The EuLisp Definition
Version 0.7.5

Julian Padget   Greg Nuyens
(Editors)

Giuseppe Attardi
Javier Béjar
Russell Bradford
Peter Broadbery
Christopher Burdorf
Jérôme Chailloux
Thomas Christaller
Jeff Dalton
Klaus Däßler
Harley Davis
David De Roure
John Fitch
Richard Gabriel
Brigitte Glas
Nicolas Graube
Peter Kolb
Pascal Kuczynski
Antonio Moreno
Marco Nanni
Eugen Neidl
Pierre Parquier
Keith Playford
Willem van der Poel
Christian Queinnec
Matthias Schneider-Hufschmidt
Enric Sesa
Herbert Stoyan
François Surirey
Richard Tobin

(Contributors)

June 10, 1991

Summary

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.

The report is divided into six major sections. Section 1 gives a brief overview of EuLisp and recounts the history of its development. Section 2 describes the interpretation of expressions and the structure of programs. Section 3 lists all the standard procedures of the different language levels for manipulating data internally and externally. Section 4 describes the operations on class system. Section 5 describes the environment related operations. Section 6 describes the standard libraries of the language. Appendix A defines the formal syntax for EuLisp programs and Appendix B gives the formal semantics.

Table of Contents

List of Figures

List of Tables

1 Introduction

EuLisp defines a dialect of Lisp. 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 and the complementary abstraction facilities provided by the class and the module mechanism. EuLisp inherits from Scheme the properties of static-scoping, a single lexical environment for all variables and the uniform treatment of operator and operands.

1.1 Introduction to 0.7.5

The aim of this version of the definition is to complete as much as possible the level-0 and level-1 descriptions and to specify fully those library modules that are well-understood from experience and prior art. Of course, there are still many inconsistencies and inaccuracies. Although some sections are incomplete—and contain a note to that effect—references to those sections are written as if they were complete. This is intentional. For instance, the summary at the beginning of this document refers to appendix B containing the formal semantics, but since they are not complete they are not included. New descriptions of (i) the thread and mutual exclusion mechanism (ii) the macro expansion mechanism (iii) the inheritance protocol are in preparation and will be available as addenda to this version. Finally, it is becoming clear that the current document structure is no longer viable and the next major version will be radically reorganised to reflect better the structure of the language.

1.2 History

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.

1.3 Acknowledgements

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], LE-Lisp [Chailloux *et al*, 1984], Lisp/VM [Alberga *et al*, 1986], Scheme [Clinger & Rees, 1986], and T [Rees *et al*, 1986] [Slade, 1987]. 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 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 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, Universitet Delft, University of Edinburgh, Universität Erlangen, Université Paris VI (LITP).

1.4 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:

1. 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.
2. The minimal requirements of a configuration that is capable of supporting an implementation of an EuLisp processor.
3. The method of preparation of an EuLisp program for execution or the method of activation of this EuLisp program once prepared.
4. The method of reporting errors, warnings or exceptions.
5. 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.

1.5 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.

1. **ISO/IEC 646: 1983**, *Information processing—ISO 7-bit coded character set for information interchange.* Note: this standard is currently under revision and interested parties should reference the 1990 Draft International Standard version of ISO/IEC 646.

2. **ISO/IEC 10646: 1990**, *Information processing—ISO multi-byte coded character set for information interchange.* Note: this is not yet a standard and the given name may not be the final one. Only the code number is correct.

3. **ISO/IEC CD 10967: 1991**, *Information technology—Programming languages—Language compatible arithmetic.* Note: this is not yet a standard and the reference cited here is to a Committee Draft (Version 3.1).

4. *Language compatible complex arithmetic.* Note: no formal reference to this standard in development was available at the time of writing.

5. *Common language independent calling mechanism and datatypes.* Language-Independent Datatypes: working draft 5, JTC1/SC22/WG11/N233 (also X3T2/91-109). Note: no formal reference for the calling mechanism was available at the time of writing.

In addition the following technical reports have influenced the drafting of this definition.

1. **ISO/IEC TR 10034: 1990**, *Information technology—Guidelines for the preparation of conformity clauses in programming language standards.*

2. **ISO/IEC TR 10176: 1991**, *Information technology—Guidelines for the preparation of programming language standards.* Note: this is currently a draft technical report.

1.6 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.

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

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

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.

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

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

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.

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

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

1.6.1 Error Definitions

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

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

- 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.
- 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.

These errors must be dealt with by a *conforming processor* so as to satisfy the following requirements:

identification: Every error is related to a condition class which has a unique identification in this definition. Further identification of the error must be provided by the parameterization of the condition object. A processor is permitted to identify and deliver errors over and above those defined in this document. These errors are implementation-defined.

handling: Where an error must be signaled, the processor must indicate, in an *implementation-defined* way, that a particular error has arisen. A conforming EuLisp program can rely on the fact that the error is signaled. An implementation can signal an error continuably. This is an *implementation-defined extension*.

environmental error: An error which is detected by the configuration supporting the EuLisp processor.

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.*

1.7 Compliance

An EuLisp processor can conform at any of the three levels defined in section 2.2. 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 rules govern the conformance of a processor at a given level.

- 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*.
- 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 conforming EuLisp program can conform at any of the three levels defined in section 2.2. Thus 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 list of all definitions or values for the *implementation-defined* features of the language standard.
- 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.
- A list of all the features of the language implemented by this *processor* which are *extensions* to the standard language.
- 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.

1.8 Conventions

1.8.1 Layout and Typography

Both layout and fonts are used to help in the description of EuLisp. Here are some examples of entries describing special forms, functions and macros:

`(special-form-name` *special-form-arguments*`)` → *result-class*	**level-88 special form**
`(standard-function-name` *standard-function-arguments*`)` → *result-class*	**level-88 function**
`(macro-name` *macro-arguments*`)`	**level-88 macro**
`(generic-function-name` *generic-function-arguments*`)` → *result-class*	**level-88 generic**
`(generic-function-name` *signature*`)` → *result-class*	**level-88** `generic-function-name` **method**
`(defining-form-name` *defining-form-arguments*`)`	**level-88 defining form**
`condition-class-name(condition-superclass-name)`	**level-88 condition**

1.8.2 Meta-language

The terms used in the following descriptions are defined in section 1.9.

A standard function denotes a constant module binding of the defined name. In addition, some operators are categorized as predicate, constructor, accessor or updator functions. These categories are all specializations of standard or generic function. No additional class information is implied. The notation is simply intended as a guide. 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. Thus the use of a class-descriptive name in the context of a generic function definition defines the intention of the definition, and is not 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. For convenience in dealing with numerical signatures, the definition observes the mathematical convention that

$$natural \subset integer \subset rational \subset real \subset complex$$

The following shorthand is used for different kinds of numbers: z for *complex*, x for the computational approximation to *real* (that is floating point), q for *rational*, i for *integers* and n for *natural* numbers. However, this implies no constraint on the class hierarchy for numbers.

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.

1.8.3 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.

"`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.

"`-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.

"`generic-`" prefix: The generic version of the function named `foo`—usually required to allow for optional or variable numbers of arguments to `foo`.

hyphenation: Function and class names made up of more than one word are hyphenated, for example, `make-vector`.

"`make-`" prefix: For most primitive or defined classes there is constructor function, which is usually named `make-`*class-name*—except where historical precedent is strong, for example, `cons` is used in preference to `make-pair`.

"`n`" prefix: The destructive version of the function named `foo` is usually named `nfoo`, for example the destructive version of `reverse` is named `nreverse`.

"`-object`" suffix: The name of a superclass of a set of related classes. For example: `structure-object` is the (remote) superclass of all user defined structures. The prefix should describe the general domain of the classes in question, but not necessarily any particular class in the set.

The name of the superclass of the class whose name is the prefix, for example `structure-object`.

"`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 `input-stream-p`.

"`q`" suffix: The version of the function named `foo` that uses `eq` for comparison is usually named `fooq`.

"`-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 for the functions that reference the slots of an object, which are called `slot-value` and `indexed-slot-value` and by historical precedent the accessors to fields of pairs are `car` and `cdr`.

“`-using-`” infix: This function name convention is used when calls are cascaded using objects derived from the original object so that users can write methods on the relevant classes. For example, in the case of `slot-value`, the class and slot description corresponding to the original object and slot name are retrieved, and the new generic functions are called (`slot-value-using-class` and `slot-value-using-slot-description`).

When an operation depends on a global value, such as the current input stream, the value might also be made accessible via a function, for example `(a-global-value)` and can be updated using the corresponding setter function, for example `((setter a-global-value) another-value)`.

1.9 Definitions

This set of definitions of basic terms and object-oriented terms, 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.

1.9.1 Basic Definitions

applicable object: An applicable object is an *instance* of any *subclass* of the class `function`.

association list: An association list is a *proper list* whose `car` fields contain *objects* of *class* `pair`.

binding: A location containing a value.

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

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`.

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

congruent:

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.

converter function:

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

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

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.

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.

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 $F1$.

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

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` *bindings* or any *binding form* occurring in E whose variable binding list contains `x`.

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`.

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

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

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

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

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.*

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.*

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

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.

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.*

must: A verbal form used to introduce a *required* property. All conforming processors must satisfy the property.

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.

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

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.

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

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

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.

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

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

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

top lexical: Bindings are created in the *top lexical* environment of a module: those made by `defclass`, `defcondition`, `defconstant`, `defgeneric`, `defmacro`, `defstruct` and `defun` are immutable and those made by `deflocal` are mutable. All such bindings are referenced by *variables* and those made by `deflocal` are modified by `setq`. Each module definition has its own distinct *top lexical* environment.

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

1.9.2 Object-oriented Definitions

This set of definitions of terms relating to classes, objects and generic functions, 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 in this or the previous 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.

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

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

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 signature* specificity.

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 the *class* named `object`, and all *instances* of `class` are *classes*.

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`.

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.

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

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

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

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*$_2$.

direct superclass: A *direct superclass* of a class *class*$_1$ is any *class* for which *class*$_1$ is a direct *subclass*.

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.

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 *signature*.

indirect instance: A indirect instance of a class *class*$_1$ is any *object* whose *class* is a *subclass* of *class*$_1$.

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*.

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*$_2$.

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*.

inherited slot description: See *indirect slot description*.

initarg: A *symbol* used as a keyword in an *initlist* to mark the value of some *slot*. Used in conjunction with `make-instance` 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*.

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*.

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.

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*.

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 eithe *direct* or *indirect*.

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 the *class* `class` to itself. The instantation graph is a tree and the *class* `class` is the root.

metaclass: A metaclass is a *class object* whose *instances* are themselves *classes*. All metaclasses in EuLisp are *instances* of *subclasses* of the class `class`, which is an *instance* of itself. Since they are *classes*, all metaclasses are also *subclasses* of `class`. `class` is a metaclass.

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

method-combination: The applicable method list for an argument list determines the next *method* called by the special form `call-next-method`: for any *method* in the list, the next *method* is simply the *method* following it in the list.

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.

method lookup: See *discrimination.*

method specificity: A signature $signature_1$ is more specific than another $signature_2$ if the first *class* in $signature_1$ is a *subclass* of the first *class* in $signature_2$, or, if they are the same, the rest of $signature_1$ is more specific than the rest of $signature_2$.

multi-method: A *method* which specializes on more than one argument. All methods in this definition are multi-methods.

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

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.*

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

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.

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

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

slot: A named component of an *object* which can be accessed using the function `slot-value`. 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 its set of *slots.*

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 a *direct* or *indirect.*

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.

specialize: A verbal form used to describe the creation of a more specific version of some entity. Normally applied to classes.

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

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 *class* is also defined as a *subclass* of itself, for terminological brevity. A *subclass* can be either *direct* or *indirect*.

superclass: A $class_1$ is a superclass of $class_2$ iff $class_2$ is a subclass of $class_1$. Thus a *class* is both a *subclass* and a superclass of itself. A *superclass* can be either *direct* or *indirect*.

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

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

2 Structure and Interpretation

2.1 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, function parameters are passed by value, and, in common with Scheme and some other Lisps, 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 TEΛOΣ. TEΛOΣ 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 TEΛOΣ 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, TEΛOΣ takes the principle of a reflective architecture, which emphasizes the power of metaclasses as an implementation strategy. From Oaklisp, TEΛOΣ takes the idea of anonymous classes. Finally, from MicroCeyx, TEΛOΣ takes the idea of a small, highly efficient kernel tightly integrated with the rest of the language. In TEΛOΣ, 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 most specific class of which it is an 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 using the function `make-thread` and orderly access to data shared between more than one control thread can be mediated by means of semaphores.

2.2 Language Structure

The EuLisp definition comprises the following items:

Level-0 comprises all the level-0 functions, macros and special forms defined in section 3.1 and all the classes and operations upon them defined in section 4.1. 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 section 3.2 and the classes defined in section 4.2. 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 has only been partiallt specified at the time of writing and is therefore not included in this version of the definition.

A *level-0 function* is a function defined by this report to be part of a conforming processor for level-0, as is a *level-0 generic function*. A (generic) function defined in terms of level-0 operations is an example of a *level-0 application*. Note that, apart from new special forms, the functionality for all **level-1 (generic) functions** can be defined in terms of level-0 operations. Thus, any **level-1 (generic) function** is a level-0 application. The same constructive definition applies to **level-2 (generic) functions** being level-1 applications.

Much of the functionality of EuLisp is defined in terms of modules. These modules might be available (and used) at any appropriate 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. See section 3.2 and section 3.3 for actual requirements. The library modules are defined in section 6.

2.3 Lexical Characteristics

This section gives informal details of the lexical format of EuLisp programs and this is defined precisely in the formal syntax given in appendix A. Case is distinguished both in character and string constants and in 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.

2.3.1 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*, *ideographic*, *special* and *digit* characters starting with a character that is not a *digit*. Appendix A.11 defines these character classes. However, because the common notations for arithmetic operations, the glyphs for plus (`+`) and minus (`-`), are necessary to indicate the sign of a number, they are also classified as identifiers. In this section, and throughout this document, the names for individual character glyphs are those used in ISO/IEC IS 646:1990.

Sometimes, it might be desireable 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* (`|`) in the standard tokenisation scheme. The *multiple-escape* denotes the beginning of an escaped *part* of an identifier and the next *multiple-escape* denotes the end of an escaped part of an identifier. A single character that would otherwise not be a legal constituent can be written by preceding it with *single-escape*, the glyph for which is called *reverse solidus* (\). Therefore, *single-escape* can be used to incorporate the *multiple-escape* or the *single-escape* character in an identifier, delimited (or not) by *multiple-escapes*. For example, `|).(|` is the identifier whose name contains the three characters `#\)`, `#\.` and `#\(`, and `a|b|` is the identifier whose name contains the characters `#\a` and `#\b`. The sequence `||` is the identifier with no name, and so is `||||`, but `|\||` is the identifier whose name contains the single character `|`, which can also be written `\|`, without delimiting *multiple-escapes*.

Any identifier can be used as a literal, in which cases it denotes a *symbol*.

2.3.2 Whitespace and Comments

Whitespace characters are `#\space` and `#\newline`. The character `#\newline` 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, which is 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.

2.3.3 Numeric Literals

There are two kinds of primitive numeric literals: integers and floating point numbers. Instances of literals of the other number classes are constructed from integers and floats: a rational is two integers separated by the *ratio-separator* glyph, which, in the standard tokenisation scheme is called *solidus* (`/`) and a complex number is a pair of numbers distinguished by the prefix `#C`, for example `#C(1 2)`, `#C(1.4 2.3)` and `#C(1/2 3.0)`. The parts of a complex can be any of `integer`, `rational` or `float` and each part can be of a different class. Integers can be read or written in any base up to base 36. For convenience, base 2, base 8 and base 16 have distinguished notations—`#b`, `#o` and `#x`, respectively. The general notation for an arbitrary base is `#`*base*`r`, where *base* is an unsigned decimal number. Informal details of external representations are given in section 2.9 and the formal definitions are given in appendix A.

2.3.4 Character and String Literals

Character literals are denoted by the *extension* glyph, which is called *hash* (`#`), in the standard tokensiation scheme, followed by the *character-extension* glyph, which is called *reverse solidus* (`\`) in the standard tokensiation scheme, followed by the character name. Examples of character literals are `#\a`, `#\\` and `#\newline`, which denote respectively the characters `a`, `\` and the newline character. If a character either does not have a name or does not have a name that can be written and then reconstructed by reading, it can be specified by giving the hexadecimal code of its position in the standard character set by means of between one and four hexadecimal digits. Examples of such character literals are `#\x0`, `#\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. String constants are delimited by the *string-begin* and *string-end* glyphs, which are both defined as the glyph called *quotation mark* (`"`), in the standard tokenisation scheme.

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* (`\`) in the standard tokenisation scheme. String escaping can also be used to include certain other characters that would otherwise be difficult to denote. Here is a summary of *string-escape* digrams:

- *alert*, or `\a`, denotes the *alert* character in a string.
- *backspace*, or `\b`, denotes the *backspace* character in a string.
- *delete*, or `\d`, denotes the *delete* character in a string.
- *formfeed*, or `\f`, denotes the *formfeed* character in a string.

- *linefeed*, or `\l`, denotes the *linefeed* character in a string.
- *newline*, or `\n`, denotes the *newline* character in a string.
- *return*, or `\r`, denotes the *return* character in a string.
- *tab*, or `\t`, denotes the *tab* character in a string.
- *vertical-tab*, or `\v`, denotes the *vertical-tab* character in a string.
- *string-begin*, or `\"`, denotes the *string-begin* character in a string.
- *string-end*, or `\"`, denotes the *string-end* character in a string.
- *string-escape*, or `\\`, denotes a *single* occurrence of *string-escape* in a string.
- *string-hex*, or `\x`, followed by up to four hexadecimal digits, denotes the character associated with that position in the current character-set.

Here are some examples of string literals.

`"a\nb"` contains `#\a`, `#\newline` and `#\b`.

`"c\\"` contains `#\c` and `#\\`.

`"\x1 "` contains `#\x1` followed by `#\space`.

`"\xabcde"` contains `#\xabcd` followed by `#\e`.

`"\x1\x2"` contains `#\x1` followed by `#\x2`.

`"\x12+"` contains `#\x12` followed by `#\+`.

`"\xabcg"` contains `#\xabc` followed by `#\g`.

`"\x00abc"` contains `#\xab` followed by `#\c`.

2.4 Classes, Objects and Generic Functions

The basic classes of EuLisp are elements of the object system class hierarchy, which is shown in Figure 1. Indentation indicates a subclass relationship to the class under which the line has been indented, for example, `condition` is a subclass of `object` and the name following the class is the name of the metaclass, for example, the metaclass of `condition` is `condition-class`. The names given here correspond to the bindings of names to classes as they are exported from the level-0, level-1 or level-2 modules.

The root of the instantiation hierarchy is the class `class`, which is an instance of itself. The root of the inheritance hierarchy is the class `object`. `class` defines the basic methods for access and modification of elements of objects. The designated class of each class is a superclass of the actual class of the class.

2.4.1 Objects

In EuLisp, every object in the system has a specific class. Classes themselves are first-class objects, and thus have classes of their own. In this respect EuLisp differs from statically-typed object-oriented languages such as C++ and μCeyx.

Programs written using TEΛOΣ 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.

```
object class
    character class
    class class
       condition-class class
       function-class class
       number-class class
       structure-class class
    condition condition-class
       execution-condition condition-class
       stream-condition condition-class
       arithmetic-condition condition-class
       ...
    function function-class
       continuation function-class
       generic-function function-class
    method class
    null class
    number class
       ...
    pair class
    stream class
    string class
    structure structure-class
    symbol class
    table class
    thread class
    vector class
```

Figure 1: Level-0 initial class hierarchy

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.

2.4.2 Classes

A class describes a set of objects, called *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 superclass will also be defined for a class, unless overridden by methods defined on subclasses of the class.

In EuLisp, classes are first-class objects and thus have classes of their own. 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 `structure-class` is an example.

Classes are defined using the `defstruct`, `defcondition` and `defclass` defining forms, which are described in detail in sections 3.2.6. New metaclasses are defined using `defclass`.

2.4.3 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 defined for instances of a particular class will be applicable for instances of all of the class's subclasses.

TEΛOΣ 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, or it might only be possible after some internal changes are made. The function, `metaclass-compatible-p` provides a means for a metaclass programmer to determine whether two metaclasses are or can be made compatible and `make-metaclass-comaptible` carries out the necessary changes.

2.4.4 Slots

The components of an object are called its *slots*. Each slot of a class 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** and above the slot description mechanism is modifiable and extensible.

Users can define 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, which are described in detail in section 4.2.2. New slot description classes are defined by `defclass`.

2.4.5 Generic Functions

A generic function is a function whose behaviour is determined by the classes of its arguments. Each potential behavior is defined by a method, which specifies a signature of classes for which it is applicable. A 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 global generic function, and `generic-lambda`, which creates an anonymous generic function. These forms are described in detail in 3.1.12.

2.4.6 Methods

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

Like slots, methods may be inherited. That is, if a method is applicable for a class $\texttt{C}_1$, it is also applicable for all of $\texttt{C}_1$'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 are defined using the `defmethod` macro, which adds a new method to a generic function. This macro is described in detail in 4.1.13.

2.5 Conditions

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.

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 number of conditions are defined at level-0 (see section 3.1.9). 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 (see section 3.1).

2.6 Modules

A module definition creates two, new, empty lexical environments—the internal and the external top-lexical environments of the module. All the definitions in the module body are stored in the former along with those definitions shared (by importation) with other modules. The latter shares those definitions from the internal top-lexical that are exported and also those of all the exported modules that are 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.

2.6.1 Imports

The import directive is expressed in terms of `except`, `only` and `rename`, the interpretation of which is defined in section 3.1.12. The import directive specifies which names from which other modules are to be visible in the current module.

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-names*, then there is a name clash which is an error.

2.6.2 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 the body of the module is then expanded according to that environment. The basic expansion mechanism examines each form in the module body. If the name of the operator of the form is bound in the syntax environment then the associated expansion function is called with a list of the (unevaluated) operands of the form. If the the operator is not defined in the syntax environment, each of the sub-forms is examined and expanded recursively.

> *NOTE—the expansion mechanism is still an open topic. Both syntactic closures and the techniques described in "Macros That Work" have been examined, but both suffer from problems, the latter in particular since they don't work.*

2.6.3 Exports

The export directives are expressed in three ways: `export`, which is used to specify individual names with a value attribute, `export-syntax`, which is used to specify individual names with a syntax attribute, and `expose`, which is used to specify collections of names from whole modules (see the example in Figure 2). The export directives of a module taken together specify which names from this and other modules are to be exported from this module.

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 an 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.

2.6.4 Definitions and Expressions

Definitions in a module only contain unqualified names—that is, *local-names*, 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.

2.6.5 Module Processing

The following steps summarize the module definition process:

1. The importations are checked. For each imported binding, the originating module must exist and the desired item must be exported from it. Each binding import specification contributes a new binding 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.

2. Syntax expansion of the body. 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.

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

4. The exportations are collected and the set of exported names is constructed.

5. The expanded body of the module is analysed. It is an error, if a variable in the body does not have a binding in the top-lexical environment.
6. The module is initialized by evaluating the forms in the body in the order they appear.

```
(defmodule example
  (classes arithmetic            ;;import classes and arithmetic
   (except (null) class-names)   ;;all but null from class-names
   (only (open) streams)         ;;but just open from streams
                                 ;;exchange the names of the bindings of car and cdr
   (rename ((car cdr) (cdr car)) lists))

  (syntax
                                 ;;rename the binding of standard lambda as eulisp-lambda
    ((rename ((lambda eulisp-lambda)) level-0-syntax)
     macros                      ;;all of macros
                                 ;;now the redefined lambda which was called new-lambda
     (rename ((new-lambda lambda)) more-macros))
    ...
  )
  ...
  (expose arithmetic extras)     ;;export arithmetic and extras
  ...
  (export example-fun1           ;;but just three functions from this module
          example-fun2
          example-var1)
  ...
)
```

Figure 2: Example of import and export directives

An example module definition is given in Figure 2. The imports to this module are specified in the first expression: all the exports of the modules `classes` and `arithmetic`, all of module `class-names` except `null`, nothing from module `streams` except `open` and everything from module `lists`—but the names of the bindings of `car` and `cdr` are exchanged. Next is the syntax section which imports the modules `level-0-syntax`, `macros` and `more-macros` to be used in the syntax expansion of the body. Of particular note here is the renaming of the expander for `lambda` as `eulisp-lambda` and the subsequent renaming of `new-lambda` defined in `more-macros` as `lambda`. In consequence all lambda forms in the body will be expanded according to the function `new-lambda` defined in `more-macros`. The remainder of the module definition is the body, comprising definitions and export specifications. In the first export directive, the `arithmetic` module is re-exported from `example` and the non-imported module `extras` is exported. In the second, `example-fun1`, `example-fun2` and `example-var1` are exported from `example`. Thus, the exports of `example` are the above three names and the exported names of the modules `arithmetic` and `extras`.

2.7 Threads and Semaphores

Threads provide a means for concurrent execution and semaphores offer a mechanism to synchronize access to data structures shared by several concurrent threads.

A semaphore is an abstract data type protecting an integer variable which can only take on non-negative values. This is called a *counting* semaphore as distinct from the primitive *binary* semaphore. A semaphore may be initialized with any non-negative value. The operations on a sempahore are `semaphore-up` and `semaphore-down`, which are also widely known as *signal* and *wait*, respectively.

A thread is an abstract data type protecting some implementation-defined data. A thread is constructed by `make-thread` and its status is set to `virgin`. A thread is made runnable by means of `thread-start`. This is a generic function to allow implementation-defined extensions specializing threads, but the method for `thread` adds the thread to the run queue of the standard scheduler. Threads in EuLisp may not be preempted. Thus a thread cedes control to another thread explicitly via the scheduler, using the function `thread-suspend`. The standard scheduler takes the next thread from its queue and, if its state is `runnable`, control is given to that thread. When control returns to the scheduler, the thread ceding control is appended to the run queue. One thread can change another thread's state to `runnable` by means of the function `proceed`.

2.8 Numbers

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 and level-1, only a limited set of number classes are defined. A general, user-extensible number mechanism is introduced at level-2.

> *NOTE—At present the class hierarchy and number operations for level-2 have not been finalized, but will probably include* `complex`*,* `variable-precision-float`*,* `integer-mod-n` *and* `integer-mod-p`*.*

2.8.1 Level-0 Numbers

In Figure 3 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 metaclass 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 3.

```
number class                                    single-precision-integer
   float number-class                              → double-float
      double-float number-class
   integer number-class
      single-precision-integer number-class
```

Figure 3: Level-0 number class hierarchy and coercion chart

Any level-0 version of a library module, for example, `elementary-function`, need only define methods for these two classes.

2.8.2 Level-1 Numbers

In Figure 4 is an example of what the initial number class hierarchy for level-1 might look like. The inheritance relationships implied by this diagram are part of this definition, but the same *caveats* listed under the description of level-0 numbers also apply here. Level-1 extends level-0 by the addition of variable precision integers (class `variable-precision-integer`), single precision floating point (class `single-float`) and rational numbers (class `ratio`). The components of an instance of `ratio` are subclasses of `integer`.

However, unlike level-0, coercion at level-1 can lead to overflow in converting an instance of `variable-precision-integer`—and, hence, an instance of `ratio`—to `single-float` or `double-float`. An error is signaled (condition: `floating-point-conversion-overflow`) if overflow is detected. Although coercion is defined as a path through a sequence of types, conversion operations should be done in one step.

```
number class                                   single-precision-integer
   float number-class                             → variable-precision-integer
      single-float number-class                      → ratio
      double-float number-class                         → single-float
   integer number-class                                    → double-float
      single-precision-integer number-class
      variable-precision-integer number-class
   ratio number-class
```

Figure 4: Level-1 number class hierarchy and coercion chart

2.9 External Representations

Objects in the following classes have defined print representations: `character`, `number`, `pair`, `string`, `symbol`, and `vector`. These representations are produced by the function `write` and permit `read` to construct a copy that is `equal` to the original object. Precise details of the syntax are given in appendix A. There are also print representations, more amenable to the eye, that are not guaranteeed to allow `read` to construct an `equal` copy of the object. These representations are produced by the function `prin`. Details of `read` and character syntax classes in the standard tokenisation scheme are in appendix A. Here is an informal description and some examples of the external representation of the classes listed some fundamental types:

character: A character is written out as `#\`*?*, where *?* is the glyph associated with the character in question, except for the special cases of *alert*, *backspace*, *delete*, *formfeed*, *linefeed*, *newline*, *return*, *tab space* and *vertical-tab*, which are displayed respectively as `#\alert`, `#\backspace`, `#\delete`, `#\formfeed`, `#\linefeed`, `#\newline`, `#\return`, `#\tab`, `#\space`, and `#\vertical-tab`. A character which does not have a glyph for its external representation is written out as `#\x` followed by up to four hexadecimal digits, the value representing the position of the character in the current character set.

number: A positive integer is written 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`. A rational number is written as two integers separated only by `/`, the first of which can be preceded by a sign. For example, `123/456`, `-1/2`, `+3/4`. A floating point number is written as: `123.456`, `-5.678E13`, `+3.421E-9` (see appendix A.9), The number styles here are the default ones used by `write` and `prin`, but more control is available via the function `format`, for example: control of field width, scientific notation.

pair: 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 this section.

string: A string is written as: `"1234567890"` except that `write` outputs a re-readable form of the escaped characters. 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
```

as output. 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 rules given in section 2.3.4 and omits the *string-begin* and *string-end* characters.

symbol: A symbol is represented by its printname, which is a true string, written without enclosing double quotes. 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 `foo.bar` 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 `foo.bar`, which `read` interprets as the symbol `a` followed by the start of a list, the symbol `b` and the symbol `foo.bar`.

vector: A vector is written as `#(`$obj_1 \ldots obj_n$`)`. The representations of obj_i are determined by the external representations defined in this section.

Some classes of objects have a distinguished external representation but cannot be reconstructed by read. Here are some examples of such classes: `function`, `class`, `table`, `thread`, `stream`.

3 Basic Expressions

3.1 Level-0 Expressions

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.

3.1.1 Primitive Expressions

`constant` **level-0 syntax**

There are two kinds of constant, literal constants and defined constants. The latter are considered under `symbols`. 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.

`symbol` **level-0 syntax**

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

3.1.2 Constant and Literal Expressions

`(quote` *datum*`)` → *datum* **level-0 special form**

The result of processing the expression `(quote` *datum*`)` is *datum*. The object *datum* can be any external representation of a EuLisp object—see section 2.9 for an informal treatment of external representations and appendix A for formal details. 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.

3.1.3 Assignments

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

`(setq` *identifier form*`)` → *obj* **level-0 special form**
The *form* is evaluated and the result is stored in either the closest lexical binding named by *identifier*. The value returned is the value of *form*. It is an error to modify an immutable binding.

`(setter` *access-function*`)` → *update-function* **level-0 function**
A simple 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-function`). 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`)

3.1.4 Conditional Expressions

`(if` *antecedent consequence alternative*`)` → *obj* **level-0 special form**
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 3.2.3.

`(cond (`*antecedent form**`)*)` **level-0 macro**
The `cond` operator 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*)
                                        ...)))
```

`(and` *form**`)` **level-0 macro**
The expansion of an `and` form leads to the evaluation of the sequence of *forms* from left to right. The the first *form* in the sequence that evaluates to `()` stops evaluation and none of the *forms* to its right will be evaluated. The result of `and` is `()`. If none of the *forms* 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 ...) ())
```

(`or` *form**) **level-0 macro**

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 $form_2 \ldots form_n$.

3.1.5 Variable Binding and Sequences

(`lambda` *lambda-list body*) → *function* **level-0 special form**

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:

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

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*.

(`let/cc` *identifier body*) → *obj* **level-0 special form**

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. The result of evaluating the last form in *body* is returned as the result of the `let/cc` form. 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.

(`let` (*binding**) *body*) **level-0 macro**

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*)
```

(`progn` *form**) → *obj* **level-0 special form**

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

3.1.6 Function Calls and Application

A function call is written (*operator operand**). Details of the processing of expressions are given in §5.1. An error is signaled (condition: `invalid-operator`) if the operator is neither a special form nor a function.

`(apply` *function* obj_1 ... obj_n`)` → *obj* **level-0 function**
Calls *function* with actual parameter list created by appending obj_n to a list of the arguments obj_1 through obj_{n-1}. An error is signaled (condition: `processing-condition`) if obj_n is not a proper list.

3.1.7 Method Combination

The following operators are used to affect the order of method application. It is an error to use either of these functions 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.

`(call-next-method )` → *obj* **level-0 special form**
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.

`(next-method-p )` → *boolean* **level-0 special form**
If there is a next most specific method, `next-method-p` returns a non-`()`, otherwise, it returns `()`.

3.1.8 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`.

`(signal` *condition continuation*`)` → *null* **level-0 function**
The function `signal` calls the dynamically closest handler with *condition*—the condition being signaled—and either *continuation* or `()`. If the second argument is a subclass of `continuation`, 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.

`(with-handler` *handler-function protected-form**`)` → *obj* **level-0 special form**
The `with-handler` sets up *handler-function* so that it can be executed in the event of a `signal` occurring during the evaluation of the sequence of *protected-form*s. A handler function takes two arguments—the condition, and a *resume* 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. A `with-handler` expression is evaluated in three steps:

1. The new *handler-function* is constructed and identifies the dynamically closest handler.
2. The dynamically closest handler is shadowed by the establishment of the new *handler-function*.
3. 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.
4. 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:

1. 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.

2. 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.

3. 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.

`(conditionp` *obj*`)` → *boolean* **level-0 predicate**
Returns *obj* if the class of *obj* is a subclass of `condition`, otherwise `()`.

`(make-condition` *condition-class init-option**`)` → *condition* **level-0 constructor**
The result of `make-condition` is an instance of a condition class containing specific information about the particular condition that has arisen. The specific information depends on the condition class and its specification depends what was defined for the *init-option*s when the class was defined. The resulting condition object can be passed to `signal`, which will then pass it to the dynamically closest *handler-function*. The *init-option*s are specified under the `initialize-instance` method for the class `condition`.

`(condition-message` *condition*`)` → *string* **level-0 function**
Returns the contents of the message slot of *condition*, which is a string.

`(initialize-instance condition` *init-option**`)` → *condition* **level-0 `initialize-instance` method**
First calls `call-next-method` to carry out initialization specified by superclasses then does the `condition` specific initialization. The following *init-option* is recognised for this method:

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

`(cerror` *error-message condition init-option**`)` → *null* **level-0 function**
`(error` *error-message condition init-option**`)` → *null* **level-0 function**
The `cerror` and `error` functions signal continuable and non-continuable errors, respectively. Each calls `signal` with an instance of a condition of class *condition* 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.

`(unwind-protect` *protected-form after-form**`)` → *obj* **level-0 special form**
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-forms* 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.

3.1.9 Defined Conditions

`execution-condition(condition)` **level-0 condition**
This is the root condition class for all conditions that are related to errors detected during, and as a consequence of, the execution of a program.

`wrong-class-object(execution-condition)` **level-0 condition**
`no-setter-function(execution-condition)` **level-0 condition**
`cannot-update-setter(execution-condition)` **level-0 condition**
`invalid-operator(execution-condition)` **level-0 condition**
`unquote-no-context(execution-condition)` **level-0 condition**
`improper-unquote-splice(execution-condition)` **level-0 condition**
The above specialized condition classes are defined to identify specific execution conditions.

`telos-condition(condition)` **level-0 condition**
This is the root condition class for all conditions that are related to the object system.

`slot-unbound(telos-condition)` **level-0 condition**
`slot-missing(telos-condition)` **level-0 condition**
`non-allocatable-object(telos-condition)` **level-0 condition**
`no-applicable-method(telos-condition)` **level-0 condition**
`non-congruent-lambda-lists(telos-condition)` **level-0 condition**
`incompatible-method-signature(telos-condition)` **level-0 condition**
`no-next-method(telos-condition)` **level-0 condition**
`method-in-use(telos-condition)` **level-0 condition**
The above specialized condition classes are defined to identify specific object system conditions.

`thread-condition(execution-condition)` **level-0 condition**
This is the root condition class for all conditions related to threads.

`arithmetic-condition(execution-condition)` **level-0 condition**
This is the root condition class for all conditions related to arithmetic.

`division-by-zero(arithmetic-condition)` **level-0 condition**
`zero-argument-in-ulp(arithmetic-condition)` **level-0 condition**

`conversion-condition(execution-condition)` **level-0 condition**
This is the root condition class for all conditions related to `convert`.
`floating-point-conversion-overflow(conversion-condition)` **level-0 condition**
`integer-conversion-overflow(conversion-condition)` **level-0 condition**
`cannot-convert-to-character(conversion-condition)` **level-0 condition**
`improper-list-conversion(conversion-condition)` **level-0 condition**

`stream-condition(execution-condition)` **level-0 condition**
This is the root condition class for all conditions related to streams.

`incompatible-streams(stream-condition)` **level-0 condition**
`cannot-open-path(stream-condition)` **level-0 condition**
`file-already-exists(stream-condition)` **level-0 condition**
`inconsistent-open-options(stream-condition)` **level-0 condition**
`invalid-stream-position(stream-condition)` **level-0 condition**
`invalid-output-base(stream-condition)` **level-0 condition**
`not-an-input-stream(stream-condition)` **level-0 condition**
`not-an-output-stream(stream-condition)` **level-0 condition**
`not-an-io-stream(stream-condition)` **level-0 condition**
`not-a-character-stream(stream-condition)` **level-0 condition**

`not-a-binary-stream(stream-condition)` **level-0 condition**
`not-a-positionable-stream(stream-condition)` **level-0 condition**
`path-does-not-exist(stream-condition)` **level-0 condition**
`stream-not-open(stream-condition)` **level-0 condition**
The above specialized condition classes are defined to identify specific stream conditions.

`environment-condition(execution-condition)` **level-0 condition**
This is the root condition class for all conditions related to the environment.

3.1.10 Quasiquotation Expressions

(`quasiquote` *skeleton*) **level-0 macro**
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*.

(`unquote` *form*) **level-0 syntax**
(`unquote-splicing` *form*) **level-0 syntax**
The holes in a quasiquoted expression are identified by “unquote” expressions and these come in two forms—expressions whose value is to be inserted at that location in the structure and expressions whose value is to be spliced into the structure at that location. The former is indicated by an `unquote` form and the latter by an `unquote-splicing` form. An “unquote-splice” expression 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. An error is signaled (condition: `unquote-no-context`) if either of these syntaxes occurs outside the scope of a `quasiquote` form.

The macros `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 an expression. Thus, `(unquote a)` may be written `,a` and `(unquote-splicing a)` can be written `,@a`.

3.1.11 Module Definition

(`defmodule` *module-name import-spec syntax-spec module-expression**) **level-0 syntax**
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*.

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*$_{\{\texttt{defmodule}\}}$

A sequence of *module-names* and or *module-directives* is treated as the union of all the names generated by each element of the sequence. It is an error is any name occurs more than once. Elements of an *import-spec* are interpreted as follows:

except: Filters the names from *module-name* or *module-directive* discarding (*module-name name*) and keeping all other names. The `except` directive is convenient when needing almost all of the names in a module by naming just the few names that are not wanted from a module.

module-name: Extracts all the exported names from *module-name*.

only: Filters the names from *module-name* or *module-directive* keeping only those names specified. If (*module-name name*) occurs in the exported names it is kept. The `only` directive is convenient when only a few names are needed from a module.

rename: Renaming is a substitution filter that takes the names from *module-name* or *module-directive* and replaces (*module-name old-name*) with (*module-name new-name*). All other names are passed unchanged.

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

export: Each of the names appearing in the `export` form is added to the set of exports of the module.

export-syntax: Each of the names appearing in the `export` form is added to the set of exports of the module with the syntax attribute set.

expose: Processes the *module-directives* appearing in the `expose` form following the rules for *import-spec* and adds the resulting set of names to the exports of the module.

3.1.12 Definitions

(`defcondition` *condition-name superclass init-option**) **level-0 defining form**
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.

(`defconstant` *identifier form*) **level-0 defining form**
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.

(`defgeneric` *gf-name gen-lambda-list init-option**) **level-0 defining form**
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-signature`) 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 a identifier followed by a corresponding value. The syntax of `defgeneric` is as follows:

gf-name ::= *identifier*
gen-lambda-list ::= *spec-lambda-list*
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*

The only *init-option* at level-0 is:

method: 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 signature, 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.

(`deflocal` *name form*) **level-0 defining form**
The value of *form* is stored as the module binding value of *name*. The binding created by a `deflocal` form is mutable.

(`defmacro` *macro-name lambda-list body*) **level-0 defining form**
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 3.1.5). The binding created by `defmacro` is immutable.

(`defstruct` *class-name superclass* (*slot-description**) *class-option**) **level-0 defining form**
`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. If *superclass* is `()`, the superclass is taken to be the root structure class `structure`. 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.

class-name ::= *identifier*
superclass ::= {`structure-class` *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*

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

initarg: The value of this option is a identifier naming a symbol, which is the name of an argument to be supplied in the *init-option*s of a call to `make-instance` on the new class. The value of this argument in the call to `make-instance` is the initial value of the slot. This option must only be specified once for a particular slot. The same initarg name may be used for several slots, in which case they will share the same initial value if the initarg is given to `make-instance`.

initform: 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-instance`. The form is evaluated in the lexical environment of the call to `defstruct` and the dynamic environment of the call to `make-instance`. The form is evaluated each time `make-instance` is called and the default value is called for. The order of evaluation of the initforms in all the slots is determined by `initialize-instance`. This option must only be specified once for a particular slot.

reader: 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 created 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: 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: The value is the identifier of the variable to which the reader function will be bound. In addition, the use of this *slot-option* causes the creation of a writer function, which is anonymous, but 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:

constructor: 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. Specifying the constructor in this way is equivalent to writing a `defconstructor` form for the class.

predicate: 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. Specifying the constructor in this way is equivalent to writing a `defpredicate` form for the class.

`(defun` *function-name lambda-list body*`)` → *symbol* **level-0 defining form**
`(defun (setter` *function-name*`)` *lambda-list body*`)` → *symbol* **level-0 defining form**
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 3.1.5). The binding created by `defun` is immutable.

3.2 Level-1 Expressions

This section gives the informal syntax of well-formed expressions and describes the semantics of the special-forms and primitive functions of the level-1 language. In the case of level-1 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.

3.2.1 Dynamic Binding

`(dynamic` *identifier*`)` → *obj* **level-1 special form**
The closest dynamic binding of `symbol` named by identifier is returned. If no such binding exists, an error is signaled (condition: `unbound-dynamic-variable`).

`(dynamic-setq` *identifier form*`)` → *obj* **level-1 special form**
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.

`(dynamic-let (`*binding**`)` *body*`)` → *obj* **level-1 special form**
The *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 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`.

3.2.2 Lexical Binding Extensions

`(labels ((`*function-name lambda-list body*`)`*`)` *labels-body*`)` **level-1 macro**
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-names* 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. The rewrite rule for `labels` is:

```
(labels ((var_1 lambda-list_1 body_1) ≡ (let ((var_1 ())
         (var_2 lambda-list_2 body_2)         (var_2 ())
         ...)                                 ...
  form*)                                   (setq var_1 (lambda lambda-list_1 body_1))
                                           (setq var_2 (lambda lambda-list_2 body_2))
                                           ...
                                           form*)
```

`(let* (`*binding**`)` *body*`)` **level-1 macro**
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* ((var_1 form_1)   ≡  (let ((var_1 form_1))
       (var_2 form_2)        (let* ((var_2 form_2)
       var_3                        var_3
       ...)                         ...)
  form*)                       form*))
```

`(generic-lambda` *lambda-list init-option**`)` **level-1 macro**
`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`. Note that an error is be signaled (condition: `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.

`(generic-labels (`*lambda-list init-option**`)` *form**`)` **level-1 macro**
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`.

3.2.3 Conditional Extensions

`(when` *antecedent form**`)` **level-1 macro**
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*$_2$ ...`) ())`

`(unless` *antecedent form**`)` **level-1 macro**
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*$_2$ ...`))`

3.2.4 Exit Extensions

`(block` *identifier form**`)` **level-1 macro**
`(return-from` *identifier* `[`*form*`])` **level-1 macro**

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(s) of the last one is returned as the value(s) of the block form.

In `return-from`, the *variable* names the continuation of the (lexical) `block` from which to return. An error is signaled (condition: `invalid-return-continuation`) if the value of the variable named by *identifier* is not a continuation. `return-from` is the invocation of the continuation of the block named by *variable*. The *form* is evaluated the value(s) are returned as the value(s) of the block named by *variable*. The rewrite rules for `block` and for `return-from` are as follows:

`(block)` ≡ Is an error
`(block` *identifier*`)` ≡ `()`
`(block` *identifier form**`)` ≡ `(let/cc` *identifier form**`)`
`(return-from)` ≡ Is an error
`(return-from` *identifier*`)` ≡ `(`*identifier* `())`
`(return-from` *identifier form*`)` ≡ `(`*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.

`(catch` *tag form**`)` **level-1 macro**
`(throw` *tag form*`)` **level-1 macro**

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.

In `throw`, the *tag* names the continuation of the `catch` from which to return. One of two conditions might arise in `throw` depending on whether the dynamic value of *tag* is undefined (condition: `unbound-dynamic-variable`) or its value is not a continuation (condition: `invalid-throw-continuation`). `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 `catch` and `throw` are:

`(catch)` ≡ Is an error
`(catch` *tag*`)` ≡ `(progn` *tag* `())`
`(catch` *tag form**`)` ≡ `(let/cc tmp (dynamic-let ((`*tag* `tmp))` *form**`))`
`(throw)` ≡ Is an error
`(throw` *tag*`)` ≡ `((dynamic` *tag*`) ())`
`(throw` *tag form*`)` ≡ `((dynamic` *tag*`)` *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.

3.2.5 Defined Conditions

`unbound-dynamic-variable(execution-condition)` **level-1 condition**
`invalid-return-continuation(execution-condition)` **level-1 condition**
`invalid-throw-continuation(execution-condition)` **level-1 condition**
`not-a-symbol(execution-condition)` **level-1 condition**
`symbol-multiply-defined(execution-condition)` **level-1 condition**

The above specialized condition classes are defined to identify execution conditions defined at level-1.

`bad-method-class(telos-condition)` **level-1 condition**
`orphan-method-call(telos-condition)` **level-1 condition**

The above specialized condition classes are defined to identify specific object system conditions at level-1.

3.2.6 Definitions

At level-1 `defgeneric` is extended to allow the use of user-defined generic function classes and method classes. This is done by extending the *init-option*s component of the specification.

`(defclass` *class-name* `(`*superclass**`)` `(`*slot-description**`)` *class-option**`)` **level-1 defining form**

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 will be `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 as follows:

class-name ::= *identifier*
superclass ::= {`class` *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* |
`slot-class` *slot-description-class* |
identifier value
class-option ::= `constructor` *constructor-spec*
`predicate` *predicate-name*
`metaclass` *class-name* |
identifier value

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

initarg: As defined at level-0. See section 3.1.12.

initform: As defined at level-0. See section 3.1.12.

reader: As defined at level-0. See section 3.1.12.

writer: As defined at level-0. See section 3.1.12.

accessor: As defined at level-0. See section 3.1.12.

slot-class: The corresponding value is an instance of a subclass of `slot-description-class`. An implementation conforming at level-1 provides the slot description classes `local-slot-description`, for slots particular to instances and `shared-slot-description`, for slots whose values are shared by all the instances of the class. New slot description classes can be defined and used here. This option can only be specified once for a particular slot. Within a class different slots can have different slot description classes.

identifier expression: The symbol named by *identifier* and the value of *expression* are passed in the call to `make-instance` 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.

constructor: As defined at level-0. See section 3.1.12.

predicate: As defined at level-0. See section 3.1.12.

metaclass: The value of this option is the class of the new class. By default, this is `class`. This option must onlybe specified once for the new class.

identifier expression: The symbol named by *identifier* and the value of *expression* are passed in the call to `make-instance` 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.

(`defgeneric` *gf-name lambda-list init-option**) **level-1 defining form**
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 as follows:

level-1-init-option ::= *level-0-init-option* |
`class` *gf-class-name* |
`method-class` *method-class-name* |
`method` (*method-description*)
gf-class-name ::= *class-name*
method-class-name ::= *class-name*
class-name ::= {`class` *or the name of one of its subclasses*}

The *init-options* are interpreted as follows:

class: The class of the new generic function. This must be a subclass of `generic-function`. The default is `generic-function`.

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`.

method: As defined at level-0. See section 3.1.12.

(`defvar` *identifier form*) **level-1 defining form**
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: `symbol-multiply-defined`), on evaluating this form more than once for the same *identifier*.

3.3 Level-2 Expressions

NOTE—Nothing has been defined for level-2 at the time of writing.

4 Classes and Objects

4.1 Level-0 Classes

4.1.1 Accessing Objects

(`class-of` *obj*) → *obj* **level-0 function**
`class-of` is a total function capable of taking any entity in the system and returning an object representing its class. The composition of this function with itself is a function that returns the metaclass of an object.

(`initialize-instance` `object` *init-option**) → *object* **level-0 `initialize-instance` method**
The default method for `initialize-instance` looks at the legal initargs for the given class, and the set of given initargs. For those initargs were given in the *init-options*, the appropriate slots are set to the specified value. For any slots not given values during this phase, the initform, if any, for the slot is called and the resulting value placed in the slot. Any other slots remain unchanged. The initialized object is returned.

(`slot-value` obj_1 obj_2) → *obj* **level-0 function**
`slot-value` returns the object associated with the slot named obj_2 in obj_1. If no slot with the given name

is defined in the object's class, an error is signaled (condition: `slot-missing`). If the slot is unbound, an error is signaled (condition: `slot-unbound`).

`((setter slot-value)` *obj*$_1$ *obj*$_2$ *obj*$_3$`)` → *obj* **level-0 function**
This is the corresponding updator for `slot-value`. The new value is returned. If no slot with the given name is defined in the object's class, an error is signaled (condition: `slot-missing`).

`(slot-exists-p` *obj symbol*`)` → *boolean* **level-0 generic**
This generic function determines if a slot of the given name exists in the given object. By default, it matches the name against the list of slot descriptors in the object's class.

`(slot-bound-p` *obj symbol*`)` → *boolean* **level-0 generic**
This generic function determines if a slot of the given name is bound in the object. If the slot does not exist in the object, an error is signalled (condition: `slot-missing`).

`(unbind-slot` *obj symbol*`)` → *obj* **level-0 generic**
This generic function makes the slot of the specified name in the object be unbound. If the slot does not exist in the object, an error is signaled (condition: `slot-missing`).

4.1.2 Comparing Objects

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:

$$\begin{array}{ccccc} \texttt{(eq}\ a\ b\texttt{)} & \Rightarrow & \texttt{(eql}\ a\ b\texttt{)} & \Rightarrow & \texttt{(equal}\ a\ b\texttt{)} \\ \texttt{(eq}\ a\ b\texttt{)} & \not\Leftarrow & \texttt{(eql}\ a\ b\texttt{)} & \not\Leftarrow & \texttt{(equal}\ a\ b\texttt{)} \end{array}$$

`(eq` *obj*$_1$ *obj*$_2$`)` → *boolean* **level-0 function**
Compares *obj*$_1$ and *obj*$_2$ and returns `t` if they are the same object, otherwise `()`. 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.

`(=` *number*$_1$ *number*$_2$`)` → *boolean* **level-0 generic**
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 2.8.1 or in section 2.8.2.

`(eql` *obj*$_1$ *obj*$_2$`)` → *boolean* **level-0 function**
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`.

`(equal` *obj*$_1$ *obj*$_2$`)` → *boolean* **level-0 generic**
The result is determined by whichever of the methods defined in Table 1 is applicable. It is implementation-defined whether or not `equal` will terminate on self-referential structures.

Argument Class	Description
`character`	Each instance of `character` is converted to a number and these values are compared using `=`. The result of `equal` is the first argument if the result of `=` is non-`()`. If not, the result is `()`.
`number`	If the class of each instance of `number` 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 `()`.
`string`	If the length of each instance of `string` 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 `()`.
`vector`	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 `()`.
`pair`	The result is the conjunction of the pairwise application of `equal` to the `car` fields and the `cdr` fields of the arguments.
`object`	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 `()`.

Table 1: Methods for `equal`

4.1.3 Copying Objects

`(copy` *obj*`)` → *obj* **level-0 generic**

Constructs of 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 below. Additional, specialized, copy functions are defined in the subsections of section 4. The methods defined on `copy` are given in Table 2.

4.1.4 Conversion

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. Table 3 details the classes for which converter functions are defined and the methods which are defined on those converter functions.

`(convert` *obj class*`)` → *obj* **level-0 function**

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*.

`(converter` *target-class*`)` → *generic-function* **level-0 function**

`((setter converter)` *target-class generic-function*`)` → *generic-function* **level-0 function**

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. 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`. 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`.

4.1.5 Classes

`(make-instance` *class init-option**`)` → *obj* **level-0 function**

There are two phases in the creation of a new instance: allocating the physical memory, and initializing

Source Class	Description
`single-precision-integer`	Constructs and returns an instance of `single-precision-integer`, whose value is the same (under `=`) as the source.
`variable-precision-integer`	Constructs and returns an instance of `variable-precision-integer`, whose value is the same (under `=`) as the source.
`float`	Constructs and returns an instance of `float`, whose value is the same (under `=`) as the source.
`ratio`	Constructs and returns an instance of `ratio`, whose value is the same (under `=`) as the source.
`complex`	Constructs and returns an instance of `complex`, whose value is the same (under `=`) as the source.
`character`	Constructs and returns an instance of `character`, whose value is the same (under `equal`) as the source.
`pair`	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.
`string`	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.
`vector`	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.
`object`	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 2: Methods defined on `copy`

the object. These phases are under control of the new object's metaclass and class respectively. Defining new classes and metaclasses often means defining new methods on the appropriate generic functions so that metainstances are allocated correctly and instances initialized correctly.

The first argument to this generic function is a class. The remaining arguments are *init-options* to be passed to `allocate-instance` and `initialize-instance`—see below. By default, `make-instance` returns a newly allocated and initialized instance of the class by calling the generic functions `allocate-instance` with the given class (thus specializing on the metaclass) and the initialization list, and then `initialize-instance` on the newly allocated instance and the initialization list. The function returns the newly allocated and initialized instance.

A new class is created by applying `make-instance` to the desired metaclass. At level-0, `make-instance` is legal on two metaclasses, `structure-class` and `condition-class` and several primitive classes: `generic-function`, `method` and their subclasses, subclasses of `condition` and instances of `structure-class`. Therefore, level-0 admits a limited class definition facility called `defstruct`, that cannot produce classes from arbitrary metaclasses. The class of classes defined with `defstruct` is `structure-class`.

`(subclassp` *class*$_1$ *class*$_2$`)` → *boolean* **level-0 function**

Determines if its first argument is a subclass of its second argument, and, if so, returns its first argument, otherwise `()`.

`(allocate-instance` *class init-option**`)` → *obj* **level-0 generic**

The first argument is a class, the remaining arguments are symbols and values to be passed to `make-instance`. `allocate-instance` returns a new instance of the class with each component unbound. Since the argument

Target Class	Source Class	Description
`character`	`integer`	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.
`integer`	`character`	Returns an instance of `single-precision-integer` which corresponds to the position of the instance of `character` in the default character set.
`string`	`single-precision-integer`	Constructs and returns a string, the characters of which correspond to the external representation of the instance of `single-precision-integer` in decimal.
	`variable-precision-integer`	Constructs and returns a string, the characters of which correspond to the external representation of the instance of `variable-precision-integer` in decimal.
	`single-float`	Constructs and returns a string, the characters of which correspond to the external representation of the instance of `single-float`.
	`double-float`	Constructs and returns a string, the characters of which correspond to the external representation of the instance of `double-float`.
	`ratio`	Constructs and returns a string, the characters of which correspond to the external representation of the instance of `ratio`.
	`complex`	Constructs and returns a string, the characters of which correspond to the external representation of the instance of `cartesian-pair`.
	`pair`	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: `improper-list-conversion`) unless all of those elements are instances of the class `character`.
`pair`	`string`	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`.
	`vector`	Constructs and returns a proper list, the elements of which correspond to the elements stored in the instance of `vector`.
`vector`	`pair`	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.

Table 3: Converter functions and Methods defined on them

is a class, the methods to this generic function specialize on the class of that class, which is the metaclass of the instance. Thus, the metaclass is responsible for the physical allocation of its metainstances. If the argument is a non-allocatable object, an error is signaled (condition: `non-allocatable-object`).

`(initialize-instance` *obj init-list*`)` → *obj* **level-0 generic**

The first argument is a newly allocated but uninitialized object; the second is a list of alternating symbols and values of information to be used to initialize the slots of the instance. The object itself is returned. Since the first argument is an instance, the generic function specialize on the class of that instance. Thus, the class is responsible for the initialization of its instances.

`(initialize-instance class` *init-list*`)` → *class* **level-0 `initialize-instance` method**

The method for `initialize-instance` for the root class `class` initializes a new class. The following initargs are accepted for this method:

name: The value must be a symbol. This initarg is only used for documentary purposes.

direct-superclasses: The value must be a list of one element, which is a class. This specifies the direct superclasses of the new class.

direct-slot-descriptions: The value must be a list of textual slot descriptions. Each textual slot description is a list of alternating keywords and values. The accepted keywords in a textual slot description are:

  name: The name of the new slot, by which it will be known to `find-slot` and to `slot-value`.

  initfunction: A function of no arguments which will yield a default value for the new slot when applied.

  initarg: A symbol specifying the initargs which can be used with *make-instance* to initialize the value of the slot by the user.

The method also takes into account any initargs specified by the class of the new class, so the method may be inherited by new metaclasses if they do not do any unusual processing of the init-list.

(`make-reader` *class slot*) → *function* **level-0 generic**
This generic function is used to create the reader functions for slots which request them. It is responsible for creating a function of one argument, an instance of the *class*, which returns the value of the slot. The function returned can be a generic function, a non-generic function, depending on the class of *class*. For instances of `class`, a generic function is created. For instances of `structure-class` a normal function is created. It is not necessary that the accessor function call `slot-value` or any of the functions in the slot access protocol if the information can be obtained in another way.

(`make-writer` *class slot*) → *function* **level-0 generic**
This generic function is used to create the writer functions for slots which request them. It is responsible for creating a function of two arguments, the first being an instance of the *class* and the second a new value for the slot, which stores the new value in the slot. The function returned may be a generic function, a non-generic function, depending on the class of *class*. For instances of `class`, a generic function is created. For instances of `structure-class` a normal function is created. It is not necessary that the returned function call `slot-value` or any of the functions in the slot access protocol if the information can be obtained in another way.

(`make-constructor` *class initarg-list*) → *function* **level-0 generic**
This generic function is used to create a constructor function for a class. The constructor function takes the same number of arguments as the initarg-list. The constructor function must return a newly allocated and initialized instance of *class*. Each element in the *initarg-list*, which must be a legal initarg for some slot of the class, specifies that the corresponding argument of the constructor function will be used to initialize the slots which specify that element as an initarg.

(`make-predicate` *class*) → *function* **level-0 generic**
This generic function is used to create a predicate function for a class. It returns a function which will return `()` if its argument of any other class than *class* and its argument otherwise.

(`defreader` *name class-name slot-name*) **level-0 macro**
This macro creates a reader function bound to *name*, a symbol, which given an instance of the class *class-name*, returns the value of the slot named *slot-name*.

(`defwriter` *name class-name slot-name*) **level-0 macro**
This macro creates a writer function bound to *name*, a symbol, which given an instance of the class *class-name*, returns the value of the slot named *slot-name*.

(`defaccessor` *name class-name slot-name*) **level-0 macro**
This macro creates a accessor function bound to *name*, a symbol, which given an instance of the class *class-name*, returns the value of the slot named *slot-name*.

`(defconstructor` *name class-name initarg-list*`)` **level-0 macro**
This macro creates a constructor function for the class *class-name* bound to *name*. The *initarg-list* is a list of initargs accepted by the class. The new function will have a lambda list congruent to this list; the new instance will be set up as though `make-instance` were called with the specified initargs and given values.

`(defpredicate` *name class-name*`)` **level-0 macro**
This macro creates a predicate function for the class *class-name* bound to *name*.

`(class-precedence-list` *class*`)` → *list* **level-0 generic**
This generic function returns the class precedence list of the given class. The class precedence list is a linearized list of all the class's superclasses, direct and remote, beginning with the class itself. This list is used to determine the specificity of slot and method inheritance. The rules for determining this list are defined by the class of the class, and should be implemented for new metaclasses by writing a new method for `compute-class-precedence-list` instead of this accessor.

`(class-direct-superclasses` *class*`)` → *list* **level-0 generic**
This generic function returns a list of the direct superclasses of the given class. This list will be of length one for instances of the primitive metaclasses and zero for the class `object`.

`(class-direct-subclasses` *class*`)` → *list* **level-0 generic**
Given a class, this generic function returns a list of the direct subclasses of the class—that is, all the classes which specified the argument as a direct superclass.

`(class-constructors` *class*`)` → *list(function)* **level-0 generic**
This function will return a list of functions taking an arbitrary number of arguments. Each of these functions can be used to create a new instance of the class, and takes as arguments a function-specific set of initialization values. These functions are specified by the `constructor` `defclass` or `defstruct` options and created by `make-constructor`.

`maximum-slot-count`:*integer* **level-0 constant**
This is a processor-defined constant. A conforming processor must support a maximum number of slots per object of at least 32767.

4.1.6 Numbers

The naming conventions described in section 1.8 are applied in the following definitions.

`(numberp` *obj*`)` → *boolean* **level-0 predicate**
If the class of *obj* is a subclass of `number` the result is *obj*, otherwise `()`.

`(make-number` *string*`)` → *number* **level-0 constructor**
The characters comprising *string* are tokenised and if the resulting lexeme is classified as a number, the internal representation of that number is constructed and returned as the result of `make-number`.

`(+` z_1 z_2 ...`)` → z **level-0 function**
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.

`(-` z_1 `[`z_2 ...`])` → z **level-0 function**
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 that argument. The arguments are combined left-associatively.

`(*` z_1 z_2 ...`)` → z **level-0 function**
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.

`(/` z_1 `[`z_2 ...`])` → z **level-0 function**
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.

`(<` x_1 x_2 ...`)` → *boolean* **level-0 function**
Determines whether the sequence of numbers x_1 up to x_n is strictly increasing according to the generic function `binary-lt`.

`(>` x_1 x_2 ...`)` → *boolean* **level-0 function**
Determines whether the sequence of numbers x_1 up to x_n is strictly decreasing, according to the generic function `binary-gt`.

`(<=` x_1 x_2 ...`)` → *boolean* **level-0 function**
Determines whether the sequence of numbers x_1 up to x_n is increasing, according to the generic function `binary-le`.

`(>=` x_1 x_2 ...`)` → *boolean* **level-0 function**
Determines whether the sequence of numbers x_1 up to x_n is decreasing, according to the generic function `binary-ge`.

`(max` x_1 `[`x_2 ...`])` → x **level-0 function**
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.

`(min` x_1 `[`x_2 ...`])` → x **level-0 function**
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.

`(gcd` z_1 `[`z_2 ...`])` → z **level-0 generic**
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.

`(lcm` q_1 `[`q_2 ...`])` → q **level-0 generic**
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.

`(abs` z`)` → z **level-0 generic**
Compute the absolute value of z.

`(zerop` x`)` → *boolean* **level-0 generic**
Compares z with the zero element of the class of z using the generic function `=`.

`(sign` x`)` → x **level-0 generic**
Returns the result of converting ±1 to the class of x with the sign of the x; 0 is considered positive.

`(positivep` x`)` → *boolean* **level-0 generic**
Compares x against the zero element of the class of x using the generic function `binary-gt`.

`(negativep` x`)` → *boolean* **level-0 generic**
Compares x against the zero element of the class of x using the generic function `binary-lt`.

`(binary-plus` z_1 z_2`)` → z **level-0 generic**
Compute the sum of z_1 and z_2.

`(binary-difference` z_1 z_2`)` → z **level-0 generic**
Compute the difference of z_1 and z_2.

`(negate` z`)` $\rightarrow z$ **level-0 generic**
Compute the additive inverse of x.

`(binary-times` z_1 z_2`)` $\rightarrow z$ **level-0 generic**
Compute the product of z_1 and z_2.

`(binary-divide` z_1 z_2`)` $\rightarrow z$ **level-0 generic**
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`).

`(binary-lt` x_1 x_2`)` $\rightarrow$ *boolean* **level-0 generic**
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.

`(binary-gt` x_1 x_2`)` $\rightarrow$ *boolean* **level-0 generic**
Compare r_1 with r_2 returning `t` if r_1 succeeds r_2 according to the ordering method of the class of higher class of r_1 and r_2.

`(binary-max` x_1 x_2`)` $\rightarrow x$ **level-0 generic**
Compare x_1 and x_2 and return whichever is the greater according to the ordering method of the class of the higher class of x_1 and x_2.

`(binary-min` x_1 x_2`)` $\rightarrow x$ **level-0 generic**
Compare x_1 and x_2 and return whichever is the greater according to the ordering method of the class of the higher class of x_1 and x_2.

`(binary-gcd` q_1 q_2`)` $\rightarrow q$ **level-0 generic**
Compute the greatest common divisor of q_1 and q_2.

`(binary-lcm` q_1 q_2`)` $\rightarrow q$ **level-0 generic**
Compute the lowest common multiple of q_1 and q_2.

4.1.7 Coercion

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

Table 4: Converter methods for level-0 numbers

4.1.8 Single Precision Integer Arithmetic

`(binary-plus` i_1 i_2`)` $\rightarrow i$ **level-0 `binary-plus` method**
`(binary-difference` i_1 i_2`)` $\rightarrow i$ **level-0 `binary-difference` method**
`(negate` i`)` $\rightarrow i$ **level-0 `negate` method**
`(binary-times` i_1 i_2`)` $\rightarrow i$ **level-0 `binary-times` method**
`(binary-lt` i_1 i_2`)` $\rightarrow i$ **level-0 `binary-lt` method**
`(binary-gt` i_1 i_2`)` $\rightarrow i$ **level-0 `binary-gt` method**

Target Class	Source Class	Description
`single-precision-integer`	`double-float`	Returns an instance of `single-precision-integer` whose value is closest to that of the 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
	`single-float`	Returns an instance of `single-precision-integer` whose value is closest to that of the 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.
	`ratio`	Returns an instance of `ratio` whose numerator is the same (under `=`) as the source and whose denominator is one.
`single-float`	`single-precision-integer`	Returns an instance of `double-float` whose value is the floating point approximation to the single precision integer source.
	`double-float`	Returns an instance of `ratio` whose numerator is the same (under `=`) as the source and whose denominator is one.
	`integer`	Returns an instance of `integer` whose value is closest to that of the source. If the source is small enough to be represented as an instance of `single-precision-integer`, this operation is the same as `round` without specifying the second argument. Otherwise this operation has the same specification as `round` (with one argument), but returns a `variable-precision-integer`.
	`ratio`	Returns an instance of `ratio` whose value is the rational approximtion to the single precision floating point source.
`double-float`	`integer`	Returns an instance of `integer` whose value is closest to that of the source. If the source is small enough to be represented as an instance of `single-precision-integer`, this operation is the same as `round` without specifying the second argument. Otherwise this operation has the same specification as `round` (with one argument), but returns a `variable-precision-integer`.
	`ratio`	Returns an instance of `ratio` whose value is the rational approximtion to the double precision floating point source.

Table 5: Converter methods for level-1 numbers

`(binary-max` i_1 i_2`)` → i **level-0 `binary-max` method**

`(binary-min` i_1 i_2`)` → i **level-0 `binary-min` method**

`(binary-gcd` i_1 i_2`)` → i **level-0 `binary-gcd` method**

`(binary-lcm` i_1 i_2`)` → i **level-0 `binary-lcm` method**

`(abs` i`)` → i **level-0 `abs` method**

`(zerop` i`)` → i **level-0 `zerop` method**

`(sign` i`)` → i **level-0 `sign` method**

`(positivep` i`)` → i **level-0 `positivep` method**

`(negativep` i`)` → i **level-0 `negativep` method**

Arithmetic operations for `variable-precision-integer` are defined by methods to be attached to the generic functions mentioned above. The non class-specific definitions of these operations is given in section 4.1.6.

`((converter` *string*`)` i`)` → i **level-0 method**

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

`(generic-prin` *i stream*`)` $\rightarrow$ *i* **level-0 `generic-prin` method**
`(generic-write` *i stream*`)` $\rightarrow$ *i* **level-0 `generic-write` method**
Output external representation of *i* on *stream* as described in section 2.9 and defined in appendix A.4.

`most-positive-single-precision-integer`:*i* **level-0 constant**
This is an implementation-defined constant. A conforming processor must support a value greater than or equal to 32767 and the value of `maximum-vector-index`.

`most-negative-single-precision-integer`:*i* **level-0 constant**
This is an implementation-defined constant. A conforming processor must support a value less than -32768.

`(single-precision-integer-p` *obj*`)` $\rightarrow$ *boolean* **level-0 predicate**
Returns *obj* if *obj* is an instance of `single-precision-integer`.

`(oddp` *number*`)` $\rightarrow$ *number* **level-0 generic**
`(oddp` *i*`)` $\rightarrow$ *i* **level-0 `oddp` method**
Returns `t` if the remainder from dividing *i* by two is non-zero, otherwise `()`.

`(evenp` *number*`)` $\rightarrow$ *number* **level-0 generic**
`(evenp` *i*`)` $\rightarrow$ *i* **level-0 `evenp` method**
Returns `t` if the remainder from dividing *i* by two is zero, otherwise `()`.

`(quotient` i_1 i_2`)` $\rightarrow i$ **level-0 generic**
`(quotient` i_1 i_2`)` $\rightarrow i$ **level-0 `quotient` method**
`(remainder` i_1 i_2`)` $\rightarrow i$ **level-0 generic**
`(remainder` i_1 i_2`)` $\rightarrow i$ **level-0 `remainder` method**
`(modulo` i_1 i_2`)` $\rightarrow i$ **level-0 generic**
`(modulo` i_1 i_2`)` $\rightarrow i$ **level-0 `modulo` method**
The definition of the behaviour of these three operations is so closely linked that they are treated together here. The arguments are related by the equation: $i_1 = i_2 \times q + r$, where r lies between integer zero (inclusive) and the integer $i_2 \times sign(i_1)$ (exclusive). Additionally, the modulus, m, is constrained by $0 \leq m < |q|$. The following three tables define sign combination for `quotient`, `remainder` and `modulus` operations:

	`quotient`		`remainder`		`modulus`	
	i_2	$-i_2$	i_2	$-i_2$	i_2	$-i_2$
i_1	q	$-q$	r	$-r$	r	`remainder(`i_2`+remainder(`i_1`,`i_2`),` i_2`)`
$-i_1$	$-q$	q	$-r$	r	`remainder(`i_2`+remainder(`i_1`,`i_2`),` i_2`)`	r

4.1.9 Double Precision Floating Point Arithmetic

`(binary-plus` x_1 x_2`)` $\rightarrow x$ **level-0 `binary-plus` method**
`(binary-difference` x_1 x_2`)` $\rightarrow x$ **level-0 `binary-difference` method**
`(negate` x`)` $\rightarrow x$ **level-0 `negate` method**
`(binary-times` x_1 x_2`)` $\rightarrow x$ **level-0 `binary-times` method**
`(binary-divide` x_1 x_2`)` $\rightarrow x$ **level-0 `binary-divide` method**
`(binary-lt` x_1 x_2`)` $\rightarrow x$ **level-0 `binary-lt` method**
`(binary-gt` x_1 x_2`)` $\rightarrow x$ **level-0 `binary-gt` method**
`(binary-max` x_1 x_2`)` $\rightarrow x$ **level-0 `binary-max` method**
`(binary-min` x_1 x_2`)` $\rightarrow x$ **level-0 `binary-min` method**
`(abs` x`)` $\rightarrow x$ **level-0 `abs` method**

`(zerop` *x* `)` → *x* **level-0 `zerop` method**
`(sign` *x* `)` → *x* **level-0 `sign` method**
`(positivep` *x* `)` → *x* **level-0 `positivep` method**
`(negativep` *x* `)` → *x* **level-0 `negativep` method**

Arithmetic operations for `double-float` are defined by methods to be attached to the generic functions metnioned above. The non class-specific definitions of these operations is given in section 4.1.6. The non class-specific definitions of these operations is given in section 4.1.6. The other definitions in this section are derived from ISO/IEC CD 10967: 1991 (Language compatible arithmetic).

`((converter` *string*`)` *x* `)` → *r* **level-0 method**

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

`(generic-prin` *x stream* `)` → *x* **level-0 `generic-prin` method**
`(generic-write` *x stream* `)` → *x* **level-0 `generic-write` method**

Output external representation of *x* on *stream* as described in section 2.9 and defined in appendix A.4.

`most-positive-double-float`:*x* **level-0 constant**
`least-positive-double-float`:*x* **level-0 constant**
`least-negative-double-float`:*x* **level-0 constant**
`most-negative-double-float`:*x* **level-0 constant**

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.

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)`.

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)`.

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.

`(floatp` *obj* `)` → *boolean* **level-0 function**
`(double-float-p` *obj* `)` → *boolean* **level-0 function**

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

`(exponent` *x* `)` → *integer* **level-0 generic**
`(exponent` *x* `)` → *integer* **level-0 `exponent` method**

Returns the exponent of the argument *x* as an integer having unbiased the value if necessary. The exponent bias is an implementation-defined value.

`(fraction` *x* `)` → *float* **level-0 generic**
`(fraction` *x* `)` → *float* **level-0 `fraction` method**

Returns the result of scaling the argument *x*, such that the result is in the range $\pm[1/r, 1)$, where r is the radix of the floating point representation.

`(scale` *x i* `)` → *float* **level-0 generic**
`(scale` *x i* `)` → *float* **level-0 `scale` method**

Returns the result of scaling the argument *x* by the i^{th} power of radix of the floating point representation.

`(succ` *x* `)` → *float* **level-0 generic**
`(succ` *x* `)` → *float* **level-0 `succ` method**

Returns the closest element of the class `float` which is greater than the argument *x*.

`(pred` *x*`)` → *float* **level-0 generic**
`(pred` *x*`)` → *float* **level-0 `pred` method**
Returns the closest element of the class `float` which is less than the argument *x*.

`(unit-last-place` *x*`)` → *float* **level-0 generic**
`(unit-last-place` *x*`)` → *float* **level-0 `unit-last-place` method**
Returns the value of one unit in the last place, that is, its value is the weight of the least significant digit of a non-zero argument. If the argument is zero, an error is signaled (condition: `zero-argument-in-ulp`).

`(truncate` *x* `[`*precision*`])` → *number* **level-0 generic**
`(truncate` *x* `[`*precision*`])` → *number* **level-0 `truncate` method**
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.

`(round` *x* `[`*precision*`])` → *number* **level-0 generic**
`(round` *x* `[`*precision*`])` → *number* **level-0 `round` method**
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.

`(intpart` *x*`)` → *float* **level-0 generic**
`(intpart` *x*`)` → *float* **level-0 `intpart` method**
Returns the integer part of the argument *x* as a floating point number.

`(fracpart` *x*`)` → *float* **level-0 generic**
`(fracpart` *x*`)` → *float* **level-0 `fracpart` method**
Returns the value the argument *x* minus its integer part.

`(floor` *x*`)` → *i* **level-0 generic**
`(floor` *x*`)` → *i* **level-0 `floor` method**
Computes the greatest integer value which is less than or equal to *x*.

`(ceiling` *x*`)` → *i* **level-0 generic**
`(ceiling` *x*`)` → *i* **level-0 `ceiling` method**
Computes the least integer value that is greater than or equal to *x*.

4.1.10 Characters

`(characterp` *obj*`)` → *boolean* **level-0 predicate**
Returns *obj* if *obj* is an instance of a subclass `character`, otherwise `()`.

`((converter` *character*`)` *integer*`)` → *integer* **level-0 method**
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.

`((converter` *integer*`)` *character*`)` → *character* **level-0 method**
Returns an instance of `single-precision-integer` which corresponds to the position of the instance of `character` in the default character set.

`(generic-prin` *character stream*`)` → *character* **level-0 `generic-prin` method**
`(generic-write` *character stream*`)` → *character* **level-0 `generic-write` method**
Output external representation of *character* on *stream* as described in section 2.9 and defined in appendix A.4.

4.1.11 Strings

`(stringp` *obj*`)` → *boolean* **level-0 predicate**
Returns *obj* if *obj* is an instance of a subclass `string`, otherwise `()`.

`(make-string` *n* `[`*character*`])` → *string* **level-0 constructor**
Allocate and return a string of *n* characters initialised to *character*, if supplied, or to `#\x0`, by default.

`(string-ref` *string n*`)` → *character* **level-0 function**
`((setter string-ref)` *string n character*`)` → *character* **level-0 function**
Access and update elements of a string. It is an error if *n* is outside the range zero to the length of the string.

`(generic-prin` *string stream*`)` → *string* **level-0 `generic-prin` method**
`(generic-write` *string stream*`)` → *string* **level-0 `generic-write` method**
Output external representation of *string* on *stream* as described in section 2.9 and defined in appendix A.4.

`((converter` *pair*`)` *string*`)` → *string* **level-0 method**
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`.

`(length` *string*`)` → *n* **level-0 `length` method**
Returns the number of characters comprising *string*.

`(string-lt` $string_1 string_2$`[`*character-set*`])` → *boolean* **level-0 function**
If the sequence of characters in $string_1$ is alphabetically less than that in $string_2$ returns `t`, else `()`.

`(string-gt` $string_1$ $string_2$`[`*character-set*`])` → *boolean* **level-0 function**
If the sequence of characters in $string_1$ is alphabetically greater than that in $string_2$ returns `t`, else `()`.

`(string-slice` *string start end*`)` → *string* **level-0 function**
Returns a newly allocated string containing the characters of *string* starting at *start* up to *end*.

`(string-append` $string_1$ $string_2$`)` → *string* **level-0 function**
Returns a newly allocated string containing the characters of $string_1$ followed by the characters of $string_2$.

4.1.12 Pairs and Lists

The class `pair` (also known as a *dotted pair*) is a 2-tuple, whose fields are called, for historical reasons, car and cdr. Pairs are created by the function `cons` and the fields 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.

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

`(consp` *obj*`)` → *boolean* **level-0 predicate**
Returns *obj* if *obj* is a subclass of `pair`, otherwise `()`.

`(atom` *obj*`)` → *boolean* **level-0 predicate**
If *obj* is not an instance `pair`, *obj* is returned, otherwise `()`.

`(cons` obj_1 obj_2`)` → *pair* **level-0 constructor**
Allocates a new pair initialized to obj_1 and obj_2.

`(car` *pair*`)` → *obj* **level-0 function**
`(cdr` *pair*`)` → *obj* **level-0 function**
`((setter car)` *pair obj*`)` → *obj* **level-0 function**
`((setter cdr)` *pair obj*`)` → *obj* **level-0 function**
Functions to access and to update the fields of objects which are instances of subclasses of `pair`.

`(generic-prin` *pair stream*`)` → *pair* **level-0 `generic-prin` method**
`(generic-write` *pair stream*`)` → *pair* **level-0 `generic-write` method**
Output the external representation of *pair* on *stream* as described in section 2.9 and defined in appendix A.4.

`(list [`obj_1 ... obj_n`])` → *list(obj)* **level-0 function**
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.

`(length` *list*`)` → *integer* **level-0 `length` method**
Returns the count of the number of top-level pairs in list.

4.1.13 Functions and Methods

The class `function` is the class of all ordinary functions. There are two subclasses of `function`: `generic--function` and `continuation`. Methods are not functions. They are objects containing functions that form a part of a generic function. Instances of `functions` are constructed by `lambda`, instances of `continuations` by `let/cc` and instances of `generic-functions` by `generic-lambda`. The external representation of functions and methods is processor defined.

`(continuationp` *obj*`)` → *boolean* **level-0 predicate**
Returns *obj* if the class of *obj* is a subclass of `continuation`.

`(functionp` *obj*`)` → *boolean* **level-0 predicate**
Returns *obj* if the class of *obj* is a subclass of `function`.

`(generic-function-p` *obj*`)` → *boolean* **level-0 predicate**
Returns *obj* if the class of *obj* is a subclass of `generic-function`.

`(methodp` *obj*`)` → *boolean* **level-0 predicate**
Returns *obj* if the class of *obj* is a subclass of `method`.

`(function-lambda-list` *function*`)` → *list* **level-0 generic**
Returns a lambda-list congruent to that specified when *function* was defined.

`(generic-function-methods` *generic-function*`)` → *list* **level-0 generic**
Returns a list of the methods attached to *generic function*.

`(generic-function-method-class` *generic-function*`)` → *class* **level-0 generic**
Returns the class of which all the methods of *generic function* are instances.

`(initialize-instance generic-function` *init-list*`)` → *generic-function*
level-0 `initialize-instance` method
The method for initializing instances of `generic-function` accepts the following initargs:

name: For documentary purposes, a name may be supplied at the creation of a generic function. If none is supplied, none will be stored.

lambda-list: The lambda list must be supplied. The syntax is as specified in section 3.1.12. An error is signaled (condition: `non-congruent-lambda-lists`) if any method defined on this generic function does not have a lambda lists isomorphic to that of the generic function. In addition, an error is signaled (condition: `incompatible-method-signature`) if the method's specialized lambda list widens the domain 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`.

method-class: A subclass of the class `method`. All methods added to the generic function must be of this class.

method: 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 signature, 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.

Any initargs specified for slots of new generic function classes will also be taken into account, so authors of new generic function classes which do not do more elaborate processing at initialization time need not write new methods for `initialize-instance`.

`(initialize-instance method` *init-list*`)` → *method* **level-0 `initialize-instance` method**
The method for initializing instances of `method` accepts the following initargs:

function: The value is a function which implements the method. This initarg is mandatory. The function parameter list must be congruent to any generic function to which the method is attached.

signature: The value is a list of names of existing classes or class objects which specify the signature of the new method — *eg*, the set of classes for which the new method is the most specific. This initarg is mandatory.

Any initargs specified for slots in subclasses of `method` will also be handled by the default method.

`(defmethod` *gf-name spec-lambda-list form**`)` **level-0 macro**
`(defmethod (converter` *class*`)` *form**`)` **level-0 macro**
This macro is used for defining new methods on generic functions. The syntactic elements of the form are all defined under `defgeneric` in section 3.1.12. A new method object is defined with the specified body and with the signature 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-signature`) if the method's specialized lambda list widens the domain of the generic function.

`(method-signature` *method*`)` → *list* **level-0 generic**
Given a method, this returns the list of the most general classes for which the method is applicable.

`(method-generic-function` *method*`)` → *boolean* **level-0 generic**
Given a method, this generic function returns the generic function of which the method is a part, or `()` if the method does not belong to any generic function.

`(method-function` *method*`)` → *function* **level-0 generic**
The function returned by the *basic-method* method of `method-function` has a lambda list congruent to that of the given method and functionality equivalent to the use of `call-method` on the given method with the same arguments.

4.1.14 Streams

Streams are created by the functions `open` and `make-io-stream`. In the following discussion, *path* is used to refer to both file names and to device names, although, on occasion, both file and device are used explicitly.

`(open` *init-option**`)` → *stream* **level-0 function**
The `open` function causes a stream to be created linked to a path characterized by the specified *init-option*s. In order that different representations of pathnames can be supported, the open operation is implemented by a generic function `generic-open`. Thus, `open` extracts the path *init-option* and call `generic-open` with the value of that option and the *init-option*s passed to `open`.

path: The value of this option specifies a pathname.

(`generic-open` *path init-option**) → *stream* **level-0 generic**

The generic function `generic-open` creates a stream linked to *path*, characterized by the specified *init-options*. The *init-options* are keywords (described below) and, where appropriate, initial values, in the same style as the arguments to `make-instance`. Descriptions of the *init-options* follow here: the first level list describes the direction options, and the second level the mode options for each direction.

input: This option does not take a value, but its presence causes the *path* to be opened for input. The three direction options are mutually exclusive. If no direction option is given, this is the default.

output: This option does not take a value, but its presence causes the *path* to be opened for output. The three direction options are mutually exclusive.

io: This option does not take a value, but its presence causes the *path* to be operned for both input and output.

character: This option does not take a value, but its presence causes the *path* to be opened for character input. The two mode options `character` and `binary` are mutually exclusive. If neither is given, this is the default.

binary: This option does not take a value, but its presence causes the *path* to be opened for binary input or output.

byte-size: The value of this option is an integer which specified the size (in bits) of the binary items to be input or output. This option is only used if `binary` is specified too.

create: This option does not take a value, but its presence causes the *path* to be created if it does not exist. If no output option is specified, `create` is the default. If only `create` is specified, an error is signaled if the file already exists (condition: `file-already-exists`).

append: This option does not take a value, but its presence causes any output to be appended to the *path*, if it exists, or to create it if it does not. `append`, `overwrite` and `new-version` are mutually exclusive. If `append` is specified but `create` is not, an error is signaled if the file does not exist (condition `path-does-not-exist`).

overwrite: This option does not take a value but its presence causes the file to be overwritten, if it exits. If it is opened for output only, it will be truncated to zero length, but if it is opened for input and output it will not be truncated. `overwrite`, `append` and `new-version` are mutually exclusive. If `overwrite` is specified but `create` is not, an error is signaled if the file does not exist (condition `path-does-not-exist`).

new-version: This option does not take a value, but its presence causes a new version of the *path* to be created if it already exists, according to some implementation-defined means. If `create` is one of the output options by default—but not by specification—`new-version` is added to the option list, unless `new-version` is not supported, in which case it is supplanted by `overwrite`. `new-version`, `append` and `overwrite` are mutually exclusive. If `new-version` is specified but `create` is not, an error is signaled if the file does not exist (condition `path-does-not-exist`).

If *path* cannot be opened, an error is signaled (condition: `cannot-open-path`). If a combination of options, which contains mutually exclusive options, is given, an error is signaled (condition: `inconsistent-open--options`). In most implementations, not all options will make sense for all *path*s.

(`generic-open string` *init-option**) → *stream* **level-0** `generic-open` **method**

At level-0, the pathname is a string. The interpretation of *string* is implementation-defined. The *init-options* is a list of directives concerning the opening of the stream and the kind of stream according to the options specified above.

`(make-io-stream` *input-stream output-stream*`)` → *io-stream* **level-0 function**
input-stream must be open for input, *output-stream* must be open for output and their modes must be compatible. The last requirement means that they must either both be `character` streams or both `binary` streams with the same `byte-size`. Unless all of these conditions hold, an error is signaled (condition: `incompatible-streams`). A new *io-stream* is created that reads from *input-stream* and writes to *output-stream*.

`(standard-input-stream )` → *input-stream* **level-0 function**
`(standard-output-stream )` → *output-stream* **level-0 function**
`(standard-error-stream )` → *output-stream* **level-0 function**
`(trace-output-stream )` → *output-stream* **level-0 function**
`(debug-io-stream )` → *io-stream* **level-0 function**
`((setter standard-input-stream)` *input-stream*`)` → *input-stream* **level-0 function**
`((setter standard-output-stream)` *output-stream*`)` → *output-stream* **level-0 function**
`((setter standard-error-stream)` *output-stream*`)` → *output-stream* **level-0 function**
`((setter trace-output)` *output-stream*`)` → *output-stream* **level-0 function**
`((setter debug-io)` *io-stream*`)` → *io-stream* **level-0 function**
The above are the basic streams provided by the system. Each of the above accessors returns the current value of the standard input stream, standard output stream, standard error stream, trace output stream and the debug-io stream. The corresponding `setter` function is used to change the specified stream to *stream*. An error is signaled if *stream* is not of the correct class (condition: `not-an-input-stream`, `not-an-output-stream`, `not-an-io-stream`, `not-a-character-stream` or `not-a-binary-stream`).

`(streamp` *obj*`)` → *boolean* **level-0 predicate**
Returns *obj* if *obj* is an instance of a subclass of `stream`.

`(open-stream-p` *obj*`)` → *boolean* **level-0 predicate**
`(input-stream-p` *obj*`)` → *boolean* **level-0 predicate**
`(output-stream-p` *obj*`)` → *boolean* **level-0 predicate**
These functions return `t` if *obj* is a *stream* which is open, open for input, or open for output, respectively.

`(close` *stream*`)` → *boolean* **level-0 generic**
`(close input-stream)` → *boolean* **level-0** `close` **method**
`(close ouptut-stream)` → *boolean* **level-0** `close` **method**
`(close io-stream)` → *boolean* **level-0** `close` **method**
The *stream* is closed, and any buffered output is flushed to the device connected to the stream. Any attempt to read from or write to *stream* signals an error to be signaled (condition: `stream-not-open`). If *stream* was open, *stream* is returned as the value of `close`. If *stream* was already closed, `close` returns `()`.

`(flush` *stream*`)` → *null* **level-0 generic**
`(flush ouptut-stream)` → *null* **level-0** `flush` **method**
`(flush io-stream)` → *null* **level-0** `flush` **method**
Any buffered output to the stream is flushed to the device connected to the stream. `flush` returns `()`. If *stream* is not open for output the condition `not-an-output-stream` is signaled.

`(stream-ready-p` *stream*`)` → *boolean* **level-0 generic**
`(stream-ready-p input-stream)` → *boolean* **level-0** `stream-ready-p` **method**
`(stream-ready-p output-stream)` → *boolean* **level-0** `stream-ready-p` **method**
`(stream-ready-p io-stream)` → *boolean* **level-0** `stream-ready-p` **method**
Returns `t` if *stream* has input available or is ready to receive ouptut.

`end-of-stream`:*eos-object* **level-0 constant**

The value of `end-of-stream` is the only instance of the class `eos-object`, which is the distinguished entity used to indicate that the stream is exhausted.

(`generic-read-char` *stream*) → {*character* | *eos-object*} **level-0 generic**
(`generic-read-byte` *stream*) → {*n* | *eos-object*} **level-0 generic**

These read and return the next character or integer from *stream*. If the end of *stream* has been reached, the end-of-stream object is returned. Either of these functions can signal the condition `not-an-input-stream` and the first can signal `not-a-character-stream` and the second can signal `not-a-binary-stream`.

(`generic-peek-char` *stream*) → {*character* | *eos-object*} **level-0 generic**
(`generic-peek-byte` *stream*) → {*n* | *eos-object*} **level-0 generic**

These return the next character or integer from *stream*, without removing it from *stream*. If the end of *stream* has been reached, the end-of-stream object is returned. Either of these functions can signal the condition `not-an-input-stream` and the first can signal `not-a-character-stream` and the second can signal `not-a-binary-stream`.

(`generic-write-char` *character stream*) → *character* **level-0 generic**
(`generic-write-byte` *n stream*) → *n* **level-0 generic**

These write the character or integer to *stream* and return their first argument. Either of these functions can signal the condition `not-an-output-stream` and the first can signal `not-a-character-stream` and the second can signal `not-a-binary-stream`.

(`stream-position` *stream*) → *n* **level-0 generic**
(`(setter stream-position)` *stream position*) → *n* **level-0 generic**

`stream-position` returns the current position of *stream*. The position is an integer which is zero at the start of the stream, and increases monotonically as *stream* is read or written. It is not guaranteed that reading or writing a single character will increase the position by one. The updator function accepts for *position* any value returned by `stream-position`, and in some implementations any integer between zero and the length of the stream is acceptable. The updator function also accepts the symbols `start` and `end`, which respectively position the stream at its start and end. An error is signaled if the position is not valid (condition: `invalid-stream-position`).

Support for stream positioning is optional, and it will not normally be available for all streams, for example, terminals. If *stream* is not positionable, an error is signaled (condition: `not-a-positionable-stream`).

(`stream-output-base` *stream*) → *n* **level-0 generic**
(`(setter stream-output-base)` *stream base*) → *n* **level-0 generic**

`stream-output-base` returns the base used when outputing objects which are a subclass of `integer` on *stream*. The updator function sets the output base of *stream* to *base*, which must be an integral value in the range [2 . . . 36] or else an error is signaled (condition: `invalid-output-base`). In either case, if *stream* is not an `output-stream` or an `io-stream` and error is signaled (condition: `not-an-output-stream`).

(`generic-prin` *obj stream*) → *obj* **level-0 generic**
(`generic-write` *obj stream*) → *obj* **level-0 generic**

Each of these functions returns *obj* as its result and has the side-effect of writing the external representation of the item on *stream*. As discussed in section 2.9, (`generic-`)`write` produces a representation which permits `read` to construct a copy which is `equal` to the original object, whilst (`generic-`)`prin` produces a representation which might not. Methods are defined at level-0 for the following classes:

`single-precision-integer`, `double-float`, `character`, `string`, `vector`, `pair`, `symbol`.

At level-1, the following classes are added:

`variable-precision-integer`, `single-float`, `ratio`

The default `write` and `prin` methods outputs an implementation-defined representation of any object.

`(read` [*stream*]`)` → {*obj* | *eos-object*} **level-0 function**
Reads and returns the next available Lisp expression—see external representations in section 2.9.

`(read-char` [*stream*]`)` → {*character* | *eos-object*} **level-0 function**
`(read-byte` [*stream*]`)` → {*n* | *eos-object*} **level-0 function**
`(peek-char` [*stream*]`)` → {*character* | *eos-object*} **level-0 function**
`(peek-byte` [*stream*]`)` → {*n* | *eos-object*} **level-0 function**
`(write-char` *character* [*stream*]`)` → *character* **level-0 function**
`(write-byte` *n* [*stream*]`)` → *n* **level-0 function**
The above functions call their generic counterparts. The input functions use the value returned by `standard-input-stream` and the output functions the value returned by `standard-output-stream` if *stream* is not specified.

`(prin` *obj* [*stream*]`)` → *obj* **level-0 function**
`(write` *obj* [*stream*]`)` → *obj* **level-0 function**
`(newline` [*stream*]`)` → *null* **level-0 function**
`(print` *obj* [*stream*]`)` → *obj* **level-0 function**
`prin` and `write` call their generic counterparts. `newline` is equivalent to calling the function `generic--prin-char` with the arguments `#\newline` and *stream*. `print` is equivalent to `newline`, followed by `prin`.

`(generic-prin` $stream_1$ $stream_2$`)` → *stream* **level-0 `generic-prin` method**
`(generic-write` $stream_1$ $stream_2$`)` → *stream* **level-0 `generic-write` method**

Outputs $stream_1$ on $stream_2$. The external representation of a stream is processor defined and, probably, cannot be re-read using `read`.

`(end-of-stream-p` *obj*`)` → *boolean* **level-0 predicate**
This predicate returns `t` if *obj* is the instance of `eos-object`, and otherwise `()`. The end-of-stream object is returned by any of the `read` functions when applied to a stream which cannot provide any more input.

4.1.15 Symbols

The following operations are defined for the creation and manipulation of symbols.

`(symbolp` *obj*`)` → *boolean* **level-0 predicate**
Returns *obj* if *obj* is an instance of a subclass of `symbol`.

`(make-symbol` *string*`)` → *symbol* **level-0 function**
If a symbol with the name *string* does not already exist, a new symbol is allocated and its name is initialized to *string*. This new symbol is returned. Otherwise the existing symbol is returned.

`(gensym` [*string*]`)` → *symbol* **level-0 function**
Makes a new symbol with a name generated by a processor-defined mechanism. Optionally, a prefix string for this name may be specified.

`(symbol-name` *obj*`)` → *string* **level-0 function**
Returns a copy of the *string* which is `equal` to that given as the argument to the call to `make-symbol` which created *symbol*.

`(symbol-exists-p` *string*`)` → *boolean* **level-0 function**
Returns the symbol whose name is *string* if that symbol has already been constructed by `make-symbol`. Otherwise, returns `()`.

`(generic-prin` *symbol stream*`)` → *symbol* **level-0 `generic-prin` method**
`(generic-write` *symbol stream*`)` → *symbol* **level-0 `generic-write` method**
Outputs the external representation of *symbol* on *stream* as described in section 2.9 and defined in appendix A.4.

4.1.16 Tables

Tables provide a general key to value association mechanism.

`(tablep` *obj*`)` → *boolean* **level-0 predicate**
Returns *obj* is *obj* is an instance of `table`.

`(make-table [`*comparator*`])` → *table(function)* **level-0 function**
Allocates a new empty table and returns it.

`(table-ref` *table key-obj* `[`*no-entry-value*`])` → *value-obj* **level-0 function**
If *key-obj* is a key in *table*, matched by the comparator function, then the associated value is returned. If there is no key *key-obj* in *table*, the value `()` is returned. However, if the optional parameter *no-entry-value* is provided and *key-obj* does not occur in *table*, the value *no-entry-value* is returned.

`((setter table-ref)` *table key-obj value-obj*`)` → *obj* **level-0 function**
If *key-obj* does not occur in *table* a new entry is made associating *key-obj* and *value-obj*. If *key-obj* does occur, then the association is changed to *value-obj*. *value-obj* is returned.

`(table-delete` *table key-obj*`)` → *table* **level-0 function**
If *key-obj* occurs in *table*, both the key and its associated value are deleted from the table. If *key-obj* does not occur in *table*, no action is taken.

`(generic-prin` *table stream*`)` → *table* **level-0 `generic-prin` method**
`(generic-write` *table stream*`)` → *table* **level-0 `generic-write` method**
Output the external representation of *table* on *stream*. The external representation of instances of `table` is implementation-defined.

`(map-table` *function table*`)` → *null* **level-0 function**
The function *function* is applied to each key and its associated value stored in *table*.

`(clear-table` *table*`)` → *table* **level-0 function**
All entries in *table* are deleted and the empty table is returned.

4.1.17 Threads and Semaphores

`(make-thread` *initial-function*`)` → *continuation* **level-0 constructor**
Create a new thread and install `initial-function` as its initial function. The size of the thread can be controlled by the optional second parameter which must be an `integer`. The initial state of the thread is `virgin`.

`(threadp` *obj*`)` → *boolean* **level-0 predicate**
Returns `obj` if *obj* is an instance of a subclass of `thread`.
`(thread-status` *thread*`)` → {`virgin` | `running` | `suspended` | `dead`} **level-0 function**
Returns a `symbol` indicating the status of *thread*.

`(generic-prin` *thread stream*`)` → *thread* **level-0 `generic-prin` method**
`(generic-write` *thread stream*`)` → *thread* **level-0 `generic-write` method**
Outputs the external representation of *thread* on *stream*. The external representation of *thread* is processor-defined.

`(thread-start` *continuation obj**`)` → *thread* **level-0 function**
Processing within the *thread*, to which *continuation* belongs, can now proceed. The values obj_1 to obj_n are passed to *thread*. On the thread which called `proceed`, execution continues and the `proceed` expression returns the value *thread*. The status of the thread containing *continuation* must be `suspended`.

`(thread-suspend )` → *null* **level-0 function**
Suspends the processing of the current thread. The system will select one of the currently active threads

for processing. The now suspended thread can only continue processing after a continuation within it is `proceed`ed. After `suspend`, the status of the current thread is `suspended`.

`(make-semaphore [`*i*`])` → *semaphore* **level-0 function**
Create a instance of the class `semaphore`, initialized to zero by default or to *i* if given.

`(generic-prin` *semaphore stream*`)` → *semaphore* **level-0 `generic-prin` method**
`(generic-write` *semaphore stream*`)` → *semaphore* **level-0 `generic-write` method**
Output the external representation of *semaphore* on *stream*. The external representation of *semaphore* is processor-defined.

`(reinitialize-semaphore` *semaphore*`)` → *semaphore* **level-0 function**
Reset *semaphore* to zero. The modified *semaphore* is returned.

`(semaphore-up` *semaphore*`)` → *semaphore* **level-0 function**
Increment the state of *semaphore*. The modified semaphore is returned. Any, some or all of the threads blocked on the semaphore may be resumed. The mechanism for choosing which of the threads succeeds in downing the semaphore is processor dependent.

`(semaphore-down` *semaphore*`)` → *semaphore* **level-0 function**
Decrement the state of *semaphore*. The modified semaphore is returned. If *semaphore* is in state zero, the downing thread is suspended.

4.1.18 Vectors

`(vectorp` *obj*`)` → *boolean* **level-0 predicate**
Returns *obj* if *obj* is an instance of `vector`.

`(make-vector` *n* `[`*obj*`])` → *vector* **level-0 function**
Returns a freshly allocated vector, whose maximum index is *n-1* and each of whose elements have been initialized to *obj*. Vectors are zero-based.

`(length` *vector*`)` → *n* **level-0 `length` method**
Returns the length of *vector*, which is the maximum index plus one.

`(vector-ref` *vector n*`)` → *obj* **level-0 function**
`((setter vector-ref)` *vector n obj*`)` → *obj* **level-0 function**
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.

`(make-initialized-vector` obj_1 obj_2 ... obj_n`)` → *vector(obj)* **level-0 function**
Allocated a vector of length *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*`)`. Returns the initialized vector.

`maximum-vector-index`:*integer* **level-0 constant**
This is an implementation-defined constant. A conforming processor must support a maximum vector index of at least 32767.

`(generic-prin` *vector stream*`)` → *vector* **level-0 `generic-prin` method**
`(generic-write` *vector stream*`)` → *vector* **level-0 `generic-write` method**
Output the external representation of *vector* on *stream* as described in section 2.9 and defined in appendix A.4.

4.2 Level-1 Classes

4.2.1 Character Sets

NOTE—No decisions have yet been made about how to handle international character sets.

4.2.2 Classes

`(initialize-instance class` *init-option** `)` → *class* **level-1** `initialize-instance` **method**
The method for `initialize-instance` for the root class `class` initializes a new class. At level-1 the following additional *init-options* are accepted for this method:

direct-superclasses: The value must be a list of one element, which is a class. This specifies the direct superclasses of the new class.

direct-slot-descriptions: The value must be a list of textual slot descriptions. Each textual slot description is a list of alternating keywords and values. The textual slot descriptions are combined with the inherited slot descriptions to produce a final list of slot descriptions for the new class. See *collect-slots* in section 4.2.2. At level-1 the following additional keyword is accepted in a textual slot description:

> **slot-class:** The value must be a slot description class. Specifies the class of the new slot description object.

`(find-slot-description` *class symbol*`)` → *slot-description* **level-1 generic**
Given a class and a slot name, this function will return the corresponding slot description object, if one exists. If not, an error is signaled (condition: `slot-missing`).

`(class-slot-descriptions` *class*`)` → *list* **level-1 generic**
This generic function returns a list of all slot description objects, inherited and direct, of the specified class.

`(class-direct-slot-descriptions` *class*`)` → *list* **level-1 generic**
This generic function returns a list of all slot description objects defined directly for the specified class. It does not return any inherited slots.

`(slot-value-using-slot-description` *slot-description obj*`)` → *obj* **level-1 generic**
This generic function is responsible for reading a slot value given an object and the appropriate slot description object. The default method retrieves the slot's position and calls `slot-value-using-class`.

`((setter slot-value-using-slot-description)` *slot-description* obj_1 obj_2`)` → *obj* **level-1 generic**
This generic function is responsible for setting a slot value given an object, the appropriate slot description object, and the new value. The default method retrieves the slot's position and calls the setter function of `slot-value-using-class`.

`(slot-value-using-class` *class obj n*`)` → *obj* **level-1 generic**
class is the class of *obj*. Methods on this function must know how to access *obj* to get the appropriate slot, whose position is the third argument. This function is called by `slot-value-using-slot-description`.

`((setter slot-value-using-class)` *class* obj_1 *n* obj_2`)` → *obj* **level-1 generic**
class is the class of *obj*. Methods on this function must know how to access obj to set the appropriate slot, whose position is the third argument. This function is called by the setter function of `slot-value-using--slot-description`).

`(compute-class-precedence-list` *class*`)` → *list(class)* **level-1 generic**
This generic function is called to compute an ordered list of superclasses of its first argument *class*, beginning with *class* and ending with *object*. Each element of the resulting list must be a class, and no elements may be repeated. Defining new metaclasses which introduce new inheritance strategies (*e.g.* multiple inheritance) may require writing new methods for this generic function. It is not specified when or how many times this generic function is called during the lifetime of a class.

`(compute-class-precedence-list` *class*`)` → *list(class)*
level-1 `compute-class-precedence-list` **method**
Instances of the root class `class` are in single inheritance. All of its instances have a single direct superclass,

except *object* which has none. This method can be considered to return a cons of its first argument and the result of a recursive application of itself to the first element of the argument's direct superclass list.

`(add-superclasses` *class* `(`*class**`))` → *class* **level-1 generic**
The first argument is to be a subclass of each of the classes in the list. Methods on this generic function must perform any necessary inheritance operations and bookkeeping. This generic function is called by the `initialize-instance` methods for the built-in metaclasses. Its built-in methods use the protocol below.

`(add-subclass` $class_1$ $class\ _2$`)` → *class* **level-1 generic**
Adds $class_2$ as a subclass of $class_1$. The built-in methods check metaclass compatibility and create the slot description objects for the new class. Returns $class_1$

`(metaclass-compatibility` $class_1$ $class_2$`)` → *obj* **level-1 generic**
$class_2$ is a potential subclass of $class_1$. This function must determine if the proposed inheritance is legal.

`(collect-slots` *class list*`)` → *class* **level-1 generic**
list is a list of property lists describing the directly defined slots of *class*. This generic function is responsible for determining the complete list of slot description objects based on this list and the slots in the superclasses of *class*. These new slot descriptions must be stored in the class. The built in methods call `make-slot-description` on each element of the list.

`(make-slot-description` *class list*`)` → *slot-description* **level-1 generic**
list is a property list of slot options. If `defclass` was used to define *class*, this will include as keys all slot options given to `defclass` and any given `slot-initargs`. In the built-in methods, if a slot with same name as the `name` property of *list* is defined on a superclass of *class*, the generic function `make-inherited-slot-description` is called with the class, old slot description, and the property list.

`(make-inherited-slot-description` *class slot-description list*`)` → *slot-description* **level-1 generic**
This generic function is called when a slot is requested for a class which has a slot of the same name defined by one of its superclasses. Generally, either uses the old slot description or creates a new slot description, perhaps reusing some information in the inherited slot, perhaps using only information in the given property list slot description description. For the kernel methods, it is an error to specify a slot for a class when a slot of the same name but a different slot description class exists in a superclass.

4.2.3 Functions and Methods

`(add-method` *generic-function method*`)` → *generic-function* **level-1 generic**
This generic function adds a method object to the given generic function. If the specialized lambda list of the method is not congruent with that of the generic function, an error is signaled (condition: `non-congruent-lambda-lists`). If the method is not an instance of the generic function's method-class, an error is signaled (condition: `bad-method-class`). If the method is already attached to some generic function an error is signaled (condition: `method-in-use`). If a method with the same signature as *method* is already attached to *generic-function*, the attached method is overwritten. The updated generic function is returned.

`(remove-method` *generic-function method*`)` → *generic-function* **level-1 generic**
This generic function removes the given method from the generic function. If the method is not one of the generic function's methods, nothing happens. The generic function is returned.

`(find-method` *generic-function signature*`)` → *boolean* **level-1 generic**
If a method with *signature* exists in the *generic-function*, then that method is returned. Otherwise, the result is `()`.

`(compute-discriminating-function` *generic-function*`)` → *function* **level-1 generic**
Constructs and returns a function of one argument, which takes a list of method specializers—either classes or some generic-function class-specific entity—with the same number of elements as the required arguments for *generic-function*, and returns a sorted list of applicable methods for the given set of specializers.

`compute-discriminating-function` is called at least once for an invocation of the generic function which is its argument for a given set of argument classes. Defining new generic function classes which use non-default discriminating algorithms involves writing new methods for this generic function.

`(compute-discriminating-function` *generic-function*`)` → *function*
level-1 `compute-discriminating-function` method
The default method implements the standard behavior for generic functions. The only supported specializers are classes. The list of methods returned is sorted from most to least specific. The ordered list of applicable methods for a set of classes can be considered to be determined by taking all the methods applicable for the first argument class, sorting them, and then eliminating those methods not applicable for further argument classes. The sort is two-keyed: first, by argument order, and second, for each argument class, according to the class precedence list of that class.

`(call-method` *method obj**`)` → *obj* **level-1 generic**
Given a method and a set of arguments, apply the method to the arguments. It is an error to apply a method to a set of arguments which does not match the method's signature. The applicable method list used in such a call includes all applicable methods for the given arguments which are less specific than the called method. If *method* is not attached to a generic function an error is signaled (condition: `orphan-method-call`).

4.2.4 Slot Descriptions

`(initialize-instance slot-description` *init-option**`)` → *slot-description*
level-1 `initialize-instance` method
This methods must call `call-next-mthod` before doing any other processing. The default method for slot description classes accepts the following *init-otion*s, as well as any specified for slots of subclasses of the class `slot-description`:

name: This mandatory initarg specifies the name by which slots described by the new slot description may be accessed using `slot-value`.

initfunction: The value of this mandatory initarg is a function of no arguments which when applied yields a default value for slots described by the slot description.

initarg: The value of this optional initarg is a symbol specifying the legal initarg which can be used to give an initial value for the slot in calls to `make-instance` of the class in which the slot description is stored.

`(slot-description-name` *slot-description*`)` → *symbol* **level-1 generic**
This function returns the name of the slot description.

`(slot-description-position` *slot-description*`)` → *n* **level-1 generic**
This function returns the position of the slot in an object.

`(slot-description-initfunction` *slot-description*`)` → *function* **level-1 generic**
This generic function returns a function of no arguments, which, when applied, yields the default value for the slot. This function is closed in the lexical environment of the class definition form—normally `defstruct` or `defclass`—and it is thus possible that this environment will be affected if the initform function is called. This function is called at most once for each instantiation of a class holding the slot description.

`(slot-value` *obj symbol*`)` → *obj* **level-1 function**
`slot-value` returns the object associated with the slot named *symbol* in *obj*. This request is divided into several parts based on the responsibilities of the objects involved. From *obj*'s class is found the appropriate slot description object. Then `slot-value-using-slot-description` is called with the original object and the slot description object. By default, this generic function finds the logical position of the slot in the slot

description object and passes this information, the object's class, and the object itself to the generic function `slot-value-using-class`, which is responsible for physically accessing this position in the object, since it discriminates off the object's metaclass.

If no slot with the given name is defined in the object's class, an error is signaled (condition: `slot-missing`). If the slot is unbound, an error is signalled (condition: `slot-unbound`).

`((setter slot-value)` *obj symbol value*`)` → *obj* **level-1 function**
This is the corresponding updator for `slot-value`. It stores *value* in the slot named *symbol* in the object *obj*. It uses a similar protocol to access the appropriate place in the object to store the new value: it calls the setter function of `slot-value-using-slot-description` and the setter function of `slot-value-using-class`. It returns the new value. If no slot with the given name is defined in the object's class, an error is signalled (condition: `slot-missing`).

4.2.5 Symbols

`(symbol-value` *symbol*`)` → *obj* **level-1 function**
`((setter symbol-value)` *symbol obj*`)` → *obj* **level-1 function**
If the first argument is not a subclass of `symbol` an error is signaled (condition: `not-a-symbol`), otherwise the top dynamic value of *symbol* is returned. The `setter` function updates the top dynamic value of *symbol* with the value of *obj*.

`(symbol-dynamic-value` *symbol*`)` → *obj* **level-1 function**
`((setter symbol-dynamic-value)` *symbol obj*`)` → *obj* **level-1 function**
If obj_1 is not a subclass of `symbol` an error is signaled (condition: `not-a-symbol`), otherwise the closest dynamic value of *symbol* is returned. The `setter` function updates the closest dynamic value of *symbol* with the value of *obj*.

`(symbol-props` *symbol*`)` → *list(obj)* **level-1 function**
If the first argument is not a subclass of `symbol` an error is signaled (condition: `not-a-symbol`), otherwise returns the property list of the symbol, which might be newly allocated, in which the alternate elements are property-name and property-value, starting with a property-name.

`(get` *symbol property-name* `[`*obj*`])` → *obj* **level-1 function**
If the first argument is not a subclass of `symbol` an error is signaled (condition: `not-a-symbol`), otherwise returns the property-value corresponding to *property-name* stored in the property list of *symbol*. If a property does not have an entry on the property list of *symbol*, `get` returns `()`. An optional second argument to `get` specifies a value to be returned in case of failure in order to distinguish between the property value `()` and the default failure value `()`.

`((setter get)` *symbol property-name property-value*`)` → *obj* **level-1 function**
If the first argument is not a subclass of `symbol` an error is signaled (condition: `not-a-symbol`), otherwise the setter function either updates the property-value corresponding to *property-name*, if *property-name* already occurs in the property list of *symbol*, or adds the association of *property-name* and *property-value* to the property list of *symbol*.

`(remprop` *symbol property-name*`)` → *obj* **level-1 function**
If the first argument is not a subclass of `symbol` an error is signaled (condition: `not-a-symbol`). If *property-name* occurs in the property list of *symbol* it is removed. The corresponding property-value is returned.

4.2.6 Single Precision Floating Point Arithmetic

`(binary-plus` x_1 x_2`)` → x **level-1 `binary-plus` method**
`(binary-difference` x_1 x_2`)` → x **level-1 `binary-difference` method**
`(negate` x`)` → x **level-1 `negate` method**

`(binary-times` x_1 x_2`)` $\rightarrow x$ **level-1** `binary-times` **method**
`(binary-divide` x_1 x_2`)` $\rightarrow x$ **level-1** `binary-divide` **method**
`(binary-lt` x_1 x_2`)` $\rightarrow x$ **level-1** `binary-lt` **method**
`(binary-gt` x_1 x_2`)` $\rightarrow x$ **level-1** `binary-gt` **method**
`(binary-max` x_1 x_2`)` $\rightarrow x$ **level-1** `binary-max` **method**
`(binary-min` x_1 x_2`)` $\rightarrow x$ **level-1** `binary-min` **method**
`(abs` x`)` $\rightarrow x$ **level-1** `abs` **method**
`(zerop` x`)` $\rightarrow x$ **level-1** `zerop` **method**
`(sign` x`)` $\rightarrow x$ **level-1** `sign` **method**
`(positivep` x`)` $\rightarrow x$ **level-1** `positivep` **method**
`(negativep` x`)` $\rightarrow x$ **level-1** `negativep` **method**
Arithmetic operations for `single-float` are defined by methods to be attached to the generic functions mentioned above. The non class-specific definitions of these operations is given in section 4.1.6.

`(exponent` x`)` $\rightarrow x$ **level-1** `exponent` **method**
`(fraction` x`)` $\rightarrow x$ **level-1** `fraction` **method**
`(scale` x i`)` $\rightarrow x$ **level-1** `scale` **method**
`(succ` x`)` $\rightarrow x$ **level-1** `succ` **method**
`(pred` x`)` $\rightarrow x$ **level-1** `pred` **method**
`(unit-last-place` x`)` $\rightarrow x$ **level-1** `unit-last-place` **method**
`(truncate` x `[`*precision*`])` $\rightarrow x$ **level-1** `truncate` **method**
`(round` x `[`*precision*`])` $\rightarrow x$ **level-1** `round` **method**
`(intpart` x`)` $\rightarrow x$ **level-1** `intpart` **method**
`(fracpart` x`)` $\rightarrow x$ **level-1** `fracpart` **method**
`(floor` x`)` $\rightarrow x$ **level-1** `floor` **method**
`(ceiling` x`)` $\rightarrow x$ **level-1** `ceiling` **method**
Additional arithmetic operations on single precision floating point are defined by methods to be attached to the generic functions mentioned above. The definitions of these operations is the same as in section 4.1.9.
`((converter` *string*`)` x`)` $\rightarrow x$ **level-1 method**

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

`(generic-prin` x *stream*`)` $\rightarrow x$ **level-1** `generic-prin` **method**
`(generic-write` x *stream*`)` $\rightarrow x$ **level-1** `generic-write` **method**
Output external representation of x on *stream* as described in section 2.9 and defined in appendix A.4.

`most-positive-single-float`:x **level-1 constant**
`least-positive-single-float`:x **level-1 constant**
`least-negative-single-float`:x **level-1 constant**
`most-negative-single-float`:x **level-1 constant**

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

The value of `least-positive-single-float` is that positive single 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)`.

The value of `least-negative-single-float` is that negative single 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)`.

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

`(single-float-p` *obj*`)` $\rightarrow$ *boolean* **level-1 function**
Returns *obj* if *obj* is an instance of `single-float`, otherwise `()`.

4.2.7 Variable Precision Integer Arithmetic

`(binary-plus` i_1 i_2`)` → i **level-1** `binary-plus` **method**
`(binary-difference` i_1 i_2`)` → i **level-1** `binary-difference` **method**
`(negate` i`)` → i **level-1** `negate` **method**
`(binary-times` i_1 i_2`)` → i **level-1** `binary-times` **method**
`(binary-lt` i_1 i_2`)` → i **level-1** `binary-lt` **method**
`(binary-gt` i_1 i_2`)` → i **level-1** `binary-gt` **method**
`(binary-max` i_1 i_2`)` → i **level-1** `binary-max` **method**
`(binary-min` i_1 i_2`)` → i **level-1** `binary-min` **method**
`(binary-gcd` i_1 i_2`)` → i **level-1** `binary-gcd` **method**
`(binary-lcm` i_1 i_2`)` → i **level-1** `binary-lcm` **method**
`(abs` i`)` → i **level-1** `abs` **method**
`(zerop` i`)` → i **level-1** `zerop` **method**
`(sign` i`)` → i **level-1** `sign` **method**
`(positivep` i`)` → i **level-1** `positivep` **method**
`(negativep` i`)` → i **level-1** `negativep` **method**
Arithmetic operations for `variable-precision-integer` are defined by methods to be attached to the generic functions mentioned above. The non class-specific definitions of these operations is given in section 4.1.6.

`((converter` *string*`)` i`)` → i **level-1 method**
Constructs and returns a string, the characters of which correspond to the external representation of the instance of `variable-precision-integer` in decimal.

`(generic-prin` i *stream*`)` → i **level-1** `generic-prin` **method**
`(generic-write` i *stream*`)` → i **level-1** `generic-write` **method**
Output external representation of i on *stream* as described in section 2.9 and defined in appendix A.4.

`(variable-precision-integer-p` *obj*`)` → *boolean* **level-1 function**
Returns *obj* if *obj* is an instance of a subclass of `variable-precision-integer`.

`(oddp` i`)` → i **level-1** `oddp` **method**
Returns `t` if the remainder from dividing i by two is non-zero, otherwise `()`.

`(evenp` i`)` → i **level-1** `evenp` **method**
Returns `t` if the remainder from dividing i by two is zero, otherwise `()`.

`(quotient` i_1 i_2`)` → i **level-1** `quotient` **method**
`(remainder` i_1 i_2`)` → i **level-1** `remainder` **method**
`(modulo` i_1 i_2`)` → i **level-1** `modulo` **method**
Additional arithmetic operations on variable precision integers are defined by methods to be attached to the generic functions `quotient`, `remainder` and `modulo`. The definitions of these operations is the same as given in section 4.1.8

4.3 Level-2 Classes

NOTE—Nothing has been defined for level-2 at the time of writing.

5 Environment

This section defines the services that EuLisp needs to be provided by the configuration and those operations that interact with the configuration so as not to be part of the language, but still necessarily part of the definition. These latter have implementation-defined dependencies.

5.1 Interactive Processing

The interactive processing of expressions in EuLisp refers to the special situation of the interactive entry and evaluation of expressions. The processing of collections of modules and of individual modules in EuLisp is covered in section 2.6. The interactive processing of expressions necessarily differs from that of a whole module, since working interactively is as if working in an incomplete module, so that conditions that would be errors or signal errors when processing a whole module must be deferred from the analysis of the expression to the time when it is evaluated.

Processing can take place in two stages depending on whether a particular implementation interprets, compiles, or does both, and the complexity of those stages might vary depending on the implementation technique. In this definition, the first stage is called "translation" and the second is called "execution". This section is an informal description of the meaning of an expression and is concerned with the second stage.

An idealization of how this processing should take place is that all definitions should be added to the current module as they are entered, but should not be analysed in any way. As soon as an expression is entered—that is something that demands execution—it is as if the module has been completed anew, so the whole module plus any new definitions can now be analysed according to the process described in section 2.6 and then instantiated. The expression can now be processed according to the following rules:

literal: a literal stands for itself;

symbol: result is the value stored in the closest lexical binding named by the symbol;

special form: process according to rules for given special form;

defining form: process according to rules for a given defining form;

macro form: expand form and evaluate result;

function form: evaluate arguments and apply function to parameters. Check arity.

5.2 Module operations

`(load-module` *name*`)` → *null* **environment special form**
Loads the module *name*.

`(dynamic-load-module` *expression*`)` → *null* **environment function**
Loads the module whose name is the value of *expression*.

`(start-module` *module-name function-name* exp_1 ... exp_n`)` → *obj* **environment special form**
Evaluates the expressions exp_1 to exp_n in the empty lexical environment and then applies the value of *function-name* exported from module *module-name* to the list of results. If the value of *function-name* is not an applicable object, an error (condition: `invalid-operator`) is signaled.

5.3 File operations

`(path-open` *path-list name* `[`*options*`])` → *stream* **environment function**
Opens a stream connected to the file *name* trying each path in *path-list* in turn, until it succeeds. In the case that *path-list* is exhausted, an error (condition: `cannot-open-path`) is signaled.

`(file-status` *path*`)` → *obj* **environment function**
Returns `()` if no file is accessible via *path*, otherwise result is a table of implementation-defined information about the state of the file.

6 Library Modules

6.1 Elementary-functions Module

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.

`pi`:*double-float* **elementary-functions constant**

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

(`sin` z) $\rightarrow$ z **elementary-functions generic**

(`cos` z) $\rightarrow$ z **elementary-functions generic**

(`tan` z) $\rightarrow$ z **elementary-functions generic**

`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.

(`acos` z) $\rightarrow$ z **elementary-functions generic**

(`asin` z) $\rightarrow$ z **elementary-functions generic**

`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.

(`atan` z) $\rightarrow$ z **elementary-functions generic**

(`atan2` x_1 x_2) $\rightarrow$ z **elementary-functions generic**

`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.

`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 3 and figure 4). 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 $(-\texttt{pi}, \texttt{pi}]$.

(`exp` z) $\rightarrow$ z **elementary-functions generic**

`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.

(`log` z) $\rightarrow$ z **elementary-functions generic**

(`log2` z) $\rightarrow$ z **elementary-functions generic**

(`log10` z) $\rightarrow$ z **elementary-functions generic**

`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.

`(sqrt` z`)` $\rightarrow z$ **elementary-functions generic**
`sqrt` returns the principal square root of z.

`(sqrt` *integer*`)` $\rightarrow z$ **elementary-functions `sqrt` method**
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.

`(sqrt` *ratio*`)` $\rightarrow z$ **elementary-functions `sqrt` method**
The method for `ratio` returns a rational if both the numerator and denominator of the argument are perfect squares, a gaussian rational if the argument is a negative rational with perfect square numerator and denominator, otherwise a `float` is returned if the argument is positive, or a `complex` if the argument is negative.

`(sqrt` *float*`)` $\rightarrow z$ **elementary-functions `sqrt` method**
The method for `float` returns a `float` if the argument is non-negative and a `complex` if it is not.

`(sqrt` *complex*`)` $\rightarrow z$ **elementary-functions `sqrt` method**
The method for `complex` returns a `complex`. In this case, the principal square root is, that with the smallest argument, where $0 \leq arg(z) < 2\pi$.

`(expt` z_1 z_2`)` $\rightarrow z$ **elementary-functions generic**
`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 6 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.

`(sinh` z`)` $\rightarrow z$ **elementary-functions generic**
`(cosh` z`)` $\rightarrow z$ **elementary-functions generic**
`(tanh` z`)` $\rightarrow z$ **elementary-functions generic**
`(asinh` z`)` $\rightarrow z$ **elementary-functions generic**
`(acosh` z`)` $\rightarrow z$ **elementary-functions generic**
`(atanh` z`)` $\rightarrow z$ **elementary-functions generic**
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.

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`

Table 6: `expt` result classes

NOTE—more detailed specification is required for this library module, in particular with respect to the handling of negative 0.0 and the stating of branches and cuts.

6.2 Rational Arithmetic Module

`(binary-plus` q_1 q_2`)` → q **rational** `binary-plus` **method**
`(binary-difference` q_1 q_2`)` → q **rational** `binary-difference` **method**
`(negate` q`)` → q **rational** `negate` **method**
`(binary-times` q_1 q_2`)` → q **rational** `binary-times` **method**
`(binary-divide` q_1 q_2`)` → q **rational** `binary-divide` **method**
`(binary-lt` q_1 q_2`)` → q **rational** `binary-lt` **method**
`(binary-gt` q_1 q_2`)` → q **rational** `binary-gt` **method**
`(binary-max` q_1 q_2`)` → q **rational** `binary-max` **method**
`(binary-mqn` q_1 q_2`)` → i **rational** `binary-mqn` **method**
`(binary-gcd` q_1 q_2`)` → q **rational** `binary-gcd` **method**
`(binary-lcm` q_1 q_2`)` → q **rational** `binary-lcm` **method**
`(abs` q`)` → q **rational** `abs` **method**
`(positivep` q`)` → q **rational** `positivep` **method**
`(negativep` q`)` → q **rational** `negativep` **method**
Arithmetic operations for `ratio` are defined by methods to be attached to the generic functions mentioned above. The non class-specific definitions of these operations is given in section 4.1.6. It is implementation-defined whether instances of `ratio` are kept in reduced (lowest terms) or unreduced form.

`((converter` *string*`)` q`)` → q **rational method**
Constructs and returns a string, the characters of which correspond to the external representation of the instance of `ratio`.

`(generic-prin` q *stream*`)` → q **rational** `generic-prin` **method**
`(generic-write` q *stream*`)` → q **rational** `generic-write` **method**
Output external representation of q on *stream* as described in section 2.9 and defined in appendix A.4.
`(numerator` q`)` → i **rational function**
`(denominator` q`)` → i **rational function**
Return the numerator and the denominator of the rational number respectively.

6.3 Complex Arithmetic Module

`(binary-plus` z_1 z_2`)` → z **complex** `binary-plus` **method**
`(binary-difference` z_1 z_2`)` → z **complex** `binary-difference` **method**

(`negate` z) → z **complex `negate` method**
(`binary-times` z_1 z_2) → z **complex `binary-times` method**
(`binary-divide` z_1 z_2) → z **complex `binary-divide` method**
(`binary-gcd` q_1 q_2) → q **complex `binary-gcd` method**
(`binary-lcm` q_1 q_2) → q **complex `binary-lcm` method**
(`zerop` z) → z **complex `zerop` method**
(`abs` z) → z **complex `abs` method**
Arithmetic operations for `complex` are defined by methods to be attached to the generic functions mentioned above. The greatest common divisor and lowest common multiple methods are only meaningful when the parameterizing class for `complex` is either a subclass of `integer` or `ratio`. The non class-specific definitions of these operations is given in section 4.1.6.

((`converter` *string*) z) → z **complex method**
Constructs and returns a string, the characters of which correspond to the external representation of the instance of `cartesian-pair`.

(`generic-prin` z *stream*) → z **complex `generic-prin` method**
(`generic-write` z *stream*) → z **complex `generic-write` method**
Output external representation of z on *stream* as described in section 2.9 and defined in appendix A.4.

(`make-complex` x_1 x_2) → z **complex function**
Constructs and returns a new instance of class `complex` whose real part and imaginary part are x_1 and x_2 respectively.

(`real-part` z) → x **complex function**
Returns the real part of z.

(`imaginary-part` z) → x **complex function**
Returns the imaginary part of z.

6.4 List-operators Module

6.4.1 Reconstructing Lists

(`append` [$list_1$ ... $list_n$ *obj*]) → *list(obj)* **listops function**
The first elements of the top-level pairs of $list_1$ to $list_n$ are copied to form a single list, which shares the structure of *obj* by using `cons`. It is an error if any $list_i$ is not a proper list. If no arguments are given, the result is `()`.

(`removeq` *obj list*) → *list(obj)* **listops function**
(`remove` *obj list* [*predicate*]) → *list(obj)* **listops function**
Constructs a top-level copy of *list* containing only those first elements of the top-level pairs in *list*, such that (*predicate obj element*) is false. In the case of `removeq`, *predicate* is `eq`. In the case of `remove`, if *predicate* is not supplied, `equal` is used. The result of these functions may share structure with the argument `list` and the result may be `eq` to `list` if *obj* does not occur in *list*.

(`reverse` *list(obj)*) → *list(obj)* **listops function**
Constructs a copy of *list* such that the first element of each top-level pair at position i in a list of length n appears at position $n - i - 1$.

(`substq` obj_1 obj_2 *list*) → *list(obj)* **listops function**
(`subst` obj_1 obj_2 *list* [*predicate*]) → *list(obj)* **listops function**
Constructs a top-level copy of *list* replacing with obj_2 those first elements of the top-level pairs in *list*, such that (*predicate* obj_1 *element*) is true. In the case of `substq`, *predicate* is `eq`. In the case of `subst`, if *predicate* is not supplied, `equal` is used. The result of these functions may share structure with the argument `list` and the result may be `eq` to `list` if *obj* does not occur in *list*.

6.4.2 Copying Lists

`not-an-alist(execution-condition)` **listops condition**
This condition is signaled by `copy-alist`. The init-options for this condition-class are:

alist: The offending object.

`(copy-alist` *alist*`)` → *alist* **listops function**
Constructs a copy of the list *alist* copying both the top-level pairs and the second level pairs (the associations). An error is signaled (condition: `not-an-alist`) if the top-level elements of *alist* are not subclasses of `pair`.

`(copy-list` *list*`)` → *list* **listops function**
Constructs a copy of *list* by copying the top-level pairs only.

`(copy-tree` *list*`)` → *list* **listops function**
Constructs 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*.

6.4.3 Updating Lists

`(nconc [`$list_1$ ... $list_n$ *obj*`])` → *list(obj)* **listops function**
The first elements of the top-level pairs of $list_1$ to $list_n$ are concatenated (destructively) to form a single list, which is then linked (destructively) to *obj*. Consequently, if *obj* is an improper list, the result will be an improper list. If no arguments are given, the result is `()`. If no arguments are given the result is `()`.

`(nreverse` *list(obj)*`)` → *list(obj)* **listops function**
The result is a list containing the same elements as *list*, but in reverse order. The argument *list* might be modified and re-used to produce the result. The result might or moight not be `eq` to *list*, so the result should always be used, since it is not guaranteed that retaining a reference to *list* and calling `nreverse` will cause that reference to contain the reversed list.

`(tconc` *list obj*`)` → *list(obj)* **listops function**
`(lconc` $list_1$ $list_2$`)` → *list(obj)* **listops function**
These two functions operate in a similar way—the difference is that `tconc` destructively adds a single object to the end of *list* whilst `lconc` adds a list of objects. The first argument is not the list of objects itself, but a pair, the fields of which hold the front and the last pair of the list being updated respectively. If the first argument is `()`, a new pair is constructed, initialized as described, and returned as the result. New data may be added to this structure using either `tconc` or `lconc`.

`(deleteq` *obj list*`)` → *list(obj)* **listops function**
`(delete` *obj list* `[`*predicate*`])` → *list(obj)* **listops function**
Destructively modifies *list* so that it only contains those first elements of the top-level pairs in *list*, such that (*predicate obj element*) is false. Note that the return value of `delete` should always be used in preferece to the pointer to the list being modified, so that in the situation of the first element of the list being deleted the correct value be used subsequently. In the case of `deleteq`, *predicate* is `eq`. In the case of `delete`, if *predicate* is not supplied, `equal` is used.

6.4.4 Converting Lists

`((converter` *vector*`)` *pair*`)` → *pair* **listops method**
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.
`((converter` *string*`)` *pair*`)` → *pair* **listops method**
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: `improper-list-conversion`) unless all of those elements are instances of the class `character`.

6.4.5 Searching Lists

`(assq` *obj alist* `[`*fail-value*`])` → *obj* **listops function**
`(assoc` *obj alist* `[[`*predicate*`]` *fail-value*`])` → *obj* **listops function**
The association list *alist* is seaarched for a key such that (*predicate obj key*) is not false. In the case of `assq`, *predicate* is `eq`. In the case of `assoc`, if *predicate* is not supplied, `equal` is used. If such a key is found, the corresponding key-value pair is returned. If none of the keys matches, the result is `()` or *fail-value* if supplied.

`(memq` *obj list*`)` → *boolean* **listops function**
`(member` *obj list* `[`*predicate*`])` → *boolean* **listops function**
Examines the first element of each top-level pair in *list* for an element such that (*predicate obj element*) is true. In the case of `memq`, *predicate* is `eq`. In the case of `member`, if *predicate* is not supplied, `equal` is used. If such an element is found, then the *list* of which *obj* is the first element is returned. If no such element is found, the result is `()`.

`(positionq` *obj list*`)` → *i* **listops function**
`(position` *obj list*`[`*predicate*`])` → *i* **listops function**
Compares the first element of each top-level pair in *list* until an element is found which such that (*predicate obj element*) is true. In the case of `posq`, *predicate* is `eq`. In the case of `pos`, if *predicate* is not supplied, `equal` is used. If such an element is found, then its position in *list*, counting from the front, is returned, such that (*predicate obj* `(list-ref (position` *obj list predicate*`)` *list*)) is true. If no such element is found, the result is `()`.

6.4.6 Dissecting Lists

`(last-car` *list*`)` → *obj* **listops function**
Returns the `car` of the last pair of *list*.

`(last-pair` *list*`)` → *pair* **listops function**
Returns the last pair of *list*.

`list-index-out-of-range(execution-condition)` **listops condition**
This condition is signaled by `list-ref` and `list-tail`. The init-options for this condition-class are:

list: The value of this option is the list in question.

index: The value of this option is the index position sought.

`(list-ref` *list n*`)` → *obj* **listops function**
`((setter list-ref)` *list n obj*`)` → *obj* **listops function**
`list-ref` returns the element at position *n* in the list, where the index of the first element is zero. `(setter list-ref)` updates the nth position in *list*. An error is signalled (condition: `list-index-out-of-range`) if *n* is not in the range zero to the length of the list minus one.

`(list-tail` *list n*`)` → *list(obj)* **listops function**
Returns the tail of *list* which starts with the element at position *n* in the list, where the index of the first element is zero. An error is signalled (condition: `list-index-out-of-range`) if *n* is not in the range zero to the length of the list minus one.

6.4.7 Lists as Sets

`(difference` [*predicate*] [$list_1$ ...]`)` → *list* **listops function**
`(differenceq` [$list_1$ ...]`)` → *list* **listops function**
If no arguments are supplied, the result is `()`. If one (*list*) argument is given, the result is that argument. If more than one argument is passed, set difference of the supplied lists is computed cumulatively using left association. `differenceq` uses `eq` for comparison. `difference` uses *equal* for comparison, or *predicate* if supplied.

`(intersection` [*predicate*] [$list_1$ ...]`)` → *list* **listops function**
`(intersectionq` [$list_1$ ...]`)` → *list* **listops function**
If no arguments are supplied, the result is `()`. If one (*list*) argument is given, the result is that argument. If more than one argument is passed, the intersection of the supplied lists is computed. `intersectionq` uses `eq` for comparison. `intersection` uses `equal` for comparison or *predicate*, if supplied.

`(null` *obj*`)` → *boolean* **listops function**
Returns `t` if *obj* is `eq` to `()` and `()` if not.

`(union` [*predicate*] [$list_1$ ...]`)` → *list* **listops function**
`(unionq` [$list_1$ ...]`)` → *list* **listops function**
If no arguments are supplied, the result is `()`. If one (*list*) argument is given, the result is that argument. If more than one argument is passed, the union of the supplied lists is computed. `unionq` uses `eq` for comparison. `union` uses `equal` for comparison or *predicate*, if supplied.

6.4.8 Mapping over Lists

`(mapc` *function* $list_1$ ... $list_n$`)` → *nul* **listops function**
`(mapcar` *function* $list_1$ ... $list_n$`)` → *list(obj)* **listops function**
`(mapcan` *function* $list_1$ ... $list_n$`)` → *list(obj)* **listops function**
These three mapping operators apply *function* the successive `cars` of $list_1$ to $list_n$. The differences between them arise from how they combine the results of these applications: `mapc` discards the results, `mapcar` constructs a list of the results using `cons` and `mapcan` constructs a list of the results using `nconc`. The number of applications of *function* is determined by the length of the shortest list. The results from side-effecting the lists which are the arguments to the map operation are undefined.

`(mapl` *function* $list_1$ ...`)` → *null* **listops function**
`(maplist` *function* $list_1$ ...`)` → *list(obj)* **listops function**
`(mapcon` *function* $list_1$ ...`)` → *list(obj)* **listops function**
These three mapping operators apply *function* the successive `cdrs` of $list_1$ to $list_n$. The differences between them arise from how they combine the results of these applications: `mapl` discards the results, `maplist` constructs a list of the results using `cons` and `mapcon` constructs a list of the results using `nconc`. The number of applications is limited by the length of the shortest list. The results from side-effecting the lists which are the arguments to the map operation are undefined.

6.5 Formatted-IO Module

The `formatted-io` module exports the functions `format` and `scan` and some related conditions.

`scan-mismatch(stream-condition)` **format condition**
This condition is signaled by `scan`. The init-options for this condition are:

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

input: 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.

(`scan` *format-string* `[`*stream*`]`) → *list(obj)* **format function**
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 vthe

`~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 Syntax

A.1 Notation

The syntax of EuLisp programs is defined in an extension of Backus-Naur form. The extensions used are as follows:

- $item^*$ indicates zero or more occurrences of *item*;
- $item^+$ indicates one or more occurrences of *item*;
- $item^n$ indicates precisely n occurrences of *item*;
- `[`*item*`]` indicates zero occurrences or one occurrence of *item*;
- $\{item_1 \mid \ldots \mid item_n\}$ indicates a single occurrence of one of $item_i$ for $1 \leq i \leq n$.

Typographical conventions employed are that terminal symbols appear in `courier` font and non-terminals appear in *italic* font.

A.2 Definition Syntax

A.2.1 Level-0 Definition Syntax

level-0-definition ::= *defconstant* | *defcondition* | *deflocal* | *defmacro* | *defmodule* |
defstruct | *defun*
defcondition ::= `(defcondition` *condition-name superclass init-option**`)`
defconstant ::= `(defconstant` *name form*`)`
deflocal ::= `(deflocal` *name 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 init-option**`)`
level-0-init-option ::= *init-option*
gf-name ::= *identifier*
gen-lambda-list ::= *spec-lambda-list*
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**`)`
defun ::= `(defun` *function-name lambda-list body*`)` |
`(defun (setter` *function-name*`)` *lambda-list body*`)`

A.2.2 Level-1 Definition Syntax

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 ::= {`class` *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* |
`slot-class` *slot-description-class* |
identifier value
class-option ::= `constructor` *constructor-spec*
`predicate` *predicate-name*
`metaclass` *class-name* |
identifier value
defgeneric ::= (`defgeneric` *gf-name gen-lambda-list init-option**)
level-1-init-option ::= *level-0-init-option* |
`class` *gf-class-name* |
`method-class` *method-class-name* |
`method` (*method-description*)
gf-class-name ::= *class-name*
method-class-name ::= *class-name*
class-name ::= {`class` *or the name of one of its subclasses*}
defvar ::= (`defvar` *name expression*)

A.3 Expression Syntax

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` *variable body*`)`
progn ::= `(progn` *expression**`)`
with-handler ::= `(with-handler` *expression expression*`)`
unwind-protect ::= `(unwind-protect` *expression expression**`)`
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**

level-1-expression ::= *dynamic-let* | *dyn-ref* | *dyn-assign* | *level-0-expression*
dyn-ref ::= `(dynamic` *identifier*`)`
dyn-assign ::= `(dynamic-setq` *identifier expression*`)`
dynamic-let ::= `(dynamic-let` *binding** *form**`)`

expression ::= *level-1-expression* | *level-0-expression*

A.4 External Representations - `write`

datum ::= *character* | *number* | *string* | *structure* | *symbol*
structure ::= *complex* | *list* | *vector*
symbol ::= *identifier*
list ::= *pair-begin datum** *pair-end*
pair ::= *pair-begin datum pair-separator datum pair-end*
vector ::= *extension vector-begin datum** *vector-end*
vector-begin ::= (
vector-end ::=)
pair-separator ::= *whitespace* . *whitespace*

A.5 External Representations - `prin`

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*

A.6 Token Syntax

token ::= *character* | *dot* | *number* | *string* | *identifier* | *pair-begin* | *pair-end* |
quotation | *anti-quotation* | *unquotation* | *unquote-splice* | *extension*
dot ::= `.`
pair-begin ::= `(`
pair-end ::= `)`
quotation ::= `'`
anti-quotation ::= `` ` ``
unquotation ::= `,`
unquote-splice ::= `,@`
extension ::= `#`
delimiter ::= *whitespace* | *pair-begin* | *pair-end* | *string-begin* | *string-end* |
comment-begin | *comment-end*
whitespace ::= *space* | *tab* | *newline*
comment-begin ::= `;`
comment-end ::= *newline*

A.7 S-expression Syntax

s-expression ::= *simple* | *compound*
simple ::= *character* | *number* | *string* | *identifier* | *abbreviation*
compound ::= *list* | *vector*
list ::= *empty-list* | *pair-begin s-expression list-tail*
list-tail ::= *pair-end* | *dot s-expression pair-end* | *s-expression list-tail*
vector ::= *extension pair-begin s-expression** *pair-end*
abbreviation ::= *prefix s-expression*
prefix ::= *quotation* | *anti-quotation* | *unquotation* | *unquote-splice* | *extension*

A.8 Character Syntax

character ::= *extension character-extension character-name*
character-name ::= *literal-name* | *control-name* | *special-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 ::= *hex-insertion*

A.9 Number Syntax

number ::= *exact-number* | *inexact-number* | *complex*
exact-number ::= `[`*sign*`]` {*uinteger* | *urational*}
inexact-number ::= `[`*sign*`]` *ufloat*
complex ::= *extension* {`c` | `C`} `(`*non-complex non-complex*`)`
non-complex ::= *exact-number* | *inexact-number*
sign ::= {`+` | `-`}
uinteger ::= *ubinary* | *uoctal* | *udecimal* | *uhexadecimal* | *uinteger-with-base*
urational ::= *uinteger ratio-separator uinteger*
ufloat ::= *udecimal exponent* |
float-separator udecimal `[`*exponent*`]` |
udecimal float-separator `[`*udecimal*`]` `[`*exponent*`]`
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)*$^{+}$
ratio-separator ::= `/`
float-separator ::= `.`
exponent ::= *expt-mark* `[`*sign*`]` *udecimal*
digit(2) ::= {`0` | `1`}
... ...
digit(10) ::= {`0` | ... | `9`}
digit(11) ::= {`0` | ... | `9` | {`a` | `A`}}
... ...
digit(36) ::= {`0` | ... | `9` | {`a` | `A`} | ... | {`y` | `Y`}}
digit(37) ::= {`0` | ... | `9` | {`a` | `A`} | ... | {`y` | `Y`} | {`z` | `Z`}}
expt-mark ::= {`e` | `d` | `E` | `D`}

A.10 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`

A.11 Identifier Syntax

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*
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`
alphabetic ::= *upper* | *lower*
alphanumeric ::= *alphabetic* | *digit*
ordinary ::= `!` | `$` | `%` | `&` | `*` | `/` | `:` | `<` | `=` | `>` | `?` | `^` | `_` | `~` | `,` |
`;` | `[` | `\` | `]` | `{` | `}`
non-digit ::= *alphabetic* | *ordinary* | *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 ::= `.`

A.12 Quasiquotation

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*$_{i-1}$`)`

B Formal Semantics

NOTE—At present the formal semantic description of EuLisp *is incomplete and therefore is not included in this definition.*

Bibliography

[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., DeMichiel 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.

[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

Index