LISP AND SYMBOLIC COMPUTATION: An International Journal, ?, ??–??, 1993
© 1993 Kluwer Academic Publishers – Manufactured in The Netherlands

An overview of EuLisp

JULIAN PADGET[*] (*jap@maths.bath.ac.uk*)
University of Bath, School of Mathematical Sciences, Bath BA2 7AY, United Kingdom
GREG NUYENS (*nuyens@ilog.com*)
Ilog Inc., 2073 Landings Drive, Mountain View, CA 94025, USA
HARRY BRETTHAUER (*bretthauer@gmd.de*)
German National Research Centre for Computer Science (GMD), P.O.Box 1316, W-5205 Sankt Augustin, FRG

Keywords: Lisp, modules, concurrency, object-oriented programming, conditions, reflection.

Abstract. This paper is an abstracted version of the EuLisp definition. As such it emphasizes those parts of the language that we consider the most important or noteworthy, while we just mention, without much detail the elements that are included for completeness. This is reflected in the structure of the paper which describes the module scheme, the object system and support for concurrent execution in the main part and consigns the majority of the datatypes to an appendix.

1. Introduction

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

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

- Classes are first-class objects. The class structure integrates the primitive classes describing fundamental datatypes, the predefined classes and user-defined classes.
- Modules together with classes are the building blocks of both the EuLisp language and of applications written in EuLisp. The module

[*]This work has been supported by too many organisations and programmes to name them individually here, although special mention should be made of the Commission of the European Communities and Jean Omnes of DG XIII (Information technology directorate). A complete list of acknowledgements appears in Section 11

system exists to limit access to objects by name. That is, modules allow for hidden definitions. Each module defines a fresh, empty, lexical environment.

- Multiple control threads can be created in EuLisp and the concurrency model has been designed to allow consistency across a wide range of architectures. Access to shared data can be controlled via locks (semaphores). Event-based programming is supported through a generic waiting function.
- Both functions and continuations are first-class in EuLisp, but the latter are not as general as in Scheme because they can only be used in the dynamic extent of their creation. That implies they can only be used once.
- A condition mechanism which is fully integrated with both classes and threads, allows for the definition of generic handlers and which supports both propagation of conditions and continuable handling.
- Dynamically scoped bindings can be created in EuLisp, but their use is restricted, as in Scheme. EuLisp enforces a strong distinction between lexical bindings and dynamic bindings by requiring the manipulation of the latter via special forms.

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

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

1.1. Language Structure

The EuLisp definition comprises the following items:

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

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

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

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

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

Table 1: Modules comprising `eulisp-level-0`

Module	Section(s)
`character`	A.1
`collection`	A.2
`compare`	A.3
`condition`	9
`convert`	A.4
`copy`	A.5
`double-float`	A.6
`elementary-functions`	A.7
`event`	10.7
`fixed-precision-integer`	A.9
`formatted-io`	A.8
`function`	10.3
`lock`	8.2
`null`	A.10
`number`	A.11
`object-0`	7
`pair`	A.12
`stream`	A.13
`string`	A.14
`symbol`	A.15
`syntax-0`	10.8
`table`	A.16
`thread`	8.1
`vector`	A.17

This definition is organized in three parts:

Sections 5–10 describes the core of level-0 of EuLisp, covering modules, simple classes, objects and generic functions, threads, conditions, control forms and events. These sections contain the information about EuLisp that characterizes the language.

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

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

Prior to these, sections 2–4 define the scope of the text and error definitions and typographical and layout conventions used in this text.

2. Scope

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

This text does not specify:

1. The size or complexity of a 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 a EuLisp processor.
3. The method of preparation of a EuLisp program for execution or the method of activation of this EuLisp program once prepared.
4. The method of reporting errors, warnings or exceptions to the client of a EuLisp processor.
5. The typographical representation of a EuLisp program for human reading.

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

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

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

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

3. Error Definitions

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

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

1. Where a *conforming processor* is required to detect the erroneous situation or behaviour and report it. This is signified by the phrase *an error is signalled*. The condition class to be signalled is specified. Signalling an error consists of identifying the condition class related to the error and allocating an instance of it. This instance is initialized with information dependent on the condition class. A conforming EuLisp program can rely on the fact that this condition will be signalled.

2. Where a *conforming processor* might or might not detect and report upon the error. This is signified by the phrase … *is an error*. A processor should provide a mode where such errors are detected and reported where possible.

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

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

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

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

4. Conventions

This section defines the conventions employed in this text, how definitions will be laid out, the typeface to be used, the meta-language used in descriptions and the naming conventions. Appendix (C) contains a glossary of definitions used in this text.

4.1. Layout and Typography

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

`a-special-form` *special form*

Syntax

(`a-special-form` form$_1$ … form$_n$)

Arguments

form$_1$: description of structure and rôle of *form*$_1$.

⋮

form$_n$: description of structure and rôle of *form*$_n$.

Result

A description of the result.

Remarks

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

Examples

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

See also:

Cross references to related entries.

`a-function` *function*

Arguments

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

⋮

`[`*argument-n*`]` : information about the class or classes of the optional argument *argument-n*.

Result

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

Remarks

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

Examples

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

See also:

Cross references to related entries.

`a-generic` *generic function*

Generic Arguments

`(`*argument-a* `<class-a>)`: means that *argument-a* of `a-generic` must be an instance of `<class-a>` and that *argument-a* is one of the arguments on which `a-generic` specializes. Furthermore, each method defined on `a-generic` may specialize only on a subclass of `<class-a>` for *argument-a*.

⋮

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

Result

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

Remarks

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

See also:

Cross references to related entries.

`a-generic` *method*

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

Specialized Arguments

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

⋮

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

Result

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

Remarks

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

See also:

Cross references to related entries.

`<a-condition>` *a-condition-superclass*

Init-options

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

⋮

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

Remarks

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

`<class-name>` *class*

Init-options

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

⋮

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

Remarks

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

4.2. Meta-Language

The terms used in the following descriptions are defined in Appendix C.

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

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

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

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

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

5. Syntax

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

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

Table 2: Minimal character set

digit	::=	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
upper	::=	`A` | `B` | `C` | `D` | `E` | `F` | `G` | `H` | `I` | `J` | `K` | `L` | `M` |
		`N` | `O` | `P` | `Q` | `R` | `S` | `T` | `U` | `V` | `W` | `X` | `Y` | `Z`
lower	::=	`a` | `b` | `c` | `d` | `e` | `f` | `g` | `h` | `i` | `j` | `k` | `l` | `m` |
		`n` | `o` | `p` | `q` | `r` | `s` | `t` | `u` | `v` | `w` | `x` | `y` | `z`
other	::=	`*` | `/` | `<` | `=` | `>` | | `+` | `-` | `.`
special	::=	`;` | `'` | `,` | `\` | `"` | `#` | `(` | `)` | `` ` `` |
		alert | *backspace* | *delete* | *formfeed* | *linefeed* | *newline*
		return | *space* | *tab* | *vertical-tab*
alphc	::=	*upper* | *lower*

5.1. Whitespace and Comments

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

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

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

5.2. Objects

The syntax of the classes of objects that can be read by EuLisp is defined in the section of this definition corresponding to the class:

`<character>` (A.1), `<double-float>` (A.6),
`<fixed-precision-integer>` (A.9), `<null>` (A.10),
`<cons>` (A.12), `<string>` (A.14),
`<symbol>` (A.15), `<vector>` (A.17).

The syntax for identifiers corresponds to that for symbols.

6. Modules

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

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

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

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

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

6.1. Directives

The list of module directives is a sequence of keywords and forms, where the keywords indicate the interpretation of the forms. This representation allows for the addition of further keywords at other levels of the definition and also for implementation-defined keywords. For the keywords given here, there is no defined order of appearance, nor is there any restriction on the number of times that a keyword can appear. Multiple occurrences of any of the directives defined here are treated as if there is a single directive whose form is the combination of each of the occurrences. This definition describes the processing of four keywords, which are now described in detail. The syntax of all the directives is given in Table 3 and an example of their use appears in Figure 1.

6.1.1. Export Directive

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

6.1.2. Import Directive

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

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

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

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

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

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

Figure 1: Example of module directives

selective importation of specified bindings from a module.

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

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

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

`only` Filters the names from each *module-descriptor* keeping only those names specified and discarding all other names. The `only` directive is convenient when only a few names exported by a module are required, since it is only necessary to name those that are wanted to include them.

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

6.1.3. Expose Directive

This is denoted by the keyword `expose` followed by a list of *module-directive*s. The purpose of this directive is to allow a module to export subsets of the external environments of various modules without importing them itself. Processing an expose directive employs the same model as for imports, namely, a pair of a module-name and a local-name with the same filtering operations. When the namelist has been filtered, the names are added to the external environment of the module begin processed. A name clash can arise in the external environment from interaction with exports or between different exposed modules. As an example of the use of `expose`, a possible implementation of the `eulisp-level-0` module is shown in Figure 2.

It is also meaningful for a module to include itself in an expose directive. In this way, it is possible to refer to all the bindings in the module being defined. This is convenient, in combination with `except` (see Section 6.1.2), as a way of exporting all but a few bindings in a module, especially if syntax expansion creates additional bindings whose names are not known, but should be exported.

```
(defmodule eulisp-level-0
  (expose
    (character collection compare condition convert copy
     double-float elementary-functions event formatted-io
     fixed-precision-integer function lock null number object-0
     pair stream string symbol syntax-0 table thread vector)))
```

Figure 2: Example module using `expose`

6.1.4. Syntax Directive

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

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

6.2. Definitions and Expressions

Definitions in a module only contain unqualified names—that is, *local-name*s, using the above terminology. A top-lexical binding is created exactly once and shared with all modules that import its exported name from the module that created the binding. A name clash can arise in the top-lexical environment from interaction between local definitions and between local definitions and imported modules. Only top-lexical bindings created by `deflocal` are mutable—both in the defining module and in any importing module. It is a static 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—that is after the creation and initialization of the top-lexical bindings. The exception to this is the `progn` form, which is descended and the forms within it are treated as if the `progn` were not present. Definitions may only appear either at top-level within a module definition or inside any number of `progn` forms. This is specified more precisely in the grammar for a module given in Table 3.

6.3. Module Processing

The following steps summarize the module definition process:

directive processing This is described in detail in Section 6.1. This step creates and initializes the top-lexical, syntax and external environments.

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

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

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

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

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

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

6.4. Module Definition

`defmodule` *syntax*

Syntax

`(defmodule` *module-name* `(`*module-directive**`)` *module-form**`)`

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

Arguments

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

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

module-form* : A sequence of defining forms, expressions and export forms.

Table 3: Module syntax

module-name	::=	*identifier*
module-directive	::=	*export* | *expose* | *import* | *syntax*
module-form	::=	*export-form* | *level-0-expression* | *defining-form*
	|	`(progn` *module-form*`)`
export	::=	`export (`*identifier**`)`
expose	::=	`expose (`*module-descriptor**`)`
import	::=	`import (`*module-descriptor**`)`
syntax	::=	`syntax (`*module-descriptor**`)`
export-form	::=	`(export` *identifier**`)`
module-descriptor	::=	*module-name* | *module-filter*
module-filter	::=	*except* | *only* | *rename*
except	::=	`(except (`*identifier**`)` *module-descriptor*$^+$`)`
only	::=	`(only (`*identifier**`)` *module-descriptor*$^+$`)`
rename	::=	`(rename (`*rename-pair**`)` *module-descriptor*$^+$`)`
rename-pair	::=	`(`*old-identifier new-identifier*`)`

Remarks

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

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

Examples

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

7. Objects

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

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

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

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

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

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

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

Classes are defined using the `defstruct` (7.3) and `defcondition` (9.1) defining forms.

Generic functions are defined using the `defgeneric` defining form, which creates a named generic function in the top-lexical environment of the module in which it appears and `generic-lambda`, which creates an anonymous

```
<object>
   <character>
   <condition>
      ...
   <function>
      <continuation>
      <simple-function>
      <generic-function>
   <list>
      <cons>
      <null>
   <lock>
   <number>
      <integer>
      <float>
         <double-float>
   <stream>
   <string>
   <structure>
   <symbol>
   <table>
   <thread>
   <vector>
```

Figure 3: Level-0 initial class hierarchy

generic function. These forms are described in detail later in this section.

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

7.1. System Defined Classes

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

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

`<object>` *class*

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

`<structure>` *class*

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

`<telos-condition>` *condition*

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

7.2. Single Inheritance

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

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

1. the *CPL* of `<object>` is a list of one element containing `<object>` itself;

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

The class precedence list controls system-defined protocols concerning:

Table 4: `defstruct` syntax

class-name	::=	*identifier*
superclass-name	::=	{*the name of a subclass of* `<structure>`}
slot-spec	::=	*slot-name* | `(`*slot-name slot-option**`)`
slot-name	::=	*identifier*
slot-option	::=	`initarg` *initarg-name*
	|	`initform` *form*
	|	`reader` *identifier*
	|	`writer` *identifier*
	|	`accessor` *identifier*
class-option	::=	`initargs` `(`*initarg-name**`)`
	|	`constructor` *constructor-spec*
	|	`predicate` *identifier*
constructor-spec	::=	`(`*identifier initarg-name**`)`
initarg-name	::=	*identifier*

1. inheritance of slot and class options when initializing a class,
2. method lookup and generic dispatch when applying a generic function.

7.3. Defining Classes

`defstruct` *defining form*

Syntax

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

The syntax of `defstruct` is defined in Table 4.

Arguments

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

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

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

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

Remarks

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

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

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

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

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

`writer` *identifier*: The value is the identifier of the variable to which the writer function will be bound. The binding is immutable. The writer function is a means to change the slot value. The creation of the writer

[1]In combination with the guarantee that the behaviour of generic functions cannot be modified once it has been defined, due to no support for method removal nor method combination, this imbues level-0 programs with static semantics.

is analogous to that of the reader function. The writer function is a function of two arguments, the first should be an instance of the new class and the second can be any new value for the slot. This option can be specified more than once for a slot. It is a static error to specify the same reader, writer, or accessor name for two different slots.

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

The class options are interpreted as follows:

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

`constructor` *constructor-spec*: Creates a constructor function for the new class. The constructor specification gives the name to which the constructor function will be bound, followed by a sequence of legal initargs for the class. The new function creates an instance of the class and fills 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.

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

7.4. Defining Generic Functions and Methods

`defgeneric` *defining form*

Syntax

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

Arguments

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

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

*level-0-init-option** : Options as specified below.

Remarks

This defining form defines a new generic function. The resulting generic function will be bound to *gf-name*. The second argument is the formal parameter list. The method's specialized lamba list must be congruent to that of the generic function. Two lambda lists are said to be *congruent* iff:

1. both have the same number of formal parameters, and
2. if one lambda list has a rest formal parameter then the other lambda list has a rest formal parameter too, and vice versa.

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

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

An error is signalled (condition class: `<method-domain-clash>`) if any methods defined on this generic function have the same domain. These conditions apply both to methods defined at the same time as the generic function and to any methods added subsequently by `defmethod`. An *init-option* is an identifier followed by a corresponding value. The syntax of `defgeneric` is given in Table 5.

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

The *init-option* is:

`method` *method-spec*: This option is followed by a method description. A method description is a list comprising the specialized lambda list of the method, which denotes the domain, and a sequence of forms,

Table 5: `defgeneric` syntax (level-0)

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

Table 6: `defgeneric` rewrite rules

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

denoting the method body. The method body is closed in the lexical environment in which the generic function definition appears. This option may be specified more than once.

The rewrite rules for the `defgeneric` form are given in Table 6.

Examples

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

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

See also:

`defmethod`, `generic-lambda`.

`defmethod` *macro*

Syntax

`(defmethod` *gf-name spec-lambda-list form**`)`

Remarks

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

`generic-lambda` *macro*

Syntax

`(generic-lambda` *gen-lambda-list level-0-init-option**`)`

Remarks

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

Examples

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

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

See also:

`defgeneric`.

`<no-applicable-method>` *telos-condition*

Init-options

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

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

Remarks

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

`<incompatible-method-domain>` *telos-condition*

Init-options

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

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

Remarks

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

`<non-congruent-lambda-lists>` *telos-condition*

Init-options

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

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

Remarks

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

`<method-domain-clash>` *telos-condition*

Init-options

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

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

Remarks

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

7.5. Specializing Methods

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

`call-next-method` *special form*

Syntax

`(call-next-method)`

Result

The result of calling the next most specific applicable method.

Remarks

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

`next-method-p` *special form*

Syntax

`(next-method-p)`

Result

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

`<no-next-method>` *telos-condition*

Init-options

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

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

Remarks

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

7.6. Method Lookup and Generic Dispatch

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

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

1. which methods attached to the generic function are applicable to the supplied arguments, and
2. the linear order of the applicable methods with respect to classes of the arguments and the argument precedence order.

A class C_1 is called *more specific* than class C_2 *with respect to* C_3 iff C_1 appears before C_2 in the class precedence list (CPL) of C_3[2].

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

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

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

A method $\mathtt{M}_1$ with the domain $D_{11} \times \ldots \times D_{1m}[\times$ `<list>`$]$ is *more specific* than a method $\mathtt{M}_2$ with the domain $D_{21} \times \ldots \times D_{2m}[\times$ `<list>`$]$ *with respect to* the arguments $a_1 \ldots a_m[a_{m+1} \ldots a_n]$ iff there exists an $i \in (1 \ldots m)$ so that D_{1i} is more specific than D_{2i} with respect to C_i, the class of a_i, and for all $j = 1 \ldots i-1$, D_{2j} is *not* more specific than D_{1j} with respect to C_j, the class of a_j.

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

1. Select the methods applicable to $a_1 \ldots a_m[a_{m+1} \ldots a_n]$ from all methods attached to `f`.
2. Sort the applicable methods $\mathtt{M}_1 \ldots \mathtt{M}_k$ into decreasing order of specificity using left to right argument precedence order to resolve otherwise equally specific methods.
3. If `call-next-method` appears in one of the method bodies, make the sorted list of applicable methods available for it.
4. Apply the most specific method on $a_1 \ldots a_m[a_{m+1} \ldots a_n]$.
5. Return the result of the previous step.

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

7.7. Creating and Initializing Objects

Objects can be created by calling

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

`make` *function*

Arguments

class: The class of the object to create.

`key`$_1$ obj_1 ... `key`$_n$ obj_n: Initialization arguments.

Result

An instance of *class*.

Remarks

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

`allocate` *function*

Arguments

class: A structure class.

initlist: A list of initialization arguments.

Result

A new uninitialized direct instance of the first argument.

Remarks

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

`initialize` *generic function*

Generic Arguments

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

initlist: The list of initialization arguments.

Result

The initialized object.

Remarks

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

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

`initialize` *method*

Specialized Arguments

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

initlist: The list of initialization arguments.

Result

The initialized object.

Remarks

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

1. Checks if the supplied initargs are legal and signals an error otherwise. Legal initargs are those specified in the class definition directly or inherited from a superclass. An initarg may be specified as a slot option or as a class option.
2. Initializes the slots of the object according to the initarg, if supplied, or according to the most specific `initform`, if specified. Otherwise, the slot remains "unbound".

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

7.8. Accessing Slots

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

8. Concurrency

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

A thread is allocated and initialized, by calling `make`. The initarg of a thread specifies the initial function, which is where execution starts the first time the thread is dispatched by the scheduler. In this discussion four states of a thread are identified: *new*, *running*, *aborted* and *finished*. These are for conceptual purposes only and a EuLisp program cannot distinguish between new and running or between aborted and finished. (Although accessing the result of a thread would permit such a distinction retrospectively, since an aborted thread will cause a condition to be signalled on the accessing thread and a finished thread will not.) In practice, the running state is likely to have several internal states, but these distinctions and the information about a thread's current state can serve no useful purpose to a running program, since the information may be incorrect as soon as it is known. The transitions between these states are summarized in Figure 4. The initial state of a thread is new. The union of the two final states is known as *determined*. Although a program can find out whether a thread is determined or not by means of `wait` with a timeout of `t` (denoting a poll), the information is only useful if the thread has been determined.

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

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

The actions of a thread can be influenced externally by `signal`. This function registers a condition to be signalled no later than when the specified thread is rescheduled for execution—when `thread-reschedule` returns. The condition must be an instance of `thread-condition`. Conditions are delivered to the thread in order of receipt. This ordering require-

Figure 4: State diagram for threads

ment is only important in the case of a thread sending more than one signal to the same thread, but in other circumstances the delivery order cannot be verified. A `signal` on a determined thread has no discernable effect on either the signalled or signalling thread unless the condition is not an instance of `<thread-condition>`, in which case an error is signalled on the signalling thread. See also Section 9.

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

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

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

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

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

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

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

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

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

7. The behaviour of i/o on the same stream by multiple threads is undefined unless it is mediated by explicit locks.

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

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

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

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

8.1. Threads

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

`<thread>` *class*

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

Init-options

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

`threadp` *function*

Arguments

object: An object to examine.

Result

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

`thread-reschedule` *function*

This function takes no arguments.

Result

The result is `()`.

Remarks

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

See also:

`thread-value`, `signal` and Section 9 for details of conditions and signalling.

`current-thread` *function*

This function takes no arguments.

Result

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

`thread-start` *function*

Arguments

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

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

Result

The thread which was supplied as the first argument.

Remarks

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

`thread-value` *function*

Arguments

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

Result

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

Remarks

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

See also:

`thread-reschedule`, `signal`.

`wait` *method*

Specialized Arguments

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

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

Result

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

See also:

`wait` and `ticks-per-second` (Section 9).

`<thread-condition>` *condition*

Init-options

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

Remarks

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

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

Init-options

`continuation` *continuation*: A continuation.

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

Remarks

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

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

Init-options

`thread` *thread*: A thread.

Remarks

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

8.2. Locks

The defined name of this module is `lock`.

`<lock>` *class*

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

`lockp` *function*

Arguments

object: An object to examine.

Result

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

`lock` *function*

Arguments

lock: the lock to be acquired.

Result

The lock supplied as argument.

Remarks

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

See also:

`unlock` and Section 9 for details of conditions and signalling.

`unlock` *function*

Arguments

lock: the lock to be released.

Result

The lock supplied as argument.

Remarks

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

See also:

`lock`.

9. Conditions

The defined name of this module is `condition`.

The condition system was influenced by the Common Lisp error system [13] and the Standard ML exception mechanism. It is a simplification of the former and an extension of the latter. Following standard practice, this text defines the actions of functions in terms of their normal behaviour. Where an exceptional behaviour might arise, this has been defined in terms of a condition. However, not all exceptional situations are errors. Following

Pitman, we use *condition* to be a kind of occasion in a program when an exceptional situation has been signalled. An error is a kind of condition—error and condition are also used as terms for the objects that represent exceptional situations. A condition can be signalled continuably by passing a continuation for the resumption to signal. If a continuation is not supplied then the condition cannot be continued.

These two categories are characterized as follows:

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

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

A condition class is defined using `defcondition` (see Section 9.1). The definition of a condition causes the creation of a new class of condition. A condition is signalled using the function `signal`, which has two required arguments and one optional argument: an instance of a condition, a resume continuation or the empty list—the latter signifying a non-continuable signal—and a thread. A condition can be handled using the special form `with-handler`, which takes a function—the handler function—and a sequence of forms to be protected. The initial condition class hierarchy is shown in Figure 5.

9.1. Condition Classes

`<condition>` *class*

Init-options

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

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

Figure 5: Level-0 initial condition class hierarchy

Remarks

The class which is the superclass of all condition classes.

`<execution-condition>` *condition*

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

`<environment-condition>` *condition*

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

`conditionp` *function*

Arguments

object: An object to examine.

Result

Returns *obj* if *obj* is an instance of `<condition>`, otherwise `()`.

`initialize` *method*

Specialized Arguments

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

initlist : A list of initialization options as follows:

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

Result

A new, initialized condition.

Remarks

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

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

`defcondition` *defining form*

Syntax

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

Arguments

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

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

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

Remarks

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

9.2. Condition Handling

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

`signal` *function*

Arguments

condition: The condition to be signalled.

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

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

Result

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

Remarks

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

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

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

See also:

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

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

Init-options

`condition` *condition*: A condition.

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

`with-handler` *special form*

Syntax

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

Arguments

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

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

Result

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

Remarks

A `with-handler` form is evaluated in four 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 first two arguments given to `signal`. The *handler-function* is executed in the dynamic extent of the call to `signal`. However, any `signal`s occurring during the execution of *handler-function* are dealt with by the 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 recognized 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 recognized by the handler function and might be preceded by some corrective action before some kind of transfer of control. This is termed *accepting* the condition.

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

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

Figure 6: Illustration of handler actions

Examples

An illustration of the use of all three cases is given in Figure 6.

See also:

`signal`.

`error` *function*

`cerror` *function*

Arguments

error-message: a string containing relevant information.

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

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

Result

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

Remarks

The `cerror` and `error` functions signal continuable and non-continuable errors, respectively. Each calls `signal` with an instance of a condition of *condition-class* initialized from *init-options*, 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 signalled continuably.

10. Expressions, Definitions and Control Forms

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

10.1. Atomic Expressions

`constant` *syntax*

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

Examples

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

`defconstant` *defining form*

Syntax

`(defconstant` *identifier form*`)`

Arguments

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

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

Remarks

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

`nil` `<null>`

Remarks

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

`t` `<symbol>`

Remarks

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

`symbol` *syntax*

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

`deflocal` *defining form*

Syntax

`(deflocal` *identifier form*`)`

Arguments

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

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

Remarks

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

See also:

`setq`.

10.2. Literal Expressions

`quote` *special form*

Syntax

(`quote` *datum*)

Arguments

datum: the *datum* to be quoted.

Result

The result is *datum*.

Remarks

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

10.3. Functions: creation, definition and application

`lambda` *special form*

Syntax

(`lambda` *lambda-list body*)

Arguments

lambda-list: The parameter list of the function conforming to the syntax specified in Table 7.

body: A sequence of forms.

Result

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

Remarks

The function construction operator is `lambda`. Access to the lexical environment of definition is guaranteed. The syntax of *lambda-list* is defined by the grammar in Table 7.

Table 7: Lambda list syntax

lambda-list	::=	*identifier* | *simple-list* | *rest-list*
simple-list	::=	(*identifier**)
rest-list	::=	(*identifier*$^+$. *identifier*)

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 indefinite extent. It is a static error if the same identifier appears more than once in a *lambda-list*. It is an error to modify *rest-list*.

`defmacro` *syntax*

Syntax

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

Arguments

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

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

body: A sequence of forms.

Remarks

The `defmacro` form defines a function named by *macro-name* and stores the definition as the top-lexical binding of *macro-name*. The interpretation of the *lambda-list* is as defined for `lambda` (see Table 7).

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

See also:

`lambda`.

Table 8: `defun` rewrite rules

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

`defun` *syntax*

Syntax

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

Arguments

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

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

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

body: A sequence of forms.

Remarks

The `defun` form defines a function named by *function-name* and stores the definition (i) as the top-lexical binding of *function-name* or (ii) as the setter function of *function-name.* The interpretation of the *lambda-list* is as defined for `lambda`.

The rewrite rules for `defun` are given in Table 8.

`function call` *syntax*

Syntax

`(`*operator operand**`)`

Arguments

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

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

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

Result

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

Remarks

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

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

See also:

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

`<invalid-operator>` *execution-condition*

Init-options

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

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

Remarks

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

`apply` *function*

Syntax

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

Arguments

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

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

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

Result

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

See also:

function call, `<invalid-operator>`.

10.4. Assignments

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

`setq` *special form*

Syntax

(`setq` *identifier form*)

Arguments

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

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

Result

The result is the value of *form*.

Remarks

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

`setter` *function*

Arguments

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

Result

The *writer* corresponding to *reader*.

Remarks

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

See also:

`defgeneric`, `defmethod`, `defstruct`, `defun`.

`<no-setter>` *execution-condition*

Init-options

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

Remarks

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

`<cannot-update-setter>` *execution-condition*

Init-options

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

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

Remarks

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

See also:

`setter`.

10.5. Conditional Expressions

`if` *special form*

Syntax

`(if` *antecedent consequent alternative*`)`

Arguments

antecedent: A form.

consequent: A form.

alternative: A form.

Result

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

Remarks

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

`cond` *macro*

Syntax

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

Remarks

The `cond` macro provides a convenient syntax for collections of *if-then-elseif…else* expressions. The rewrite rules for `cond` are given in Table 9.

Table 9: `cond` rewrite rules

`(cond)`	≡	`()`
`(cond (`*antecedent*`) ...)`	≡	`(or` *antecedent* `(cond ...))`

`(cond`
`(`*antecedent*$_1$`)`
`(`*antecedent*$_2$ *form**`)`
`...)`
	≡	`(or` *antecedent*$_1$
		`(cond`
		`(`*antecedent*$_2$ *form**`)`
		`...))`

`(cond`
`(`*antecedent*$_1$ *form**`)`
`(`*antecedent*$_2$ *form**`)`
`...)`
	≡	`(if` *antecedent*$_1$
		`(progn` *form**`)`
		`(cond`
		`(`*antecedent*$_2$ *form**`)`
		`...))`

`and` *macro*

Syntax

`(and` *form**`)`

Remarks

The expansion of an `and` form leads to the evaluation of the sequence of *form*s from left to right. The first *form* in the sequence that evaluates to `()` stops evaluation and none of the *form*s to its right will be evaluated—that is to say, it is non-strict. The result of `(and)` is `()`. If none of the *form*s evaluate to `()`, the value of the last *form* is returned. The rewrite rules for `and` are given in Table 10.

`or` *macro*

Syntax

`(or` *form**`)`

Remarks

The expansion of an `or` form leads to the evaluation of the sequence of *form*s from left to right. The value of the first *form* that evaluates to *true* is the result of the `or` form and none of the *form*s to its right will be evaluated—that is to say, it is non-strict. If none of the forms evaluate to *true*, the value of the last *form* is returned. The rewrite rules for `or` are given in Table 10. Note that `x` does not occur free in any of *form*$_2$ ... *form*$_n$.

Table 10: `and` and `or` rewrite rules

`(and)`	$\equiv$	`t`
`(and` *form*`)`	$\equiv$	*form*
`(and` $form_1$ $form_2$ `...)`	$\equiv$	`(if` $form_1$ `(and` $form_2$ `...) ())`
`(or)`	$\equiv$	`()`
`(or` *form*`)`	$\equiv$	*form*
`(or` $form_1$ $form_2$ `...)`	$\equiv$	`(let ((x` $form_1$`))` `(if x x (or` $form_2$ `...)))`

10.6. Variable Binding and Sequences

`let/cc` *special form*

Syntax

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

Arguments

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

body: A sequence of forms.

Result

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

Remarks

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

Examples

An example of the use of `let/cc` is given in Figure 7. The function `path-open` takes a list of paths, the name of a file and list of options to

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

Figure 7: Example using `let/cc`

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

See also:

`block`, `return-from`.

`block` *macro*

Syntax

`(block` *identifier form**`)`

Remarks

The block expression is used to establish a statically scoped binding of an escape function. The block *variable* is bound to the continuation of the block. The continuation can be invoked anywhere within the block by using `return-from`. The *form*s are evaluated in sequence and the value of the last one is returned as the value of the block form. See also `let/cc`. The rewrite rules for `block` are given in Table 11.

Table 11: `block` and `return-from` rewrite rules

`(block` *identifier*`)`	≡	`()`
`(block` *identifier form**`)`	≡	`(let/cc` *identifier form**`)`
`(return-from` *identifier*`)`	≡	`(`*identifier* `())`
`(return-from` *identifier form*`)`	≡	`(`*identifier form*`)`

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

See also:

`return-from`.

`return-from` *macro*

Syntax

`(return-from` *identifier* `[`*form*`])`

Remarks

In `return-from`, the *identifier* names the continuation of the (lexical) `block` from which to return. `return-from` is the invocation of the continuation of the block named by *identifier*. The *form* is evaluated and the value is returned as the value of the block named by *identifier*. The rewrite rules for `return-from` are given in Table 11.

See also:

`block`.

`labels` *special form*

Syntax

`(labels ((`*identifier lambda-list body*`)*)` *labels-body*`)`

Arguments

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

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

body: A sequence of forms.

labels-body: A sequence of forms.

Result

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

`let` *macro*

Syntax

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

Remarks

The optional *identifier* denotes that the let form can be called from within its *body*. This is an abbreviation for `labels` form in which *identifier* is bound to a function whose parameters are the identifiers of the *binding*s of the `let`, whose body is that of the `let` and whose initial call passes the values of the initializing form of the *binding*s. A binding is specified by either an identifier or a two element list of an identifier and an initializing form. All the initializing forms are evaluated in order from left to right in the current environment and the variables named by the identifiers in the *binding*s are bound to new locations holding the results. Each form in *body* is evaluated in order in the environment extended by the above bindings. The result of evaluating the last form in *body* is returned as the result of the `let` form. The rewrite rules for `let` are given in Table 12.

`let*` *macro*

Syntax

(`let*` (*binding**) *body*)

Table 12: `let` rewrite rules

(let () $form^*$)	$\equiv$	(progn $form^*$)
(let ((id_1 $form_1$)	$\equiv$	((lambda (id_1 id_2 id_3 ...)
(id_2 $form_2$)		$form^*$)
id_3		$form_1$ $form_2$ () ...)
...)		
$form^*$)		
(let id_0	$\equiv$	(labels
((id_1 $form_1$)		((id_0 (id_1 id_2 ...)
id_2		$form^*$))
...)		(id_0 $form_1$ () ...))
$form^*$)		

Table 13: `let*` rewrite rules

(let* () $form^*$)	$\equiv$	(progn $form^*$)
(let* ((var_1 $form_1$)	$\equiv$	(let ((var_1 $form_1$))
(var_2 $form_2$)		(let* ((var_2 $form_2$)
var_3		var_3
...)		...)
$form^*$)		$form^*$))

Remarks

A *binding* is specified by a two element list of a variable and an initializing form. The first initializing form is evaluated in the current environment and the corresponding variable is bound to a new location containing that result. Subsequent bindings are processed in turn, evaluating the initializing form in the environment extended by the previous binding. Each form in *body* is evaluated in order in the environment extended by the above bindings. The result of evaluating the last form is returned as the result of the `let*` form. The rewrite rules for `let*` are given in Table 13.

`progn` *special form*

Syntax

(`progn` $form^*$)

Arguments

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

Result

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

Remarks

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

`unwind-protect` *special form*

Syntax

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

Arguments

protected-form: A form.

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

Result

The value of *protected-form*.

Remarks

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

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

Figure 8: Interaction of `unwind-protect` with non-local exits

Examples

The code fragment in Figure 8 illustrates both the use of `unwind-protect` and of a difference between the semantics of EuLisp and some other Lisps. Stepping through the evaluation of this form: `k1` is bound to the continuation of its `let/cc` form; a recursive function named `loop` is constructed, `loop` is called from the body of the `labels` form; `k2` is bound to the continuation of its `let/cc` form; `unwind-protect` calls `k1`; the after forms of `unwind-protect` are evaluated in order; `k2` is called; `loop` is called; etc.. This program loops indefinitely.

10.7. Events

The defined name of this module is `event`.

`wait` *generic function*

Generic Arguments

obj: An object.

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

Result

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

Remarks

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

1. A condition associated with *obj* returns true;
2. *timeout* time units elapse;
3. A condition is raised by another thread on this thread.

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

A `timeout` argument of `()` or zero denotes a polling operation. A `timeout` argument of `t` denotes indefinite blocking (cases 1 or 3 above). A `timeout` argument of a non-negative integer denotes the minimum number of time units before timeout. The number of time units in a second is given by the implementation-defined constant `ticks-per-second`.

Examples

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

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

See also:

threads (section 8.1), streams (section A.13).

`ticks-per-second` `<double-float>`

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

10.8. Quasiquotation Expressions

`quasiquote` *macro*

Syntax

`(quasiquote` *skeleton*`)` or `` ` ``*skeleton*

Remarks

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

`unquote` *syntax*

Syntax

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

Remarks

See `unquote-splicing`.

`unquote-splicing` *syntax*

Syntax

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

Remarks

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

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

Examples

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

11. History and Acknowledgements

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

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

The ancestors of EuLisp (in alphabetical order) are Common Lisp[17], InterLISP[19], Le-Lisp [4], Lisp/VM [1], Scheme [6], and T [14] [16]. Thus, the authors of this report are pleased to acknowledge both the authors of the manuals and definitions of the above languages and the many who have dissected and extended those languages in individual papers. The various papers on Standard ML [11] and the draft report on Haskell [8] have also provided much useful input.

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

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

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

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

References

1. Alberga, C.N., Bosman-Clark, C., Mikelsons, M., Van Deusen, M., and Padget, J.A. Experience with an Uncommon Lisp. In *Proceedings of 1986 ACM Symposium on Lisp and Functional Programming*, ACM Press, New York (1986) 39–53. Also available as IBM Research Report RC-11888.

2. Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S.E, Kiczales, G., and Moon, D.A. Common Lisp Object System Specification. *SIGPLAN Notices*, 23, 9 (September 1988).

3. Bretthauer, H. and Kopp, J. *The Meta-Class-System MCS. A Portable Object System for Common Lisp. –Documentation–*. Arbeitspapiere der GMD 554, Gesellschaft für Mathematik und Datenverarbeitung

(GMD), Sankt Augustin (July 1991).

4. Chailloux, J., Devin, M., and Hullot, J-M. LELISP: A Portable and Efficient Lisp System. In *Proceedings of 1984 ACM Symposium on Lisp and Functional Programming*, ACM Press, New York (1984) 113–122.

5. Chailloux, J., Devin, M., Dupont, F., Hullot, J-M., Serpette, B., and Vuillemin, J. *Le-Lisp de l'INRIA, Version 15.2, Manuel de référence.* INRIA, Rocquencourt (1987).

6. Clinger, W. and Rees, J.A. (editors). The Revised^3 Report on Scheme. *SIGPLAN Notices*, 21, 12 (December 1986).

7. Cointe, P. Metaclasses are First Class: the ObjVlisp model. In *Proceedings of OOPSLA '87*, ACM Press (December 1987) 156–167. published as SIGPLAN Notices, Vol 22, No 12.

8. Hudak, P. and Wadler, P. (editors). Report on the Functional Programming Language Haskell. *SIGPLAN Notices*, 27, 7 (May 1992).

9. Lang, K.J. and Pearlmutter, B.A. Oaklisp: An Object-Oriented Dialect of Scheme. *Lisp and Symbolic Computation*, 1, 1 (June 1988) 39–51.

10. MacQueen, D. Modules for Standard ML. In *Proceedings of 1984 ACM Symposium on Lisp and Functional Programming*, ACM Press, New York (1984) 198–207.

11. Milner, R. and et al. *Standard ML.* Technical Report, Laboratory for the Foundations of Computer Science, University of Edinburgh (1986).

12. Padget, J.A. et al. Desiderata for the Standardisation of Lisp. In *Proceedings of 1986 ACM Symposium on Lisp and Functional Programming*, ACM Press, New York (1986) 54–66.

13. Pitman, K.M. An Error System for Common Lisp. (1988). ISO-IEC JTC1 SC22 WG16 document N24.

14. Rees, J.A. *The T Manual.* Technical Report, Yale University (1986).

15. Shalit, A. *Dylan, an object-oriented dynamic language.* Apple Computer Inc. (1992).

16. Slade, S. *The T Programming Language, a Dialect of Lisp.* Prentice-Hall (1987).

17. Steele Jr, G.L. *Common Lisp the Language.* Digital Press (1984). Second edition, Digital Press, 1990.

18. Stoyan, H. et al. Towards a Lisp Standard. In *Proceedings of 1986 European Conference on Artificial Intelligence* (1986) 46–52.

19. Teitelman, W. *The Interlisp Reference Manual.* Xerox Palo Alto Research Center (1978).

A. Level-0 Module Library

This part of the definition contains entries for each of the remaining modules comprising level-0 of EuLisp. Most of them export a class and operations on that class. The rest export functions implementing useful operations, such as copying, comparison and conversion. This section has purposely been highly compressed, since there is little that is very different from other Lisps, although each of these is documented in detail in the full version of the definition.

A.1. Characters

Character comparison is supported via methods on `<` and other such generic functions. However, it is only meaningful to compare a lower case character with another lower case, upper with upper and digit with digit. All other combinations are undefined. There is primitive support for the input of two-byte characters by specifying the character's index position in the current character set, for example: `#\x0` and `#\xabcd`, which denote, respectively, the characters at position 0 and at position 43981 in the current character set.

A.2. Collections

Exports a set of generic functions, the methods for which are defined in various other modules, to provide a set of operators for all the predefined aggregates (`<list>`, `<string>`, `<table>`, `<vector>`). The specification of collections is still being finalized, but has been influenced by the operations outlined in Dylan [15].

A.3. Comparison

Exports the functions `eq`, `eql`, `binary=`, `binary<` and `equal`. Both `binary=` and `binary<` are generic but the domain of the former is restricted to subclasses of `<number>`. The function `equal` is also generic.

A.4. Conversion

The function `convert` takes an object and a class as its argument and returns a direct or indirect instance of class which is the result of converting the object to the class. This works by associating a (generic) converter function with each target class. The methods attached to this generic function for each source class are responsible for the conversion of the object to an instance of the target class. The syntax of `defgeneric` and `defmethod` are both extended to help with the definition of converter functions and methods.

A.5. Copying

Exports two generic functions called `deep-copy` and `shallow-copy` to which the class-specific modules add methods.

A.6. Double floats

Defines methods on the standard arithmetic operators, functions for rounding and maximum and minimum positive and negative values.

A.7. Elementary functions

Exports the same set of trigonometric functions as provided by ISO-C.

A.8. Formatted input-output

Exports a function named `format` which takes the same parameters as Common Lisp's format, but accepts only a subset of its formatting directives.

A.9. Fixed precision integers

Defines methods on the standard arithmetic operators.

A.10. The empty list

Exports the class `<null>` and the function `null`. Note: this class is a subclass of `<list>` and disjoint from `<cons>`.

A.11. Numbers

Exports the abstract class `<number>` and the generic functions for the standard arithmetic operators.

A.12. Pairs

Exports the class `<cons>` its constructor `cons` and accessors `car` and `cdr`, and the copying functions `copy-list` and `copy-tree`. Note: this class is a subclass of `<list>` and disjoint from `<null>`.

A.13. Streams

Currently, the only defined stream class is `<file-stream>`. Input is via the function `read` and output via `prin` and `write`, which call generic counterparts to do the actual output operations.

A.14. Strings

Exports the class `<string>` which is largely indistinguishable from any other Lisp.

A.15. Symbols

Almost the same syntax as for Common Lisp, except that case is significant.

A.16. Tables

Exports the class `<table>` which provides a key to value mapping similar to that in most other Lisps.

A.17. Vectors

The class `<vector>` corresponds to Common Lisp's simple vector type, that is, there are no displaced arrays, nor adjustable arrays.

B. Level-1 Extensions

The part gives a brief overview of level-1 of EuLisp concentrating on the facilities of the metaobject protocol. These items are described in more detail in the full version of the definition.

B.1. Syntax Extensions

B.1.1. Classes

The `defclass` form extends the `defstruct` form of level-0 (see Section 7.3) to define any kind of class. It has the following syntax:

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

It differs from `defstruct` in allowing multiple superclasses and additional

Table 14: `defclass` syntax (level-1)

class-name	::=	*identifier*
superclass-name	::=	{*the name of a subclass of* `<object>`}
slot-spec	::=	*slot-name* | `(`*slot-name slot-option**`)`
slot-name	::=	*identifier*
slot-option	::=	`initarg` *identifier*
	|	`initform` *form*
	|	`reader` *identifier*
	|	`writer` *identifier*
	|	`accessor` *identifier*
	|	*identifier expression*
class-option	::=	`initargs` `(`*identifier**`)`
	|	`constructor` *constructor-spec*
	|	`predicate` *identifier*
	|	`class` *class-name*
	|	*identifier expression*

slot and class options as shown in Table 14.

The value of the `class` *class-option* specifies the class of the new class, whose default is `<class>`. The value of a non-standard slot or class option (*identifier expression*) is evaluated in the lexical and dynamic environment of `defclass` and passed to `make` of the slot description or class, respectively. This option is used for new slot descriptions or metaclasses which need extra information beside the standard options.

B.1.2. Generic Functions

The syntax of `generic-lambda` is an extension of the level-0 syntax allowing additional init-options (see Table 15):

`(generic-lambda` *gen-lambda-list level-1-init-option**`)`

The additional options include the specification of the class of the new generic function, which defaults to `<generic-function>`, the class of all methods, which defaults to `<method>`, and non-standard options. The latter are evaluated in the lexical and dynamic environment of `defgeneric` and passed to `make` of the generic function as additional initialization arguments.

The `defgeneric` defining form extends the one of level-0 in the same way as `generic-lambda` is extended. Thus, the `defgeneric` form can be rewritten as shown in Table 6, except that *level-0-init-option*s are replaced by *level-1-init-option*s as per Table 15.

Table 15: `generic-lambda` syntax (level-1)

level-1-init-option	::=	*level-0-init-option*
	|	`class` *gf-class-name*
	|	`method-class` *method-class-name*
	|	`method` *level-1-method-description*
	|	*identifier expression*
gf-class-name	::=	{*the name a subclass of* `<generic-function>`}
method-class-name	::=	{*the name of a subclass of* `<method>`}
level-1-method-description	::=	`(`*method-init-option** *spec-lambda-list form**`)`
method-init-option	::=	`class` *method-class-name*
	|	*identifier expression*

B.1.3. Methods

The `method-lambda` form defines and returns an unattached method. It is the counterpart to `generic-lambda` and its syntax is:

`(method-lambda` *method-init-option** *spec-lambda-list form**`)`

The additional *method-init-option*s includes `class`, for specifying the class of the method to be defined, and non-standard options, which are evaluated in the lexical and dynamic environment of `method-lambda` and passed to `initialize` of that method.

The `defmethod` form of level-1 extends that of level-0 to accept *method-init-options*. The syntax is:

`(defmethod` *gf-name method-init-option** *spec-lambda-list form**`)`

B.2. The Metaobject Protocol

B.2.1. System Defined Classes

The basic classes of level-1 of EuLisp are shown in Figure 9, which extend the class hierarchy found in Section 7.1. The class of each class is shown after it enclosed in square brackets. It may be a direct or indirect instance of that class.

Standard classes are not redefinable and support single inheritance only. General multiple inheritance or mixin inheritance can be provided by extensions. Nor is it possible to use a class as a superclass which is not defined

```
<object> [<abstract-class>]
   <class> [<class>]
      <abstract-class> [<class>]
      <function-class> [<class>]
   <slot-description> [<abstract-class>]
      <local-slot-description> [<class>]
   <function> [<abstract-class>]
      <generic-function> [<function-class>]
   <method> [<class>]
```

Figure 9: Class Hierarchy

at the time of class definition. Again, such forward reference facilities can be provided by extensions.

Standard classes support local slots only. Shared slots can be provided by extensions. The class `<slot-description>` is the abstract class of all slot descriptions.

B.2.2. Introspection

The minimal information associated with an object is its class. The corresponding introspection function `class-of` is defined for all objects.

The minimal information associated with a class metaobject is: The *class precedence list*, ordered most specific first, beginning with the class itself, the list of (effective) *slot descriptions*, the list of (effective) *initargs*, and the *instance size.*

Access to this information is provided by functions which take a class as their only argument:

```
class-precendence-list
class-slot-descriptions
class-initargs
class-instance-size
```

The minimal information associated with a slot description metaobject is: the *name*, which is required to perform inheritance computations, the *initfunction*, called by default to compute the initial slot value when creating a new instance, the *reader*, which is a function to read the corresponding slot value of an instance, the *writer*, which is a function to write the corresponding slot of an instance, and the *initarg*, which is a symbol to access the value which can be supplied to a `make` call in order to initialize the corresponding slot in a newly-created object.

The information associated with slot descriptions can be accessed via operations which take a slot description as their only argument. These functions are:

```
slot-description-name
slot-description-initfunction
slot-description-slot-reader
slot-description-slot-writer
slot-description-initarg
```

The minimum information associated with a generic function metaobject is: the *domain*, restricting the domain of each added method to a subdomain, the *method class*, restricting each added method to be an instance of that class, the list of all attached *methods*, the *method look-up function* used to collect and sort the applicable methods for a given domain, and the *discriminating function* used to perform the generic dispatch.

The associated introspection operations which take a generic function as their only argument are:

```
generic-function-domain
generic-function-method-class
generic-function-methods
generic-function-method-lookup-function
generic-function-discriminating-function
```

The minimal information associated with a method metaobject is the *domain*, which is a list of classes. The associated introspection operation which take a method as its only argument is `method-domain`.

B.2.3. Class Initialization and Inheritance

The init-options for classes are:

```
direct-superclasses
direct-slot-descriptions
direct-initargs
```

The init-options for slot descriptions are:

```
name
initfunction
reader
writer
initarg
```

The default initialization of a class takes place in four steps. First compatibility of the direct superclasses is checked. Then the logical inheritance computations are done. This includes the class precedence list, the initargs, the effective slot descriptions, and the instance size. The third step computes the new slot accessors and ensures all (new and inherited) accessors work correctly on instances of the new class. Finally, the results are made

```
compatible-superclasses-p cl direct-superclasses → boolean
  compatible-superclass-p cl superclass → boolean
compute-class-precedence-list cl direct-superclasses → (cl*)
compute-inherited-initargs cl direct-superclasses → ((initarg*)*)
compute-initargs cl direct-initargs inh-initargs → (initarg*)
compute-inherited-slot-descriptions cl direct-superclasses → ((sd*)*)
compute-slot-descriptions cl slot-specs inh-sds → (sd*)
  either
  compute-defined-slot-description cl slot-spec → sd
    compute-defined-slot-description-class cl slot-spec → sd-class
  or
  compute-specialized-slot-description cl inh-sds slot-spec → sd
    compute-specialized-slot-description-class
                                  cl inh-sds slot-spec → sd-class
compute-instance-size cl eff-sds → integer
compute-and-ensure-slot-accessors cl eff-sds inh-sds → (sd*)
  compute-slot-reader cl sd eff-sds → function
  compute-slot-writer cl sd eff-sds → function
  ensure-slot-reader cl sd eff-sds reader → function
    compute-primitive-reader-using-slot-description
                                    sd cl eff-sds → function
      compute-primitive-reader-using-class cl sd eff-sds → function
  ensure-slot-writer cl sd eff-sds writer → function
    compute-primitive-writer-using-slot-description
                                    sd cl eff-sds → function
      compute-primitive-writer-using-class cl sd eff-sds → function
```

Figure 10: Initialization Call Structure

accessible by the class introspection operations. The basic call structure of the first three steps is laid out in Figure 10. It uses the following abbreviations: cl – class, sd – slot-description, inh – inherited, eff – effective. Note that it is implementation-defined whether any of these steps are performed completely at initialization time or lazily when needed.

The generic function `compatible-superclasses-p` checks the compatibility between class and its *direct-superclasses* by calling `compatible-superclass-p` for class and each of its superclasses. `compute-class-precedence-list` returns a list of classes which represents the linearized inheritance hierarchy of class *cl* and the given list of *direct-superclasses*, beginning with *cl* and ending with `<object>`. `compute-initargs` computes and returns all legal initargs for a class. Therefore, it takes the result of `compute-inherited-initargs` which returns a list of the legal initarg lists of the (direct) superclasses. The computation of the effective slot descriptions is done by `compute-slot-descriptions` which takes the class *cl*, the direct slot specifications *slot-specs*, and the inherited slot descriptions *inh-sds* as its arguments. The latter are computed by `compute-inherited-slot-descriptions` which takes the class *cl* and its *direct-superclasses* and returns a list containing lists of inherited slot descriptions. `compute-slot-descriptions` distinguishes between the computation of specialized and newly-defined slot descriptions and calls either

`compute-specialized-slot-description`
or
`compute-defined-slot-description`

The former takes for each slot name as arguments the class *cl*, the list of inherited slot descriptions *inh-sds* and the canonicalized slot specification *slot-spec* and the latter takes *cl* and *slot-spec*. Both generic functions return a new effective slot description. They call

`compute-specialized-slot-description-class`
or
`compute-defined-slot-description-class`,

respectively, to get the class for the new effective slot description corresponding to its arguments which defaults to `<local-slot-description>`. The instance size computation is described in Section B.2.6. The third step is described as a subprotocol in the next section.

All of the default methods profit from the single inheritance assumption, but the call structure and the supplied arguments take into account that there will exist classes with different inheritance strategies.

B.2.4. Slot Accessor Computation

Rather than have a dynamic slot access protocol, Telos provides a standard protocol for computing readers and writers. Every slot description contains one reader and one writer capable of extracting and updating the corresponding slot within instances. Accessor defining slot options within a `defclass` merely bind the slot's single reader or writer to the appropriate name, therefore two readers for the same slot bound to different names will always be `eq`.

All slot accesses take place through calls to these accessor functions. No counterpart of `slot-value` found in CLOS is provided, although it can be written in terms of accessors.

Accessors are computed and updated as part of the initialization of a class calling `compute-and-ensure-slot-accessors` (see Figure 10).

A new slot, that is one which is not inherited, has a reader computed for it by `compute-slot-reader` and a writer by `compute-slot-writer`. The results which are

1. simple functions for structure classes, and
2. generic function without any methods for standard classes

are then stored in its slot description.

Inherited slots, either specialized in some way or left unchanged, take the reader and writer from the corresponding slot description objects of the superclasses.

Accessor functions computed or inherited in this way are updated to work for direct instances of a particular class calling `ensure-slot-reader` and `ensure-slot-writer` for that class:

1. For structure classes, the ensure operations need do nothing since the slot position can never change in subclasses.
2. For standard classes, they add a method to *accessor* capable of accessing the appropriate slot of direct instances of class *cl*. In cases where the slot has not "moved" relative to its position within instances of the superclasses of *cl*, there may be no need to update the accessor function.

The standard ensuring methods use a subprotocol for computing *primitive accessors* used in the new method bodies – standard functions capable of accessing a particular slot in direct instances of a given class.

These subprotocol functions are the direct counterpart of `slot-value-using-class` in CLOS and are the generic functions most commonly used to change the behaviour of slot access.

By default, `compute-primitive-reader-using-slot-description` returns the result of calling `compute-primitive-reader-using-class`. For standard classes and local slot descriptions this is a function of one argument that when applied to a direct instance of class *cl*, returns the value of the slot described by slot description *sd*. Its behaviour on instances of other classes, even subclasses of the specified class, is undefined in general. For structure classes, its behaviour on direct and indirect instances of *cl* is the same.

Similarly for writers, the result is a function of two arguments: a direct instance of class *cl* and a new value for the slot in question.

B.2.5. Method Lookup and Generic Dispatch

The default generic dispatch scheme is class-based; that is, methods are class specific. The argument precedence order is by default left-to-right. This functionality specified at level-0 in detail is provided by the following protocol. A newly-created generic function is prepared for calling by the corresponding `initialize` method. Generic functions can be created calling `make` or `generic-lambda`, while methods are created only by `method-lambda`. The only init-option which can be used within the `initialize` method called by `method-lambda` is `domain`. It specifies the list of the argument classes. The init-options for generic functions are:

```
domain
method-class
methods
```

The basic call structure inside the `initialize` method is:

`add-method` *gf method* `->` *gf*
`compute-method-lookup-function` *gf domain* `->` *function*
`compute-discriminating-function` *gf domain lookup-fn methods* `->` *function*

The (generic) function `add-method` attaches the specified methods to the generic function and its slots are initialized from the information passed in *initlist* and from the results of calling `compute-method-lookup-function` and its partner, `compute-discriminating-function` on the generic function. Note that these two functions might not be called during the call to `initialize`, and that they might be called several times for the generic function.

The generic function `add-method` adds a *method* to the generic function *gf* and returns *gf* as its result. The *method* will be taken into account when *gf* is called with appropriate arguments the next time. New methods may be defined on `add-method` for new generic function and method classes. The default method checks that the domain classes of the method are subclasses of those of the generic function, that the method is an instance of the generic function's method class, and that a method with the same domain is not already attached to the generic function. An error is signalled if any of these conditions does not hold.

NOTE — In contrast to CLOS, `add-method` does not remove a method with the same domain as the method being added.

If no error occurs, the method is added to the generic function *gf*. Depending on the kind of optimizations employed for generic dispatch, adding a method may cause the recomputation of the method lookup function and the discriminating function.

The former computes and returns a function which will be called at least once for each domain to select and sort the applicable methods by the default dispatch mechanism. New methods may be defined for this function to implement different method lookup strategies. Although only one method lookup function generating method is provided by the system, each generic function has its own specific lookup function which may vary from generic function to generic function.

The latter computes and returns a function which is called whenever the generic function is called. The returned function controls the generic dispatch. Users may define methods on this function for new generic function classes to implement alternative dispatch strategies. The default method implements the standard dispatch strategy: the generic function's methods are sorted using the function returned by `compute-method-lookup-function`, and the first is called as if by `call-method`, passing the others as the list of next methods. Note that `call-method` need not be called directly for standard generic functions. However, user-defined extensions might need `call-method` or `apply-method` to implement other generic dispatch strategies. The interfaces of these functions are:

(`call-method` *method next-methods arg**)
(`apply-method` *method next-methods arg** *args*)

The first calls *method* with the sequence of arguments *arg**. The argument *next-methods* is a list of methods which are used as the applicable method list for *args*; it is an error if this list is different from the methods which would be produced by the method lookup function of the generic function of *method*. If *method* is not attached to a generic function, its behaviour is unspecified. The *next-methods* are used to determine the next method to call when `call-next-method` is called within *method*.

`apply-method` is identical to `call-method` except that its last argument is a list whose elements are the other arguments to pass to *method*. The difference is identical to that between normal function application and `apply`.

B.2.6. Low Level Allocation Primitives

The high level allocation construct is the generic function `allocate` which takes a class and an initlist as arguments. At level-0, it was specified just as a function. At level-1 it is generic, and, thus, extensible by the user. In order to implement new allocation methods portably low level primitives are necessary. Examples requiring this are persistent objects, or the `change-class` functionality, or redefinable classes with automatically updatable instances, etc. The primitives are defined in such a way that objects cannot be destroyed unintentionally. The protocol should be both secure and efficient.

The operations are:

(`primitive-allocate` *class*) → *primitive-allocated-object*
(`primitive-class-of` *primitive-allocated-object*) → *class*
(`primitive-ref` *primitive-allocated-object index*) → *value*
(`(setter primitive-class-of)` *primitive-allocated-object new-class*)
(`(setter primitive-ref)` *primitive-allocated-object index new-value*)

In order to make this interface work, the class initialization protocol is extended by a function `class-instance-size` which returns the value computed by the generic function `compute-instance-size` and stored once for each class. That means `class-instance-size` always returns the same value for a particular class. The default `compute-instance-size` method returns the number of local slot descriptions in a class. The function `primitive-allocate` uses the result of `class-instance-size` applied on its argument to create an object of that size. Thus, there is no way to create instances of a class with the wrong size. The index passed to `primitive-ref` must be a non-negative fixed precision integer smaller than the corresponding instance size. `(setter primitive-class-of)` checks that the results of `class-instance-size` on the old and the new classes are equal. Otherwise, an error is signalled. An error is signalled if either of `primitive-class-of`, `primitive-ref` or their setters are applied on objects not created by `primitive-allocate`, or if `primitive-allocate` is called on a direct instance of a system-defined metaclass.

Thus, due to the above restrictions type inference is safely applicable to primitive allocated objects.

NOTE — The `change-class` module can be implemented reducing the semantic difficulties of CLOS.

C. Glossary

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

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

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

defining form: Any form or any *macro expression* expanding into a form whose operator is one of:

`defclass`, `defcondition`, `defconstant`, `defgeneric`, `deflocal`, `defmacro`, `defstruct`, `defun`, or `defvar`.

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

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.*

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

function: A function is one of *continuation*, *simple function* or *generic function.*

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

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

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 *direct subclass* relations between the nodes. This graph has a distinguished root, the *class* `<object>`, which is a *superclass* of every *class.*

inherited slot description: A *slot description* is inherited for a *class*$_1$ if the *slot description* is defined for another *class*$_2$ which is a direct or indirect *superclass* of *class*$_1$.

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

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 *class.*

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.*

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

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

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.

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

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

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

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

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

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.

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

simple function: A function comprises at least: an expression, a set of identifiers, which occur in the expression, called the parameters and the closure of the expression with respect to the *lexical environment* in which it occurs, less the parameter identifiers. Note: this is not a definition of the class `<simple-function>`.

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

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

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.

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

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

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

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

subclass: The behaviour and structure defined by a class $class_1$ are inherited by a set of *classes* which are termed *subclasses* of $class_1$. A *subclass* can be either *direct* or *indirect* or itself.

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

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

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

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