . . 17
  syntax . . . iii, 13
  writer . . . 16
`<keyword>` . . . 56
`keyword` . . . 16, 56
  syntax table . . . 56
`keyword-exists?` . . . 57
`keyword-name` . . . 56
`keyword?` . . . 56
`keywords` . . . 17
`lambda` . . . 21
  syntax table . . . 22
language structure . . . 3
`last` . . . 43
`lcm` . . . 63
`least-negative-double-float` . . . 50
`least-positive-double-float` . . . 50
LeLisp . . . 11
LE-LISP . . . vi, 2
`let` . . . 26
  rewrite rules . . . 26
  syntax table . . . 26
`let*` . . . 27
  rewrite rules . . . 27
  syntax table . . . 27
`let/cc` . . . 25
  see also `block` and `return-from` . . . 25
  syntax table . . . 25
`letfuns` . . . 26
  syntax table . . . 26
level-0 . . . 3, 21
lexical environment . . . 8
lexical scope . . . 8
lexical syntax . . . 9
`lispin` . . . 67
LISP/VM . . . vi, 2
list . . . 58
  module . . . 58
`<list>` . . . 58
`list` . . . 59
  syntax table . . . 58
literal . . . 21
  arbitrary base . . . 55
  binary . . . 55
  character . . . 38
  hexadecimal . . . 55
  modification of . . . 21
  octal . . . 55
  quotation . . . 21
`<local-slot>` . . . 82
lock . . . 38
  module . . . 38
`<lock>` . . . 38
`lock` . . . 38
`lock?` . . . 38
`log` . . . 50, 61
`log10` . . . 50, 61
`make` . . . 19
`map` . . . 43
mathlib . . . 60
  module . . . 60
`max` . . . 47
`maximum-vector-index` . . . 76
MCS . . . 2
`member` . . . 43
metaclass . . . 8
`<metaclass>` . . . 82
method . . . 8
  applicable . . . 7
  bindings . . . 18
  specificity . . . 8
`<method>` . . . 84
`method` . . . 17
method function . . . 8
method specificity . . . 8
`method-domain` . . . 84
`method-function` . . . 84
`method-function-lambda` . . . 80
`method-generic-function` . . . 84
`method-lambda` . . . 80
  syntax table . . . 80
MicroCeyx . . . 2
`min` . . . 47
`mod` . . . 63
module . . . 2, 11
  directives . . . 12
  environments . . . 2
  `except` . . . 12
  `export` . . . 12
  `expose` . . . 13
  `import` . . . 12
  name bindings . . . 11, 12
  `only` . . . 12
  `rename` . . . 12
  `syntax` . . . 13
modules
  level-0 . . . 3
`most-negative-double-float` . . . 50
`most-negative-int` . . . 54
`most-positive-double-float` . . . 50
`most-positive-int` . . . 54
multi-method . . . 8
multiple inheritance . . . 78
multiple inheritance . . . 19

<name> . . . 20
negate . . . 50, 54, 63
negative? . . . 63
new instance . . . 8
next-method? . . . 19
    function signature . . . 19
normativeReferences . . . 3
null . . . 58
<null> . . . 58
null? . . . 58
number . . . 62
    coercion . . . 62
    module . . . 62
<number> . . . 62
number? . . . 62
Oaklisp . . . 2
object . . . 19
    syntax . . . 77
    syntax-1 . . . 97
<object> . . . 15
object
    syntax table . . . 10
objects . . . 10
ObjVLisp . . . 2
octal literals . . . 55
odd? . . . 56
only . . . 12
open-input-file . . . 71
open-output-file . . . 71
or . . . 25
    rewrite rules . . . 25
    syntax table . . . 25
pair . . . 58
pair . . . 58
pi . . . 60
positive? . . . 63
pow . . . 50, 61
predicate . . . 17
primitive-allocate . . . 94
primitive-class-of . . . 94
primitive-ref . . . 94
prin-char . . . 71
print . . . 70
processing
    constants . . . 21
    symbols . . . 21
processor . . . 4
processor-defined
    gensym names . . . 74
progn . . . 27
    syntax table . . . 27
quasiquotation . . . 27
quasiquote . . . 27
    abbreviation with ` . . . 28
    syntax table . . . 27
quote . . . 21
    abbreviation with ' . . . 21
    syntax table . . . 21
range-condition . . . 61
read . . . 68
read-line . . . 70
reader . . . 16
reconnect . . . 66, 66
reflective . . . 8
remove . . . 44
rename . . . 12
required? . . . 17
return-from . . . 26
    rewrite rules . . . 26
    see also let/cc . . . 26
    syntax table . . . 26
reverse . . . 44
rewrite rules
    and . . . 25
    block . . . 26
    catch . . . 96
    cond . . . 24
    defgeneric . . . 18
    defun . . . 22
    generic function . . . 18
    let . . . 26
    let* . . . 27
    or . . . 25
    return-from . . . 26
    throw . . . 96
    unless . . . 25
    when . . . 24
round . . . 50, 52
scan . . . 52
scope . . . 8
scope and extent
    of lambda bindings . . . 22
scope and extent
    in letfuns expressions . . . 26
    of let/cc binding . . . 25
    of dynamic-let bindings . . . 96
self-instantiated class . . . 8
<sequence> . . . 40
sequence? . . . 44
setq . . . 23
    syntax table . . . 23
setter . . . 23
setter function . . . 8
(setter car) . . . 59
(setter cdr) . . . 59
(setter converter) . . . 48
(setter element) . . . 42
(setter method-function) . . . 84
(setter primitive-class-of) . . . 94
(setter primitive-ref) . . . 95
sflush . . . 70
sformat . . . 52
shallow-copy . . . 49, 49, 59, 73, 76
signal . . . 33
signum . . . 63
simple function . . . 8
<simple-class> . . . 82
<simple-function> . . . 17
<simple-generic-function> . . . 17
<simple-method> . . . 84
<simple-thread> . . . 38
sin . . . 50, 61
single inheritance . . . 15
sinh . . . 50, 61
size . . . 44
slice . . . 44
slot . . . 9
<slot> . . . 82
slot description . . . 9
slot option . . . 9
slot specification . . . 9
slot-default-function . . . 83
slot-name . . . 82
slot-slot-reader . . . 83
slot-slot-writer . . . 83
sort . . . 44
special form . . . 3, 9
special operator . . . 9
special-0-forms
    syntax table . . . 14
specialize . . . 9
specialize on . . . 9

sprin-char ... 71
sprint ... 70
sqrt ... 50, 61
sread ... 71
Standard ML ... 11
Standard ML ... vi, 31
standard module ... 3
static error ... 4
stderr ... 67
stdin ... 67
stdout ... 67
stream ... 65
  module ... 65
<stream> ... 65
stream-buffer ... 65
stream-buffer-size ... 65
stream-lock ... 65
stream-sink ... 65
stream-source ... 65
stream? ... 65
string ... 72
  escaping in ... 72
  module ... 72
  string-escape glyph ... 72
<string> ... 39, 72
string ... 72
  syntax table ... 72
<string-stream> ... 65
string-stream? ... 65
string? ... 72
subclass ... 9
  direct ... 7
  indirect ... 8
superclass ... 9
  direct ... 7
swrite ... 69
symbol ... 74
  module ... 74
<symbol> ... 74
symbol ... 21, 21, 74
  syntax table ... 74
symbol-exists? ... 74
symbol-name ... 74
symbol? ... 74
syntax ... 14, 77
  generic function lambda-list ... 17
syntax ... iii, 13
syntax operator ... 22
syntax expansion ... 14
syntax operator ... 8, 14
  definition by **defsyntax** ... 22
syntax-1 ... 97
T ... vi, 2
t ... 21
table ... 75
  module ... 75
<table> ... 75
table? ... 75
tan ... 50, 61
tanh ... 50, 61
telos0 ... 15
  module ... 15
thread ... 36
  module ... 36
<thread> ... 36
thread-reschedule ... 36
thread-start ... 36
thread-value ... 37
thread? ... 36
throw ... 96
  rewrite rules ... 96
  syntax table ... 96
ticks-per-second ... 37
to-stream ... 65
top dynamic ... 9
top lexical ... 9
true ... 5, 10
truncate ... 50, 52
unless ... 24
  rewrite rules ... 25
  syntax table ... 24
unlock ... 38
unquote ... 28
  abbreviation with **,** ... 28
  syntax table ... 28
unquote-splicing ... 28
  syntax table ... 28
unwind-protect ... 27
  syntax table ... 27
vector ... 76
  module ... 76
<vector> ... 76
vector ... 76
  syntax table ... 76
vector? ... 76
violation ... 4
wait ... 37, 37
when ... 24
  rewrite rules ... 24
  syntax table ... 24
whitespace ... 9
with-handler ... 34
  syntax table ... 34
with-input-file ... 71
  syntax table ... 71
with-output-file ... 71
  syntax table ... 71
with-sink ... 71
  syntax table ... 71
with-source ... 71
  syntax table ... 71
write ... 70
writer ... 16
zero? ... 50, 54, 63