Programming Language Eulisp

Version, 0.96

Contents

Annexes

Figures

Tables

Programming Language EuLisp, version 0.96

Foreword

The EULISP group first met in September 1985 at IRCAM in Paris to discuss the need for a common European dialect of Lisp. Subsequent meetings formulated the view of EULISP that was presented at the 1986 ACM Conference on Lisp and Functional Programming held at MIT, Cambridge, Massachussetts [Padget *et al*, 1986] and at the European Conference on Artificial Intelligence (ECAI-86) held in Brighton, Sussex [Stoyan *et al*, 1986]. 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 this document, the EULISP definition. Since then, affairs have returned to their previous state, but with the evolution of the implementations of EULISP and the background of the foundations laid by the INRIA supported work, there is convergence to a consistent and practical definition.

The acknowledgements for this report fall into three categories: intellectual, personal, and financial.

The ancestors of EULISP (in alphabetical order) are Common Lisp[Steele, 1984/90], INTERLISP[Teitelman, 1978], LELISP [Chailloux *et al*, 1984], LISP/VM [Alberga *et al*, 1986], Scheme [Clinger & Rees, 1986], and T [Rees *et al*, 1986] [Slade, 1987]. There has also been some feedback from a language which has been influenced by EuLisp, namely Dylan [Shalit, 1992]. 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 [Milner *et al*, 1986] and the draft report on Haskell [Hudak, Wadler *et al.*, 1988] have also provided much useful input.

The writing of this report has, at various stages, been supported by Bull S.A., 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 EULISP.

Initially, funding for the EULISP group came from individuals' institutions or companies, but since 1987 the Commission of the European Communities (CEC, as the EULISP Technical Interest Group (TIG), also called the EULISP committee, supported by DG XIII) has provided the assistance without which this effort would have faded away. 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 with Deutscher Akademischer Austauschdienst (Academic Research Collaboration programme), British Standards Institute, Centre d'Estudis Avançats de Blanes, CSIC, Departament de Llenguatges i Sistemes Informàtics (LSI, Universitat Politècnica de Catalunya), 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), Rank Xerox France, Science and Engineering Research Council, Siemens AG, University of Bath, University of Technology, Delft, University of Edinburgh, Universität Erlangen, 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, Russell Bradford, Harry Bretthauer, Peter Broadbery, Christopher Burdorf, Jérôme Chailloux, Thomas Christaller, Jeff Dalton, Klaus Däßler, Harley Davis, David DeRoure, John Fitch, Richard Gabriel, Brigitte Glas, Nicolas Graube, Dieter Kolb, Jürgen Kopp, Pascal Kuczynski, Antonio Moreno, Eugen Neidl, Pierre Parquier, Keith Playford, Willem van der Poel, Christian Queinnec, Enric Sesa, Herbert Stoyan, and Richard Tobin.

The editors wish particularly to acknowledge the work of Harley Davis on the first version of the description of the object system and of Harry Bretthauer on the second version.

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

Greg Nuyens (`nuyens@ilog.fr`)
ILOG SA
2, Avenue Galliéni
94353 Gentilly CEDEX, FRANCE

editors.

Introduction

The purpose of this document is to define the programming language EuLisp. EuLisp is a dialect of Lisp and as such owes much to the great body of work that has been done on language design in the name of Lisp over the last thirty years. EuLisp is the outcome of efforts on the part of many people in countries of the European Community since 1986. The guiding principles of the language are simplicity, expressiveness, completeness, orthogonality of constructs, formal definition and efficient implementation.

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 necessary. It takes a class system, but extends the notion by integrating the primitive types (classes) with user-defined classes. It has a condition system. It introduces a module mechanism for information hiding and separate compilation and it has first-class continuations. But this is not the place for a detailed language comparison. That can be drawn from the rest of this report. However, it is important to stress that the distinguishing features of EuLisp are the integration of the classical Lisp type system and the object system into a single class hierarchy, the complementary abstraction facilities provided by the class and the module mechanism and suport for concurrent execution. EuLisp inherits from Scheme the properties of static-scoping, a single lexical environment for all variables and the uniform treatment of operator and operands.

NOTE (version 0.96) — Changes between this version and 0.8 are: further improvements to the parallel processing model and the addition and revision of some of the thread operations; a major rewrite of input/output functions and the associated stream classes; a major rewrite of telos to account for the revised meta-object protocol; a major reorganisation of the document to reflect better the structure of the language and (hopefully) to make it easier to grasp the essentials of the design on first reading.

Overview

The operator and the operands of forms are treated in a uniform manner. Here, EuLisp continues the tradition exemplified in Scheme, T, Lisp/VM and Cambridge Lisp [Fitch & Norman, 1977]. In common with other Lisp-like languages, operand expressions are evalueated and the results are passed by reference, and, in common with Scheme and some other Lisps, operator expressions are evalutated uniformly as operand expressions and the result which must be an applicable object is applied to the arguments. Functions themselves are first-class values.

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) [Bobrow *et al.*, 1988], ObjVLisp [Cointe, 1987], Oaklisp [Lang & Pearlmutter, 1988], and MicroCeyx [Chailloux *et al*, 1987]. The greatest debt of Telos is to CLOS, from which it takes the ideas of generic functions and multi-methods. In addition, most of the terminology, the names and format of the user-level macros, and the names of many of the functions in the internal protocol are inspired by CLOS. From ObjVLisp, Telos takes the principle of a reflective architecture, which emphasizes the power of metaclasses as an implementation strategy. From Oaklisp, Telos takes the idea of anonymous classes. Finally, from MicroCeyx, Telos takes the idea of a small, highly efficient kernel tightly integrated with the rest of the language. In Telos, this integration is achieved through the total merging of types with classes and message-passing through normal function application. Classes are first-class values. The class structure integrates the primitive classes describing fundamental datatypes, the defined classes and user-defined classes. The function `class-of`, given an object, returns the class of which it is a direct instance.

Modules and classes are the building blocks of both the EuLisp language and of applications written in EuLisp. The module system exists to limit access to items by name. That is, modules allow for hidden definitions. Each module defines a fresh, empty, lexical environment. This fresh environment is the top-lexical environment of that module. A defining form creates a new binding in the top lexical environment of the lexical environment in which it is evaluated.

Continuations are first-class in EuLisp, but they are not as general as in Scheme. They are weaker because they can only be used within the dynamic extent of their creation. That also implies they can only be used once. These weaker continuations are suitable for controlling simple non-local exits and form the basis of the condition system of *handlers*. Functions, too, are first-class, comprising the environment of definition (the closure of the definition) and an expression as described by Landin in ISWIM [Landin, 1966].

Dynamically scoped bindings can be created in EuLisp, but their use is much more restricted than in most Lisps up to now—except Scheme. EuLisp enforces a strong distinction between lexical bindings and dynamic bindings by requiring the use of a special form (called `dynamic-let`) for their creation and two other special forms (called `dynamic` and `dynamic-setq`) for access and update, respectively.

Multiple control threads can be created in EuLisp and orderly access to data shared between more than one control thread can be mediated by means of semaphores.

Language Structure

The EuLisp definition comprises the following items:

Level-0 — comprises all the level-0 functions, macros and special forms, which is this document minus annex B. The class system can be extended by user-defined structure classes, and generic functions.

Level-1 — extends level-0 with the functions, macros and special forms defined in annex B The class 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.

Level-2 — is yet to be elaborated.

A *level-0 function* is a (generic) function defined by this report 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 certain levels. Whenever a module depends on the operations available a given level, that dependency will be specified.

The main part of this document defines the kernel of level-0 of EuLisp. The annex A defines all the remaining classes and modules which comprise level-0. The defined name of the module providing level-0 of EuLisp is `level-0-eulisp` which imports and re-exports the modules specified in Table 1.

Table 1 — Modules comprising level-0

Module	Section
`character`	A.1
`collection`	A.2
`compare`	A.3
`convert`	A.4
`copy`	A.5
`double`	A.6
`elementary-functions`	A.7
`formatted-io`	A.8
`null`	A.9
`number`	A.10
`pair`	A.11
`semaphore`	13.2
`spint`	A.12
`stream`	A.13
`string`	A.14
`symbol`	A.15
`table`	A.16
`thread`	13.1
`vector`	A.17

Level-1 of EuLisp is defined in annex B.

1 Scope

This document 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 represenation of data and algorithms).

This document 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 an EULISP processor.

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

d) The method of reporting errors, warnings or exceptions.

e) The typographical representation of an EULISP program for human reading.

To clarify certain instances of the use of English in this document 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. Implementers of processors are urged (but not required) to satisfy the property.

2 Normative References

The following standards contain provisions, which through references in this document 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, *Information processing — ISO 7-bit coded character set for information interchange, 1983. Note: this standard is currently under revision and interested parties should reference the 1990 Draft International Standard version of ISO/IEC 646.*

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.*

BS 6154, *Method of defining — Syntactic metalanguage, 1981.*

3 Conformance Definitions

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

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

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

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

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

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

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

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

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

4 Error Definitions

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

4.1 dynamic error: An error which is detected during the execution of an EULISP program or which is a violation of the dynamic semantics of EULISP. Dynamic errors have two classifications:

a) Where a *conforming processor* is required to detect the erroneous situation or behaviour and report it. This is signified by the phrase *an error is signaled*. The condition class to be signaled 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 signaled.

b) Where a *conforming processor* might or might not detect and report upon the error. This is signified by the phrase ... *is an error*. Such errors should be detected and reported.

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

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

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

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

5 Compliance

An EULISP processor can conform at any of the three levels defined under Language Structure in the Introduction. Thus a level-0 conforming processor must support all the basic expressions, classes and class operations defined at level-0. A level-1 conforming processor must support all the basic expressions, classes, class operations and modules defined at level-1. A level-2 conforming processor must support all the classes, class operations and all of the modules defined at level-2.

The following two rules 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. A level-1 conforming program is also a level-2 conforming program. Hence, by extension, a level-0 strictly-conforming program is also a level-2 conforming program.

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

The documentation of a *conforming processor* must include:

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

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

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

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

6 Conventions

This section defines the conventions employed in this document, how definitions will be will be laid out, the typeface to be used, the meta-language used in descriptions and the naming conventions. A later section (7) contains definitions of the terms used in this document.

6.1 Layout and Typography

Both layout and fonts are used to help in the description of EULISP. A language element is defined with its name as the heading of a clause, coupled with its classification, for example:

`special-form-name`	*special form*
`function-name`	*function*
`generic-function-name`	*generic*
`generic-function-name`	*method*
`condition-name`	*super condition*

Thus a sample entry might appear as:

`sample-function`	*function*

Arguments

`arg1`: — An instance of class x_1.

`arg2`: — An instance of class x_2.

`[arg3]`: — An optional argument of class x_3.

6.1.0.1 Result

The result class.

6.1.0.2 Remarks

Some clarifying background to the actions of the function.

6.1.0.3 Examples

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

6.1.0.4 See also: Cross references to other sections or to other individual relevant language elements.

6.2 Meta-language

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

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

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. In the case of a generic function, the domain can be widened arbitrarily by the definition of new methods, similarly the range, except when the generic function was defined with a particular domain and/or range. In this case, any new methods must respect the domain and/or range of the generic function to which they are to be attached. The use of a class-descriptive name in the context of a generic function definition defines the intention of the definition, and is not necessarily a policed restriction.

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

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

6.3 Naming

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

6.1 **"`<name>`" wrapping:** By convention, the initial bindings of classes have names which begin with "`<`" and end with "`>`".

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

6.3 **"`-class`" suffix:** The name of a metaclass of a set of related classes. For example, `function-class`, which is the metaclass of `function`, `generic-function` and any of their subclasses and `condition-class` is the class of all conditions. The exception is `class` itself which is the default metaclass. The prefix should describe the general domain of the classes in question, but not necessarily any particular class in the set.

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

6.5 **hyphenation:** Function and class names made up of more than one word are hyphenated, for example: `compute-primitive-reader-using-slot-description`.

6.6 **"`p`" suffix:** A predicate function is named by a "`p`" suffix if the function or class name is not hyphenated, for instance, `consp`, and is named by a "`-p`" suffix if it is, for instance `compatible-superclass-p`.

6.7 "-ref" suffix: For each builtin or defined class, there is a field accessor named *class-name*`-ref`—where appropriate—and a corresponding field updator `(setter` *class-name*`-ref)`—also where appropriate, for example `table-ref`. This convention is broken by historical precedent for the accessors to slots of pairs, which are called `car` and `cdr`.

7 Definitions

This set of definitions, which will 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. Some of the terms defined here are redundant. Names in `courier` font refer to entities defined in the language.

7.1 accessor: An accessor is a association of a *reader* and a *writer*.

7.2 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.

7.3 applicable method list: An applicable method list is a list of all the *methods* applicable for a particular list of arguments to a generic function, sorted according to *method domain* specificity.

7.4 applicable object: An applicable object is an *instance* of `function`.

7.5 binding: A location containing a value.

7.6 binding form: Any form or any *macro expression* expanding into a form which causes the creation of *inner dynamic* or *inner lexical bindings*.

7.7 bound variable: A *variable* `x` is bound in an expression E if `x` occurs in the *scope* of a *defining* form which creates *inner-lexical binding*s or of a *binding form* occurring in E whose variable binding list contains `x`.

7.8 class: A class is an *object* which describes the structure and behavior of a set of *objects* which are its *instances*. A *class* object contains *inheritance* information and a set of *slot descriptions* which define the structure of its *instances*. A *class object* is an *instance* of a *metaclass*. All *classes* in EULISP are *subclasses* of `object`, and all *instances* of `class` are *classes*.

7.9 class precedence list: Each *class* has a linearized list of all its *superclasses*, *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`.

7.10 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.

7.11 closure: The closure of an expression E is the set of all *free variables* that occur in E.

7.12 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.

7.13 constant: A number, string, character or the empty list.

7.14 constructor: A constructor is a *function* which creates an *instance* of a particular *class*.

7.15 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. Such a function is called a continuation.

7.16 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.

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

7.18 direct instance: A direct instance of a class $class_1$ is any *object* whose *class* is $class_1$.

7.19 direct slot description: A *class's* direct *slot descriptions* are defined specifically for the *class*.

7.20 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 $class_2$.

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

7.22 discrimination: *Generic function* application consists of two parts: finding a set of *methods* applicable to the given set of arguments, and application of the *method functions* of those *methods*. The first part is called *discrimination*

or *method lookup*. *Generic functions* have an associated *function* called the *discriminating function* which implements the discrimination. Users can define new *classes* of *generic functions* which implement discrimination in new ways.

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

7.24 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.

7.25 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.

7.26 dynamically closer: If a form $F2$ is executed in the *dynamic extent* of a form $F1$ then within the *dynamic extent* of $F2$, $F2$ is dynamically closer than $F1$.

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

7.28 form: One of *constant*, *symbol*, *literal*, function call or *special form*.

7.29 free variable: A *variable* `x` is free in an expression E if `x` does not occur in the *lexical scope* of any *defining* which creates *inner-lexical binding*s or any *binding form* occurring in E whose variable binding list contains `x`.

7.30 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 `function`.

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

7.32 identifier: An identifier is the syntactic representation of a *variable*.

7.33 improper list: An improper list is a list whose final pair contains something other than the empty list in its `cdr` field.

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

7.35 indefinite scope: An access constraint, such that the *scope* of the *variable* is unlimited.

7.36 indirect instance: A indirect instance of a class $class_1$ is any *object* whose *class* is a *subclass* of $class_1$.

7.37 indirect slot description: A *slot description* is indirect for a $class_1$ if the *slot description* is defined for $class_1$, but was originally defined for another $class_2$ which is a *superclass* of $class_1$ and incorporated into $class_1$ through *inheritance*. An indirect slot description is also called an *inherited slot description*.

7.38 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 $class_2$.

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

7.40 inherited slot description: See *indirect slot description*.

7.41 initarg: A *symbol* used as a keyword in an *initlist* to mark the value of some *slot*. Used in conjunction with `make` and the other *object* initialization functions to specify initial *slot* values. An initarg may be declared for a *slot* in a class definition form using the `initarg` *slot option*.

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

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

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

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

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

7.47 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 behavior described by its *class*. An instance can be either *direct* or *indirect*.

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

7.49 lexically closer: If a *form* $F2$ occurs in a form $F1$, then any *binding*s created by $F2$ are lexically closer than those of $F1$.

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

7.51 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*.

7.52 literal: An object created by use of the `quote` operator.

7.53 macro: A macro is a function. A macro is distinguished from a function by when it is used: macro functions are only used during the syntax expansion of modules to transform expressions.

7.54 macro expression: A form whose operator names a macro expansion function.

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

7.56 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 behavior of each of the *classes* in its *domain*. Methods are not *functions* but *objects* which contain, among other information, a *function* representing the method's behavior.

7.57 method function: A *function* which implements the behavior 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 forms `call-next-method` and `next-method-p` are only valid within the lexical scope of a method function.

7.58 method lookup: See *discrimination*.

7.59 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 $domain_2$.

7.60 multi-method: A *method* which specializes on all its arguments. All methods in this definition are multi-methods.

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

7.62 object: Any entity that can be bound to a *variable*—including entities from outside LISP's memory. Every object is an *instance* of some *class*.

7.63 proper list: A proper list is a list whose final pair contains the empty list in its `cdr` field, or is just the empty list.

7.64 reader: A reader is a *function* of one argument which returns the value of a particular *slot* in *instances* of a particular *class*.

7.65 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.

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

7.67 self-instantiated class: A *class* which is an *instance* of itself. In EULISP, `class` is the only example of a self-instantiated class.

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

7.69 shadow: If two entities are created for which the same means of reference is used, and either the form creating one occurs lexically in the form creating the other (where the means of reference has *lexical scope*) or the form creating one is executed in the dynamic extent of the form creating the other (where the means of reference has *dynamic scope*), then the outer entity is shadowed by the inner one.

7.70 domain: A domain is a list of *classes* derived from a list of arguments, or the list of *classes* for which a *method* is *applicable*.

7.71 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 object* associated with the *class* of the *object*. When we refer to the *structure* of an *object*, this usually means its set of *slots*.

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

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

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

any form which must be evaluated in a specific way.

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

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

7.77 subclass: The behavior 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*.

7.78 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*.

7.79 symbol: A symbol is a data structure, often used to represent an *identifier*.

7.80 textual slot description: A list of alternating keywords and values (starting with a keyword) which represents a not-yet-created *slot description* during class initialisation.

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

7.82 top lexical: Top lexical bindings are created in the *top lexical* environment of a module by `defclass`, `defcondition`, `defconstant`, `defgeneric`, `defmacro`, `defmetaclass`,`defstruct` and `defun`. All these bindings are immutable. `deflocal` 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.

7.83 variable: A variable denotes a *binding* and is a means to reference the value stored in the *binding*.

7.84 writer: A writer is a *function* of two arguments which changes the value of a particular *slot* in *instances* of a particular *class*.

8 Syntax

Case is distinguished in each of characters, strings and identifiers, so that `variable-name` and `Variable-name` are different, but where a character is used in a positional number representation (e.g. `\#x3Ad`) the case is ignored. Thus, case is also significant in this document and, as will be observed later, all the special form and standard function names are lower case. In this section, and throughout this document, the names for individual character glyphs are those used in ISO/IEC DIS 646:1990.

The minimal character set to support EULISP is defined in Table 2. The language as defined in this document 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: *digit*, *upper*, *lower*, *other*, *special* and *alphabetic*.

The syntax of the classes of objects that can be read by EULISP is defined in the section of this document corresponding to the class: `<character>`(A.1), `<double-float>`(**??**), `<null>`(A.9), `<pair>`(A.11), `<spint>`(A.12), `<string>`(A.14), `<symbol>`(A.15) and `<vector>`(A.17). The syntax for identifiers corresponds to that for symbols.

8.1 Whitespace and Comments

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

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

9 Modules

The EULISP module scheme has several influences: the existing Le-Lisp implementation, the model defined in the functional language Haskell, the ML module system [MacQueen, 1984], the `make-environment` module mechanism used in C-Scheme and T's `locale`s.

The module mechanism provides a means of abstraction and a means of security for programs in a complementary style to that provided by the object system. Indeed, although objects do support data abstraction, they do not support all forms of information hiding. For this reason it is important to provide a mechanism offering the complete hiding of names. A module defines a mapping between a set of names and the bindings of those names in the imported module. Most such bindings are immutable. The exception are those bindings created by `deflocal` which may be modified by the defining and importing modules. There are no implicit imports into a module, and not even the special forms are available in a module that imports nothing. A module exports nothing by default.

A module definition creates two, new, empty lexical environments—the internal and the external top-lexical environments of the module. All the bindings in the module body are stored in the internal top-lexical along with those bindings shared (by importation) with other modules. The external top-lexical shares those bindings from the internal top-lexical that are exported and also those of all the exposed modules (modules which are exported but not imported). 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 representation of the module environment is implementation-defined. The body of a module definition comprises an import directive followed by a syntax directive and a sequence of definitions, expressions and export directives. The processing of each of these is now discussed in detail.

9.1 Imports

The import directive is expressed in terms of module names and the filters `except`, `only` and `rename`. The syntax of import specifications is given in Table 3. An *import-spec* is a sequence of *module-name*s and or *module-directive*s and denotes the union of all the names generated by each element of the sequence.

In processing import directives, every name should be thought of as a pair of (*module-name local-name*) coupled with some attributes (mutable, immutable, syntax, value). Intuitively, a namelist of module-name/local-name pairs is generated by giving the module name and then filtered by `except`, `only` and `rename`. In addition, all names with a `syntax` attribute are filtered out because syntax functions can have no use at execution time. In an import directive, when a namelist has been filtered, the names are regarded as being defined in the internal top-lexical environment of the module into which they have been imported. Should any two instances of *local-name* have different *module-name*s, then there is a name clash which is a static error. Elements of an *import-spec* are interpreted as follows:

except — Filters the names from each *module-directive* discarding those specified and keeping all other names. The `except` directive is convenient when almost all of the

Table 2 — Minimal character set

digit	::=	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
upper	::=	`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	::=	`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`
other	::=	`!` | `$` | `%` | `&` | `*` | `/` | `:` | `<` | `=` | `>` | `?` | `^` | `_` | `~` | `[` | `]` | `{` | `}` | `+` | `-` | `@` | `.`
special	::=	`;` | `'` | `,` | `\` | `"` | `#` | `(` | `)` | `` ` ``
		alert | *backspace* | *delete* | *formfeed* | *linefeed* | *newline* | *return* | *space* | *tab* | *vertical-tab*
alphabetic	::=	*upper* | *lower*

names exported by a module are required, since it is only necessary to name those few that are not wanted to exclude them.

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

only — Filters the names from each *module-directive* 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 requied, since it is only necessary to name those that are wanted to include them.

rename — Filters the names from each *module-directive* 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.

9.2 Syntax

The syntax section defines the expansion functions for the body of the module. This section comprises an import directive for access to expanders defined in other modules and a sequence of definitions. The import directive is processed as described above except that all names which do *not* have a `syntax` attribute are filtered out. The body of the syntax section is expanded according to the syntax environment defined by the import directive of the syntax section. All the resulting functions are added to the syntax environment and the body of the module is then expanded according to that environment.

NOTE (version 0.96) — Currently the semantics of syntax expansion are unsatisfactory because expanders are not opaque to the user. That is to say, the resulting expansion can contain references to bindings imported into the syntax module and those modules must also be imported explicitly by the user of the macro using the same binding renamings as the syntax module. Neither is there a requirement for hygienic expansion, so expander and user bound variables may interfere with one another. It is intended that the next version of this document should specify a syntax expansion scheme that is opaque to the user.

The basic expansion mechanism examines each form in the module body, applying the following process:

a) If the form is not a list, the result is the original form.

b) If the form is a list and the operator is an identifier which is bound in the syntax environment then the associated expansion function is called on a list of the (unevaluated) operands of the form. The expansion process is applied to the resulting form.

c) If the identifier names a special form then the form specific expansion function is invoked.

d) If the form is a list the expansion process is applied to the operator and the operands of the form. The result is a form containing the results of the expansion of the operator and the operands in the order corresponding to the order of the operator and the operands in the original form. The expansion process is applied to the resulting form.

The result of each step of the expansion process is a form containing references to the bindings visible from the module in which the expansion function is defined.

9.3 Exports

The export directive is expressed either in terms of bindings names, using `export` or `export-syntax`, or module names using `expose`.. The syntax of export specifications is given in Table 3.

Processing export directives employs the same model as for imports, namely, a module-name/local-name pair with the same filtering operations. When the namelist has been filtered, the names are added to the set of exportations of this module. It is the union of all the export directives in the body of a module defines the externally visible top-lexical environment of the module. Should any two instances of *local-name* have different *module-name*s, then there is a name clash, which is a static error. Note that the external top-lexical environment might not be a subset of the internal top-lexical environment because the external one can reference modules which have not been imported.

The sequence of *export-spec*s in the module body is treated as the union of all the names generated by each *export-spec*. It is a static error if any name occurs more than once. Each *export-spec* is interpreted as follows:

export — Each of the names appearing in the `export` form is added to the external top-lexical environment of the module.

export-syntax — Each of the names appearing in the `export` form is added to the external top-lexical environment of the module with the syntax attribute set.

expose — Processes each of the *module-directive*s appearing in the `expose` form following the rules for *import-spec* and adds the resulting set of names to the external top-lexical environment of the module.

Figure 1 — Example of import and export directives

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

  (syntax
     syntax-module-1                                 ;;all of the module syntax-module-1
    ((rename ((syntax-a syntax-b)) syntax-module-2)  ;;rename the binding of syntax-a from syntax-module-2 to
syntax-b

     (rename ((syntax-c syntax-a)) syntax-module-3))  ;;now rename syntax-c from syntax-module-3 as syntax-a
    ...
  )
  ...
  (expose module-5                                   ;;export all of module-5
          (except (function-e) module-6))            ;;export all of module-6 except function-e
  ...
  (export local-definition-1                         ;;but just three bindngs from this module
          local-definition-2
          local-definition-3)
  ...
)
```

9.4 Definitions and Expressions

Definitions in a module only contain unqualified names—that is, *local-name*s, using the above terminology. All top-lexical module bindings are only ever created once and are shared with all modules that import the module creating the bindings. Only top-lexical bindings created by `deflocal` are mutable and it is an error 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. 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.

9.5 Module Processing

The following steps summarize the module definition process:

Import Processing — Module import clauses either specify the importation of a module in its entirety or the selective importation of specified bindings from a module. For each import specification, the originating module must exist and, in the case of selective specifications, the named binding(s) must be exported from it. Each import specification contributes a set of bindings to the top-lexical environment of the module being defined. Each such binding is checked for name-conflict, since no two imported names can be the same. Note that mutually referential modules are not possible because of the definition before use requirement. Hence, the importation dependencies form a DAG.

Syntax Expansion — The `syntax` section specifies the modules required for syntax expansion and any locally defined syntax. The body of the module is expanded according to the operators defined in the syntax environment constructed from the syntax import directive and the local definitions.

Environment Construction — All the defined variables are collected and added to the module's top-lexical environment.

Export Processing — The exportations are collected and the set of exported names is constructed.

Static Analysis — The expanded body of the module is analysed. It is a static error, if a variable in the body does not have a binding in the top-lexical environment.

Initialization — The module is initialized by evaluating the forms in the body in the order they appear.

9.6 Module Definition

9.6.1 `defmodule` *syntax*

9.6.1.1 Syntax

The syntax of the text of a module is given in Table 3.

9.6.1.2 Arguments

module-name: A symbol used to name the module.

import-spec: An expression specifying the modules on which the execution of this module depends and how their exports are to be named for reference in this module.

Table 3 — Module syntax

import-spec	::=	`(`*module-directive**`)`
syntax-spec	::=	`()` | `(syntax` *import-spec defmacro**`)`
export-spec	::=	*export* | *export-syntax* | *expose*
export	::=	`(export` *name**`)`
export-syntax	::=	`(export-syntax` *name**`)`
expose	::=	`(expose` *module-directive**`)`
module-directive	::=	*module-name* | *module-filter*
module-filter	::=	*except* | *only* | *rename*
except	::=	`(except (`*name**`)` *module-directive*$^+$`)`
only	::=	`(only (`*name**`)` *module-directive*$^+$`)`
rename	::=	`(rename ((`*old-name new-name*`)`*`)` *module-directive*$^+$`)`
module-expression	::=	*export-spec* | *level-0-expression* | *definition* | `(progn` *expression*`)`
definition	::=	*level-0-definition*$_{\{\texttt{defmodule}\}}$

syntax-spec: An expression specifying the modules on which the expansion of this module depends and how their exports are to be named for reference in this module

module-expression*: A sequence of definitions, expressions and export specifications.

9.6.1.3 Remarks

The `defmodule` form defines a module named by *module-name* and stores a module object in the module binding environment under the name *module-name*.

9.6.1.4 Examples

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

10 Objects

In EULISP, every object in the system has a specific class. Classes themselves are first-class objects. In this respect EULISP differs from statically-typed object-oriented languages such as C++ and μCEYX. The EULISP object system is called TELOS.

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 behavior of its instances. *Structure* is the information contained in the class's instances and *behavior* is the way in which the instances are treated by the protocol defined for them.

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 behavior depends on the classes of their arguments. The particular class-specific behavior 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.

It may be possible in an implementation to acquire a pointer that does not correspond to any EULISP object. It is an error to pass such a pointer to a function. The default domain for arguments of functions is `<object>`.

10.1 Creating and Initializing Objects

Objects can be created calling

— constructors (predefined or user defined) or

— `make`, the general constructor function.

The general constructor `make` creates a new object calling `allocate` and initializes it calling `initialize`. `allocate` discriminates on the class, while `initialize` discriminates on the new object. Both `initialize` and `allocate`, as well as more specific constructors, are described in more detail in section **??**. The default `allocate` method creates a new uninitialized object, that is, all slots are unbound. The default `initialize` method (see below) is applicable to any object, although there are also more specific methods for classes, slot descriptors, generic functions and methods.

10.1.1 `initialize` *generic function*

10.1.1.1 Generic Arguments

(*object* `<object>`): The object to initialize.

(*initlist* `<list>`): The list of initialization arguments.

10.1.1.2 Result

The initialized object.

10.1.1.3 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`.

10.1.2 `initialize` *method*

10.1.2.1 Specialized Arguments

(*object* `<object>`): The object to initialize.

(*initlist* `<list>`): The list of initialization arguments.

10.1.2.2 Result

The initialized object.

10.1.2.3 Remarks

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

a) Check if the supplied initargs are legal and signal an error otherwise. Legal initargs are those specified in the class definition directly or inherited from a superclass. Initargs may be slot-names or other symbols.

b) Initialize the slots of the object according to the initarg, if supplied, or according to the initfunction stored in the slot-description followed by calling the anonymous writer stored in the slot-description. The initfunction may return "unbound" if no `initform` has been specified directly in the class definition or none was inherited from a superclass.

Legal initargs which do not name a slot are ignored by the default `initialize` method. The default method can be specialized by calling `call-next-method` from more specific `initialize` methods.

10.1.2.4 See also: `initialize` methods for classes, slot descriptors, generic functions and methods, `make`, `allocate`.

10.2 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.

At the metalevel there is a protocol which allows user defined methods for the computation of special accessors, for example, those checking the type of a value when storing a new one.

Table 4 — `generic-prin` output syntax

prin-datum	::=	*prin-character* | *number* | *prin-string* | *structure* | *prin-symbol*
prin-character	::=	*any-character*
prin-string	::=	*prin-string-char**
prin-symbol	::=	*prin-string*
prin-string-char	::=	*any-character*

10.3 External Representation

10.3.1 `generic-prin` *generic function*

10.3.1.1 Generic Arguments

(*object* `<object>`): An object to be output on *stream*.

(*stream* `<stream>`): The stream on which *object* is to be output.

10.3.1.2 Result

The object supplied as the first argument.

10.3.1.3 Remarks

The individual methods for specific classes define the format of the output. The representation produced by `generic-prin`, may be more convenient for human reading, but is not guaranteed to by syntactically correct input for a EuLisp processor. Specifically, `generic-prin` will not output escaping information in characters, strings or symbols. This is summarised in the grammar in Table 4.

10.3.2 `generic-write` *generic function*

10.3.2.1 Generic Arguments

(*object* `<object>`): An object to be output on *stream*.

(*stream* `<stream>`): The stream on which *object* is to be output.

10.3.2.2 Result

The object supplied as the first argument.

Table 5 — `generic-write` output syntax

datum	::=	*character* | *number* | *string* | *structure* | *symbol*
structure	::=	*list* | *vector*

10.3.2.3 Remarks

The individual methods for specific classes define the format of the output. The representation produced by `generic-write` is guaranteed to be syntactically correct input for a EuLisp processor and will result in an object `equal` to the original entity. The syntax of objects output by `generic-write` is given in Table 5.

11 Classes and Slot Desciptions

A class describes a set of objects, called its *instances*, in the problem domain. Classes define the structure of their instances through a set of slots which each instance contains. Classes also define the behavior of their instances through the methods which specialize on them.

Inheritance is implemented through classes. Each class has a *class precedence list*, a linearized list of all the class's superclasses, which defines the classes from which the class inherits structure and behavior. Slots and methods defined for a class will also be defined for its subclasses but a subclass may specialize them.

In EULISP, classes are first-class objects and are instances of some specific class. These classes of classes are called *metaclasses*. Extensions, such as multiple inheritance, support for the `change-class` functionality of CLOS, and persistent objects can be supported through metaclasses. In addition, metaclasses can provide new kinds of classes with reduced power but increased efficiency; the class `<structure-class>` is an example.

Classes are defined using the `defstruct` (11.4.1), `defcondition` (**??**) and `defclass` (**??**) defining forms. New metaclasses are defined using `defmetaclass` (**??**).

11.1 Inheritance

The structure and behaviour defined for a class is *inherited* by all of its subclasses. 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 for direct and indirect instances of the class.

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

Metaclasses control the structure and behaviour of their instances and the representation of their metainstances. It might not be possible to form a subclass link between two classes of different metaclasses. The meta-object protocol (MOP) provides means to control compatibility between classes with respect to the subclass relationhsip.

11.2 Slot Descriptions

The components of an object are called its *slots*. Each slot of an object is defined by its class. It is represented by a *slot description object*, which defines where the slot is to be stored, how it can be accessed, and its default value. At level-1 of EULISP and above the slot description mechanism is specializable and extensible. New slot description classes to support extensions such as the facets found in many knowledge representation languages, multi-valued slots, typed slots, and slots whose values are not stored in the instance. Slots are defined within a `defstruct`, `defcondition` or `defclass` defining form. New slot description classes are defined by `defclass`.

11.3 System Defined Classes

The basic classes of EULISP are elements of the object system class hierarchy, which is shown in Figure 2. 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.

Figure 2 — Level-0 initial class hierarchy

```
<object>
     <character>
     <condition>
     <function>
        <continuation>
        <generic-function>
     <method>
     <null>
     <number>
        <integer>
        <ratio>
        <float>
        <complex>
     <pair>
     <semaphore>
     <stream>
     <string>
     <structure>
     <symbol>
     <table>
     <thread>
     <vector>
```

The root of the inheritance hierarchy is the class `<object>`. `<object>` defines the basic methods for initialisation and external representation of objects. In this definition, unless otherwise specified, classes declared to be subclasses of other classes may be indirect subclasses. Furthermore, unless otherwise specified, all objects declared to be of a certain class may be indirect instances of that class.

11.4 Defining Classes

11.4.1 `defstruct` — *defining form*

11.4.1.1 Syntax

`(defstruct` *class-name superclass* `(`*slot-description**`)` *class-option**`)`

11.4.1.2 Arguments

class-name: A symbol naming a binding to be initialised with the new structure class.

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

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

class-option*: A sequence of keys and values (see below).

11.4.1.3 Remarks

`defstruct` creates a new structure class. The first argument is the name to which the new class will be bound. The second is identifier which names a variable to which the superclass is bound. The list of *slot-description*s is described below. Finally, a *class-option* is a identifier followed by a corresponding value, which, taken together, apply to the class as a whole. The syntax of `defstruct` is defined in Table 6.

The *slot-option*s are interpreted as follows:

`initform` *form*: The value of this option is a form, which is evaluated as the default value of the slot, to be used if no initarg is defined for the slot or given to a call to `make`. The form is evaluated in the lexical environment of the call to `defstruct` and the dynamic environment of the call to `make`. The form is evaluated each time `make` is called and the default value is called for. The order of evaluation of the initforms in all the slots is determined by `initialize`. This option must only be specified once for a particular slot.

`reader` *symbol*: The value is the identifier of the variable to which the reader function will be bound. The reader function is a means to access the slot. The reader function is a function of one argument, which should be an instance of the new class. No writer function is automatically bound with this option. This option can be specified more than once for a slot, creating several readers. It is an error to specify the same reader, writer, or accessor name for two different slots.

`writer` *symbol*: The value is the identifier of the variable to which the writer function will be bound. The writer function is a means to change the slot value. The creation of the writer is analogous to that of the reader function. This option can be specified more than once for a slot. It is an error to specify the same reader, writer, or accessor name for two different slots.

`accessor` *symbol*: The value is the identifier of the variable to which the reader function will be bound. In addition, the use of this *slot-option* causes that the writer function is associated to the reader *via* the `setter` mechanism. This option can be specified more than once for a slot. It is an error to specify the same reader, writer, or accessor name for two different slots.

The class options are interpreted as follows:

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

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

Table 6 — `defstruct` syntax

class-name	::=	*identifier*
superclass	::=	{`<structure>` *or the name of one of its subclasses*}
slot-description	::=	*slot-name* | (*slot-name slot-option***)
slot-name	::=	*identifier*
slot-option	::=	`initarg` *identifier* |
		`initform` *form* |
		`reader` *reader-name* |
		`writer` *writer-name* |
		`accessor` *reader-name*
reader-name	::=	*identifier*
writer-name	::=	*identifier*
class-option	::=	`constructor` *constructor-spec* |
		`predicate` *predicate-name*
constructor-spec	::=	(*constructor-name init-option***)
constructor-name	::=	*identifier*
predicate-name	::=	*identifier*

`predicate` *symbol*: Creates a predicate function for the new class. The predicate specification gives the name to which the predicate function will be bound. This option may be specified any number of times for a class.

11.4.2 `defclass` — *defining form*

11.4.2.1 Syntax

(`defclass` *class-name* (*superclass**) (*slot-description**) *class-option**)

11.4.2.2 Arguments

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

(*superclass**): A list of a single symbol naming the binding of the class to be used as the superclass of the new class. Multiple superclasses can be specified at level-1 (see section B.1.1).

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

*class-option**: A sequence of keys and values (see below).

11.4.2.3 Remarks

This defining form defines a new class. The resulting class will be bound to *class-name*. The second argument is a list containing the superclass. If this list is empty, the superclass is `<object>`. The third argument is a list of slot-descriptions, the format of which is an extension of that for `defstruct`. The remaining arguments are class options. The syntax of `defclass` is given in Table 7. All the slot options and class options are exactly the same way as for `defstruct` (11.4.1).

11.5 Creating Objects

11.5.1 `make` — *function*

11.5.1.1 Arguments

class: The class of the object to create.

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

11.5.1.2 Result

An instance of *class*.

11.5.1.3 Remarks

The general constructor `make` creates a new object calling `allocate` and initializes it by calling `initialize`. `allocate` discriminates on the class, while `initialize` discriminates on the new object.

11.5.2 `<telos-condition>` — *condition*

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

Table 7 — `defclass` syntax (level-0)

class-name	::=	*identifier*
superclass	::=	{`<object>` *or the name of one of its subclasses*}
slot-description	::=	*slot-name* | (*slot-name slot-option**)
slot-name	::=	*identifier*
level-0-slot-option	::=	`initarg` *identifier* |
		`initform` *form* |
		`reader` *reader-name* |
		`writer` *writer-name* |
		`accessor` *reader-name* |
level-0-class-option	::=	`constructor` *constructor-spec*
		`predicate` *predicate-name*

12 Generic Functions and Methods

A generic function is a function whose application behaviour is determined by the classes of its arguments. Each potential behavior is defined by a method, which specifies a domain of classes for which it is applicable. r-na'J¨program's protocol is a set of generic functions and the relationships between them.

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*.

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, `generic-lambda`, which creates an anonymous generic function, and `generic-labels`, which creates a named generic function in the inner-lexical environment. These forms are described in detail later in this section.

A method describes the application behaviour of a generic function for a particular sequence of classes, called the method's *domain*. Methods are not functions themselves, but objects attached to a generic function containing closed expressions.

Like slots, methods are inherited from the superclass(es) of an object. That is, if a method is applicable for a class C_1, it is also applicable for all of C_1's subclasses as well. New methods may also be defined for these subclasses, and these methods are said to be *more specific* than the methods defined on the super classes. However, the more general methods are accessible from the more specific through the `call-next-method` form. Thus, behavior can be inherited and extended in subclasses.

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

12.1 Defining Generic Functions and Methods

12.1.1 `defgeneric` *defining form*

12.1.1.1 Syntax

`(defgeneric` *gf-name gen-lambda-list level-0-init-option** `)`

12.1.1.2 Arguments

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

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

*init-option**: Options as specified below.

12.1.1.3 Remarks

This defining form defines a new generic function. The resulting generic function will be bound to *gf-name*. The second argument is the formal parameter list. An error is signaled (condition: `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. In addition, an error is signaled (condition: `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. 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 an identifier followed by a corresponding value. The syntax of `defgeneric` is given in Table 8:

The *init-option* is:

`method` *method-spec*: This option is followed by a method description. A 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.

12.1.1.4 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 subclass of `<class-a>`.

```
(defgeneric gf-0 (arg1 (arg2 <class-a>))
  method (((m1-arg1 <class-b>) (m1-arg2 <class-c>)) ...)
```

Table 8 — `defgeneric` syntax (level-0)

gf-name	::=	*identifier* | `(setter` *identifier*`)` | `(converter` *identifier*`)`
gen-lambda-list	::=	*spec-lambda-list*
level-0-init-option	::=	`method` *method-description*
method-description	::=	`(`*spec-lambda-list form**`)`
spec-lambda-list	::=	`(`*spec-variable** `[.` *variable*`])`
spec-variable	::=	`(`*variable class*`)` | *variable*
class	::=	*class-name*

```
  method (((m2-arg1 <class-d>) (m2-arg2 <class-e>)) ...)
  method (((m3-arg1 <class-f>) (m3-arg2 <class-g>)) ...)
)
```

12.1.1.5 See also: `add-method`.

12.1.2 `defmethod` *macro*

12.1.2.1 Syntax

`(defmethod` *gf-name spec-lambda-list form**`)`
or
`(defmethod (setter` *identifier*`)` *spec-lambda-list form**`)`
or
`(defmethod (converter` *identifier*`)` *spec-lambda-list form**`)`

12.1.2.2 Remarks

This macro 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* or convertor function associated with *class*. In the former case, if the specialized-lambda-list is not congruent with that of the generic function, an error is signaled (condition: `non-congruent-lambda-lists`). In addition, an error is signaled (condition: `incompatible-method-domain`) if the method's specialized lambda list would widen the domain of the generic function.

12.1.3 `no-applicable-method` *telos-condition*

12.1.3.1 Init-options

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

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

12.1.3.2 Remarks

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

12.1.4 `incompatible-method-domain` *telos-condition*

12.1.4.1 Init-options

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

`method` *method*: The method to be added.

12.1.4.2 Remarks

Signalled by `add-method` if the domain of the method would not be a subdomain of the generic functions domain.

12.1.5 `non-congruent-lambda-lists` *telos-condition*

12.1.5.1 Init-options

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

`method` *method*: The method to be added.

12.1.5.2 Remarks

Signalled by `add-method` if the lambda list of the method is not congruent to that of the generic function.

12.2 Specializing Methods

The following operators (`call-next-method` and `next-method-p`) are used to specialize more general methods by calling them and perhaps performing some additional computations before and/or after calling the next method. 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.

12.2.1 `call-next-method` *special form*

12.2.1.1 Syntax

`(call-next-method)`

12.2.1.2 Result

The result of calling the next most specific applicable method.

12.2.1.3 Remarks

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

12.2.2 `no-next-method` *telos-condition*

12.2.2.1 Init-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.

12.2.2.2 Remarks

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

12.2.3 `next-method-p` *special form*

12.2.3.1 Syntax

```
(next-method-p)
```

12.2.3.2 Result

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

13 Threads and Semaphores

The basic elements of parallel processing in EULISP are processes and mutal exclusion, which are provided by the classes `thread` and `semaphore` respectively.

A thread is an abstract data type protecting some implementation-defined data. A thread is allocated and initialised like all other object, by using `make`. This function is called the initial function and is where execution starts the first time the thread is dispatched by the scheduler. The discussion identifies four states of a thread: new, running, aborted and finished. These are for conceptual purposes only and are not distinguishable in practice. The transitions between these states are summarised in figure 3. The initial state of a thread is new. The union of the two final states is known as *determined*.

A thread is made available for dispatch by starting it, using the function `thread-start`, which changes its state to running. From running a thread becomes either finished or aborted. When a thread has 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, then `thread-value` will signal the condition on the thread accessing the value. Note that `thread-value` blocks until the specified thread has been determined.

Access to shared resources or an undetermined thread while a thread is running may cause it to become blocked. Thus, a thread may be blocked on a semaphore, input-output or `thread-value`. In each of these cases, `thread-resechedule` is called to allow another thread to execute. This function may also be called voluntarily. A thread may be unblocked by some other thread executing `close-semaphore` on the semaphore on which the first thread is blocked. Thus, the call to `thread-reschedule` returns.

The actions of a thread can be influenced externally by `signal`. This function registers a condition to be signalled when the specified thread is rescheduled for execution. The condition must be an instance of a subclass of `thread-condition`. Conditions are delivered to the thread in order of receipt. A `signal` on a determined thread has no effect on either the signalled or signalling thread. See also section 14.

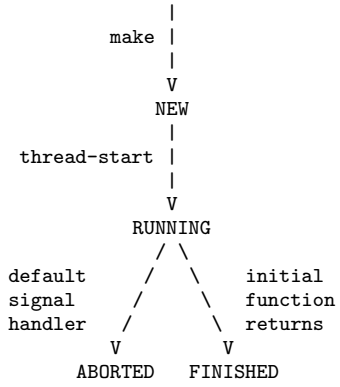
A semaphore is an abstract data type protecting a binary value; call these values zero and one. The operations on a sempahore are `close-semaphore` and `open-semaphore`. The `open-semaphore` operations sets the value to zero if it is one, or changes the state of the thread executing the operation to `blocked`, if the value is zero and reschedules the thread. The `close-semaphore` operation sets the value to one and if there are any threads blocked on this semaphore, one will be selected and unblocked.

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

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

b) A conforming program must assume the possibility of concurrent execution of threads and will have the same

Figure 3 — State diagram for threads



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) An error is signaled (condition: `wrong-thread`), if a continuation is called from a thread other than the one on which it was created. That is to say, a continuation must only be called from within its dynamic extent.

e) The lexical environment (inner and top) associated with the initial function is shared by the thread, as is the top-dynamic environment, but each thread has a distinct inner-dynamic environment.

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 signaled 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.

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 semaphores—both the thread primitives and semaphores 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.

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 semaphore, the other thread is guaranteed to have the opportunity to change the data. If it is not using a semaphore, the fairness guarantee ensures that in the same scenario the following loop will exit eventually:

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

13.1 Threads

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

13.1.1 `<thread>` *class*

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

13.1.1.1 Init-options

`initfn` *initial-function*: an instance of `function` which will be called when the resulting thread is started.

`size` *integer*: a positive integer specifying, in implementationdefined units, the size of the thread to be allocated.

13.1.2 `threadp` *function*

13.1.2.1 Arguments

obj: object to examine.

13.1.2.2 Result

The supplied argumentMultillan instance of `thread`, otherwise `()`.

13.1.3 `thread-reschedule` *function*

This function takes no arguments.

13.1.3.1 Result

The result is `()`.

13.1.3.2 Remarks

This function is called for side-effect only and causes the thread which executes it to become blocked. If conditions are pending on the thread when `thread-reschedule` continues, one is selected arbitrarily and signalled. No further pending conditions will be signalled until the handler processing the condition has exited. Upon exit from the handler the thread is rescheduled again.

13.1.3.3 See also: `thread-value`, `thread-signal`.

13.1.4 `current-thread` *function*

This function takes no arguments.

13.1.4.1 Result

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

13.1.5 `thread-start` *function*

13.1.5.1 Arguments

thread: the thread to be started, which must be in state `new`. If not an error is signaled (condition: `old-thread`).

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

13.1.5.2 Result

The thread which was was supplied as the first argument.

13.1.5.3 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.

13.1.6 `thread-value` *function*

13.1.6.1 Arguments

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

13.1.6.2 Result

The result of the initial function applied to the arguments passed from `thread-start`. However, if a condition is signalled on *thread* which is handled by the default handler the condition will now be signalled on the thread executing `thread-value`.

13.1.6.3 Remarks

If *thread* is not determined, the thread executing `thread-value` is blocked until *thread* is determined.

13.1.6.4 See also: `thread-reschedule`, `thread-signal`.

13.1.6.5 Result

The result is `()`

13.1.6.6 Remarks

Registers the specified condition, or, by default, an instance of `thread-condition`, to be signalled when *thread* is rescheduled for execution. A `thread-signal` on a determined thread has no effect on either the signalled or signalling thread.

13.1.6.7 See also: `thread-reschedule`, `thread-value`.

13.1.7 `wait` *method*

13.1.7.1 Specialized Arguments

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

(*timeout* `<object>`): The timeout period.

13.1.7.2 Result

Returns *thread* if *thread* is *determined*.

13.1.7.3 See also: `wait`.

13.1.8 `thread-condition` *condition*

13.1.8.1 Init-options

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

13.1.8.2 Remarks

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

13.1.9 `wrong-thread` *thread-condition*

13.1.9.1 Init-options

`continuation` *continuation*: A continuation.

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

13.1.9.2 Remarks

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

13.1.10 `old-thread` *thread-condition*

13.1.10.1 Init-options

`thread` *thread*: A thread.

13.1.10.2 Remarks

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

13.1.11 `generic-prin` *method*

13.1.12 `generic-write` *method*

13.1.12.1 Specialized Arguments

(*thread* `<thread>`): The thread to be ouptut on stream

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

13.1.12.2 Result

The thread supplied as the first argument.

13.1.12.3 Remarks

Outputs the external representation of thread on stream. The external representation of thread is processor-defined.

13.2 Semaphores

The defined name of this module is `semaphore`.

13.2.1 `<semaphore>` *class*

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

13.2.2 `semaphorep` *function*

13.2.2.1 Arguments

obj: object to examine.

13.2.2.2 Result

The supplied argument if it is an instance of `semaphore`, otherwise `()`.

13.2.3 `open-semaphore` *function*

13.2.3.1 Arguments

semaphore: the semaphore to be opened.

13.2.3.2 Result

The semaphore supplied as argument.

13.2.3.3 Remarks

Set the value of semaphore to zero if it is one or block the thread executing `open-semaphore` if the value is to zero and call `thread-reschedule`. This operation is atomic. On being unblocked the call to `open-semaphore` will continue by attempting to open the semaphore.

13.2.3.4 See also: `close-semaphore`

13.2.4 `close-semaphore` *function*

13.2.4.1 Arguments

semaphore: the semaphore to be closed.

13.2.4.2 Result

The semaphore supplied as argument.

13.2.4.3 Remarks

Set the value of semaphore to one and if there are any threads blocked on this semaphore, select one and unblock it. This operation is atomic. That thread may then attempt to open the semaphore again.

13.2.4.4 See also: `open-semaphore`

13.2.5 `generic-prin` *method*

13.2.5.1 Arguments

semaphore: the semaphore to be ouptut on stream

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

13.2.5.2 Result

The thread supplied as the first argument.

13.2.5.3 Remarks

Outputs the external representation of semaphore on stream. The external representation of semaphore is processor-defined.

13.2.6 `generic-write` *method*

13.2.6.1 Arguments

semaphore: the semaphore to be ouptut on stream

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

13.2.6.2 Result

The thread supplied as the first argument.

13.2.6.3 Remarks

Outputs the external representation of semaphore on stream. The external representation of semaphore is processor-defined.

14 Conditions

The defined name of this module is `condition`.

The condition system owes much to the Common Lisp error system [Pitman, 1988] and to the Standard ML exception mechanism. It is a simplification of the former and an extension of the latter. Following standard practice, this document has defined the behaviour 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 signaled. An error is a kind of condition—error and condition are also used as terms for the objects that describe exceptional situations. A condition can be signaled continuably or non-continuably.

These two classes are characterised as follows:

a) A condition might be signaled 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 signaled to indicate divergence of an operation. A continuable condition should be signaled when there is a strategy for recovery from the condition.

b) Alternatively, a condition might be signaled when some catastrophic situation is recognised, such as the memory manager being unable to allocate more memory or unable to recover sufficent memory from that already allocated. a non-continuable condition should be signaled when there is no reasonable way to resume processing.

A condition class is defined with `defcondition` or `defclass`. The definition of a condition causes the creation of a new class of condition, including a new condition class constructor. A condition is signaled using the function `signal`, which takes an instance of a condition and a resume continuation or the empty list, signifying a non-continuable condition, as arguments. A condition can be handled using the special form `with-handler`, which takes a function—the handler function—and a sequence of forms to be protected. The initial condition class hierarchy is shown in Figure 4.

Figure 4 — Level-0 initial condition class hierarchy

```
<condition>
  <execution-condition>
    <invalid-operator>
    <bad-apply-argument>
    <cannot-update-setter>
    <no-setter>
    <improper-unquote-splice>
  <environment-condition>
  <arithmetic-condition>
    <division-by-zero>
  <conversion-condition>
    <no-converter>
  <stream-condition>
  <syntax-error>
  <thread-condition>
  <telos-condition>
    <no-next-method>
    <non-congruent-lambda-lists>
    <incompatible-method-signature>
    <no-applicable-method>
```

14.0.1 `<condition>` *class*

14.0.1.1 Init-options

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

14.0.1.2 Remarks

The class which is the superclass of all condition classes.

14.0.2 `execution-condition` *condition*

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

14.0.3 `environment-condition` *condition*

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

14.1 Condition Handling

Conditions are handled with a function called a *handler*. Handlers are established dynamically and have dynamic scope and extent. Thus, when a condition is signaled, the processor will call the dynamically closest handler. Note that it is the first handler accepting to process the condition that is used and not necessarily the most specific. Handlers are established by the special form `with-handler`.

14.1.1 `signal` *function*

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

14.1.1.1 Arguments

condition: The condition to be signaled.

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

[*thread*]: If this argument is not supplied, the condition is signalled on the thread which called `signal`, otherwise, *thread* indicates the thread on which *condition* is to be signalled.

14.1.1.2 Result

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

14.1.1.3 Remarks

If the third argumentMis not supplied, `signal` calls the dynamically closest handler with *condition*—the condition being signaled—and either *continuation* or `()`. If the second argument is a subclass of `function`, that 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 signaled 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 when *thread* is rescheduled for execution. The condition must be an instance of `thread-condition`, otherwise an error is signalled (condition: `wrong-condition-class`) on the thread calling `signal`. A `signal` on a determined thread has no effect on either the signalled or signalling thread.

14.1.1.4 See also: `thread-reschedule`, `thread-value`, `with-handler`.

14.1.2 `wrong-condition-class` *thread-condition*

14.1.2.1 Init-options

`condition` *condition*: A condition.

Signalled by `signal` if the given condition is not an instance of `thread-condition`.

14.1.3 `with-handler` *special form*

14.1.3.1 Syntax

`(with-handler` handler-function protected-form`)`

14.1.3.2 Arguments

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

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

14.1.3.3 Result

The value of the last form in the sequence of *protected-form*s.

14.1.3.4 Remarks

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

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

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

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

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

The above is the normal behaviour of `with-handler`. The exceptional behaviour of `with-handler` happens when there is a call to `signal` during the evaluation of *protected-form*. `signal` calls the dynamically closest *handler-function* passing on the 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 dynamically closest handler outside the extent of the form which established *handler-function*. A *handler-function* takes one of three actions:

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

b) Call the *resume* continuation. This action might be taken if the condition is recognised by the handler function and might be preceded by some corrective action. This is termed *resuming* the condition.

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

It is an error if the condition is declined and there is no next closest enclosing handler. In this circumstance the identified error is delivered to the configuration to be dealt with in an implementation-defined way.

14.1.3.5 Examples

An illustration of the use of all three cases is given in the following (unrealistic) example:

```
(let/cc accept
  (with-handler
    (generic-lambda (condition resume)
      method ;;error fixable, return to cerror (resume)
       (((c continuable-condition) resume)
        (resume (fix (condition))))
      method ;;serious error, exit from with-handler (accept)
       (((c condition) resume)
        (accept))
      ;;otherwise, by omission, let another handler deal
      ;;with it (decline)
      )
    ;;the protected expression
    (something-which-might-signal-an-error)))
```

14.1.3.6 See also: `signal`.

14.2 Conditions

14.2.1 `conditionp` *function*

14.2.1.1 Arguments

obj: object to examine.

14.2.1.2 Result

Returns *obj* if *obj* is a subclass of `condition`, otherwise `()`.

14.2.2 `condition-message` *function*

14.2.2.1 Arguments

condition: an instance of `condition`.

14.2.2.2 Result

Returns the contents of the message slot of *condition*, which is a string.

14.2.3 `initialize` *method*

14.2.3.1 Specialized Arguments

(*condition* `<condition>`): a condition.

(*initlist* `<list>`): A list of initialisation options as follows:

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

14.2.3.2 Result

A new, initialised condition.

14.2.3.3 Remarks

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

message: — The value must be a string, which should be used to convey information about the condition that has arisen.

14.2.4 `error` *function*

14.2.5 `cerror` *function*

14.2.5.1 Arguments

error-message: a string containing relevant information.

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

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

14.2.5.2 Result

The result of both of these functions is `()`.

14.2.5.3 Remarks

The `cerror` and `error` functions signal continuable and non-continuable errors, respectively. Each calls `signal` with an instance of a condition of *condition-class* initialized from *init-option*s, the *error-message* and a *resume* continuation. In the case of `cerror` the *resume* continuation is the continuation of the `cerror` expression. In the case of `error`, it is `()`, signifying that the condition was not signaled continuably. `cerror` and `error` can be defined in terms of more primitive operations:

```
(cerror error-message condition init-arg*)
                    ≡
    (let/cc cerror-fixed-up
      (let ((c (make condition init-arg*)))
        (signal c cerror-fixed-up)
        ;; return comes here
        (no-handler c cerror-fixed-up)))
    ;; resume comes here

(error error-message condition init-arg*)
                    ≡
     (let ((c (make condition init-arg*)))
       (signal c ())
       ;; return comes here
       (no-handler c ())))
```

The function *no-handler* is that which changes the state of the thread to `aborted` and saves the signaled condition for future reference. This function is called after all handlers have declined the condition. That is, *no-handler* is only called when none of the handlers can deal with the condition. Note that both the condition and the resume continuation are given to *no-handler*, as for any other handler function, so that, for instance, execution could be resumed from the debugger. Also note that *no-handler* is called in the environment of the `signal`, so that all the handlers in force at the time of signaling are also in force during the call to *no-handler*.

14.2.6 `defcondition` *defining form*

14.2.6.1 Syntax

`(defcondition` condition-name superclass init-option*`)`

14.2.6.2 Arguments

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

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

*init-option**: A sequence of symbols and expressions to be passed to *allocate-instance* and `initialize-instance`.

14.2.6.3 Remarks

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

15 Expressions, Definitions and Control Forms

This section gives the informal syntax of well-formed expressions and describes the semantics of the special-forms, functions and macros of the level-0 language. In the case of level-0 macros, the description is augmented with an expansion which has the required semantics. However, these descriptions are not prescriptive of any processor and a conforming program cannot rely on adherence to these expansions.

15.1 Atomic Expressions

15.1.1 `constant` *syntax*

There are two kinds of constant, literal constants and defined constants. The latter are considered under `symbol`s. 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. The external representation of the empty list is `()`. The empty list is the only instance of the class `null`. For historical reasons, the symbol `nil` is defined to be immutably bound to the empty list.

15.1.1.1 Examples

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

15.1.2 `defconstant` *defining form*

15.1.2.1 Syntax

`(defconstant` *identifier form*`)`

15.1.2.2 Arguments

identifier: A symbol naming an immutable top-lexical binding containing the value of *form*.

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

15.1.2.3 Remarks

The value of *form* is stored as the module value of *name*. It is an error to set the value of a defined constant to a different value.

15.1.3 `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. The defined constant `t` has the value `t` . The defined constant `nil` has the value `()` , which represents the abstract boolean value *false*. The abstract boolean value *true* can represented by any value other than *false*—that is, other than `()`.

15.1.4 `deflocal` *defining form*

15.1.4.1 Syntax

(`deflocal` *identifier form*)

15.1.4.2 Arguments

identifier: A symbol naming a binding containing the value of *form*.

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

15.1.4.3 Remarks

The value of *form* is stored as the module binding value of *name*. The binding created by a `deflocal` form is mutable.

15.2 Literal Expressions

15.2.1 `quote` *special form*

15.2.1.1 Syntax

(`quote` *datum*)

15.2.1.2 Arguments

datum: the *datum* to be quoted.

15.2.1.3 Result

The result is *datum*.

15.2.1.4 Remarks

The result of processing the expression (`quote` *datum*) is *datum*. The object *datum* can be any external representation of a EULISP object. The special form `quote` can be abbreviated using *apostrophe*—graphic representation ' in the standard tokenisation scheme—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 the contents of a literal expression. Within a single module, multiple references to the same (`eq`) literal produce the same literal.

15.3 Functions, Application, Definition

15.3.1 `lambda` *special form*

15.3.1.1 Syntax

(`lambda` *lambda-list body*)

15.3.1.2 Arguments

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

body: A sequence of forms.

15.3.1.3 Result

A function with the specified *lambda-list* and *body*.

15.3.1.4 Remarks

The function construction operator is `lambda`. Access to the lexical environment of definition is guaranteed, which may cause the creation of a closure. The syntax of *lambda-list* is defined by the following grammar:

$$\begin{array}{rcl} \textit{lambda-list} & ::= & \textit{identifier} \mid \textit{simple-list} \mid \textit{rest-list} \\ \textit{simple-list} & ::= & (\textit{identifier}^*) \\ \textit{rest-list} & ::= & (\textit{identifier}^+ \ . \ \textit{identifier}) \end{array}$$

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 *identifiers*. 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 extent. It is an error if the same identifier appears more than once in a *lambda-list*.

15.3.2 function call *special form*

15.3.2.1 Syntax

(*operator operand**)

15.3.2.2 Arguments

operator: This may be a symbol—being either the name of a special form, or a lexical variable—or a function call, which must result in an instance of `function`. An error is signaled (condition: `invalid-operator`) if the operator is neither the name of a special form nor a function.

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

15.3.2.3 Result

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

15.3.2.4 Remarks

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

15.3.2.5 See also: `constant`, `symbol`, `quote`.

15.3.3 `invalid-operator` *execution-condition*

15.3.3.1 Init-options

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

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

15.3.3.2 Remarks

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

15.3.4 `defmacro` *defining form*

15.3.4.1 Syntax

(`defmacro` *macro-name lambda-list body*)

15.3.4.2 Arguments

macro-name: A symbol naming a binding containing the function with 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.

15.3.4.3 Remarks

The `defmacro` form defines a function named by *macro-name* and stores the definition as the module binding value of *macro-name*. In addition, the function *macro-name* is exported with the syntax attribute set. The interpretation of the *lambda-list* is as defined for `lambda` (see section 15.6). The binding created by `defmacro` is immutable.

15.3.4.4 See also: `lambda`.

15.3.5 `defun` *defining form*

15.3.5.1 Syntax

(`defun` function-name lambda-list body)
or
(`defun` (`setter` function-name) lambda-list body)

15.3.5.2 Arguments

: The first argument can take two forms:

function-name — A symbol naming a binding containing the function with 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.

15.3.5.3 Remarks

The `defun` form defines a function named by *function-name* and stores the definition as the module value of *function-name*. The interpretation of the *lambda-list* is as defined for `lambda` (see section 15.6). The binding created by `defun` is immutable.

15.3.6 `apply` *function*

15.3.6.1 Syntax

(`apply` *function* obj_1 ... obj_n)

15.3.6.2 Arguments

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

obj_1 ... obj_{n-1}: A sequence of expressions, which wil be evaluated according to the rules given in **function call**.

obj_n: An expression which must evaluate to a proper list. An error is signaled (condition: `bad-apply-argument`) if obj_n is not a proper list.

15.3.6.3 Result

The result is the result of calling *function* with the actual parameter list created by appending obj_n to a list of the arguments obj_1 through obj_{n-1}. An error is signalled (condition: `invalid-operator`) if the first argument is not an instance of `function`.

15.3.6.4 See also: **function call**.

15.3.7 `bad-apply-argument` *execution-condition*

15.3.7.1 Init-options

`arglist` *list*: A list of the objects passed to `apply`.

15.3.7.2 Remarks

Signalled by `apply` if its first argument is not an instance of `function`.

15.4 Assignments

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

15.4.1 `setq` *special form*

15.4.1.1 Syntax

(`setq` *identifier expression*)

15.4.1.2 Arguments

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

form: An expression whose value is to be stored in the binding of *identifier*

15.4.1.3 Result

The result is the value of *form*.

15.4.1.4 Remarks

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

15.4.2 `setter` — *function*

15.4.2.1 Arguments

access-function: An expression which must evaluate to an instance of `function`.

15.4.2.2 Result

The *update-function* corresponding to *access-function*.

15.4.2.3 Remarks

A generalized place update facility is provided by `setter`. Given *access-function*, `setter` returns the corresponding update function. If no such function is known to `setter`, an error is signaled (condition: `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) an-accessor an-updator)` installs the function which is the value of `an-updator` as the updator of the accessor function which is the value of `an-accessor`. Defined updator functions in this report 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: `cannot-update-setter`)

15.4.2.4 See also: `defun`, `defmethod`.

15.4.3 `no-setter` — *execution-condition*

15.4.3.1 Init-options

`object` *object*: The object given to `setter`.

15.4.3.2 Remarks

Signalled by `setter` if there is no updator for the given function.

15.4.4 `cannot-update-setter` — *execution-condition*

15.4.4.1 Init-options

`accessor` $object_1$: The given accessor object.

`updator` $object_2$: The given updator object.

15.4.4.2 Remarks

Signalled by `(setter setter)` if the updator of the given accessor is immutable.

15.4.4.3 See also: `setter`.

15.5 Conditional Expressions

15.5.1 `if` — *special form*

15.5.1.1 Syntax

`(if` *expression expression expression*`)`

15.5.1.2 Arguments

antecedent: An expression which may evaluate to any object.

consequence: An expression which may evaluate to any object.

alternative: An expression which may evaluate to any object.

15.5.1.3 Result

Either the value of *consequence* or *alternative* depending on the value of *antecedent*.

15.5.1.4 Remarks

The *antecedent* is evaluated. If the result is *true* the *consequence* is evaluated, otherwise the *alternative* is evaluated. Both *consequence* and *alternative* must be specified. The result of `if` is the result of the evaluation of whichever of *consequence* or *alternative* is chosen. *consequence* is a single form, but *alternative* is a sequence of forms. Each form in *alternative* is evaluated in order and the result of the last form is the result of the `if` expression. Additional conditional forms (`when`, `unless`) are given in section B.20.

15.5.2 `cond` — *macro*

15.5.2.1 Syntax

`(cond (`*antecedent form**`)`*`)`

15.5.2.2 Remarks

The `cond` macro provides a convenient syntax for collections of *if-then-elif...else* expressions. The rewrite rules for `cond` are:

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

15.5.3 `and` — *macro*

15.5.3.1 Syntax

`(and` *form**`)`

15.5.3.2 Remarks

The expansion of an `and` form leads to the evaluation of the sequence of *form*s from left to right. The the first *form* in the sequence that evaluates to `()` stops evaluation and none of the *form*s to its right will be evaluated. The result of `and` is `()`. If none of the *form*s evaluate to `()`, the value of the last *form* is returned. The rewrite rules for `and` are:

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

15.5.4 `or` *macro*

15.5.4.1 Syntax

`(or` *form**`)`

15.5.4.2 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 *true* is the result of the `or` form and none of the *form*s to its right will be evaluated. If none of the forms evaluate to *true*, the value of the last *form* is returned. The rewrite rules for `or` are:

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

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

15.6 Variable Binding and Sequences

15.6.1 `let/cc` *special form*

15.6.1.1 Syntax

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

15.6.1.2 Arguments

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

body: A sequence of forms.

15.6.1.3 Result

The result of evaluating the last form in *body*.

15.6.1.4 Remarks

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

15.6.1.5 See also: `block`, `return-from`.

15.6.2 `labels` *special form*

15.6.2.1 Syntax

`(labels (`*function-name lambda-list body*$)^*$ *labels-body*`)`

15.6.2.2 Remarks

The `labels` operator provides for local mutually recursive function creation. Each *function-name* is bound to a new location holding an unspecified value, making a new environment extended by those bindings. Then for each set of formal parameters and *body*, a function is constructed, using `lambda`, and the binding of the corresponding *function-name* is updated to have the value of the lambda expression. The scope of the *function-name*s is the entire `labels` form. The *lambda-list* is either a single variable or a list of variables—see `lambda`. Each form in *labels-body* is evaluated in order in the above extended environment. The result of evaluating the last form is returned as the result of the `labels` form.

15.6.3 `let` *macro*

15.6.3.1 Syntax

`(let [`*identifier*`] (`*binding**`)` *body*`)`

15.6.3.2 Remarks

The optional *identifier* denotes that the let form can be called from within its *body*. This is an abbreviation for `labels` combined with `let`. 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. The rewrite rule for `let` is:

```
(let () form*)          ≡   (progn form*)
(let ((id1 form1)       ≡   ((lambda (id1 id2 id3 ...)
      (id2 form2)               form*)
      id3                    form1 form2 () ...)
      ...)
  form*)
```

15.6.4 `let*` *macro*

15.6.4.1 Syntax

`(let* [`*identifier*`] (`*binding**`)` *body*`)`

15.6.4.2 Remarks

The optional *identifier* denotes that the let form can be called from within its *body*. This is an abbreviation for `labels` combined with `let*`. A *binding* is specified by a two element list of a variable and an initializing form. The first initializing form is evaluated in the current environment and the corresponding variable is bound to a new location containing that

result. Subsequent bindings are processed in turn, evaluating the initializing form in the environment extended by the previous binding. Each form in *body* is evaluated in order in the environment extended by the above bindings. The result of evaluating the last form is returned as the result of the `let*` form. The rewrite rules for `let*` are:

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

15.6.5 `progn` *special form*

15.6.5.1 Syntax

`(progn` *form**`)`

15.6.5.2 Arguments

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

15.6.5.3 Result

The sequence of *form*s is evaluated in order, returning the value of the last one as the result of the `progn` form.

15.6.5.4 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. It is a static error if this rule is violated.

15.6.6 `unwind-protect` *special form*

15.6.6.1 Syntax

`(unwind-protect` *protected-form after-form** `)`

15.6.6.2 Arguments

protected-form: A form.

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

15.6.6.3 Result

The value of *protected-form*.

15.6.6.4 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-protect`s between the dynamic extent of the target of control transfer and the current `unwind-protect` are evaluated in increasing order of dynamic extent.

15.7 Waiting on Events

15.7.1 `wait` *generic function*

15.7.1.1 Arguments

obj: An object.

timeout: One of `()`, `t` or an instance of `integer`.

15.7.1.2 Result

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

15.7.1.3 Remarks

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

a) A predicate associated with *obj* returns true;

b) After *timeout* time units;

c) A signal is received.

`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. 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`.

15.7.1.4 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.

```
(labels (
  (loop ()
    (when (wait s cuckoo-heartbeat)
          (print (read-char s))
          (loop))))
  (loop))
```

15.7.1.5 See also: threads, streams.

15.7.2 `ticks-per-second` *double-float*

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

Table 9 — Quasiquote Syntax

anti-quotation	::=	‘
unquotation	::=	,
unquote-splice	::=	,@
quasiquotation	::=	$quasiquotation_1$
$template_0$	::=	*expression*
$quasiquotation_i$	::=	‘$template_i$ | (**quasiquote** $template_i$)
$template_i$	::=	*oneofsimple-datum, list-*$template_i$*, vector-*$template_i$*, unquotation*$_i$
list-$template_i$	::=	(*template-or-splice*$_i^*$) | (*template-or-splice*$_i^+$. $template_i$) | ’$template_i$ | $quasiquotation_{i+1}$
vector-$template_i$	::=	#(*template-or-splice*$_i^*$)
$unquotation_i$	::=	,$template_{i-1}$ | (**unquote** $template_{i-1}$)
template-or-splice$_i$	::=	$template_i$ | *splicing-unquotation*$_i$
splicing-unquotation$_i$	::=	,@$template_{i-1}$ | (**unquote-splicing** $template_{i-1}$)

15.8 Quasiquotation Expressions

15.8.1 quasiquote *macro*

15.8.1.1 Syntax

(**quasiquote** *skeleton*) or ‘*skeleton*

15.8.1.2 Remarks

Quasiquotation is also known as “backquoting”. A **quasiquote**d expression is a convenient way of building a structure. The *skeleton* describes the shape and, generally, many of the entries in the structure but some holes remain to be filled. The **quasiquote** macro might be abbreviated by using the glyph called *grave accent* (‘), so that (**quasiquote** *expression*) can be written ‘*expression*. A complete definition of the syntax of quasiquote expressions is given in Table 9.

15.8.2 unquote *syntax*

15.8.2.1 Syntax

(**unquote** *form*) or ,*form*

15.8.2.2 Remarks

See **unquote-splicing**.

15.8.3 unquote-splicing *syntax*

15.8.3.1 Syntax

(**unquote-splicing** *form*) or ,@*form*

15.8.3.2 Remarks

The holes in a quasiquoted expression are identified by unquote expressions of which there are two kinds—forms whose value is to be inserted at that location in the structure and forms whose value is to be spliced into the structure at that location. The former is indicated by an **unquote** expression and the latter by an **unquote-splicing** expression. In **unquote-splice** the *form* must result in a proper list. An error is signaled (condition: **improper-unquote-splice**) on attempting to **unquote-splice** an improper 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**.

15.8.4 improper-unquote-splice *execution-condition*

15.8.4.1 Init-options

skeleton *skeleton*: The skeleton that **unquote** is processing.

splice-list *list*: The improper list which has lead to the error.

15.8.4.2 Remarks

Signalled by **quasiquote** if the result of an **unquote-splicing** form is not a proper list.

15.9 Summary of Level-0 Expressions and Definitions

The syntax of all level-0 expressions and definitions is given in Table 10. Any productions undefined here appear elsewhere in the definition, specifically: the syntax of certain classes of data are defined in their own sections.

Table 10 — Expressions and Definitions (level-0)

level-0-definition	::=	*defconstant* | *defcondition* | *deflocal* | *defmacro* | *defmodule* | *defstruct* | *defclass* | *defun*
defcondition	::=	(`defcondition` *condition-name superclass init-option**)
defconstant	::=	(`defconstant` *identifier form*)
deflocal	::=	(`deflocal` *identifier form*)
defmacro	::=	(`defmacro` *macro-name lambda-list body*)
defmodule	::=	(`defmodule` *module-name import-spec syntax-spec module-expression**)
import-spec	::=	(*module-directive**)
syntax-spec	::=	() | (`syntax` *import-spec defmacro**)
export-spec	::=	*export* | *export-syntax* | *expose*
export	::=	(`export` *name**)
export-syntax	::=	(`export-syntax` *name**)
expose	::=	(`expose` *module-directive**)
module-directive	::=	*module-name* | *module-filter*
module-filter	::=	*except* | *only* | *rename*
except	::=	(`except` (*name**) *module-directive*$^+$)
only	::=	(`only` (*name**) *module-directive*$^+$)
rename	::=	(`rename` ((*old-name new-name*)*) *module-directive*$^+$)
module-expression	::=	*export-spec* | *level-0-expression* | *definition* | (`progn` *expression*)
definition	::=	*level-0-definition*$_{\{\texttt{defmodule}\}}$
defgeneric	::=	(`defgeneric` *gf-name gen-lambda-list level-0-init-option**)
gf-name	::=	*identifier* | (`setter` *identifier*) | (`converter` *identifier*)
gen-lambda-list	::=	*spec-lambda-list*
level-0-init-option	::=	`method` *method-description*
method-description	::=	(*spec-lambda-list form**)
spec-lambda-list	::=	(*spec-variable** [`.` *variable*])
spec-variable	::=	(*variable class*) | *variable*
class	::=	*class-name*
defstruct	::=	(`defstruct` *class-name superclass* (*slot-description**) *class-option**)
defclass	::=	(`defclass` *class-name* (*superclass**) (*slot-description**) *class-option**)
defun	::=	(`defun` *function-name lambda-list body*) | (`defun` (`setter` *function-name*) *lambda-list body*)
level-0-expression	::=	*lex-ref* | *literal* | *procedure-call* | *macro-call* | *level-0-special* | *quasiquotation*
level-0-special	::=	*conditional* | *lambda* | *let/cc* | *progn* | *lex-assign* | *unwind-protect* | *with-handler*
lex-ref	::=	*identifier*
literal	::=	*quotation* | *self-evaluating*
quotation	::=	(`quote` *datum*)
lex-assign	::=	(`setq` *identifier expression*)
conditional	::=	(`if` *expression expression expression*)
lambda	::=	(`lambda` *lambda-list body*)
lambda-list	::=	*identifier* | *simple-list* | *rest-list*
simple-list	::=	(*identifier**)
rest-list	::=	(*identifier*$^+$ `.` *identifier*)
let/cc	::=	(`let/cc` *identifier body*)
progn	::=	(`progn` *form**)
with-handler	::=	(`with-handler` *handler-function protected-form*)
unwind-protect	::=	(`unwind-protect` *protected-form after-form**)
self-evaluating	::=	*character* | *number* | *string* | *vector*
procedure-call	::=	(*procedure-operator operand**)
macro-call	::=	(*macro-operator operand**)
procedure-operator	::=	*expression*
operand	::=	*expression*
macro-operator	::=	*symbol*
variable	::=	*identifier*
body	::=	*expression**

Annex A
(normative)
Level-0 Module Library

A.1 Characters

The defined name of this module is `character`.

A.1.1 `character` *syntax*

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

Table A.2 — Special Character Syntax

Name	Syntax
alert	`#\alert`
backspace	`#\backspace`
delete	`#\delete`
formfeed	`\formfeed`
linefeed	`#\linefeed`
newline	`#\newline`
return	`#\return`
tab	`#\tab`
space	`#\space`
vertical-tab	`#\vertical-tab`

Any character which does not have a name, and thereby an external representation dealt with by the above cases is represented by `#\x` followed by up to four hexadecimal digits. The value of the hexadecimal number represents the position of the character in the current character set. Examples of such character literals are `#\x0` and `#\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 Table A.1.

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

A.1.2 `<character>` *class*

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

A.1.3 `characterp` *function*

A.1.3.1 Arguments

obj: Object to examine.

A.1.3.2 Result

Returns *obj* if *obj* is an instance of a subclass `character`, otherwise `()`.

A.1.4 `(converter integer)` *method*

A.1.4.1 Arguments

character: A character.

A.1.4.2 Result

Returns an instance of `single-precision-integer` which corresponds to the position of the instance of `character` in the default character set.

A.1.5 `equal` *method*

A.1.5.1 Arguments

$character_1$: an instance of `character`.

$character_2$: an instance of `character`.

A.1.5.2 Result

Each instance of `character` is converted to a integer and the two values are compared using `=`. The result of `equal` is the first argument if the result of `=` is non-`()`. If not, the result is `()`.

A.1.6 `copy` *method*

A.1.6.1 Arguments

character: A character

A.1.6.2 Result

Constructs and returns an instance of `character`, whose value is the same (under `equal`) as the source.

A.1.7 `generic-prin` *method*

A.1.8 `generic-write` *method*

A.1.8.1 Arguments

character: Character to be ouptut on *stream*.

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

Table A.1 — Character Syntax

character	::=	*extension character-extension character-name*
character-name	::=	*literal-name* | *special-name* | *control-name* | *numeric*
literal-name	::=	*alphanumeric* | *non-alpha*
control-name	::=	*control-extension literal-name*
special-name	::=	`alert` | `backspace` | `delete` | `formfeed` | `linefeed` | `newline` | `return` | `tab` | `space` | `vertical-tab`
character-extension	::=	`\`
control-extension	::=	`^`
numeric	::=	*string-hex digit(16)* [*digit(16)* [*digit(16)* [*digit(16)*]]]

A.1.8.2 Result

The character *character*.

A.1.8.3 Remarks

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

A.1.9 `generic-write` *method*

A.1.9.1 Arguments

character: Character to be ouptut on *stream*.

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

A.1.9.2 Result

The character *character*.

A.1.9.3 Remarks

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

A.2 Collections

The defined name of this module is `collection`. A *collection* is defined as an instance of one of `string`, `list`, `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.

A.2.1 `empty-p` *generic function*

A.2.2 `size` *generic function*

A.2.3 `member` *generic function*

A.2.4 `do` *generic function*

A.2.5 `map` *generic function*

A.2.6 `reduce` *generic function*

A.2.7 `reduce1` *generic function*

A.2.8 `fill` *generic function*

A.2.9 `catenate` *generic function*

A.2.10 `filter` *generic function*

A.3 Comparing Objects

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

```
(eq a b)   ⇒   (eql a b)   ⇒   (equal a b)
(eq a b)   ⇍   (eql a b)   ⇍   (equal a b)
```

A.3.1 `eq` *function*

A.3.1.1 Arguments

obj$_1$: an object.

obj$_2$: an object.

A.3.1.2 Result

Compares *obj*$_1$ and *obj*$_2$ and returns `t` if they are the same object, otherwise `()`.

A.3.1.3 Remarks

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

A.3.2 `=` *generic function*

A.3.2.1 Arguments

number$_1$: an instance of `number`

number$_2$: an instance of `number`

A.3.2.2 Result

One of the arguments, or `()`.

A.3.2.3 Remarks

Defined over all number types. If both numbers are of the same class, they are compared according to the comparison function for numbers of that class. If the two instances are numerically equal, the result is the first argument (a non-`()` value). If not, the result is `()`. Methods are defined for the following classes: `single-precision-integer`, `variable-precision-integer`, `ratio`, `float` and `complex`. In the case of `complex`, the result is determined by the conjunction of the pairwise application of `=` to the real parts and the imaginary parts.

If the numbers are not of the same class, then one of the numbers is converted to the class of the other number according to the protocol given in section A.10.

A.3.2.4 See also: Class specific sections which define methods on `copy`—single precision integer and double float.

A.3.3 `eql` *function*

A.3.3.1 Arguments

obj$_1$:

obj$_2$:

A.3.3.2 Result

If the class of *obj*$_1$ and of *obj*$_2$ is the same and is a subclass of `number`, the result is that of comparing them under `=`. If the class of *obj*$_1$ and of *obj*$_2$ is the same and is a subclass of `character`, the result is that of comparing them under `equal`. Otherwise the result is that of comparing them under `eq`.

A.3.4 `equal` *generic function*

A.3.4.1 Arguments

obj$_1$:

obj$_2$:

A.3.4.2 Result

The result is determined by whichever of the methods defined here is applicable. It is implementation-defined whether or not `equal` will terminate on self-referential structures.

A.3.4.3 See also: Class specific sections which define methods on `copy`.

A.3.5 `equal` *method*

A.3.5.1 Arguments

object$_1$: an object.

object$_2$: an object.

A.3.5.2 Result

If the class of each instance of `object` is the same, then the result is the conjunction of the pairwise application of `equal` to the contents of the slots of the arguments. If not the result is `()`.

A.4 Conversion

The defined name of this module is `convert`.

It may seem that the natural way to define (`convert` *obj target-class*) is as a generic function specializing on both parameters. However, because we want the behaviour to depend on the target-class and not on the class of target-class, we would have to use class prototypes (as in up to version 0.7) or `eql` methods in order to make it work. Neither is desirable. Some classes that might be targets for conversion (for instance `pair`, `number`) cannot easily have prototypes because they cannot have direct instances. `eql` methods, if they are desirable at all, are too great a complication for level-0. This suggests that it may have been a mistake to think in terms of multi-methods. Fortunately, if we forget them and consider classical methods, a fairly reasonable solution appears.

Advantages are that it is object-oriented in a natural way, there is no need for class prototypes or `eql` methods and it is asymmetric in the right direction (the converter of a class converts to rather than from instance of the class).

Disadvantages are that methods are not inherited from more general target classes—although this could be argued the other way round too.

Conversion between classes is provided by the function `convert` which accesses a set of converter functions using the target class (the second argument to `convert`) as a key. The resulting converter function is a generic function which discriminates on the class of the object which is to be converted.

A.4.1 `convert` *function*

A.4.1.1 Arguments

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

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

A.4.1.2 Result

Returns an instance of *class* which is equivalent in some class-specific sense to *obj*, which may be an instance of any type. Calls the converter function associated with the class *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 *obj* to an instance of *class*.

A.4.2 `conversion-condition` *condition*

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

A.4.3 `no-converter` *conversion-condition*

A.4.3.1 Init-options

`source` *object*: The object to be converted.

`class` *class*: The class with which no converter function is associated.

A.4.3.2 Remarks

Signalled by `convert` if there is no converter function for the given target class.

A.4.4 `converter` *function*

A.4.4.1 Arguments

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

A.4.4.2 Result

The accessor returns the converter function for the class *target-class*. The converter is a generic-function with methods specialized on the class of the object to be converted. Note that all converters defined here whose target class is *string* produce a string containing a representation of the source object as if it had output by `write`.

A.4.5 (`setter converter`) *setter*

A.4.5.1 Arguments

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

generic-function: The new converter function.

A.4.5.2 Result

The new converter function. The setter function replaces the converter function for the class *target-class* by *generic-function*. The new converter function must be an instance of `generic-function`.

A.4.5.3 See also: Converter methods from one class to another are defined in the section pertaining to the source class.

A.5 Copying Objects

The defined name of this module is `copy`.

A.5.1 `copy` *generic function*

A.5.1.1 Arguments

obj: An object to be copied.

A.5.1.2 Result

Constructs and returns a copy of the source which is the same (under some class specific predicate) as the source. The exact behaviour for each class of *obj* is defined by the most applicable method for *obj*.

A.5.1.3 See also: Class specific sections which define methods on `copy`.

A.5.2 `copy` *method*

A.5.2.1 Arguments

object: An object (the default method).

A.5.2.2 Result

Constructs and returns an instance of the same class as the source, whose slot values are the same as those of the source (under `eql`), so that the resulting object is the same (under `equal`) as the source.

Table A.4 — Methods for double precision floats

`binary-plus`
`binary-difference`
`negate`
`binary-times`
`binary-divide`
`binary-lt`
`abs`
`zerop`
`signum`
`positivep`
`negativep`

A.6 Double Precision Floats

A.6.1 `double-float` *syntax*

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.` (*simple-float-1*), `-.456` (*simple-float-2*), `123.456` (*simple-float-3*); and with exponents: `+123456.D3`-, `1.23455D2`, `-.123456D3`,

The syntax for the external representation of double precision floating point literals is defined in Table A.3. The representation used by `write` and `prin` is that based on *simple-float* without an exponent, namely: `[`*sign*`]` *simple-float-3*. Finer control over the format of the output of floating point numbers is provided by some of the formatting specifications of `format` (see section A.8).

The defined name of this module is `double`. Arithmetic operations for `double-float` are defined by methods to be attached to the generic functions listed in Table A.4.

A.6.2 `<double-float>` *class*

The class of all instances of double precision float.

A.6.3 `double-float-p` *function*

A.6.3.1 Arguments

obj: Object to examine.

A.6.3.2 Result

If *obj* is an instance of `float` the result is *obj*, otherwise `()`.

The function `float` returns *obj* if *obj* is a subclass of `float` and the `double-float-p` returns *obj* if it is an instance of `double-float`. Otherwise both return `()`.

Table A.3 — Floating Point Syntax

float	::=	[*sign*] *ufloat*
sign	::=	{+ | -}
ufloat	::=	*simple-float* [*exponent*]
simple-float	::=	*simple-float-1* | *simple-float-2* | *simple-float-3*
simple-float-1	::=	*udecimal float-separator*
simple-float-2	::=	*float-separator udecimal*
simple-float-3	::=	*udecimal float-separator udecimal*
float-separator	::=	.
exponent	::=	*dexpt-mark* [*sign*] *udecimal*
dexpt-mark	::=	{d | D}
udecimal	::=	*digit(10)*$^+$
digit(10)	::=	{0 | ... | 9}

A.6.4 `most-positive-double-float` *double-float*

A.6.4.1 Remarks

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

A.6.5 `least-positive-double-float` *double-float*

A.6.5.1 Remarks

The value of `least-positive-double-float` is that positive double precision floating point number closest in value to (but not equal to) zero that the processor provides. This value is the same as the result of `(succ 0.0)`.

A.6.6 `least-negative-double-float` *double-float*

A.6.6.1 Remarks

The value of `least-negative-double-float` is that negative double precision floating point number closest in value to (but not equal to) zero that the processor provides. Even if the processor provide negative zero, this value must not be negative zero. This value is the same as the result of `(pred 0.0)`.

A.6.7 `most-negative-double-float` *double-float*

A.6.7.1 Remarks

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

A.6.8 `truncate` *generic function*

A.6.8.1 Arguments

float: An instance of `float`.

[*precision*]: A single precision integer.

A.6.8.2 Result

Given one argument, returns the greatest integer value whose magnitude is less than or equal to x. Given two arguments with an integer value as the second to specify precision, returns a floating point number which is the result of zeroing out the low $(n - precision)$ digits, where n is the number of digits of precision provided by the representation. It is an error if *precision* is greater than n.

A.6.9 `truncate` *method*

A.6.9.1 Remarks

Implements `truncate` for `double-float`.

A.6.10 `round` *generic function*

A.6.10.1 Arguments

float: An instance of `float`.

[*precision*]: A single precision integer.

A.6.10.2 Result

Given one argument, returns the integer whose value is closest to x, except in the case when x is exactly half-way between two integers, when it is rounded to the one that is even. Given two arguments with an integer value as the second to specify precision, returns a floating point number which is the result of zeroing out the low $(n - precision)$ digits, where n is the number of digits of precision provided by the representation. The number of digits of precision and the radix of the precision are implementation-defined values. If the resulting value is exactly half-way between two *precision*-digit floating point numbers the result is the one with the even least significant digit. It is an error if *precision* is greater than n.

A.6.11 `round` *method*

A.6.11.1 Remarks

Implements `round` for `double-float`.

A.6.12 `floor` *generic function*

A.6.12.1 Arguments

float: An instance of `float`.

A.6.12.2 Result

Computes the greatest integer value which is less than or equal to *float*.

A.6.13 `floor` *method*

A.6.13.1 Remarks

Implements `floor` for `double-float`.

A.6.14 `ceiling` *generic function*

A.6.14.1 Arguments

float: An instance of `float`.

A.6.14.2 Result

Computes the least integer value that is greater than or equal to *float*.

A.6.15 `ceiling` *method*

A.6.15.1 Remarks

Implements `ceiling` for `double-float`.

A.6.16 `(converter string)` *method*

A.6.16.1 Arguments

double-float: A double precision float.

A.6.16.2 Result

Constructs and returns a string, the characters of which correspond to the external representation of the instance of `double-float`.

A.6.17 `(converter single-precision-integer)` *method*

A.6.17.1 Arguments

double-float: A double precision float.

A.6.17.2 Result

Returns an instance of `single-precision-integer` whose value is closest to that of the floating point source. This is the same function as `round` without specifying the second argument. An error is signaled (condition: `integer-conversion-overflow`) if the floating point number cannot be represented as a single precision integer.

A.6.18 `integer-conversion-overflow` *conversion-condition*

A.6.18.1 Init-options

`source` *double*: The double float to be converted.

A.6.18.2 Remarks

Signalled by the `double` method of `(converter spint)` if the magnitude of the double float is greater than can be reprented by a single precision integer.

A.6.19 `copy` *method*

A.6.19.1 Arguments

double-float: A double precision float.

A.6.19.2 Result

Constructs and returns an instance of `double-float`, whose value is the same (under `=`) as the source.

A.6.20 `generic-prin` *method*

A.6.21 `generic-write` *method*

A.6.21.1 Arguments

x: The double float to be output on *stream*.

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

A.6.21.2 Result

The double float supplied as the first argument.

A.6.21.3 Remarks

Output the external representation of *x* on *stream*, as described in the introduction to this section, namely: `[`*sign*`]` *simple-float-3*. Finer control over the format of the output of floating point numbers is provided by some of the formatting specifications of `format` (see section A.8).

A.7 Elementary Functions

The defined name of this module is `elementary-functions`. The contents of this module are defined as if all the number classes of EULISP exist (including `complex`. Depending on the level of conformance of a given implementation, only the methods for the number classes defined at the level of the processor need be supplied to provide a compliant `elementary-functions` library module.

A.7.1 `pi` *double-float*

A.7.1.1 Remarks

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

A.7.2 `sin` *generic function*

A.7.3 `cos` *generic function*

A.7.4 `tan` *generic function*

A.7.4.1 Arguments

z: A number.

A.7.4.2 Result

`sin` returns the sine of its argument, `cos` the cosine and `tan` the tangent. The unit of the argument is radians. Methods are defined for the appropriate subclasses of `integer` and `float` and for `ratio` and `complex`. The methods for `integer` and `ratio` coerce their argument to `float` and then compute the result. The methods for `float` produce a `float` result, the methods for `complex` produce a `complex` result.

A.7.5 `acos` *generic function*

A.7.6 `asin` *generic function*

A.7.6.1 Arguments

z: A number.

A.7.6.2 Result

`acos` returns the principal arc cosine and `asin` the principal arc sine of its argument. The unit of the result is radians. Methods are defined for the appropriate subclasses of `integer` and `float` and for `ratio` and `complex`. The methods for `integer` and `ratio` coerce their argument to `float` and then compute the result. The methods for `float` produce a `float` result when $-1 \leq z \leq 1$, otherwise a `complex` result. The methods for `complex` produce a `complex` result.

A.7.7 `atan` *generic function*

A.7.7.1 Arguments

z: A number.

A.7.7.2 Result

`atan` returns the arc tangent of its argument. The unit of the argument is radians. Methods are defined for the appropriate subclasses of `integer` and `float` and for `ratio` and `complex`. The methods for `integer` and `ratio` coerce their argument to `float` and then compute the result. The method for `float` produces a `float` result, the method for `complex` produces a `complex` result.

A.7.8 `atan2` *generic function*

A.7.8.1 Arguments

x_1 x_2: Two numbers.

A.7.8.2 Result

`atan2` returns the arc tangent of the quantity x_1/x_2, treating the case $x_2 = 0$ correctly. Methods are defined for (`integer integer`), (`float float`) and (`ratio ratio`). If the arguments are not of the same subclass of `number` but in the set given above, the lower one is coerced to the class of the higher according to the protocol for the level being used (see figure A.1). The methods for `integer` and `ratio` coerce their arguments to `float` and then compute the result. A `float` result is returned.

The range of the real-part of the values returned by `atan` and `atan2` is $(-\texttt{pi}, \texttt{pi}]$.

A.7.9 `exp` *generic function*

A.7.9.1 Arguments

z: A number.

A.7.9.2 Result

`exp` returns e raised to the power of x, where e is the base of the natural logarithms. Methods are defined for the appropriate subclasses of `integer` and `float` and for `ratio` and `complex`. The methods for `integer` and `ratio` coerce their argument to `float` and then compute the result. The method for `float` produces a `float` result, the method for `complex` produces a `complex` result.

A.7.10 `log` *generic function*

A.7.11 `log2` *generic function*

A.7.12 `log10` *generic function*

A.7.12.1 Arguments

z: A number.

A.7.12.2 Result

`log` returns the logarithm of z to the base of the natural logarithms. `log2` returns the logarithm of z to base 2. `log10` returns the logarithm of z to base 10. The result can be either `float` or `complex`. Methods are defined for the appropriate subclasses of `integer` and `float` and for `ratio` and `complex`. The methods for `integer` and `ratio` coerce their argument to `float` and then compute the result. The methods for `float` produce a `float` result when z is real and positive, otherwise a `complex` result. The methods for `complex` produce a `complex` result.

A.7.13 `sqrt` *generic function*

A.7.13.1 Arguments

z: A number.

`sqrt` returns the principal square root of z.

A.7.14 `sqrt` *method*

A.7.14.1 Arguments

integer: An integer.

A.7.14.2 Result

The method for `integer` returns an integer if the argument is a positive perfect square, a gaussian integer if the argument is a negative perfect square, otherwise a `float` is returned if the argument is positive, or a `complex` if the argument is negative.

A.7.15 `sqrt` *method*

A.7.15.1 Arguments

double-float: A double float.

A.7.15.2 Result

The method for `double-float` returns a `double-float` if the argument is non-negative and a `complex` if it is not.

A.7.16 `expt` *generic function*

A.7.16.1 Arguments

z_1 z_2: Two numbers.

A.7.16.2 Result

`expt` returns the principal value that results from raising z_1 to the power z_2. The complexity in the definition of `expt` stems from the different combinations of argument classes and what might be a reasonable result class for a given pair of argument classes. For the purpose of defining the behaviour of this function, the number classes are considered to form a tower as follows:

`complex`
`complex(ratio)`
`complex(integer)`
`float`
`ratio`
`integer`

where the classes correspond to and approximate the abstract mathematical objects: $\mathcal{C}$, $\mathcal{Q}[i]$, $\mathcal{Z}[i]$, $\mathcal{R}$, $\mathcal{Q}$, $\mathcal{Z}$. For each argument class combination, the entry in Table A.5 shows the lowest class in which the result might be expressed. In this sense, we define the lower bound class in which the result can occur for a given pair of arguments. The result of `expt` should be in the lowest class possible for a given argument combination without loss of information.

A.7.17 `sinh` *generic function*

A.7.18 `cosh` *generic function*

A.7.19 `tanh` *generic function*

A.7.20 `asinh` *generic function*

A.7.21 `acosh` *generic function*

A.7.22 `atanh` *generic function*

A.7.22.1 Arguments

z: A number.

A.7.22.2 Result

These functions compute the hyperbolic sine, cosine, tangent, arc sine, arc cosine and arc tangent functions. The result can be `float` or `complex`. Methods are defined for the appropriate subclasses of `integer` and `float` and for `ration` and `complex`. The methods for `integer` and `rational` coerce their argument to `float` and then compute the result. For the sine, cosine, tangent and arc sine, the methods for `float` produce a `float` result. For the arc cosine, the method for `float` produces a `float` if $z > 1$, otherwise a `complex`. For the arc tangent, the method for `float` produces a `float` if $-1 \leq z \leq 1$, otherwise a `complex`.

All methods produce a `complex` result for a `complex` argument.

NOTE—more detailed specification is required for this library module, in particular with respect to the han-

Table A.5 — `expt` result classes

Base Class	Exponent Class			
	`integer`	`ratio`	`float`	`complex`
`integer`	`integer`	`integer`	`float`	`complex`
`ratio`	`integer`	`integer`	`float`	`complex`
`float`	`float`	`float`	`float`	`complex`
`complex(integer)`	`integer`	`integer`	`complex`	`complex`
`complex(ratio)`	`integer`	`integer`	`complex`	`complex`
`complex`	`complex`	`complex`	`complex`	`complex`

dling of negative 0.0 and the stating of branches and cuts.

A.8 Formatted-IO

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

A.8.1 `scan-mismatch` *stream-condition*

A.8.1.1 Init-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.

A.8.1.2 Remarks

This condition is signaled by `scan` if the format string does not agree with the data received from *stream*.

A.8.2 `scan` *function*

A.8.2.1 Arguments

format-string: A string containing format directives.

[*stream*]: A stream from which input is to be taken.

A.8.2.2 Result

Returns a list of objects which have been read.

A.8.2.3 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 campatible 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 the stream which is the value of calling `standard-input-stream`. Scan returns a list of the values read in.

— `~a` any: accept any object

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

— `~c` character: a single character

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

— `~`*m.n*`e` a fixed-format floating-point number in FORTRAN "E" format.

— `~`*m.n*`f` an exponential-format floating-point number in FORTRAN "F" format.

— `~`*m.n*`g` a generalized floating-point number in either fixed or exponential format. either fixed-format or exponential notation as appropriate.

— `~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.

A.8.3 `format` *function*

A.8.3.1 Arguments

stream: One of `()`, `t` or an instance of `stream`.

format-string: A string containing format directives.

[*obj**]:

A.8.3.2 Result

Returns a list of unconsumed *obj*s.

A.8.3.3 Remarks

Has side-effect of printing according to *format-string*. If *stream* is `t` the output is to the current output stream. If *stream* is `()`, a formatted string is returned as the result of the call. Otherwise *stream* must be a valid output stream. Characters are output as if the string were output by the `prin` function with the exception of those prefixed by *tilde*–graphic representation `~`–as follows:

— `~a` any: use `prin` to output the object.

— `~b` binary: the integer argument is output in binary format.

— `~c` character: the next argument is displayed as a character.

— `~d` decimal: the integer argument is output in decimal format.

— `~`*m.n*`e` fixed-format floating-point: the floating-point argument is output in FORTRAN "E" format.

— `~`*m.n*`f` exponential floating-point: the floating-point argument is output in FORTRAN "F" format.

— `~`*m.n*`g` generalized floating-point: output the floating-point argument using either fixed-format or exponential notation as appropriate.

— `~o` octal: the integer argument is output in octal format.

— `~`*n*`r` radix: the integer argument is output in radix *n*.

— `~s` s-expression: uses `write` to output the object.

— `~t` tab: output sufficient spaces to reach the next tab-stop.

— `~x` hexadecimal: the integer argument is output in hexadecimal format.

— `~%` newline: output a `#\newline` character.

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

— `~|` page separator: output a page separator.

— `~~` tilde: output a tilde.

A.9 The empty list

The defined name of this module is `null`. The empty list is disjoint from the class `<pair>`. The combination of `<null>` and `<pair>` allows the creation of proper lists, since a proper list is one whose last pair contains the empty list in its `cdr` field.

A.9.1 `()` *syntax*

A.9.1.1 Remarks

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

A.9.2 `<null>` *class*

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

A.9.3 `null` *function*

A.9.3.1 Arguments

obj: Object to examine.

A.9.3.2 Result

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

A.9.4 `length` *method*

A.9.4.1 Arguments

null: The empty list.

A.9.4.2 Result

Returns zero.

A.9.5 `generic-prin` *method*

A.9.6 `generic-write` *method*

A.9.6.1 Arguments

null: The empty list.

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

A.9.6.2 Result

The empty list.

A.9.6.3 Remarks

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

A.10 Numbers

The defined name of this module is `number`. The naming conventions described in section 6 are applied in the following definitions.

Numbers can take on many forms with unusual properties, specialized for different tasks, but two classes of number normally suffice for the majority of needs. Thus, at level-0, only a limited set of number classes are defined.

In Figure A.1 is an example of what the initial number class hierarchy for level-0 might look like. 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` and the class of each number class might be a subclass of `number-class`. Since there are only two concrete number classes at level-0, coercion is simple, as shown in figure A.1. Any level-0 version of a library module, for example, `elementary-functions`, need only define methods for these two classes.

A.10.1 `<number>` *class*

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

A.10.2 `numberp` *function*

A.10.2.1 Arguments

obj: Object to examine.

A.10.2.2 Result

If the class of *obj* is a subclass of `number` the result is *obj*, otherwise `()`.

A.10.3 `<integer>` *class*

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

A.10.4 `integerp` *function*

A.10.4.1 Arguments

obj: Object to examine.

A.10.4.2 Result

If the class of *obj* is a subclass of `integer` the result is *obj*, otherwise `()`.

A.10.5 `<float>` *class*

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

A.10.6 `floatp` *function*

A.10.6.1 Arguments

obj: Object to examine.

A.10.6.2 Result

If the class of *obj* is a subclass of `float` the result is *obj*, otherwise `()`.

A.10.7 `arithmetic-condition` *condition*

A.10.7.1 Init-options

`operator` *object*: The operator which signalled the condition.

`operand-list` *list*: The operands passed to the operator.

A.10.7.2 Remarks

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

A.10.8 `equal` *method*

A.10.8.1 Arguments

$number_1$: an instance of `number`.

$number_2$: an instance of `number`.

A.10.8.2 Result

If the class of $number_1$ and $number_2$ is the same subclass of `number`, the result of `equal` is the result of `=`. If the instances are not of the same subclass of `number`, the result is `()`.

A.10.9 `+` *function*

A.10.9.1 Arguments

[z_1 z_2 ...]: A sequence of instances of `number`.

A.10.9.2 Result

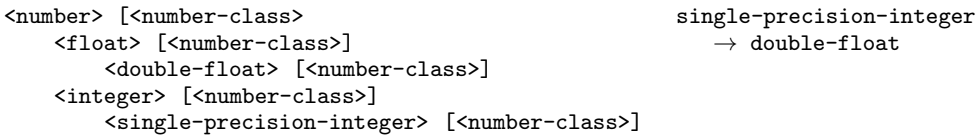
Computes the sum of the arguments using the generic function `binary-plus`. Given zero arguments, `+` returns `0` of class `integer`. One argument returns that argument. The arguments are combined left-associatively.

A.10.10 `-` *function*

A.10.10.1 Arguments

z_1 [z_2 ...]: A non-empty sequence of instances of `number`.

Figure A.1 — Level-0 number class hierarchy and coercion chart



A.10.10.2 Result

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

A.10.11 `*` *function*

A.10.11.1 Arguments

[z_1 z_2 ...]: A sequence of instances of `number`.

A.10.11.2 Result

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

A.10.12 `/` *function*

A.10.12.1 Arguments

z_1 [z_2 ...]: A non-empty sequence of instances of `number`.

A.10.12.2 Result

Computes the result of dividing the first argument by its succeeding arguments using the generic function `binary-divide`. Zero arguments is an error. One argument computes the reciprocal of the argument.

A.10.13 `<` *function*

A.10.13.1 Arguments

x_1 x_2 ...: A sequence of at least two instances of `number`.

A.10.13.2 Result

Determines whether the sequence of numbers x_1 up to x_n is strictly increasing according to the generic function `binary-lt`.

A.10.14 `>` *function*

A.10.14.1 Arguments

x_1 x_2 ...: A sequence of at least two instances of `number`.

A.10.14.2 Result

Determines whether the sequence of numbers x_1 up to x_n is strictly decreasing, according to the generic function `binary-lt`.

A.10.15 `<=` *function*

A.10.15.1 Arguments

x_1 x_2 ...: A sequence of at least two instances of `number`.

A.10.15.2 Result

Determines whether the sequence of numbers x_1 up to x_n is increasing, according to the generic function `binary-le`.

A.10.16 `>=` *function*

A.10.16.1 Arguments

x_1 x_2 ...: A sequence of at least two instances of `number`.

A.10.16.2 Result

Determines whether the sequence of numbers x_1 up to x_n is decreasing, according to the generic function `binary-ge`.

A.10.17 `max` *function*

A.10.17.1 Arguments

x_1 [x_2 ...]: A non-empty sequence of instances of `number`.

A.10.17.2 Result

Determines the maximal element of the numbers x_1 up to x_n using the generic function `binary-lt`. Zero arguments is an error. One argument returns x_1.

A.10.18 `min` *function*

A.10.18.1 Arguments

x_1 [x_2 ...]: A non-empty sequence of instances of `number`.

A.10.18.2 Result

Determines the minimal element of the numbers x_1 up to x_n using the generic function `binary-lt`. Zero arguments is an error. One argument returns x_1.

A.10.19 `gcd` *generic function*

A.10.19.1 Arguments

z_1 [z_2 ...]: A non-empty sequence of instances of number.

A.10.19.2 Result

Computes the greatest common divisor of z_1 up to z_n using the generic function `binary-gcd`. Zero arguments is an error. One argument returns z_1.

A.10.20 `lcm` *generic function*

A.10.20.1 Arguments

q_1 [q_2 ...]: A non-empty sequence of instances of `number`.

A.10.20.2 Result

Computes the least common multiple of q_1 up to q_n using the generic function `binary-lcm`. Zero arguments is an error. One argument returns q_1.

A.10.21 `abs` *generic function*

A.10.21.1 Arguments

z: An instance of `number`.

A.10.21.2 Result

Compute the absolute value of z.

A.10.22 `zerop` *generic function*

A.10.22.1 Arguments

x: An instance of `number`.

A.10.22.2 Result

Compares z with the zero element of the class of z using the generic function `=`.

A.10.23 `signum` *generic function*

A.10.23.1 Arguments

x: An instance of `number`.

A.10.23.2 Result

If `zerop` x then returns x. Otherwise returns the result of converting ± 1 to the class of x with the sign of x.

A.10.24 `positivep` *generic function*

A.10.24.1 Arguments

x: An instance of `number`.

A.10.24.2 Result

Compares x against the zero element of the class of x using the generic function `binary-lt`.

A.10.25 `negativep` *generic function*

A.10.25.1 Arguments

x: An instance of `number`.

A.10.25.2 Result

Compares x against the zero element of the class of x using the generic function `binary-lt`.

A.10.26 `binary-plus` *generic function*

A.10.26.1 Arguments

z_1 z_2: Two instances of `number`.

A.10.26.2 Result

Compute the sum of z_1 and z_2.

A.10.27 `binary-difference` *generic function*

A.10.27.1 Arguments

z_1 z_2: Two instances of `number`.

A.10.27.2 Result

Compute the difference of z_1 and z_2.

A.10.28 `negate` *generic function*

A.10.28.1 Arguments

z: An instance of `number`.

A.10.28.2 Result

Compute the additive inverse of x.

A.10.29 `binary-times` *generic function*

A.10.29.1 Arguments

z_1 z_2: Two instances of `number`.

A.10.29.2 Result

Compute the product of z_1 and z_2.

A.10.30 `binary-divide` *generic function*

A.10.30.1 Arguments

z_1 z_2: Two instances of `number`.

A.10.30.2 Result

Compute the ratio of z_1 and z_2. If the divisor is the zero element of the class an error is signaled (condition: `division-by-zero`).

A.10.31 `binary-lt` *generic function*

A.10.31.1 Arguments

x_1 x_2: Two instances of `number`.

A.10.31.2 Result

Compare x_1 with x_2 returning `t` if x_1 precedes x_2 according to the ordering method of the class of higher class of x_1 and x_2.

A.10.32 `binary-gcd` *generic function*

A.10.32.1 Arguments

q_1 q_2: Two instances of `number`.

A.10.32.2 Result

Compute the greatest common divisor of q_1 and q_2.

A.10.33 `binary-lcm` *generic function*

A.10.33.1 Arguments

q_1 q_2: Two instances of `number`.

A.10.33.2 Result

Compute the lowest common multiple of q_1 and q_2.

A.11 Pairs and Lists

The defined name of this module is `pair`.

A.11.1 `pair` *syntax*

A pair is written as `(`obj_1 `.` obj_2`)`, where obj_1 is the `car` and obj_2 is the `cdr`. There are two special cases in the print representation of pair. If obj_2 is the empty list, then the pair is written as `(`obj_1`)`. If obj_2 is an instance of `pair`, then the pair is written as `(`obj_1 obj_3 `.` obj_4`)`, where obj_3 is the `car` of obj_2 and obj_4 is the `cdr` with the above rule for the empty list applying. By induction, a list of length n is written as `(`obj_1 $\ldots$ obj_{n-1} `.` obj_n`)`, with the above rule for the empty list applying. The representations of obj_1 and obj_2 are determined by the external representations defined in other sections of this definition (see `<character>`(A.1), `<double-float>`(A.6), `<null>`(A.9), `<spint>`(A.12), `<string>`(A.14), `<symbol>`(A.15) and `<vector>`(A.17), as well as instances of `<pair>` itself. The syntax for the external representation of pairs and lists is defined in Table A.6.

A.11.2 `<pair>` *class*

The class of all instances of `<pair>`. Instances of the class `<pair>` (also known informally as a *dotted pair*) is a 2-tuple, whose slots are called, for historical reasons, car and cdr. Pairs are created by the function `cons` and the slots are accessed by the functions `car` and `cdr`. The major use of pairs is in the construction of (proper) lists. A (proper) list is defined as either the empty list (denoted by `()`) or a pair whose `cdr` is a proper list. An improper list is one containing a `cdr` which is not a list (see Table A.6.

A.11.3 `consp` *function*

A.11.3.1 Arguments

obj: Object to examine.

A.11.3.2 Result

Returns *obj* if *obj* is a subclass of `pair`, otherwise `()`.

A.11.4 `atom` *function*

A.11.4.1 Arguments

obj: Object to examine.

A.11.4.2 Result

If *obj* is not an instance `pair`, *obj* is returned, otherwise `()`.

A.11.5 `cons` *function*

A.11.5.1 Arguments

Table A.6 — Pair and List Syntax

pair	::=	*pair-begin* obj_1 *pair-separator* obj_2 *pair-end*
pair-begin	::=	`(`
pair-separator	::=	*whitespace* `.` *whitespace*
pair-end	::=	`)`
list	::=	*empty-list* | *list-head*
empty-list	::=	`()`
list-head	::=	*pair-begin* obj_1 $list\text{-}tail_1$
$list\text{-}tail_i$	::=	$improper\text{-}tail_i$ | *pair-end* | obj_i $list\text{-}tail_{i+1}$
$improper\text{-}tail_i$	::=	*pair-separator* obj_i

obj_1: An object to be stored in the car field of the result pair.

obj_2: An object to be stored in the cdr field of the result pair.

A.11.5.2 Result

Allocates a new pair initialized to obj_1 and obj_2.

A.11.6 `car` *function*

A.11.7 `cdr` *function*

A.11.7.1 Arguments

pair: An instance of `pair`.

A.11.7.2 Result

Given an instance of `pair`, such as the result of `(cons` obj_1 obj_2`)`, the function `car` returns obj_1 and `cdr` returns obj_2.

A.11.7.3 Remarks

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

A.11.8 `(setter car)` *setter*

A.11.9 `(setter cdr)` *setter*

A.11.9.1 Arguments

pair: An instance of `pair`.

obj: An object.

A.11.9.2 Result

Given an instance of `pair`, such as the result of `(cons` obj_1 obj_2`)`, the function `(setter car)` replaces obj_1 with *obj* and `(setter cdr)` replaces obj_2 with *obj*. The setter functions return *obj*.

A.11.9.3 Remarks

Note that if *obj* is not `()`, then the use of `(setter cdr)` will make *pair* an improper list. It is an error to apply these `setter` functions to anything other than a pair.

A.11.10 `(converter string)` *method*

A.11.10.1 Arguments

pair: A list of characters.

A.11.10.2 Result

Constructs and returns a string, the characters of which correspond to the characters comprising the first elements of the top-level pairs of the instance of `pair`. It is an error if the source is not a proper list. An error is signaled (condition: `not-a-character`) unless all of those elements are instances of the class `character`.

A.11.11 `not-a-character` *conversion-condition*

A.11.11.1 Init-options

`source` *list*: The list of objects to be converted into a string.

A.11.11.2 Remarks

Signalled by the `pair` method of `(converter string)` unless the source list contains only characters.

A.11.12 `(converter string)` *method*

A.11.12.1 Arguments

pair: A list of characters.

A.11.12.2 Result

Constructs and returns a vector the elements of which correspond to first elements of the top-level pairs in the instance of `pair`. It is an error if the source is not a proper list.

A.11.13 `equal` *method*

A.11.13.1 Arguments

pair$_1$: an instance of `pair`.

pair$_2$: an instance of `pair`.

A.11.13.2 Result

The result is the conjunction of the pairwise application of `equal` to the `car` fields and the `cdr` fields of the arguments.

A.11.14 `copy` *method*

A.11.14.1 Arguments

pair: A pair.

A.11.14.2 Result

Constructs and returns an instance of `pair` whose elements are the same as those of the source (under `eql`), so that the resulting pair is the same (under `equal`) as the source.

A.11.15 `list` *function*

A.11.15.1 Arguments

[*obj*$_1$ ... *obj*$_n$]: A sequence of objects.

A.11.15.2 Result

Allocates a set of pairs each of which has been initialized with *obj*$_i$ in the `car` field and the pair whose `car` field contains *obj*$_{i+1}$ in the `cdr` field. Returns the pair whose `car` field contains *obj*$_1$.

A.11.16 `length` *method*

A.11.16.1 Arguments

pair: A pair.

A.11.16.2 Result

Returns the count of the number of top-level pairs in list.

A.11.16.3 Remarks

The *list* need not be a proper list.

A.11.16.4 Examples

```
(length ())                     ;;result is 0
(length (cons 1 ()))            ;;result is 1
(length (cons 1 . 2))           ;;result is 1
(length (cons 1 (cons 2 . 3)))  ;;result is 2
```

A.11.17 `copy-alist` *function*

A.11.17.1 Arguments

alist: A proper list of pairs.

A.11.17.2 Result

Constructs and returns a copy of the list *alist* copying both the top-level pairs and the second level pairs (the associations).

A.11.18 `copy-list` *function*

A.11.18.1 Arguments

list: A list—proper or improper.

A.11.18.2 Result

Constructs and returns a copy of *list* by copying the top-level pairs only.

A.11.19 `copy-tree` *function*

A.11.19.1 Arguments

list: A list—proper or improper.

A.11.19.2 Result

Constructs and returns a copy of *list* by copying the top-level pairs and then operates recursively on each of those pairs, thus copying every pair in *list*.

A.11.20 `generic-prin` *method*

A.11.21 `generic-write` *method*

A.11.21.1 Arguments

pair: The pair to be output on *stream*.

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

A.11.21.2 Result

The pair supplied as the first argument.

A.11.21.3 Remarks

Output the external representation of *pair* on *stream* as described in the introduction to this section. Both methods call the generic function again to produce the external representation of the car and cdr slots of the pair.

Table A.8 — Methods for single precision integers

`binary-plus`
`binary-difference`
`negate`
`binary-times`
`binary-lt`
`binary-gcd`
`binary-lcm`
`abs`
`zerop`
`signum`
`positivep`
`negativep`

A.12 Single Precision Integers

The defined name of this module is `spint`.

A.12.1 `single-precision-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 syntax for the external representation of integer literals is defined in Table A.7.

NOTE — At present this document 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`, with the shorter alias `vpint`. The syntax given here is applicable to both single and variable precision integers.

The class is named `<single-precision-integer>`, but it is also called by the shorter alias `<spint>`. Arithmetic operations for `single-precision-integer` are defined by methods to be attached to the generic functions listed in Table A.8.

A.12.2 `<single-precision-integer>` *class*

The class of all instances of single precision integers. Also named `<spint>`.

A.12.3 `single-precision-integer-p` *function*

A.12.3.1 Arguments

obj: Object to examine.

A.12.3.2 Result

Returns *obj* if *obj* is an instance of `<spint>`.

A.12.4 `evenp` *generic function*

A.12.4.1 Arguments

number: An instance of `number`.

A.12.4.2 Result

If *number* is an instance of a suitable subclass of `number`, returns `t` if the remainder from dividing i by two is zero, otherwise `()`.

A.12.5 `evenp` *method*

A.12.5.1 Remarks

Implements `evenp` for single precision integers.

A.12.6 `oddp` *generic function*

A.12.6.1 Arguments

number: An instance of `number`.

A.12.6.2 Result

If *number* is an instance of a suitable subclass of `number`, returns `t` if the remainder from dividing i by two is non-zero, otherwise `()`.

A.12.7 `oddp` *method*

Implements `oddp` for single precision integer.

A.12.8 `division-by-zero` *arithmetic-condition*

Signalled by any of `quotient`, `remainder` and `modulo` if their second argument is zero.

A.12.9 `quotient` *generic function*

A.12.9.1 Arguments

$integer_1$ $integer_2$: Two instances of `integer`.

A.12.9.2 Result

If *number* is an instance of a suitable subclass of `number`, returns q by solving the equation $number_1 = number_2 \times q + r$, where r lies between zero (inclusive) and the $number_2 \times sign(number_1)$ (exclusive). Sign combination for `quotient` is as follows:

Table A.7 — Integer Syntax

integer	::=	[*sign*] *uinteger*
sign	::=	{`+` | `-`}
uinteger	::=	*ubinary* | *uoctal* | *udecimal* | *uhexadecimal* | *uinteger-with-base*
ubinary	::=	*extension* {`b` | `B`} *digit(2)*$^+$
uoctal	::=	*extension* {`o` | `O`} *digit(8)*$^+$
udecimal	::=	*digit(10)*$^+$
uhexadecimal	::=	*extension* {`x` | `X`} *digit(16)*$^+$
uinteger-with-base	::=	*extension udecimal* {`r` | `R`} *digit(udecimal)*$^+$
digit(2)	::=	{`0` | `1`}
...		...
digit(10)	::=	{`0` | ... | `9`}
digit(11)	::=	{`0` | ... | `9` | {`a` | `A`}}
...		...
digit(35)	::=	{`0` | ... | `9` | {`a` | `A`} | ... | {`y` | `Y`}}
digit(36)	::=	{`0` | ... | `9` | {`a` | `A`} | ... | {`y` | `Y`} | {`z` | `Z`}}

	$integer_2$	$-integer_2$
$integer_1$	q	$-q$
$-integer_1$	$-q$	q

A.12.10 `quotient` *method*

A.12.10.1 Remarks

Implements `quotient` for single precision integer. An error is signalled (condition: `division-by-zero`) if $integer_2$ is zero.

A.12.11 `remainder` *generic function*

i$_1$ i$_2$i

A.12.11.1 Arguments

$integer_1$ $integer_2$: Two instances of `integer`.

A.12.11.2 Result

If *integer* is an instance of a suitable subclass of `integer`, returns r by solving the equation $integer_1 = integer_2 \times q + r$, where r lies between zero (inclusive) and the $integer_2 \times sign(integer_1)$ (exclusive). Sign combination for `remainder` is as follows:

	$integer_2$	$-integer_2$
$integer_1$	r	$-r$
$-integer_1$	$-r$	r

A.12.12 `remainder` *method*

A.12.12.1 Remarks

Implements `remainder` for single precision integer. An error is signalled (condition: `division-by-zero`) if $integer_2$ is zero.

A.12.13 `modulo` *generic function*

i$_1$ i$_2$i

A.12.13.1 Arguments

i_1 i_2: Two instances of `integer`.

A.12.13.2 Result

If i is an instance of a suitable subclass of `integer`, returns m by solving the equation $integer_1 = integer_2 \times q + r$, where r lies between zero (inclusive) and the $integer_2 \times sign(integer_1)$ (exclusive) and m is constrained by $0 \leq m < |q|$. Sign combination for `modulo` is given in Table A.9.

A.12.14 `modulo` *method*

A.12.14.1 Remarks

Implements `modulo` for single precision integer. An error is signalled (condition: `division-by-zero`) if $integer_2$ is zero.

A.12.15 `most-positive-single-precision-integer` *single-precision-integer*

A.12.15.1 Remarks

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

A.12.16 `most-negative-single-precision-integer` *single-precision-integer*

A.12.16.1 Remarks

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

A.12.17 `(converter character)` *method*

A.12.17.1 Arguments

single-precision-integer: A single-precision-integer.

Table A.9 — Sign combination in `modulo`

	$integer_2$	$-integer_2$
$integer_1$	r	`remainder(`i_2`+remainder(`i_1`,`$i_2`),`$i_2`)`
$-integer_1$	`remainder(`i_2`+remainder(`i_1`,`$i_2`),`$i_2`)`	r

A.12.17.2 Result

Returns an instance of `character` whose position in the default character set corresponds to that specified by the instance of `integer`. An error is signaled (condition: `cannot-convert-to-character`) if the specified position does not exist.

A.12.18 `no-such-character` *conversion-condition*

Signalled by the conversion method from `<spint>` to `<character>` if there is no character in the defaul character set corresponding to the position specified by the integer.

A.12.18.1 See also: `(converter character)`.

A.12.19 `(converter string)` *method*

A.12.19.1 Arguments

single-precision-integer: A single-precision-integer.

A.12.19.2 Result

Constructs and returns a string, the characters of which correspond to the external representation of the instance of `single-precision-integer` in decimal.

A.12.20 `(converter double-float)` *method*

A.12.20.1 Arguments

single-precision-integer: A single precision integer.

A.12.20.2 Result

Returns an instance of `double-float` whose value is the floating point approximation to the single precision integer source.

A.12.21 `copy` *method*

A.12.21.1 Arguments

single-precision-integer: A single precision integer.

A.12.21.2 Result

Constructs and returns an instance of `single-precision-integer`, whose value is the same (under `=`) as the source.

A.12.22 `generic-prin` *method*

A.12.23 `generic-write` *method*

A.12.23.1 Arguments

i: The single precision integer to be output on *stream*.

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

A.12.23.2 Result

The single precision integer supplied as the first argument.

A.12.23.3 Remarks

Output external representation of *i* on *stream* in decimal as described in the introduction to this section.

Table A.10 — Initial stream class hiearchy

```
<stream> [<stream-class>]
  <file-stream> [<stream>]
<stream-properties> [<stream-properties-class>]
  <stream-direction> [<stream-properties>]
    <input-stream>
    <output-stream>
    <io-stream>
  <stream-unit> [<stream-properties>]
    ...
  <stream-positionable-p> [<stream-properties>]
    ...
```

A.13 Streams

The defined name of this module is `stream`.

A.13.1 `<stream>` *class*

The abstract class of all instances of `<stream>`.

The initial class hiearchy of streams and stream properties is shown in Table A.10.

A.13.2 `<file-stream>` *class*

The class of all instances of `<file-stream>`. This is the only subclass of stream defined at level-0.

A.13.2.1 Init-options

`direction` *direction*: An instance of `<stream-direction>`.

`transaction-unit` *unit*: An instance of `<stream-unit>`, or `<character>`(indicating a character stream) or `<spint>`(indicating a binary stream).

`positionable` *boolean*: Either `()` or non-`()`.

A.13.3 `input-stream` *stream-direction*

Used to indicate stream direction when making instances of `<file-stream>`.

A.13.4 `io-stream` *stream-direction*

Used to indicate stream direction when making instances of `<file-stream>`.

A.13.5 `ouput-stream` *stream-direction*

Used to indicate stream direction when making instances of `<file-stream>`.

A.13.6 `file-streamp` *function*

A.13.6.1 Arguments

obj: object to examine.

A.13.6.2 Result

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

A.13.7 `stream-condition` *condition*

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

A.13.8 `syntax-error` *condition*

This is the general condition class for conditions arising from using the function `read`.

A.13.9 `input-stream-p` *function*

A.13.9.1 Arguments

obj: Object to examine.

A.13.9.2 Result

The supplied argument if one of its stream direction is an instance of `<input-stream>`, otherwise `()`.

A.13.10 `output-stream-p` *function*

A.13.10.1 Arguments

obj: Object to examine.

A.13.10.2 Result

The supplied argument if one of its stream direction is an instance of `<output-stream>`, otherwise `()`.

A.13.11 `io-stream-p` *function*

A.13.11.1 Arguments

obj: Object to examine.

A.13.11.2 Result

The supplied argument if one of its stream direction is an instance of `<output-stream>`, otherwise `()`.

A.13.12 `character-stream-p` *function*

A.13.12.1 Arguments

obj: Object to examine.

A.13.12.2 Result

The supplied argument if one of its properties is an instance of `<character-stream>`, otherwise `()`.

A.13.13 `binary-stream-p` *function*

A.13.13.1 Arguments

obj: Object to examine.

A.13.13.2 Result

The supplied argument if one of its properties is an instance of `<binary-stream>`, otherwise `()`.

A.13.14 `open` *generic function*

A.13.14.1 Generic Arguments

(*stream* `<stream>`): The stream to be opened.

(*handle* `<object>`): The name of the object (internal or external to the processor).

(*initlist* `<list>`): A list of initialization options.

A.13.14.2 Result

A stream.

A.13.15 `open` *method*

A.13.15.1 Specialized Arguments

(*stream* `<file-stream>`): The stream to be opened.

(*handle* `<string>`): The name of the file to be openend.

(*initlist* `<list>`): A list of initialization options as follows:

`direction` *direction*: An instance of `<stream-direction>`.

`transaction-unit` *unit*: An instance of `<stream-unit>`.

`positionable` *boolean*: Either `()` or non-`()`.

A.13.15.2 Result

A stream.

NOTE (version 0.96) — In a future version it is forseen that a more general approach to file naming will be adopted, such as the pathnames of Common Lisp. At that time it will be appropriate to add another method to `open`, where the *handle* argument is an instance of `pathname`. Thus, at present, no implementation independent way of naming files is provided.

A.13.16 `open-p` *generic function*

A.13.16.1 Generic Arguments

(*stream* `<stream>`): A stream.

A.13.16.2 Result

A non-`()` value if *stream* is open, otherwise `()`.

A.13.17 `open-p` *method*

A.13.17.1 Specialized Arguments

(*stream* `<file-stream>`): A file stream.

A.13.17.2 Result

A non-`()` value if *stream* is open, otherwise `()`.

A.13.18 `close` *generic function*

A.13.18.1 Generic Arguments

(*stream* `<stream>`): A stream.

A.13.18.2 Result

If *stream* is open, it is closed. Returns `()`).

A.13.19 `close` *method*

A.13.19.1 Specialized Arguments

(*stream* `<file-stream>`): A file stream.

A.13.19.2 Result

If *stream* is open, it is closed. Returns `()`).

A.13.20 `write-unit` *generic function*

A.13.20.1 Generic Arguments

(*stream* `<stream>`): A stream.

(*object* `<object>`): An object.

A.13.20.2 Result

The *object* is written to *stream*. Returns `()`.

A.13.21 `write-unit` *method*

A.13.21.1 Specialized Arguments

(*stream* `<file-stream>`): A file stream.

(*character* `<character>`): A character.

A.13.21.2 Result

If the transaction unit of *stream* is `<character>`, the *character* is written to *stream*. Returns `()`.

A.13.22 `write-unit` *method*

A.13.22.1 Specialized Arguments

(*stream* `<file-stream>`): A file stream.

(*integer* `<spint>`): A single precision integer.

A.13.22.2 Result

If the transaction unit of *stream* is `<spint>`, the *integer* is written to *stream*. Returns `()`.

A.13.23 `write` *function*

A.13.23.1 Arguments

object:

[*stream*]:

A.13.23.2 Result

If *stream* is open, *object* is output on *stream*.

A.13.24 `generic-write` *generic function*

A.13.24.1 Generic Arguments

(*stream* `<stream>`): The stream on which *object* is to be output.

(*object* `<object>`): The object to be output on *stream*.

A.13.24.2 Result

If *stream* is open, *object* is output on *stream*.

A.13.24.3 See also: Sections on the different classes defined at level-0 for methods oon this function.

A.13.25 `prin` *function*

A.13.25.1 Arguments

object:

[*stream*]:

A.13.25.2 Result

If *stream* is open, *object* is output on *stream*.

A.13.26 `generic-prin` *generic function*

A.13.26.1 Generic Arguments

(*stream* `<stream>`): The stream on which *object* is to be output.

(*object* `<object>`): The object to be output on *stream*.

A.13.26.2 Result

If *stream* is open, *object* is output on *stream*.

A.13.26.3 See also: Sections on the different classes defined at level-0 for methods oon this function.

A.13.27 `read-unit` *generic function*

A.13.27.1 Generic Arguments

(*stream* `<stream>`): A stream.

A.13.27.2 Result

The next object input from *stream*. The class of the object is restricted to that of the transaction unit of the *stream*.

A.13.28 `read-unit` *method*

A.13.28.1 Specialized Arguments

(*stream* `<file-stream>`): A file stream.

A.13.28.2 Result

An object representing the next transaction unit input from *stream*. The class of this object is determined by the transaction unit property of *stream*.

A.13.29 `read` *function*

A.13.29.1 Arguments

[*stream*]: An instance of `stream`.

A.13.29.2 Result

An object corresponding to the next element processed by `read` from *stream*.

A.13.29.3 Remarks

When a syntax error is detected an error is signaled (condition: `syntax-error`). The position of the stream being read is undefined after such an error occurs.

A.13.30 `generic-read` *generic function*

A.13.30.1 Generic Arguments

(*stream* `<stream>`): A stream.

(*prototype* `<object>`): An object.

A.13.30.2 Result

The next object input from *stream*.

A.13.31 `generic-read` *method*

A.13.31.1 Specialized Arguments

(*stream* `<file-stream>`): A file stream.

(*prototype* `<object>`): An object.

A.13.31.2 Result

The next object input from *stream*. What is *protptype* for?

A.13.32 `peek-unit` *generic function*

A.13.32.1 Generic Arguments

(*stream* `<stream>`): A stream.

A.13.32.2 Result

The next object that could be input from *stream*. The class of the object is restricted to that of the transaction unit of the *stream*.

A.13.33 `peek-unit` *method*

A.13.33.1 Specialized Arguments

(*stream* `<file-stream>`): A file stream.

A.13.33.2 Result

An object representing the next transaction unit that oculd be input from *stream*. The class of this object is determined by the transaction unit property of *stream*.

A.13.34 `flush` *generic function*

A.13.34.1 Generic Arguments

(*stream* `<stream>`): A stream.

A.13.34.2 Result

Any remaining buffered output to stream is forced out.

A.13.35 `flush` *method*

A.13.35.1 Specialized Arguments

(*stream* `<file-stream>`): A file stream.

A.13.35.2 Result

Any remaining buffered output to stream is forced out.

A.13.36 `wait` *method*

A.13.36.1 Specialized Arguments

(*stream* `<file-stream>`): The `file-stream` on which to wait.

(*timeout* `<object>`): The timeout period.

A.13.36.2 Result

Returns *stream* if *stream* is an input-stream or an io-stream and has input available.

A.13.36.3 See also: `wait`.

A.14 Strings

The defined name of this module is `string`.

A.14.1 `string` *syntax*

String literals are delimited by the *string-begin* and *string-end* glyphs, which are both defined as 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 section A.1) have a particular syntax to allow their appearance in strings. To allow arbitrary characters to appear in strings, the argument to the hex-insertion digram is an integer denoting the position of the character in the current character set. In the case of hex insertion, each *d* denotes any hexadecimal digit. A summary of *string-escape* digrams appears in Table A.11. Some examples of string literals appear in Table A.12.

Table A.11 — String escape digrams

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

The syntax for the external representation of strings is defined in Table A.13.

NOTE — At present this document refers to the "current character set" but defines no means of selecting alternative character sets. This is to allow for future extensions and 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 `#\newline`, `#\a`, `#\\` and `#\b`. The function `write` produces `"a\n\\b"`, whilst `prin` produces

```
a
\b
```

Characters which do not have a glyph associated with their position in the character set are output as a hex insertion in which all four hex digits are specified, even if there are leading zeros. The function `prin` outputs the interpretation of the characters according to the definitions in section A.1 and omits the *string-begin* and *string-end* characters.

Table A.12 — Examples of string literals

Example	Contents
`"a\nb"`	`#\a`, `#\newline` 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`

A.14.2 `<string>` *class*

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

A.14.2.1 Init-options

`size` *spint*: The number of characters in the string. Strings are zero-based and thus the maximum index is *size-1*. If not supplied the *size* is zero.

`fill` *character*: A character with which to initialize the string. If not supplied the fill character is `#\x0`.

A.14.3 `stringp` *function*

A.14.3.1 Arguments

obj: Object to examine.

A.14.3.2 Result

Returns *obj* if *obj* is an instance of a subclass `string`, otherwise `()`.

A.14.4 `string-ref` *function*

A.14.5 `(setter string-ref)` *setter*

A.14.5.1 Arguments

string: Source string.

n: An instance of `single-precision-integer`.

character: An instance of character (for setter).

A.14.5.2 Result

Access and update elements of a string. It is an error if *n* is outside the range zero to one less than the length of the string.

A.14.6 `(converter pair)` *method*

A.14.6.1 Specialized Arguments

Table A.13 — String Syntax

string	::=	*string-begin string-char* string-end*
string-begin	::=	"
string-char	::=	*normal-char* | *escape-diphthong*
string-end	::=	"
normal-char	::=	*any-character-begin-end-or-escape-excepted*
escape-diphthong	::=	*string-escape string-insertion*
string-escape	::=	\
string-insertion	::=	*string-begin* | *string-end* | *string-escape* | *hex-insertion* | *string-newline*
hex-insertion	::=	*string-hex digit(16)* [*digit(16)* [*digit(16)* [*digit(16)*]]]
string-newline	::=	n | N
string-hex	::=	x | X

(*string* <string>): A string to be converted to a list of characters.

A.14.6.2 Result

Constructs and returns a proper list of characters, the elements of which correspond to the characters in the external representation of the instance of `string` as would be generated by `write`.

A.14.7 `equal` *method*

A.14.7.1 Specialized Arguments

(*string*$_1$ <string>): an instance of `string`.

(*string*$_2$ <string>): an instance of `string`.

A.14.7.2 Result

If the length of each instance of `string` is the same, then the result is the conjunction of the pairwise application of `equal` to the elements of the arguments. If not the result is `()`.

A.14.8 `copy` *method*

A.14.8.1 Specialized Arguments

(*string* <string>): A string.

A.14.8.2 Result

Constructs and returns an instance of `string`, whose characters are the same as the source and such that the resulting string is the same (under `equal`) as the source.

A.14.9 `length` *method*

A.14.9.1 Specialized Arguments

(*string* <string>): An instance of `string`.

A.14.9.2 Result

Returns the number of characters in *string*.

A.14.10 `string-lt` *function*

A.14.10.1 Arguments

string$_1$ *string*$_2$: Two instances of `string`.

A.14.10.2 Result

If the sequence of characters in *string*$_1$ is alphabetically less than that in *string*$_2$ returns `t`, else `()`.

A.14.11 `string-slice` *function*

A.14.11.1 Arguments

string: An instance of `string`.

start: An instance of `single-precision-integer`.

: An instance of `single-precision-integer`.

A.14.11.2 Result

Returns a newly allocated string containing the characters of *string* starting at *start* up to *end*.

A.14.12 `string-append` *function*

A.14.12.1 Arguments

string$_1$ *string*$_2$: Two instances of `string`.

A.14.12.2 Result

Returns a newly allocated string containing the characters of *string*$_1$ followed by the characters of *string*$_2$.

A.14.13 `generic-prin` *method*

A.14.13.1 Specialized Arguments

(*string* <string>): String to be ouptut on *stream*.

(*stream* <stream>): Stream on which *string* is to be ouptut.

A.14.13.2 Result

The string *string*.

Output external representation of *string* on *stream* as described in the introduction to this section, interpreting each of the characters in the string. The opening and closing quotation marks are not output.

A.14.14 `generic-write` *method*

A.14.14.1 Specialized Arguments

(*string* `<string>`): String to be ouptut on *stream*.

(*stream* `<stream>`): Stream on which *string* is to be ouptut.

A.14.14.2 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.

A.15 Symbols

The defined name of this module is `symbol`.

A.15.1 `symbol` *syntax*

The syntax of symbols also serves as the syntax for identifiers. Identifiers in EuLisp are very similar lexically to identifiers in other Lisps and in other programming languages. Informally, an identifier is a sequence of *alphabetic*, *digit* and *other* characters starting with a character that is not a *digit*. Characters which are *special* (see section A.1) 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 (`-`) have another use 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 otherwise illegal. 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-escape*s. 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-escape*s.

Any identifier can be used as a literal, in which cases it denotes a *symbol*. Because there are two escaping mechanisms and because the first character of a token affects the interpretation of the remainder, there are many ways in which to input the same identifier. If this identifier is used as a literal the results of processing each token denoting the identifier will be `eq` to one another. For example, the following tokens all denote the same symbol:

`|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`.

The syntax of the external representation of identifiers and

Table A.14 — Identifier/Symbol Syntax

symbol	::=	*identifier*
identifier	::=	*normal-identifier* | *peculiar-identifier*
normal-identifier	::=	*non-digit constituent**
peculiar-identifier	::=	*sign-identifier* | *point-identifier*
sign-identifier	::=	{*sign* | *sign* {*non-digit* | *sign* | *point*}} *constituent**
point-identifier	::=	*point* {*non-digit* | *sign* | *point*} *constituent**
constituent	::=	*non-digit* | *digit* | *sign* | *point*
alphanumeric	::=	*alphabetic* | *digit*
non-digit	::=	*alphabetic* | *other* | *escaped*
escaped	::=	*single-escaped* | *multiple-escaped*
multiple-escaped	::=	*multiple-escape* {*single-escaped* | *non-escape*}* *multiple-escape*
single-escaped	::=	*single-escape* {*any-character*}
non-escape	::=	not(*single-escape, multiple-escape*)
multiple-escape	::=	|
single-escape	::=	\
point	::=	.

symbols is defined in Table A.14.

A.15.2 `<symbol>` *class*

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

A.15.2.1 Init-options

`string` *string*: The string containing the characters to be used to name the symbol.

A.15.3 `symbolp` *function*

A.15.3.1 Arguments

obj: Object to examine.

A.15.3.2 Result

Returns *obj* if *obj* is an instance of a subclass of `symbol`.

A.15.4 `gensym` *function*

A.15.4.1 Arguments

[*string*]: A string contain characters to be prepended to the name of the new symbol.

A.15.4.2 Result

Makes a new symbol with a name generated by a processor-defined mechanism. Optionally, a prefix string for this name may be specified.

A.15.5 `symbol-name` *function*

A.15.5.1 Arguments

symbol: An instance of `symbol`.

A.15.5.2 Result

Returns a *string* which is `equal` to that given as the argument to the call to `make-symbol` which created *symbol*.

A.15.6 `symbol-exists-p` *function*

A.15.6.1 Arguments

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

A.15.6.2 Result

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

A.15.7 `generic-prin` *method*

A.15.8 `generic-write` *method*

A.15.8.1 Specialized Arguments

(*symbol* `<symbol>`): The symbol to be output on *stream*.

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

A.15.8.2 Result

The symbol supplied as the first argument.

A.15.8.3 Remarks

Outputs the external representation of *symbol* on *stream* as described in the introduction to this section, interpreting each of the characters in the name.

A.15.9 `generic-write` *method*

A.15.9.1 Specialized Arguments

(*symbol* `<symbol>`): The symbol to be output on *stream*.

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

A.15.9.2 Result

The symbol supplied as the first argument.

A.15.9.3 Remarks

Outputs the external representation of *symbol* on *stream* as described in the introduction to this section. If any characters in the name would not normally be legal constituents of an identifier or symbol, the output is preceded and succeeded by multiple-escape characters.

A.16 Tables

The defined name of this module is `table`. Tables provide a general key to value association mechanism.

A.16.1 `<table>` *class*

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

A.16.1.1 Init-options

`comparator` *function*: The function to be used to compare keys.

A.16.2 `tablep` *function*

A.16.2.1 Arguments

obj: Object to examine.

A.16.2.2 Result

Returns *obj* is *obj* is an instance of `table`.

A.16.3 `table-ref` *function*

A.16.3.1 Arguments

table: An instance of `table`.

key: An object to be used to identify an entry in the table.

[*no-entry-value*]: An object to be returned in the case that *key* is not found.

A.16.3.2 Result

If *key* is in *table*, matched by the comparator function, then the associated value is returned. If *key* is not in *table*, the value `()` is returned. However, if the optional parameter *no-entry-value* is provided and *key* is not in *table*, the value *no-entry-value* is returned.

A.16.4 `(setter table-ref)` *setter*

A.16.4.1 Arguments

table: An instance of `table`.

key: An object to be used to identify an entry (new or existing) in the table.

value: An object to be associated with *key* in *table*.

A.16.4.2 Result

If *key* does not occur in *table* a new entry is made associating *key* and *value*. If *key* does occur, then the association is changed to *value*. *value-obj* is returned.

A.16.5 `table-delete` *function*

A.16.5.1 Arguments

table: An instance of `table`.

key: An object to be used to identify an entry (new or existing) in the table.

A.16.5.2 Result

If *key* occurs in *table*, both the key and its associated value are deleted from the table. If *key* does not occur in *table*, no action is taken.

A.16.6 `generic-prin` *method*

A.16.7 `generic-write` *method*

A.16.7.1 Arguments

table: The table to be output on *stream*.

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

A.16.7.2 Result

The table supplied as the first argument.

A.16.7.3 Remarks

Output the external representation of *table* on *stream*. The external representation of instances of `table` is implementation-defined.

Table A.15 — Vector Syntax

vector	::=	*extension vector-begin obj* vector-end*
extension	::=	`#`
vector-begin	::=	`(`
vector-end	::=	`)`

A.17 Vectors

The defined name of this module is `vector`.

A.17.1 `vector` *syntax*

A vector is written as `#(`obj_1 ... 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>`(A.1), `<double-float>`(A.6), `<null>`(A.9), `<pair>`(A.11), `<spint>`(A.12), `<string>`(A.14) and `<symbol>`(A.15), as well as instances of `<vector>` itself. The syntax for the external representation of vectors is defined in Table A.15.

A.17.2 `<vector>` *class*

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

A.17.2.1 Init-options

`size` *spint*: 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` *object*: An object with which to initialize the string. If not supplied the fill object is `()`.

A.17.3 `vectorp` *function*

A.17.3.1 Arguments

obj: Object to examine.

A.17.3.2 Result

Returns *obj* if *obj* is an instance of `vector`.

A.17.4 `length` *method*

A.17.4.1 Arguments

vector: A vector.

A.17.4.2 Result

Returns the length of *vector*, which is the maximum index plus one.

A.17.5 `vector-ref` *function*

A.17.6 `(setter vector-ref)` *setter*

A.17.6.1 Arguments

vector: A vector.

n: An integer to specify the index of an element in the *vector*.

obj: An object to be stored as element *n* of *vector*.

A.17.6.2 Result

The accessor returns and the updator changes the contents of the *n*th index of *vector*. The value stored in index position *n* is *obj*, which is returned.

A.17.7 `make-initialized-vector` *function*

A.17.7.1 Arguments

obj_1 obj_2 ... obj_n: A sequence of objects.

A.17.7.2 Result

Allocate a vector of size *n* and store obj_1 in `(vector-ref v 0)`, obj_2 in position `(vector-ref v 1)`, up to obj_n in `(vector-ref v n-1)`. Returns the initialized vector.

A.17.8 `maximum-vector-index` *integer*

A.17.8.1 Remarks

This is an implementation-defined constant. A conforming processor must support a maximum vector index of at least 32767.

A.17.9 `(converter pair)` *method*

A.17.9.1 Arguments

vector: A vector.

A.17.9.2 Result

Constructs and returns a proper list, the elements of which correspond to the elements stored in the instance of *vector*.

A.17.10 `equal` *method*

A.17.10.1 Arguments

$vector_1$: an instance of `vector`.

$vector_2$: an instance of `vector`.

A.17.10.2 Result

If the maximum index of each instance of `vector` is the same (under `=`), then the result is the conjunction of the pairwise application of `equal` to the elements of the arguments. If not the result is `()`.

A.17.11 `copy` *method*

A.17.11.1 Arguments

vector: A vector.

A.17.11.2 Result

Constructs and returns an instance of `vector`, whose elements are the same as those of the source (under `eql`), so that the resulting vector is the same (under `equal`) as the source.

A.17.12 `generic-prin` *method*

A.17.13 `generic-write` *method*

A.17.13.1 Arguments

vector: the vector to be ouptut on stream

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

A.17.13.2 Result

The vector supplied as the first argument.

A.17.13.3 Remarks

Output the external representation of *pair* on *stream* as described in the introduction to this section. Both methods call the generic function again to produce the external representation of the elements stored in the vector.

Annex B
(normative)
Programming Language EuLisp, Level-1

B.1 Classes and Objects

B.1.1 `defclass` *defining form*

B.1.1.1 Syntax

(`defclass` *class-name* (*superclass**) (*slot-description**) *class-option**)

B.1.1.2 Arguments

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

(*superclass**): A list of symbols naming bindings of classes to be used as the superclasses of the new class. This is different from `defcalss` at level-0, where only one superclass may be specified.

(*slot-description**): A list of slot specifications (see below), comprising either a *slot-name* or a list of a *slot-name* followed by some *slot-option*s. An additional class option at level-1 allows for the specification of the class of the slot description.

*class-option**: A sequence of keys and values (see below). An additional class options at level-1 allows for the specification of the metaclass of the class.

B.1.1.3 Remarks

This defining form defines a new class. The resulting class will be bound to *class-name*. The second argument is a list of superclasses. If this list is empty, the superclass is `<object>`. The third argument is a list of slot-descriptions. The remaining arguments are class options. The syntax of `defclass` is given in Table B.1.

Table B.1 — `defclass` syntax (level-1)

class-name	::=	*identifier*
superclass	::=	{`<object>` *or one of its subclasses*}
slot-description	::=	*slot-name* |
		(*slot-name slot-option**)
slot-name	::=	*identifier*
slot-option	::=	*identifier expression*
		level-0-slot-option
class-option	::=	`metaclass` *class-name* |
		identifier expression
		level-0-class-option

The additional *slot-option*s are:

identifier expression: The symbol named by *identifier* and the value of *expression* are passed to `make` of the slot description class along with other slot options. The values are evaluated in the lexical and dynamic environment of the `defclass`. For the language defined slot description classes, no slot initargs are defined which are not specified by particular `defclass` slot options.

The additional *class-option*s are:

`metaclass` *class*: The value of this option is the class of the new class. By default, this is `<class>`. This option must only be specified once for the new class.

identifier expression: The symbol named by *identifier* and the value of *expression* are passed to `make` on the class of the new class. This list is appended to the end of the list that `defclass` constructs. The values are evaluated in the lexical and dynamic environment of the `defclass`. This option is used for metaclasses which need extra information not provided by the standard options.

B.2 Generic Functions

B.2.1 `defgeneric` *defining form*

B.2.1.1 Syntax

(`defgeneric` *gf-name gen-lambda-list level-1-init-option**`)`

B.2.1.2 Arguments

gf-name: As level-0. See section 12.1.1.

lambda-list: As level-0. See section 12.1.1.

init-option*: Format as level-0, but with additional options, which are defined below.

B.2.1.3 Remarks

This defining form defines a new generic function. The resulting generic function will be bound to *gf-name*. The second argument is the formal parameter list. An error is signaled (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—see Table B.2.

The additional *init-option*s 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 or of one of its subclasses. The method class must be a subclass of `<method>` and is, by default, `<method>`.

identifier expression: The symbol named by *identifier* and the value of *expression* are passed to `make` as initargs. The values are evaluated in the lexical and dynamic environment of the `defgeneric`. This option is used for classes which need extra information not provided by the standard options.

B.2.1.4 Examples

In the following example of the use of `defgeneric` a generic function named `gf-1` is defined. The differences between this function and `gf-0` (see 12.1.1) are

a) The class of the generic function is specified (`<another-gf-class>`) along with some init-options related to the creation of an instance of that class.

b) The class of the methods to be attached to the generic function is specified (`<another-method-class>`) along with an init-option related to the creation of an instance of that class.

```
(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-a-key-a method-class-a-value-a
          ((m1-arg1 <class-b>) (m1-arg2 <class-c>))
          ...)
  method (class <another-method-class-b>
          method-class-b-key-a method-class-b-value-a
          ((m2-arg1 <class-d>) (m2-arg2 <class-e>))
          ...)
  method (class <another-method-class-c>
          method-class-c-key-a method-class-c-value-a
          ((m3-arg1 <class-f>) (m3-arg2 <class-g>))
          ...)
)
```

B.2.2 `defmethod` *macro*

B.2.2.1 Syntax

(`defmethod` *gf-name method-init-option** *spec-lambda-list form**`)`
or
(`defmethod (setter` *identifier*`)` *method-init-option** *spec-lambda-list form**`)`
or
(`defmethod (converter` *identifier*`)` *method-init-option** *spec-lambda-list form**`)`

B.2.2.2 Remarks

The syntax of `defmethod` is extended to accept init-options for the method-class of the generic function to which the method is to be attached. Otherwise, the behaviour is as that defined in level-0.

The additional *init-option*s 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 initargs to `make`.

identifier expression: The symbol named by *identifier* and the value of *expression* are passed to `make` creating a new method as initargs. The values are evaluated in the lexical and dynamic environment of the `defgeneric`. This option is used for classes which need extra information not provided by the standard options.

B.2.3 `generic-lambda` *macro*

B.2.3.1 Syntax

(`generic-lambda` *lambda-list level-1-init-option**`)`

B.2.3.2 Remarks

`generic-lambda` creates and returns an anonymous generic function that can be applied immediately, much like the normal `lambda`. The first argument is a lambda list, while the *init-option*s are interpreted exactly as for the level-1 definition of `defgeneric`. An error is signaled (condition:

Table B.2 — `defgeneric` syntax (level-1)

level-1-init-option	::=	*level-0-init-option* |
		`class` *gf-class* |
		`method-class` *method-class* |
		`method` (*level-1-method-description*) |
		gf-init-option
gf-class	::=	*a subclass of* `<generic-function>`
method-class	::=	*a subclass of* `<method>`
level-1-method-description	::=	(*method-init-option** *spec-lambda-list form**)
gf-init-option	::=	*identifier expression*
method-init-option	::=	`class` *method-class* |
		identifier expression

`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.

B.2.3.3 Examples

In the following example an anonymous version of `gf-1` (see `defgeneric` above) 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 (method-class <another-method-class-a>
          method-class-a-key-a method-class-a-value-a
          ((m1-arg1 <class-b>) (m1-arg2 <class-c>))
          ...)
  method (method-class <another-method-class-b>
          method-class-b-key-a method-class-b-value-a
          ((m2-arg1 <class-d>) (m2-arg2 <class-e>))
          ...)
  method (method-class <another-method-class-c>
          method-class-c-key-a method-class-c-value-a
          ((m3-arg1 <class-f>) (m3-arg2 <class-g>))
          ...)

)
```

B.2.3.4 See also: `defgeneric`.

B.2.4 `generic-labels` *macro*

B.2.4.1 Syntax

(`generic-labels` (*lambda-list level-1-init-option**) *form**)

B.2.4.2 Remarks

This form is analogous to the normal `labels`. The first argument is a binding list, of the same form as that specified for the level-1 definition of `defgeneric`. The lexical environment of each defined generic function includes the others, just like `labels`.

B.2.4.3 Examples

In the following example, two generic functions (`gf-1a` and `gf-1b`) are defined. In addition the same init-options, namely `class` and `method-class` can be specified.

```
(generic-labels (
  (gf-1a (arg1 (arg2 <class-a>))
    method (method-class <another-method-class-a>
            method-class-a-key-a method-class-a-value-a
            ((m1-arg1 <class-b>) (m1-arg2 <class-c>))
            ...)
    method (method-class <another-method-class-b>
            method-class-b-key-a method-class-b-value-a
            ((m2-arg1 <class-d>) (m2-arg2 <class-e>))
            ...)
    method (method-class <another-method-class-c>
            method-class-c-key-a method-class-c-value-a
            ((m3-arg1 <class-f>) (m3-arg2 <class-g>))
            ...))
  (gf-1b ((arg1 <class-b>) (arg2 <class-c>))
    method (((m1-arg1 <class-b>) (m1-arg2 <class-c>))
            ...)
    method (((m2-arg1 <class-d>) (m2-arg2 <class-e>))
            ...)))
  ...)
```

B.3 Reflection on Objects

The only reflective capability of any object is the access to its class, that is, introspection.

B.3.1 `class-of` *function*

B.3.1.1 Arguments

object: An object.

B.3.1.2 Result

The class of the object.

B.3.1.3 Remarks

`class-of` is a total function capable of taking any entity in the system and returning an object representing the class of that entity. The composition of this function with itself is a function that returns the metaclass of an object. `class-of` is a reflective operation.

B.3.1.4 Examples

In the following examples we explore the class hierarchy starting from the single precision integer `1`.

```
(class-of 1)
   → #<single-precision-integer>
(class-of (class-of 1))
   → #<class>
(class-of (class-of (class-of 1)))
   → #<metaclass>
(class-of (class-of (class-of (class-of 1))))
   → #<metaclass>
```

Table B.3 — Class and Slot Description Classes

```
<object> [<abstract-class>]
   <class> [<metaclass>]
      <metaclass> [<metaclass>]
      <abstract-class> [<metaclass>]
   <slot-description> [<class>]
```

B.4 Reflection on Classes and Slot Descriptions

Standard classes are not redefinable and support single inheritance only. General multiple inheritance or mixin inheritance can be provided by extensions. Nor is it possible to use a class as a superclass which is not defined at the time of class definition. Again, such forward reference facilities can be provided by extensions. The distinction between metaclasses and non-metaclasses is made explicit by a special class, named `<metaclass>`, which is the class of all metaclasses. This is different from ObjVlisp, where whether a class is a metaclass depends on the superclass list of the class in question. It is implementation-defined whether `<metaclass>` itself is specializable or not. This implies that implementations are free to restrict the instantiation tree (excluding the selfinstantiation loop of `<metaclass>`) to a depth of three levels.

The minimal information associated with a class metaobject is:

a) The name, which has no semantic effect.

b) The class precedence list, ordered most specific first, beginning with the class itself.

c) The list of (effective) slot descriptions.

d) The list of (effective) initargs.

Standard classes support local slots only. Shared slots can be provided by extensions. The minimal information associated with a slot description metaobject is:

a) The name, which is required to perform inheritance computations.

b) The initfunction, called by default to compute the initial slot value when creating a new instance.

c) The reader, which is a function to read the corresponding slot value of an instance.

d) The writer, which is a function to write the corresponding slot of an instance.

The metaobject classes defined for slot descriptions at level-1 are shown in Table B.3.

B.4.1 `<slot-description>` *class*

The abstract class of all slot descriptions.

B.4.2 `<local-slot-description>` *class*

The class of all local slot descriptions.

B.4.2.1 Init-options

`name` *string*: The name of the slot; useful for debugging only.

`reader` *function*: The function to access the slot.

`writer` *function*: The function to update the slot.

`initfunction` *function*: The function to compute the initial value in the absence of supplied value.

B.4.3 `class-name` *function*

B.4.3.1 Arguments

class: A class.

B.4.3.2 Result

The symbol which was given as the first argument to `defstruct` or `defclass` when the class was defined.

B.4.3.3 Remarks

The class name has no significance other than for debugging.

B.4.4 `class-precedence-list` *function*

B.4.4.1 Arguments

class: A class.

B.4.4.2 Result

A list of classes, which are the superclasses of *class* in order of increasing generality.

B.4.4.3 Remarks

The class precedence list controls the inheritance of slots and methods.

B.4.5 `class-slot-descriptions` *function*

B.4.5.1 Arguments

class: A class.

B.4.5.2 Result

A list of slot-descriptions, one for each of the slots of an instance of *class*.

B.4.5.3 Remarks

The slot-descriptions determine the instance size (number of slots) and the slot access.

B.4.6 `class-initargs` *function*

B.4.6.1 Arguments

class: A class.

B.4.6.2 Result

A list of symbols, corresponding to the slot names specified when the class was defined and any additional keywords for slot-options or class-options.

B.4.6.3 Remarks

The initargs contain the legal keywords which can be used to initialize instances of the class.

B.4.7 `slot-description-name` *function*

B.4.7.1 Arguments

slot-description: A slot description.

B.4.7.2 Result

The symbol which was used to name the slot when the class, of which the slot-description is part, was defined.

B.4.7.3 Remarks

The slot description name is used to identify a slot description in a class. It has no effect on bindings.

B.4.8 `slot-description-initfunction` *function*

B.4.8.1 Arguments

slot-description: A slot description.

B.4.8.2 Result

A function.

B.4.9 `slot-description-slot-reader` *function*

B.4.9.1 Arguments

slot-description: A slot description.

B.4.9.2 Result

A function.

B.4.10 `slot-description-slot-writer` *function*

B.4.10.1 Arguments

slot-description: A slot description.

B.4.10.2 Result
A function.

B.5 Defining Metaclasses

B.5.1 `defmetaclass` *defining form*

B.5.1.1 Syntax
`(defmetaclass` *class-name superclass* `(`*slot-description**`)` *class-option**`)`

B.5.1.2 Arguments

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

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

(*slot-description**): A list of slot specifications (see below).

class-option*: A sequence of symbols (see below).

B.5.1.3 Remarks
This defining form defines a new metaclass. The resulting metaclass will be bound to *class-name*. The second argument is a superclass. A valid *class-option* is `predicate`.

B.6 Initializing Classes

B.6.1 `initialize` *method*

B.6.1.1 Specialized Arguments

(*class* `<class>`): A class.

(*initlist* `<list>`): A list of initialization options as follows:

`name` *symbol*: Name of the class being initialized.

`direct-superclasses` *list*: List of direct superclasses.

`direct-slot-descriptions` *list*: List of slot specifications.

`direct-initargs` *list*: List of direct initargs.

B.6.1.2 Result
The initialized class.

B.6.1.3 Remarks
The initialization of a class takes place as follows:

a) Check compatibility of direct superclasses

b) Perform the logical inheritance computations of:

1) class precedence list

2) initargs

3) slot descriptions

c) Compute new slot accessors and ensure all (new and inherited) accessors to work correctly on instances of the new class.

d) Make the results accessible by class readers.

The basic call structure is laid out in Figure B.1

B.7 Initializing Slot Descriptions

B.7.1 `initialize` *method*

B.7.1.1 Specialized Arguments

(*slot-description* `<slot-description>`): A slot description.

(*initlist* `<list>`): A list of initialization options as follows:

`name` *symbol*: The name of the slot.

`initfunction` *function*: A function.

B.7.1.2 Result
The initialized slot description.

B.8 Inheritance Protocol

B.8.1 `compatible-superclasses-p` *generic function*

B.8.1.1 Generic Arguments

(*class* `<class>`): Class being defined.

(*direct-superclasses* `<list>`): List of potential direct superclasses.

B.8.1.2 Result
True or false.

B.8.1.3 Remarks
Checks compatibility between class and the list of direct superclasses.

B.8.2 `compatible-superclasses-p` *method*

B.8.2.1 Specialized Arguments

(*class* `<class>`): Class being defined.

(*direct-superclasses* `<list>`): List of potential direct superclasses.

Figure B.1 — Initialization Call Structure

```
COMPATIBLE-SUPERCLASSES-P cl direct-superclasses -> boolean
  COMPATIBLE-SUPERCLASS-P cl superclass -> boolean
COMPUTE-CLASS-PRECEDENCE-LIST cl direct-superclasses -> list(class)
COMPUTE-INHERITED-INITARGS cl direct-superclasses -> list(list(initarg))
COMPUTE-INITARGS cl direct-initargs <inherited-initargs> -> list(initarg)
COMPUTE-INHERITED-SLOT-DESCRIPTIONS cl direct-superclasses -> list(list(slotd))
COMPUTE-SLOT-DESCRIPTIONS cl direct-slotds <inherited-slotds> -> list(slotd)
  either
  COMPUTE-DEFINED-SLOT-DESCRIPTION cl slotd-init-list -> slotd
    COMPUTE-DEFINED-SLOT-DESCRIPTION-CLASS cl slotd-init-list -> slotd-class
  or
  COMPUTE-SPECIALIZED-SLOT-DESCRIPTION cl inherited-slotds slotd-init-list -> slotd
    COMPUTE-SPECIALIZED-SLOT-DESCRIPTION-CLASS cl inherited-slotds slotd-init-list -> slotd-class
COMPUTE-AND-ENSURE-SLOT-ACCESSORS cl <effective-slotds> <inherited-slotds> -> list(slotd)
  COMPUTE-SLOT-READER cl slotd <effective-slotds> -> function
  COMPUTE-SLOT-WRITER cl slotd <effective-slotds> -> function
  ENSURE-SLOT-READER cl slotd <effective-slotds> reader -> function
    COMPUTE-PRIMITIVE-READER-USING-SLOT-DESCRIPTION slotd cl <effective-slotds> -> function
      COMPUTE-PRIMITIVE-READER-USING-CLASS cl slotd <effective-slotds> -> function
  ENSURE-SLOT-WRITER cl slotd writer -> function
    COMPUTE-PRIMITIVE-WRITER-USING-SLOT-DESCRIPTION slotd cl <effective-slotds> -> function
      COMPUTE-PRIMITIVE-WRITER-USING-CLASS cl slotd <effective-slotds> -> function
```

B.8.2.2 Result

True or false.

B.8.2.3 Remarks

The default method calls `compatible-superclass-p` on *class* and the first element of the *direct-superclasses* (single inheritance assumption).

B.8.3 `compatible-superclass-p` *generic function*

B.8.3.1 Generic Arguments

(*subclass* `<class>`): Class being defined.

(*superclass* `<class>`): Potential direct superclass.

B.8.3.2 Result

True or false.

B.8.3.3 Remarks

Checks compatibility between a subclass being defined (first argument) and a potential superclass (second argument).

B.8.4 `compatible-superclass-p` *method*

B.8.4.1 Specialized Arguments

(*subclass* `<class>`): Class being defined.

(*superclass* `<class>`): Potential direct superclass.

B.8.4.2 Result

True or false.

B.8.4.3 Remarks

This method returns true, if the class of the first argument is a subclass of the class of the second argument, false otherwise.

B.8.5 `compatible-superclass-p` *method*

B.8.5.1 Specialized Arguments

(*subclass* `<class>`): Class being defined.

(*superclass* `<abstract-class>`): Potential direct superclass.

B.8.5.2 Result

True.

B.8.5.3 Remarks

This method always returns true.

B.8.6 `compatible-superclass-p` *method*

B.8.6.1 Specialized Arguments

(*subclass* `<abstract-class>`): Class being defined.

(*superclass* `<class>`): Potential direct superclass.

B.8.6.2 Result

False.

B.8.6.3 Remarks

This method always returns false.

B.8.7 `compute-class-precedence-list` *generic function*

B.8.7.1 Generic Arguments

(*class* `<class>`): Class being defined.

(*direct-superclasses* `<list>`): List of direct superclasses.

B.8.7.2 Result

List of classes.

B.8.7.3 Remarks

Computes and returns a list of classes which represents the linearized inheritance hierarchy of *class* and the given list of direct superclasses, beginning with *class* and ending with `<object>`.

B.8.8 `compute-class-precedence-list` *method*

B.8.8.1 Specialized Arguments

(*class* `<class>`): Class being defined.

(*direct-superclasses* `<list>`): List of direct superclasses.

B.8.8.2 Result

List of classes.

B.8.8.3 Remarks

This method can be considered to return a cons of its first argument and the class precedence list of the first element of *list* (single inheritance assumption).

B.8.9 `compute-slot-descriptions` *generic function*

B.8.9.1 Generic Arguments

(*class* `<class>`): Class being defined.

(*direct-slot-descriptions* `<list>`): A list of direct slot descriptions.

(*inherited-slot-descriptions* `<list>`): A list of lists of inherited slot descriptions.

B.8.9.2 Result

List of effective slot descriptions.

B.8.9.3 Remarks

Computes and returns the list of effective slot descriptions of class.

B.8.9.4

See also: `compute-inherited-slot-descriptions`.

B.8.10 `compute-slot-descriptions` *method*

B.8.10.1 Specialized Arguments

(*class* `<class>`): Class being defined.

(*slot-specs* `<list>`): List of (direct) slot specifications.

(*inherited-slot-description-lists* `<list>`): A list of lists (in fact one list) of inherited slot descriptions.

B.8.10.2 Result

List of effective slot descriptions.

B.8.10.3 Remarks

The default method computes two sublists:

a) Specialized slot descriptions by calling `compute-specialized-slot-description` on *class*, each *inherited-slot-description* wrapped by a list and the corresponding (same name) *slot-specification* if there exits one or `()` otherwise.

b) New slot descriptions by calling `compute-defined-slot-description` on *class* and each *slot-specification* which has no corresponding (same name) *inherited-slot-description*.

The method returns the concatenation of these two lists as its result. The order of elements in the list is significant. All specialized slot descriptions have the same position as in the effective slot descriptions list of the direct superclass (due to the single inheritance). The slot accessors (computed later) may rely on this assumption minimizing the number of methods to one for all subclasses and minimizing the access time to an indexed reference.

B.8.10.4

See also: `compute-specialized-slot-description`, `compute-defined-slot-description`, `compute-and-ensure-slot-accessors`.

B.8.11 `compute-initargs` *generic function*

B.8.11.1 Generic Arguments

(*class* `<class>`): Class being defined.

(*initargs* `<list>`): List of direct initargs.

(*inherited-initarg-lists* `<list>`): A list of lists of inherited initargs.

B.8.11.2 Result

List of symbols.

B.8.11.3 Remarks

Computes and returns all legal initargs for *class*.

B.8.11.4 See also: `compute-inherited-initargs`.

B.8.12 `compute-initargs` *method*

B.8.12.1 Specialized Arguments

(*class* `<class>`): Class being defined.

(*initargs* `<list>`): List of direct initargs.

(*inherited-initarg-lists* `<list>`): A list of lists of inherited initargs.

B.8.12.2 Result

List of symbols.

B.8.12.3 Remarks

This method appends the second argument with the first element of the third argument (single inheritance assumption), removes duplicates and returns the result.

B.8.13 `compute-inherited-slot-descriptions` *generic function*

B.8.13.1 Generic Arguments

(*class* `<class>`): Class being defined.

(*direct-superclasses* `<list>`): List of direct superclasses.

B.8.13.2 Result

List of lists of inherited slot descriptions.

B.8.13.3 Remarks

Computes and returns a list of effective slot description lists.

B.8.13.4 See also: `compute-slot-descriptions`.

B.8.14 `compute-inherited-slot-descriptions` *method*

B.8.14.1 Specialized Arguments

(*class* `<class>`): Class being defined.

(*direct-superclasses* `<list>`): List of direct superclasses.

B.8.14.2 Result

List of lists of inherited slot descriptions.

B.8.14.3 Remarks

The result of the default method is a list of one element: a list of effective slot descriptions of the first element of the second argument (single inheritance assumption). Its result is used by `compute-slot-descriptions` as an argument value.

B.8.15 `compute-inherited-initargs` *generic function*

B.8.15.1 Generic Arguments

(*class* `<class>`): Class being defined.

(*direct-superclasses* `<list>`): List of direct superclasses.

B.8.15.2 Result

List of lists of symbols.

B.8.15.3 Remarks

Computes and returns a list of initarg-lists. Its result is used by `compute-initargs` as an argument value.

B.8.15.4 See also: `compute-initargs`.

B.8.16 `compute-inherited-initargs` *method*

B.8.16.1 Specialized Arguments

(*class* `<class>`): Class being defined.

(*direct-superclasses* `<list>`): List of direct superclasses.

B.8.16.2 Result

List of lists of symbols.

B.8.16.3 Remarks

The result of the default method contains one list of legal initargs of the first element of the second argument (single inheritance assumption).

B.8.17 `compute-defined-slot-description` *generic function*

B.8.17.1 Generic Arguments

(*class* `<class>`): Class being defined.

(*slot-spec* `<list>`): Canonicalized slot specification.

B.8.17.2 Result

Slot description.

B.8.17.3 Remarks

Computes and returns a new effective slot description. It is called by `compute-slot-descriptions` on each slot specification which has no corresponding inherited slot descriptions.

B.8.17.4
See also: `compute-defined-slot-description-class`.

B.8.18 compute-defined-slot-description *method*

B.8.18.1 Specialized Arguments

(*class* <class>): Class being defined.

(*slot-spec* <list>): Canonicalized slot specification.

B.8.18.2 Result

Slot description.

B.8.18.3 Remarks

Computes and returns a new effective slot description. Its class is determined by calling compute-defined-slot-description-class.

B.8.18.4

See also: compute-defined-slot-description-class.

B.8.19 compute-defined-slot-description-class *generic function*

B.8.19.1 Generic Arguments

(*class* <class>): Class being defined.

(*slot-spec* <list>): Canonicalized slot specification.

B.8.19.2 Result

Slot description class.

B.8.19.3 Remarks

Determines and returns the slot description class corresponding to *class* and *list* .

B.8.19.4 See also: compute-defined-slot-description.

B.8.20 compute-defined-slot-description-class *method*

B.8.20.1 Specialized Arguments

(*class* <class>): Class being defined.

(*slot-spec* <list>): Canonicalized slot specification.

B.8.20.2 Result

The class <slot-description>.

B.8.20.3 Remarks

This method just returns the class <slot-description>.

B.8.21 compute-specialized-slot-description *generic function*

B.8.21.1 Generic Arguments

(*class* <class>): Class being defined.

(*inherited-slot-descriptions* <list>): List of inherited slot descriptions.

(*slot-spec* <list>): Canonicalized slot specification or ().

B.8.21.2 Result

Slot description.

B.8.21.3 Remarks

Computes and returns a new effective slot description. It is called by compute-slot-descriptions on the class, each list of inherited slots with the same name and with the specialising slot specification list or () if no one is specified with the same name.

B.8.21.4

See also: compute-specialized-slot-description-class.

B.8.22 compute-specialized-slot-description *method*

B.8.22.1 Specialized Arguments

(*class* <class>): Class being defined.

(*inherited-slot-descriptions* <list>): List of inherited slot descriptions.

(*slot-spec* <list>): Canonicalized slot specification or ().

B.8.22.2 Result

Slot description.

B.8.22.3 Remarks

Computes and returns a new effective slot description. Its class is determined by calling compute-specialized-slot-description-class.

B.8.22.4

See also: compute-specialized-slot-description-class.

B.8.23 compute-specialized-slot-description-class *generic function*

B.8.23.1 Generic Arguments

(*class* <class>): Class being defined.

(*inherited-slot-descriptions* `<list>`): List of inherited slot descriptions.

(*slot-spec* `<list>`): Canonicalized slot specification or `()`.

B.8.23.2 Result

Slot description class.

B.8.23.3 Remarks

Determines and returns the slot description class corresponding to (i) the class being defined, (ii) the inherited slot descriptions being specialized (iii) the specializing information in *slot-spec*.

B.8.23.4

See also: `compute-specialized-slot-description`.

B.8.24 `compute-specialized-slot-description-class` *method*

B.8.24.1 Specialized Arguments

(*class* `<class>`): Class being defined.

(*inherited-slot-descriptions* `<list>`): List of inherited slot descriptions.

(*slot-spec* `<list>`): Canonicalized slot specification or `()`.

B.8.24.2 Result

The class `<slot-description>`.

B.8.24.3 Remarks

This method just returns the class `<slot-description>`.

B.9 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 slot-descriptions and classes is performed once at class definition time rather than again and again at slot access time.

Each slot-description has exactly one reader and one writer as anonymous objects. If a reader/writer slot-option is specified in a class definition, the anonymous reader/writer of that slot-description is bound to the specified identifier. Thus, if a reader/writer option is specified more than once, the same object is bound to all the identifiers. If the accessor slot-option is specified the anonymous writer will be installed as the setter of the reader. Specialized slot-descriptions refer to the same objects as those in the superclasses (single inheritance makes that possible). Since the readers/writers are generic, it is possible for a subclass (at the meta-level) to add new methods for inherited slot-descriptions in order to make the readers/writers applicable on instances of the subclass. A new method might be necessary if the subclasses have a different instance allocation or if the slot positions cannot be kept the same as in the superclass (in multiple inheritance extensions). This can be done during the initialization computations.

B.9.1 `compute-and-ensure-slot-accessors` *generic function*

B.9.1.1 Generic Arguments

(*class* `<class>`): Class being defined.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

(*inherited-slot-descriptions* `<list>`): List of inherited slot descriptions.

B.9.1.2 Result

List of effective slot descriptions.

B.9.1.3 Remarks

Computes new accessors or ensures that inherited accessors work correctly for each effective slot description.

B.9.2 `compute-and-ensure-slot-accessors` *method*

B.9.2.1 Specialized Arguments

(*class* `<class>`): Class being defined.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

(*inherited-slot-descriptions* `<list>`): List of inherited slot descriptions.

B.9.2.2 Result

List of effective slot descriptions.

B.9.2.3 Remarks

The default method checks if it is a new slot description (not an inherited one). If yes,

a) calls `compute-slot-reader` to compute a new slot reader and stores the result;

b) calls `compute-slot-writer` to compute a new slot writer and stores the result;

If not, assumes that the inherited values remain valid. Last, it ensures the reader and writer to work correctly calling `ensure-slot-reader` and `ensure-slot-writer`.

B.9.3 `compute-slot-reader` *generic function*

B.9.3.1 Generic Arguments

(*class* `<class>`): Class.

(*slot-description* `<slot-description>`): Slot description.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

B.9.3.2 Result

Function.

B.9.3.3 Remarks

Computes and returns a new slot reader applicable to instances of *class* returning the slot value corresponding to *slot-description*. The third argument can be used in order to compute the logical slot position.

B.9.4 `compute-slot-reader` *method*

B.9.4.1 Specialized Arguments

(*class* `<class>`): Class.

(*slot-description* `<slot-description>`): Slot description.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

B.9.4.2 Result

Generic function.

B.9.4.3 Remarks

The default method returns a new generic function of one argument without any methods. Its domain is *class*.

B.9.5 `compute-slot-writer` *generic function*

B.9.5.1 Generic Arguments

(*class* `<class>`): Class.

(*slot-description* `<slot-description>`): Slot description.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

B.9.5.2 Result

Function.

B.9.5.3 Remarks

Computes and returns a new slot writer applicable to instances of *class* and any value to be stored as the new slot value corresponding to *slot-description*. The third argument can be used in order to compute the logical slot position.

B.9.6 `compute-slot-writer` *method*

B.9.6.1 Specialized Arguments

(*class* `<class>`): Class.

(*slot-description* `<slot-description>`): Slot description.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

B.9.6.2 Result

Generic function.

B.9.6.3 Remarks

The default method returns a new generic function of two arguments without any methods. Its domain is the product of *class* and `<object>`.

B.9.7 `ensure-slot-reader` *generic function*

B.9.7.1 Generic Arguments

(*class* `<class>`): Class.

(*slot-description* `<slot-description>`): Slot description.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

(*reader* `<function>`): The slot reader.

B.9.7.2 Result

Function.

B.9.7.3 Remarks

Ensures *function* works correctly on instances of class.

B.9.8 `ensure-slot-reader` *method*

B.9.8.1 Specialized Arguments

(*class* `<class>`): Class.

(*slot-description* `<slot-description>`): Slot description.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

(*reader* `<generic-function>`): The slot reader.

B.9.8.2 Result

Generic function.

B.9.8.3 Remarks

The default method checks if there is a method in the *generic-function*. If not, it creates and adds a new one. The new method has

domain: — (*class*)

```
(method-function-lambda ((object class))
   (primitive-reader object))
```

`compute-primitive-reader-using-slot-description` is called by `ensure-slot-reader` method to compute the primitive reader used in the function of the new created reader method.

B.9.9 `ensure-slot-writer` *generic function*

B.9.9.1 Generic Arguments

(*class* `<class>`): Class.

(*slot-description* `<slot-description>`): Slot description.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

(*writer* `<function>`): The slot writer.

B.9.9.2 Result

Function.

B.9.9.3 Remarks

Ensures *function* to work correctly on instances of class.

B.9.10 `ensure-slot-writer` *method*

B.9.10.1 Specialized Arguments

(*class* `<class>`): Class.

(*slot-description* `<slot-description>`): Slot description.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

(*writer* `<generic-function>`): The slot writer.

B.9.10.2 Result

Generic function.

B.9.10.3 Remarks

The default method checks if there is a method in the *generic-function*. If not, creates and adds a new one. The new method has

domain: — (*class* `<object>`)

```
(method-function-lambda ((obj class)
                         (new-value <object>))
   (primitive-writer obj new-value))
```

`compute-primitive-writer-using-slot-description` is called by `ensure-slot-writer` method to compute the primitive writer used in the function of the new created writer method.

B.9.11 `compute-primitive-reader-using-slot-description` *generic function*

B.9.11.1 Generic Arguments

(*slot-description* `<slot-description>`): Slot description.

(*class* `<class>`): Class.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

B.9.11.2 Result

Function.

B.9.11.3 Remarks

Computes and returns a function which returns a slot value when applied on an instance of *class*.

B.9.12 `compute-primitive-reader-using-slot-description` *method*

B.9.12.1 Specialized Arguments

(*slot-description* `<slot-description>`): Slot description.

(*class* `<class>`): Class.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

B.9.12.2 Result

Function.

B.9.12.3 Remarks

Calls `compute-primitive-reader-using-class`. This is the default method.

B.9.13 `compute-primitive-reader-using-class` *generic function*

B.9.13.1 Generic Arguments

(*class* `<class>`): Class.

(*slot-description* `<slot-description>`): Slot description.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

B.9.13.2 Result

Function.

B.9.13.3 Remarks

Computes and returns a function which returns the slot value when applied on an instance of *class*.

B.9.14 `compute-primitive-reader-using-class` *method*

B.9.14.1 Specialized Arguments

(*class* `<class>`): Class.

(*slot-description* `<slot-description>`): Slot description.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

B.9.14.2 Result

Function.

B.9.14.3 Remarks

The default method returns a function of one argument.

B.9.15 `compute-primitive-writer-using-slot-description` *generic function*

B.9.15.1 Generic Arguments

(*slot-description* `<slot-description>`): Slot description.

(*class* `<class>`): Class.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

B.9.15.2 Result

Function.

B.9.15.3 Remarks

Computes and returns a function which stores a new slot value when applied on an instance of *class* and a new value.

B.9.16 `compute-primitive-writer-using-slot-description` *method*

B.9.16.1 Specialized Arguments

(*slot-description* `<slot-description>`): Slot description.

(*class* `<class>`): Class.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

B.9.16.2 Result

Function.

B.9.16.3 Remarks

Calls `compute-primitive-writer-using-class`. This is the default method.

B.9.17 `compute-primitive-writer-using-class` *generic function*

B.9.17.1 Generic Arguments

(*class* `<class>`): Class.

(*slot-description* `<slot-description>`): Slot description.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

B.9.17.2 Result

Function.

B.9.17.3 Remarks

Computes and returns a function which stores the new slot value when applied on an instance of *class* and new value.

B.9.18 `compute-primitive-reader-using-class` *method*

B.9.18.1 Specialized Arguments

(*class* `<class>`): Class.

(*slot-description* `<slot-description>`): Slot description.

(*slot-descriptions* `<list>`): List of effective slot descriptions.

B.9.18.2 Result

Function.

B.9.18.3 Remarks

The default method returns a function of two arguments.

B.10 Predicates and Constructors

B.10.1 `compute-predicate` *generic function*

B.10.1.1 Generic Arguments

(*class* `<class>`): Class.

B.10.1.2 Result

Function.

B.10.1.3 Remarks

Computes and returns a predicate function of one argument.

B.10.2 `compute-predicate` *method*

B.10.2.1 Specialized Arguments

(*class* `<class>`): Class.

B.10.2.2 Result

Function.

B.10.2.3 Remarks

Computes and returns a predicate function of one argument, which returns true when applied on direct or indirect instances of *class* and false otherwise.

B.10.3 `compute-constructor` *generic function*

B.10.3.1 Generic Arguments

(*class* `<class>`): Class.

(*parameters* `<list>`): Argument list of the function being created.

B.10.3.2 Result

Function.

B.10.3.3 Remarks

Computes and returns a constructor function.

B.10.4 `compute-constructor` *method*

B.10.4.1 Specialized Arguments

(*class* `<class>`): Class.

(*parameters* `<list>`): Argument list of the function being created.

B.10.4.2 Result

Function.

B.10.4.3 Remarks

Computes and returns a constructor function, which returns a new instance of *class* when applied.

B.11 Instance Allocation

B.11.1 `allocate` *generic function*

B.11.1.1 Generic Arguments

(*class* `<class>`): A class.

(*initlist* `<list>`): A list of initialization arguments.

B.11.1.2 Result

An instance of the first argument.

B.11.1.3 Remarks

Creates an instance of the first argument. Users may define new methods for new metaclasses.

B.11.2 `allocate` *method*

B.11.2.1 Specialized Arguments

(*class* `<class>`): A class.

(*initlist* `<list>`): A list of initialization arguments.

B.11.2.2 Result

An instance of the first argument.

B.11.2.3 Remarks

The default method creates a new uninitialized instance of the first argument. The initlist is not used by this `allocate` method.

B.12 Low Level Allocation Primitives

This module provides primitives which are necessary to implement new allocation methods portably. However, they should be defined in such a way that objects cannot be destroyed unintentionally. In consequence it is an error to use `primitive-class-of`, `primitive-ref` and their setters on objects not created by `primitive-allocate`.

B.12.1 `primitive-allocate` *function*

B.12.1.1 Arguments

class: A class.

size: An integer.

B.12.1.2 Result

An instance of the first argument.

B.12.1.3 Remarks

This function returns a new instance of the first argument which has a vector like structure of length *size*. The components of the new instance can be accessed by means of `primitive-ref` and `(setter primitive-ref)`. It is intended to be used in new `allocate` methods defined for new metaclasses.

B.12.2 `primitive-class-of` *function*

B.12.2.1 Arguments

object: An object created by `primitive-allocate`.

B.12.2.2 Result

A class.

B.12.2.3 Remarks

This function returns the class of an object. It is similar to `class-of`, which has a defined behaviour on any object. It is an error to use `primitive-class-of` on objects which were not created by `primitive-allocate`.

B.12.3 `(setter primitive-class-of)` *setter*

B.12.3.1 Arguments

object: An object created by `primitive-allocate`.

class: A class.

B.12.3.2 Result

The *class*.

B.12.3.3 Remarks

This function supports portable implementations of

a) dynamic classification like `change-class` in CLOS.

b) automatic instance updating of redefined classes.

B.12.4 `primitive-ref` *function*

B.12.4.1 Arguments

object: An object created by `primitive-allocate`.

index: The index of a component.

B.12.4.2 Result

An object.

B.12.4.3 Remarks

Returns the value of the objects component corresponding to the supplied index. It is an error if the index is out of range. This function is intended to be used when defining new kinds of accessors for new metaclasses.

B.12.5 `(setter primitive-ref)` *function*

B.12.5.1 Arguments

object: An object created by `primitive-allocate`.

index: The index of a component.

value: The new value, which can be any object.

B.12.5.2 Result

The new value.

B.12.5.3 Remarks

Stores and returns the new value as the objects component corresponding to the supplied index. It is an error if the index is out of range. This function is intended to be used when defining new kinds of accessors for new metaclasses.

Table B.4 — Metaobject Classes

Table B.5 — Generic Function Metaobject Classes

```
<object> [<abstract-class>]
   <class> [<metaclass>]
      <metaclass> [<metaclass>]
      <abstract-class> [<metaclass>]
      <function-class> [<metaclass>]
   <generic-function> [<function-class>]
   <method> [<class>]
```

B.13 Reflection on Generic Functions and Methods

The generic dispatch is class based, i.e. methods are class specific. Instance specific methods can be provided (even portably) in an extension module by defining a new generic function class. We think it is confusing to have generic functions with both, class specific and instance specific methods. Combining different discrimination strategies in a single generic function is rather a shortcoming than an advantage. The argument precedence order is always left-to-right.

The minimal information associated with a generic function metaobject is:

a) The name, which has no semantic effect.

b) The domain, restricting the domain of each added method to a subdomain.

c) The range, restricting the range of each added method to a subrange.

d) The method class, restricting each added method to be an instance (direct or indirect) of that class.

e) The list of all added methods.

f) The method look-up function to collect and sort the applicable methods.

g) The discriminating function to perform the generic dispatch.

The minimal information associated with a method metaobject is:

a) The domain, which is a list of classes.

b) The range, which is a class.

c) The function comprising the code of the method.

d) The generic function, if the method has been added to one (at the most).

The metaobject classes for generic functions defined at level-1 are shown in Table B.5.

B.14 Introspection

B.14.1 `generic-function-name` *function*

B.14.1.1 Arguments

generic-function: A generic function.

B.14.1.2 Result

A symbol.

B.14.1.3 Remarks

The name has only debugging purposes.

B.14.1.4 See also: `class-name`.

B.14.2 `generic-function-domain` *function*

B.14.2.1 Arguments

generic-function: A generic function.

B.14.2.2 Result

List of classes.

B.14.2.3 Remarks

Returns the domain of a generic function. All methods attached to a generic function have subdomains of it.

B.14.3 `generic-function-range` *function*

B.14.3.1 Arguments

generic-function: A generic function.

B.14.3.2 Result

A class.

B.14.3.3 Remarks

Returns the range of a generic function. Each method attached to a generic function must have a subrange of it.

B.14.4 `generic-function-method-class` *function*

B.14.4.1 Arguments

generic-function: A generic function.

B.14.4.2 Result

A class.

B.14.4.3 Remarks

Returns the method class of a generic function. Each method attached to a generic function must be an instance of that class. When a method is defined using `defmethod` it will be an instance of that class by default.

B.14.5 `generic-function-methods` *function*

B.14.5.1 Arguments

generic-function: A generic function.

B.14.5.2 Result

A list of methods.

B.14.5.3 Remarks

Returns a list of methods attached to the generic function.

B.14.6 `generic-function-method-lookup-function` *function*

B.14.6.1 Arguments

generic-function: A generic function.

B.14.6.2 Result

A function.

B.14.6.3 Remarks

Returns a function applicable on same arguments as the generic function which returns a sorted list of applicable methods when applied.

B.14.7 `generic-function-discriminating-function` *function*

B.14.7.1 Arguments

generic-function: A generic function.

B.14.7.2 Result

A function.

B.14.7.3 Remarks

Returns a function applicable on same arguments as the generic function which is called everytime when the generic function is called. It performs the generic dispatch and calls the applicable methods.

B.14.8 `method-domain` *function*

B.14.8.1 Arguments

method: A method.

B.14.8.2 Result

List of classes.

B.14.8.3 Remarks

Returns the domain of a method.

B.14.9 `method-range` *function*

B.14.9.1 Arguments

method: A method.

B.14.9.2 Result

A class.

B.14.9.3 Remarks

Returns the range of a method.

B.14.10 `method-function` *function*

B.14.10.1 Arguments

method: A method.

B.14.10.2 Result

A function.

B.14.10.3 Remarks

Returns the function which is called when a method is called. The method itself can not be applied or called as a function.

B.14.11 `method-generic-function` *function*

B.14.11.1 Arguments

method, ¡method¿: A method.

B.14.11.2 Result

A generic function or false.

B.14.11.3 Remarks

Returns the generic function to which the method is attached, false otherwise.

B.15 Special forms (or macros)

The followin macros hide the implementation details of calling methods, arranging the next methods so that they are accessible by `call-next-method`, etc.

B.15.1 `method-function-lambda` *macro*

B.15.2 `call-method` *macro*

B.15.3 `apply-method` *macro*

B.16 Initializing Generic Functions and Methods

B.16.1 `initialize` *method*

B.16.1.1 Specialized Arguments

(*generic-function* `<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.

`range` *class*: The class of the result.

`method-class` *class*: Class of attached method.

`methods` *list*: List of methods to be attached.

B.16.1.2 Result

The initialized generic function.

B.16.1.3 Remarks

Initializes and returns the *generic-function*. It calls `compute-method-lookup-function`, `compute-discriminating-function` and stores their results as well as the direct information passed as arguments.

The basic call structure is:

```
COMPUTE-METHOD-LOOKUP-FUNCTION
 generic-function domain -> function
COMPUTE-DISCRIMINATING-FUNCTION
 generic-function domain lookup-fn methods -> function
```

B.16.2 `initialize` *method*

B.16.2.1 Specialized Arguments

(*method* `<method>`): A method.

(*initlist* `<list>`): A list of initialization options as follows:

`domain` *list*: List of argument classes.

`range` *class*: The class of the result.

`function` *function*: A function.

`generic-function` *generic-function*: A generic function.

B.16.2.2 Result

The initialized method.

B.16.2.3 Remarks

Initializes and returns the *method*. There is nothing special to specify for this method.

B.17 Method Lookup and Generic Dispatch

B.17.1 `compute-method-lookup-function` *generic function*

B.17.1.1 Generic Arguments

(*generic-function* `<generic-function>`): A generic function.

(*domain* `<list>`): A list of classes which span the domain.

B.17.1.2 Result

A function.

B.17.1.3 Remarks

Computes and returns a function which is called at least ones for a particular domain in order to select and sort the applicable methods by the system provided dispatch mechanism. Users may define new methods for it in order to implement different method lookup strategies. Althought, there is just one lookup strategie provided by the system, each generic function may have its own (more efficient) lookup function.

B.17.2 `compute-method-lookup-function` *method*

B.17.2.1 Specialized Arguments

(*generic-function* `<generic-function>`): A generic function.

(*domain* `<list>`): A list of classes which span the domain.

B.17.2.2 Result

A function.

B.17.2.3 Remarks

Computes and returns a function which is called at least ones for a particular domain in order to select and sort the applicable methods by the system provided dispatch mechanism. It is not specified, whether or not each generic function gets its own lookup function.

B.17.3 `compute-discriminating-function` *generic function*

B.17.3.1 Generic Arguments

(*generic-function* `<generic-function>`): A generic function.

(*domain* `<list>`): A list of classes which span the domain.

(*lookup-function* `<function>`): The method lookup function.

(*methods* `<list>`): A list of methods attached to the *generic-function*.

B.17.3.2 Result

A function.

B.17.3.3 Remarks

Computes and returns a function which is called whenever the generic function is called. It is the controller of the generic dispatch. Users may define new methods for new generic function classes in order to implement other dispatch strategies, e.g. eql-discrimination as in CLOS.

B.17.4 `compute-discriminating-function` *method*

B.17.4.1 Specialized Arguments

(*generic-function* `<generic-function>`): A generic function.

(*domain* `<list>`): A list of classes which span the domain.

(*lookup-function* `<function>`): The method lookup function.

(*methods* `<list>`): A list of methods attached to the *generic-function*.

B.17.4.2 Result

A function.

B.17.4.3 Remarks

Computes and returns a function which is called whenever the generic function is called. It is unspecified which optimizations are provided by this function.

B.18 Extending Generic Functions by New Methods

These operations provide means to add and remove methods dynamically to/from generic functions. They are intended to support portable programming environment implementations. In contrast to CLOS, `add-method` does not remove a method with the same domain as the method being added. Instead, `remove-method` must be used explicitly before adding the new one.

B.18.1 `add-method` *generic function*

B.18.1.1 Generic Arguments

(*generic-function* `<generic-function>`): A generic function.

(*method* `<method>`): A method to be attached.

B.18.1.2 Result

The generic function.

B.18.1.3 Remarks

Adds a method to a generic function. User may define new methods to it for new generic function and method classes.

B.18.2 `add-method` *method*

B.18.2.1 Specialized Arguments

(*generic-function* `<generic-function>`): A generic function.

(*method* `<method>`): A method to be attached.

B.18.2.2 Result

The generic function.

B.18.2.3 Remarks

Checks if the domain, the range as well as the class of the method are more special than those of the generic function. If not, signals an error. Checks if there is a method with the same domain attached to the generic function already. If yes, signals an error. If no error occures, adds the method to the generic function. Depending on particular optimizations of the generic dispatch, adding a method may cause some updating computations.

B.18.3 `remove-method` *generic function*

B.18.3.1 Generic Arguments

(*generic-function* `<generic-function>`): A generic function.

(*method* `<method>`): A method to be removed.

B.18.3.2 Result

The generic function.

B.18.3.3 Remarks

Removes a method from a generic function. User may define new methods for new generic function and method classes.

B.18.4 `remove-method` *method*

B.18.4.1 Specialized Arguments

(*generic-function* `<generic-function>`): A generic function.

(*method* `<method>`): A method to be removed.

B.18.4.2 Result

The generic function.

B.18.4.3 Remarks

If the method is attached to the generic function it will be removed as well as the backpointer from the method to the generic function. Depending on particular optimizations of the generic dispatch, removing a method may cause some updating computations.

B.19 Dynamic Binding

B.19.1 `dynamic` *special form*

B.19.1.1 Syntax

(`dynamic` *identifier*)

B.19.1.2 Arguments

identifier: A symbol naming a dynamic binding.

B.19.1.3 Result

The value of closest dynamic binding of `symbol` named by identifier is returned. If no such binding exists, an error is signaled (condition: `unbound-dynamic-variable`).

B.19.2 `dynamic-setq` *special form*

B.19.2.1 Syntax

(`dynamic-setq` *identifier form*)

B.19.2.2 Arguments

identifier: A symbol naming a dynamic binding to be updated.

form: An expression whose value will be stored in the dynamic binding of *identifier*.

B.19.2.3 Result

The value of *form*.

B.19.2.4 Remarks

The *form* is evaluated and the result is stored in the closest dynamic binding of `symbol` named by *identifier*. An error is signaled (condition: `unbound-dynamic-variable`) if *symbol* is not dynamically apparent and has no dynamic global value.

B.19.3 `unbound-dynamic-variable` *execution-condition*

B.19.3.1 Init-options

`symbol` *symbol*: A symbol naming the (unbound) dynamic variable.

B.19.3.2 Remarks

Signalled by `dynamic` or `dynamic-setq` if the given dynamic variable has no visible dynamic binding.

B.19.4 `dynamic-let` *special form*

B.19.4.1 Syntax

(`dynamic-let` *binding** *form**)

B.19.4.2 Arguments

binding*: A list of binding specificiers.

body: A sequence of forms.

B.19.4.3 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.

B.19.4.4 Remarks

A binding specifier is either an identifier or a two element list of an identifier and an initializing form. All the initializing forms are evaluated from left to right in the current environment and the new bindings for the symbols named by the identifiers are created in the dynamic environment to hold the results. These bindings have dynamic scope and dynamic extent. Each form in *body* is evaluated in order in the environment extended by the above bindings. The result of evaluating the last form in *body* is returned as the result of `dynamic-let`.

B.19.5 `defvar` *defining form*

B.19.5.1 Syntax

`(defvar` *name expression*`)`

B.19.5.2 Arguments

identifier: A symbol naming a top dynamic binding containing the value of *form*.

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

B.19.5.3 Remarks

The value of *form* is stored as the top dynamic value of the symbol named by *identifier*. The binding created by `defvar` is mutable. An error is signaled (condition: `dynamic-multiply-defined`), on evaluating this form more than once for the same *identifier*.

B.19.6 `dynamic-multiply-defined` *execution-condition*

B.19.6.1 Init-options

`symbol` *symbol*: A symbol naming the dynamic variable which has already been defined.

B.19.6.2 Remarks

Signalled by `defvar` if the named dynamic variable already exists.

B.20 Conditional Extensions

B.20.1 `when` *macro*

B.20.1.1 Syntax

`(when` *antecedent form**`)`

B.20.1.2 Remarks

The `when` operator evaluates *antecedent* and if the result is not `()`, the *form*s are evaluated from left to right. It is equivalent to `if` with a null alternative. If the evaluation of *antecedent* is not `()`, the result of the `when` form is that of the evaluation of the last *form*, otherwise the result is `()`. The rewrite rule for `when` is:

`(when)`	≡	Is an error
`(when` *antecedent*`)`	≡	`()`
`(when` *form*$_1$ *form*$_2$ ...`)`	≡	`(if` *form*$_1$ `(progn` *form*$_2$ ...`) ())`

B.20.2 `unless` *macro*

B.20.2.1 Syntax

`(unless` *antecedent form**`)`

B.20.2.2 Remarks

The `unless` operator evaluates the first form and if the result is `()`, the remaining forms are evaluated from left to right. It is equivalent to `if` with a null consequence. If the evaluation of the first form is `()`, the result of the `unless` form is the result of the evaluation of the last form, otherwise the result is `()`. The rewrite rule for `unless` is:

`(unless)`	≡	Is an error
`(unless` *antecedent*`)`	≡	`()`
`(unless` *form*$_1$ *form*$_2$ ...`)`	≡	`(if` *form*$_1$ `() (progn` *form*$_2$ ...`))`

B.21 Exit Extensions

B.21.1 `block` *macro*

B.21.1.1 Syntax

`(block` *identifier form**`)`

B.21.1.2 Remarks

The block expression is used to establish a statically scoped binding of an escape function. The block *variable* is bound to the continuation of the block. The continuation can be invoked anywhere within the block by using `return-from`. The *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`. The rewrite rules for `block` are:

`(block)`	≡	Is an error
`(block` *identifier*`)`	≡	`()`
`(block` *identifier form**`)`	≡	`(let/cc` *identifier form**`)`

Exiting from a `block`, by whatever means, causes the restoration of the lexical environment and dynamic environment that existed before `block` entry. The above rewrite for `block`, does not prevent the `block` being exited from anywhere in its dynamic extent, since the `block`-exit function is a first-class item and can be passed as an argument like other values.

B.21.1.3 See also: `return-from`.

B.21.2 `return-from` *macro*

B.21.2.1 Syntax

`(return-from` *identifier* [*form*]`)`

B.21.2.2 Remarks

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

`(return-from)`	≡	Is an error
`(return-from` *identifier*`)`	≡	`(`*identifier*` ())`
`(return-from` *identifier form*`)`	≡	`(`*identifier form*`)`

B.21.2.3 See also: `block`.

B.21.3 `catch` *macro*

B.21.3.1 Syntax

`(catch` *tag form**`)`

B.21.3.2 Remarks

The `catch` operator is similar to `block`, except that the scope of the name (*tag*) of the exit function is dynamic. The catch *tag* must be a `symbol` because it is used as a dynamic variable to create a dynamically scoped binding of *tag* to the continuation of the `catch` form. The continuation can be invoked anywhere within the dynamic extent of the `catch` form by using `throw`. The *form*s are evaluated in sequence and the value of the last one is returned as the value of the `catch` form. The rewrite rules for `catch` are:

```
(catch)              ≡  Is an error
(catch tag)          ≡  (progn tag ())
(catch tag form*)    ≡  (let/cc tmp
                          (dynamic-let ((tag tmp))
                            form*))
```

Exiting from a `catch`, by whatever means, causes the restoration of the lexical environment and dynamic environment that existed before the `catch` was entered. The above rewrite for `catch`, causes the variable `tmp` to be shadowed. This is an artifact of the above presentation only and a conforming processor must not shadow any variables that could occur in the body of `catch` in this way.

B.21.3.3 See also: `throw`.

B.21.4 `throw` *macro*

B.21.4.1 Syntax

`(throw` *tag form*`)`

B.21.4.2 Remarks

In `throw`, the *tag* names the continuation of the `catch` from which to return. `throw` is the invocation of the continuation of the catch named *tag*. The *form* is evaluated and the value are returned as the value of the catch named by *variable*. The *tag* ia a symbol because it used to access the current dynamic binding of the symbol, which is where the continuation is bound. The rewrite rules for `throw` are:

`(throw)`	≡	Is an error
`(throw` *tag*`)`	≡	`((dynamic` *tag*`) ())`
`(throw` *tag form*`)`	≡	`((dynamic` *tag*`)` *form*`)`

B.21.4.3 See also: `catch`.

B.22 Summary of Level-1 Expressions and Definitions

The syntax of all level-1 expressions and definitions is given in Table B.6. Any productions undefined here appear elsewhere in the definition, specifically: the syntax of most expressions and definitions is completed in the section defining level-0.

Table B.6 — Expressions and Definitions (level-1)

module-expression	::=	*export-spec* | *level-1-expression* | *level-0-expression* | *definition* | `(progn` *expression*`)`
definition	::=	*level-1-definition* | *level-0-definition*$_{\{\texttt{defmodule}\}}$
level-1-definition	::=	*defclass* | *defcondition* | *defgeneric* | *defvar* | *level-0-definition*
defclass	::=	`(defclass` *class-name* `(`*superclass**`)` `(`*slot-description**`)` *class-option**`)`
class-name	::=	*identifier*
superclass	::=	{`<object>` *or one of its subclasses*}
slot-description	::=	*slot-name* | `(`*slot-name slot-option**`)`
slot-name	::=	*identifier*
slot-option	::=	*identifier expression* | *level-0-slot-option*
class-option	::=	`metaclass` *class-name* | *identifier expression* | *level-0-class-option*
defgeneric	::=	`(defgeneric` *gf-name gen-lambda-list level-1-init-option**`)`
level-1-init-option	::=	*level-0-init-option* | `class` *gf-class* | `method-class` *method-class* | `method (`*level-1-method-description*`)` | *gf-init-option*
gf-class	::=	*a subclass of* `<generic-function>`
method-class	::=	*a subclass of* `<method>`
level-1-method-description	::=	`(`*method-init-option** *spec-lambda-list form**`)`
gf-init-option	::=	*identifier expression*
method-init-option	::=	`class` *method-class* | *identifier expression*
defvar	::=	`(defvar` *name expression*`)`
level-1-expression	::=	*dynamic-let* | *dyn-ref* | *dyn-assign* | *level-0-expression*
dyn-ref	::=	`(dynamic` *identifier*`)`
dyn-assign	::=	`(dynamic-setq` *identifier form*`)`
dynamic-let	::=	`(dynamic-let` *binding** *form**`)`

References

[Alberga *et al*, 1986] 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).

[Bobrow *et al.*, 1988] 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.

[Chailloux *et al*, 1984] 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.

[Chailloux *et al*, 1987] 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.

[Clinger & Rees, 1986] Clinger W. & Rees J.A. (eds.), *The Revised^3 Report on Scheme*, SIGPLAN Notices, Vol. 21, No. 12, 1986.

[Cointe, 1987] Cointe P., *Mateclasses are First Class: the ObjVlisp model*, Proceedings of OOPSLA '87, published as SIGPLAN Notices, Vol 22, No 12 pp156-167.

[Fitch & Norman, 1977] Fitch J.P. & Norman A.C., *Implementing Lisp in a High-Level Language*, Software Practice and Experience, Vol 7, pp713-725.

[Friedman & Haynes, 1985] 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.

[Hudak, Wadler *et al.*, 1988] 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.

[Landin, 1966] Landin P.J., *The Next 700 Programming Languages*, Communications of the ACM, Vol 9, No 3., 1966, pp156-166.

[Lang & Pearlmutter, 1988] 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.

[MacQueen, 1984] 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.

[Milner *et al*, 1986] Milner R., *et al*, *Standard ML*, Laboratory for the Foundations of Computer Science, University of Edinburgh, Technical Report.

[Padget *et al*, 1986] 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.

[Padget, 1989] Padget J.A., *A Simple Light-weight Process Mechanism in Lisp*, in preparation.

[Pitman, 1988] Pitman K.M., *An Error System for Common Lisp*, ISO///WG16 paper N24.

[Rees *et al*, 1986] Rees J.A., *The T Manual*, YALEU Technical Report, 1986.

[Slade, 1987] Slade S., *The T Programming Language, a Dialect of Lisp*, Prentice-Hall 1987.

[Shalit, 1992] Shalit A., *Dylan, an object-oriented dynamic language*, Apple Computer Inc., 1992.

[Steele, 1984/90] Steele G.L. Jr., *Common Lisp the Language*, Digital Press, 1984, and *Common Lisp the Language (second edition)*, Digital Press, 1990.

[Stoyan *et al*, 1986] Stoyan H. *et al*, *Towards a Lisp Standard*, published in the Proceedings of the 1986 European Conference on Artificial Intelligence.

[Teitelman, 1978] Teitelman W., *The Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1978.

Function Index

Macro Index

Generic Function Index

Method Index

Condition Index

Constant Index

General Index