Eclector User's Manual

Table of Contents

1 Introduction 1

2 External protocols 2

2.1 Packages 2
2.1.1 Package for basic features 2
2.1.2 Package for ordinary reader features 2
2.1.3 Package for readtable features 2
2.1.4 Package for parse result construction features 2
2.1.5 Package for CST features 2
2.2 Basic features 2
2.3 Ordinary reader features 4
2.3.1 Common Lisp reader compatible interface 4
2.3.2 Reader behavior protocol 5
2.3.3 Labeled objects and references 11
2.3.4 S-expression creation and quasiquotation 17
2.3.5 Readtable initialization 18
2.4 Readtable features 19
2.5 Parse result construction features 19
2.6 CST reader features 22

3 Recovering from errors 23

3.1 Error recovery features 23
3.2 Recoverable errors 23
3.3 Potential problems 24

4 Side effects 25

4.1 Potential side effects for the default client 25
4.1.1 Symbols and packages (default client) 25
4.1.2 Read-time evaluation (default client) 25
4.1.3 Standard reader macros (default client) 25
4.2 Potential side effects for non-default clients 26
4.2.1 Symbols and packages 26
4.2.2 Read-time evaluation 26
4.2.3 Structure instance creation 26
4.2.4 Circular structure 26
4.2.5 Standard reader macros 26

5 Interpretation of unclear parts of the specification 27

5.1 Interpretation of Sharpsign C and Sharpsign S 27
5.2 Interpretation of Backquote and Sharpsign Single Quote 28
5.3 Circular objects and custom reader macros 28

Concept index **30**

Function and macro and variable and type index . . **31**

1 Introduction

Eclector is a portable, implementation-independent version of the Common Lisp function `read`, a corresponding readtable and a quasiquotation facility. As opposed to existing implementation-specific versions of `read`, Eclector uses generic functions to allow clients to customize the exact behavior, such as the interpretation of tokens.

Another unusual feature of Eclector is its ability to, at the discretion of the client, recover from many syntax errors, continue reading and return a result that somewhat resembles what would have been returned in case the syntax had been valid.

Furthermore, Eclector can be used as a *source tracking* reader, which is accomplished through a mode of operation that produces *parse results* which wrap the Common Lisp expressions in objects that can also contain information about the positions in the source code of those expressions. One example of such parse results are *concrete syntax trees*[1].

[1] See: `https://github.com/s-expressionists/Concrete-Syntax-Tree`

2 External protocols

2.1 Packages

2.1.1 Package for basic features

The package for basic features such as customizable source location construction is named `eclector.base`. Although this package does not shadow any symbol in the `common-lisp` package, we still recommend the use of explicit package prefixes to refer to symbols in this package.

2.1.2 Package for ordinary reader features

The package for ordinary reader features is named `eclector.reader`. To use features of this package, we recommend the use of explicit package prefixes, simply because this package shadows and exports names that are also exported from the `common-lisp` package. Importing this package will likely cause conflicts with the `common-lisp` package otherwise.

2.1.3 Package for readtable features

The package for readtable-related features is named `eclector.readtable`. To use features of this package, we recommend the use of explicit package prefixes, simply because this package shadows and exports names that are also exported from the `common-lisp` package. Importing this package will likely cause conflicts with the `common-lisp` package otherwise.

2.1.4 Package for parse result construction features

The package for features related to the creation of client-defined parse results is named `eclector.parse-result`. To use features of this package, we recommend the use of explicit package prefixes, simply because this package shadows and exports names that are also exported from the `common-lisp` package. Importing this package will likely cause conflicts with the `common-lisp` package otherwise.

2.1.5 Package for CST features

The package for features related to the creation of concrete syntax trees is named `eclector.concrete-syntax-tree`. To use features of this package, we recommend the use of explicit package prefixes, simply because this package shadows and exports names that are also exported from the `common-lisp` package. Importing this package will likely cause conflicts with the `common-lisp` package otherwise.

2.2 Basic features

In this section, symbols written without package marker are in the `eclector.base` package (see Section 2.1.1 [Package for basic features], page 2).

This package provides the mechanism that enables clients to customize the behavior of the reader. Furthermore this package provides a protocol for customizing a particular aspect of the behavior, namely the construction of source positions and source ranges. Eclector uses source positions and source ranges in signaled conditions and parse results (see Section 2.5 [Parse result construction features], page 19).

`stream-position-condition` `[eclector.base]` [Class]

This condition type is the supertype of all conditions which are signaled by Eclector functions. An instance of this condition type stores an approximate position in an input stream and an offset from that position. The condition is associated with the stream content at the designated position and offset. The position uses a representation which is controlled by the respective client by adding a method on the `source-position` generic function. The offset indicates a distance in characters which must be added to the approximate position to produce the exact position.

`stream-position` `[eclector.base]` [Generic Function]

condition

This generic function can be called by clients in order to obtain the approximate position in the input stream to which *condition* pertains. The type and interpretation of the returned object depend on the client, namely the presence of client-specific methods on the `source-position` generic function.

Applicable methods exist for all conditions of type `stream-position-condition`.

`position-offset` `[eclector.base]` [Generic Function]

condition

This generic function is called in order to compute the exact position in the input stream to which *condition* pertains by refining the approximate position obtained by calling `stream-position`. The returned value is an integer (possibly negative) which indicates the offset in characters from the approximate position to the exact position. Since the representation of the approximate position is chosen by the client, applying the offset to that position in a suitable way is also the responsibility of the client.

Applicable methods exist for all conditions of type `stream-position-condition`.

`*client*` `[eclector.base]` [Variable]

This variable is used by several generic functions which are called by `eclector.reader:read`. The default value of the variable is `nil`. Clients that want to override or extend the default behavior of some generic function of Eclector should bind this variable to some standard object and provide a method on that generic function, specialized to the class of that standard object.

`source-position` `[eclector.base]` [Generic Function]

client stream

This generic function is called in order to determine the current position in *stream*. Eclector does not inspect or manipulate the objects returned by this generic function beyond storing them in signaled conditions and passing them as arguments to the `make-source-range` generic function. A client is therefore free to define methods on this generic function that return arbitrary objects.

The default method on this generic function calls `cl:file-position`.

`make-source-range` `[eclector.base]` [Generic Function]

client start end

This generic function is called in order to turn the source positions *start* and *end* into a range representation suitable for *client*. The returned representation designates the

range of input characters from and including the character at position *start* to but not including the character at position *end*. The default method returns `(cons start end)`.

2.3 Ordinary reader features

In this section, symbols written without package marker are in the `eclector.reader` package (see Section 2.1.2 [Package for ordinary reader features], page 2)

The features provided in this package fall into two categories:

- The functions `read`, `read-preserving-whitespace`, `read-from-string` and `read-delimited-list` which, together with standard special variables, replicate the interface of the standard Common Lisp reader (except functions related to readtables which Eclector provides separately, see Section 2.4 [Readtable features], page 19). These functions are discussed in the section Section 2.3.1 [Common Lisp reader compatible interface], page 4.
- The second category is comprised of the `eclector.base:*client*` special variable and a collection of protocols which allow customizing the behavior of the reader by defining methods specialized to a particular client on the generic functions of the protocols.

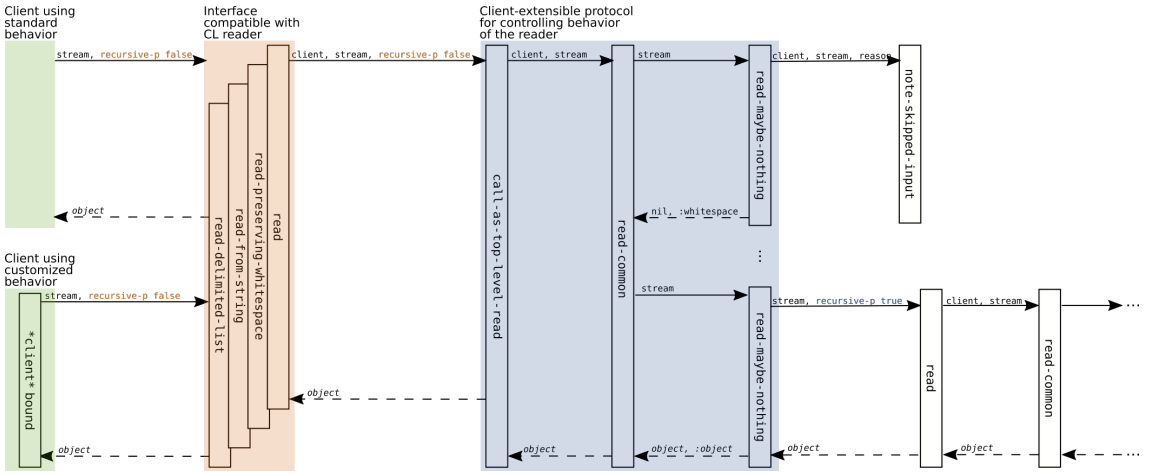

Figure 2.1: Functions and typical function call sequences. Solid arrows represent calls, dashed arrows represent returns from function calls. Labels above arrows represent arguments and return values.

Figure 2.1 illustrates the categorization into the Common Lisp reader compatible interface and the extensible behavior protocol as well as typical function call patterns that arise when the functions `read`, `read-preserving-whitespace`, `read-from-string` and `read-delimited-list` are called by client code.

2.3.1 Common Lisp reader compatible interface

The following functions are like their standarad Common Lisp counterparts with the two differences that their names are symbols in the `eclector.reader` package and that their

behavior can deviate from that of the standard reader depending on the value of the `eclector.base:*client*` variable.

`read` `[eclector.reader]` [Function]
`&optional`(input-stream `*standard-input*`) (eof-error-p `t`) (eof-value `nil`) (recursive-p `nil`)

This function is the main entry point for the ordinary reader. It is entirely compatible with the standard Common Lisp function with the same name.

`read-preserving-whitespace` `[eclector.reader]` [Function]
`&optional`(input-stream `*standard-input*`) (eof-error-p `t`) (eof-value `nil`) (recursive-p `nil`)

This function is entirely compatible with the standard Common Lisp function with the same name.

`read-from-string` `[eclector.reader]` [Function]
string `&optional` (eof-error-p `t`) (eof-value `nil`) `&key` (start `0`) (end `nil`) (preserve-whitespace `nil`)

This function is entirely compatible with the standard Common Lisp function with the same name.

`read-delimited-list` `[eclector.reader]` [Function]
char `&optional` (input-stream `*standard-input*`) (recursive-p `nil`)

This function is entirely compatible with the standard Common Lisp function with the same name.

2.3.2 Reader behavior protocol

By defining methods on the generic functions of this protocol, clients can customize the high-level behavior of the reader.

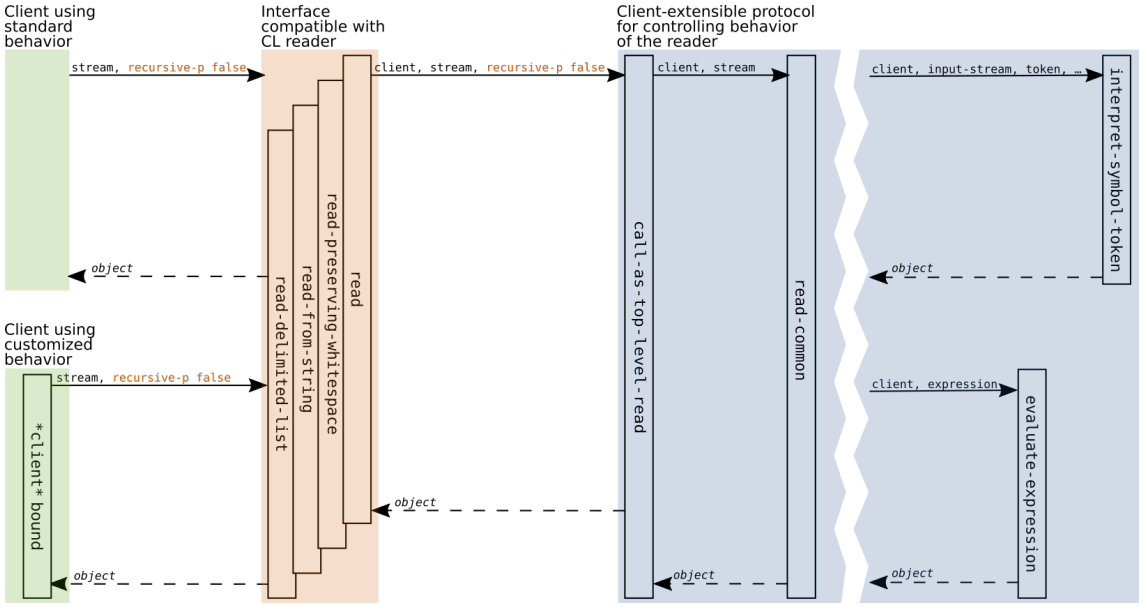

Figure 2.2: Functions and typical function call sequences terminating on the right hand side of the diagram at customizable generic functions which implement aspects of the reader algorithm and standard reader macros. Solid arrows represent calls, dashed arrows represent returns from function calls. Labels above arrows represent arguments and return values.

Figure 2.2 illustrates how the customizable generic functions described in this section are called through the client interface and the implementation of the reader algorithm.

`call-as-top-level-read` `[eclector.reader]` [Generic Function]

client thunk input-stream eof-error-p eof-value preserve-whitespace-p

This generic function is called by `read` if `read` is called with a false value for the *recursive-p* parameter. It calls *thunk* with the necessary context for a global `read` call. *thunk* should read and return an object without consuming any whitespace following the object. If *preserve-whitespace-p* is false, this function reads up to one character of whitespace after *thunk* returns. This function returns the object or *eof-value* returned by *thunk* as its first value. It may return additional values.

The default method on this generic function performs two tasks:

1. It establishes a context in which labels (`#N=`) and references (`#N#`) work.
2. It realizes the requested *preserve-whitespace-p* behavior.

`read-common` `[eclector.reader]` [Generic Function]

client input-stream eof-error-p eof-value

This generic function is called by `read`, passing it the value of the variable `eclector.base:*client*` and the corresponding parameters. Client code can add methods on this function, specializing them to the client class of its choice. The actions that `read` needs to take for different values of the parameter *recursive-p* have already been taken before `read` calls this generic function.

`read-maybe-nothing` `[eclector.reader]` [Generic Function]
client input-stream eof-error-p eof-value

This generic function can be called directly by the client or by the generic function `read-common` to read an object or consume input without returning an object. If called directly by the client, the call has to be in the dynamic scope of a `call-as-top-level-read` call. The function `read-maybe-nothing` either

- encounters the end of input on *input-stream* and, depending on *eof-error-p* either signals an error or returns the values *eof-value* and `:eof`
- or reads one or more whitespace characters an returns the values `nil` and `:whitespace`
- or reads an object. If `*read-suppress*` is true, the function returns nil and `:suppress`. Otherwise it returns the object and `:object`.
- or consumes a macro character and the characters consumed by the associated reader macro function if that reader macro function does not return a value. In this case the function returns `nil` and `:skip`.

`note-skipped-input` `[eclector.reader]` [Generic Function]
client input-stream reason

This generic function is called whenever the reader skips some input such as a comment or a form that must be skipped because of a reader conditional. It is called with the value of the variable `eclector.base:*client*`, the input stream from which the input is being read and an object indicating the reason for skipping the input. The default method on this generic function does nothing. Client code can supply a method that specializes to the client class of its choice.

When this function is called, the stream is positioned immediately *after* the skipped input. Client code that wants to know both the beginning and the end of the skipped input must remember the stream position before the call to `read` was made as well as the stream position when the call to this function is made.

`*skip-reason*` `[eclector.reader]` [Variable]
This variable is used by the reader to determine why a range of input characters has been skipped. To this end, internal functions of the reader as well as reader macros can set this variable to a suitable value before skipping over some input. Then, after the input has been skipped, the generic function `note-skipped-input` is called with the value of the variable as its *reason* argument.

As an example, the method on `note-skipped-input` specialized to `eclector.parse-result:parse-result-client` relays the reason and position information to the client by calling the `eclector.parse-result:make-skipped-input-result` generic function (see Section 2.5 [Parse result construction features], page 19).

`read-token` `[eclector.reader]` [Generic Function]
client input-stream eof-error-p eof-value

This generic function is called by `read-common` when it has been detected that a token should be read. This function is responsible for accumulating the characters of the token and then calling `interpret-token` (see below) in order to create and return a token.

`interpret-token` `[eclector.reader]` [Generic Function]
client input-stream token escape-ranges

This generic function is called by `read-token` in order to create a token from accumulated token characters. The parameter *token* is a string containing the characters that make up the token. The parameter *escape-ranges* indicates ranges of characters read from *input-stream* and preceded by a character with single-escape syntax or delimited by characters with multiple-escape syntax. Values of *escape-ranges* are lists of elements of the form `(`*`start`*`\ .\ `*`end`*`)` where *`start`* is the index of the first escaped character and *`end`* is the index *following* the last escaped character. Note that *`start`* and *`var`* can be identical indicating no escaped characters. This can happen in cases like `a||b`. The information conveyed by the *escape-ranges* parameter is used to convert the characters in *token* according to the *readtable case* of the current readtable before a token is constructed.

`check-symbol-token` `[eclector.reader]` [Generic Function]
client input-stream token escape-ranges position-package-marker-1 position-package-marker-2

This generic function is called by the default method on `interpret-token` when the syntax of the token corresponds to that of a symbol. This function checks the syntactic validity of the symbol token and signals an error in case of a syntax error. If there are no syntax errors (or error recovery has been performed, see Chapter 3 [Recovering from errors], page 23), this function returns three values:

1. *token* or a value derived from *token* by error recovery operations.
2. *position-package-marker-1* or a value derived from *position-package-marker-1* by error recovery operations.
3. *position-package-marker-2* or a value derived from *position-package-marker-2* by error recovery operations.

The parameter *input-stream* is the input stream from which the characters were read. The parameter *token* is a string that contains all the characters of the token. The parameter *escape-ranges* indicates ranges within *token* that were preceded by a character with single-escape syntax or delimited by characters with multiple-escape syntax. The parameter *position-package-marker-1* contains the index into *token* of the first package marker, or `nil` if the token contains no package markers. The parameter *position-package-marker-2* contains the index into *token* of the second package marker, or `nil` if the token contains no package markers or only a single package marker.

The default method on this generic function checks the positions of the package markers taking into account escape ranges. The method signals errors and allows error recovery as described above.

`interpret-symbol-token` `[eclector.reader]` [Generic Function]
client input-stream token position-package-marker-1 position-package-marker-2

This generic function is called by the default method on `interpret-token` when the syntax of the token corresponds to that of a valid symbol. The parameter *input-stream* is the input stream from which the characters were read. The parameter *token* is a

string that contains all the characters of the token. The parameter *position-package-marker-1* contains the index into *token* of the first package marker, or `nil` if the token contains no package markers. The parameter *position-package-marker-2* contains the index into *token* of the second package marker, or `nil` if the token contains no package markers or only a single package marker.

The default method on this generic function calls `interpret-symbol` (see below) with a symbol name string and a package indicator.

`interpret-symbol` `[eclector.reader]` [Generic Function]

client input-stream package-indicator symbol-name internp

This generic function is called by the default method on `interpret-symbol-token` as well as the default `#:` reader macro function to resolve a symbol name string and a package indicator to a representation of the designated symbol. The parameter *input-stream* is the input stream from which *package-indicator* and *symbol-name* were read. The parameter *package-indicator* is a either

- a string designating the package of that name
- the keyword `:current` designating the current package
- the keyword `:keyword` designating the keyword package
- `nil` to indicate that an uninterned symbol should be created

The *symbol-name* is the name of the desired symbol.

The default method uses `cl:find-package` (or `cl:*package*` when *package-indicator* is `:current`) to resolve *package-indicator* followed by `cl:find-symbol` or `cl:intern`, depending on *internp*, to resolve *symbol-name*.

A second method which is specialized on *package-indicator* being `nil` uses `cl:make-symbol` to create uninterned symbols.

`call-reader-macro` `[eclector.reader]` [Generic Function]

client input-stream char readtable

This generic function is called when the reader has determined that some character is associated with a reader macro. The parameter *char* has to be used in conjunction with the *readtable* parameter to obtain the macro function that is associated with the macro character. The parameter *input-stream* is the input stream from which the reader macro function will read additional input to accomplish its task.

The default method on this generic function simply obtains the reader macro function for *char* from *readtable* and calls it, passing *input-stream* and *char* as arguments. The default method therefore does the same thing that the standard Common Lisp reader does.

`find-character` `[eclector.reader]` [Generic Function]

client designator

This generic function is called by the default `#\` reader macro function to find a character. *designator* is either

- a `string` that is the name of the character to be found with single and multiple escapes removed, but with the case of all characters as it was in the input.
- or a character designating itself.

The function has to either return the character designated by *designator* or `nil` if no such character exists.

If *designator* is a `string`, it is the responsibility of the client to disregard the case of characters in *designator*, for example by producing an uppercase string from *designator* before looking up the designated character.

A default method on this generic function that is not specialized to any particular client but is specialized to *designator* being a `string` recognizes the mandatory character names listing in HyperSpec Section 13.1.7 Character Names. Another default method on this generic function that is not specialized to any particular client but is specialized to *designator* being a `character` just returns *designator*.

`make-structure-instance` `[eclector.reader]` [Generic Function]
client name initargs

This generic function is called by the default `#S` reader macro function to construct structure instances. *name* is a symbol naming the structure type of which an instance should be constructed. *initargs* is a list the elements of which alternate between string designators naming structure slots and values for those slots.

It is the responsibility of the client to coerce the string designators to symbols as if by `(intern (string slot-name) (find-package 'keyword))` as described in the Common Lisp specification.

There is no default method on this generic function since there is no portable way to construct structure instances given only the name of the structure type.

`call-with-current-package` `[eclector.reader]` [Generic Function]
client thunk package-designator

This generic function is called by the reader when input has to be read with a particular current package. This is currently only the case in the `#+` and `#-` reader macro functions which read feature expressions in the keyword package. *thunk* is a function that should be called without arguments. *package-designator* designates the package that should be the current package around the call to *thunk*.

The default method on this generic function simply binds `cl:*package*` to the result of `(cl:find-package` *`package-designator`*`)` around calling *thunk*.

`evaluate-expression` `[eclector.reader]` [Generic Function]
client expression

This generic function is called by the default `#.` reader macro function to perform read-time evaluation. *expression* is the expression that should be evaluated as it was returned by a recursive `read` call and potentially influenced by *client*. The function has to either return the result of evaluating *expression* or signal an error.

The default method on this generic function simply returns the result of `(cl:eval` *`expression`*`)`.

`check-feature-expression` `[eclector.reader]` [Generic Function]
client feature-expression

This generic function is called by the default `#+` and `#-` reader macro functions to check the well-formedness of *feature-expression* which has been read from the input

stream before evaluating it. For compound expressions, only the outermost expression is checked regarding the atom in operator position and its shape – child expressions are not checked. The function returns an unspecified value if *feature-expression* is well-formed and signals an error otherwise.

The default method on this generic function accepts standard Common Lisp feature expression, i.e. expressions recursively composed of symbols, `:not`-expressions, `:and`-expressions and `:or`-expressions.

`evaluate-feature-expression` `[eclector.reader]` [Generic Function]
client feature-expression

This generic function is called by the default `#+` and `#-` reader macro functions to evaluate *feature-expression* which has been read from the input stream. The function returns either true or false if *feature-expression* is well-formed and signals an error otherwise.

For compound feature expressions, the well-formedness of child expressions is not checked immediately but lazily, just before the child expression in question is evaluated in a subsequent `evaluate-feature-expression` call. This allows expressions like `#+(and my-cl-implementation (special-feature a b))` *`form`* to be read without error when the `:my-cl-implementation` feature is absent.

The default method on this generic function first calls `check-feature-expression` to check the well-formedness of *feature-expression*. It then evaluates *feature-expression* according to standard Common Lisp semantics for feature expressions.

2.3.3 Labeled objects and references

Eclector includes implementations of the `#=` and `##` reader macros and they are present in the default readtable. One way to customize the behavior of the reader around the `#=` and `##` syntax is replacing the reader macro functions with custom ones but with this approach the client code has to reimplement a lot of functionality. As a finer grained and more composable mechanism for customization, Eclector provides a protocol for implementing and customizing the behavior of the `#=` and `##` reader macros, with or without modifying the readtable. The remainder of this section describes that protocol.

To start with a bit of terminology, we call the object created by reading `#`*`N`*`=`*`expression`* a *labeled object*. We call *N* the *label* of the labeled object and the result of reading *`expression`* the *object* of the labeled object. We say that `#`*`N`*`=`*`expression`* *defines* the labeled object and `#`*`N`*`#` *references* the labeled object. We call the reference *circular* if `#`*`N`*`#` occurs within *expression*. Labeled objects are internal to the reader and only exist during `eclector.reader:read` calls: before such a call returns an object, each labeled object within the returned object is replaced by its respective final object. Callers of `eclector.reader:read` and related functions will therefore only ever see the object, never the labeled object[1].

On a technical level, a labeled object is represented as a data type with a current state and a single (possibly unbound) slot containing the object. The following diagrams depicts

[1] Reader macro functions which call `eclector.reader:read` may receive labeled objects under certain circumstances (see Section 5.3 [Circular objects and custom reader macros], page 28).

the possible states of a labeled object together with input patterns and corresponding transitions:

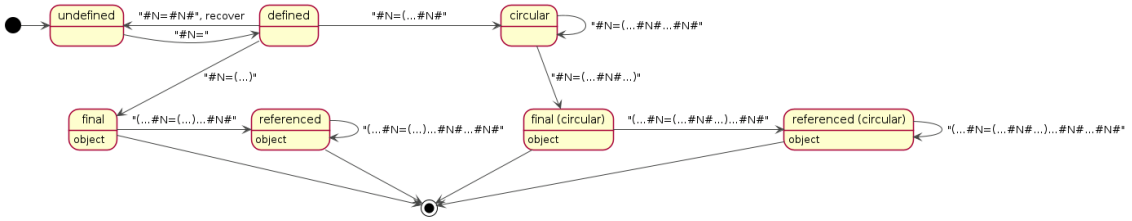

Figure 2.3: Possible states of a labeled object and input patterns which correspond to state transitions.

Put differently, a labeled object can be in the following states:

State	**Object slot**
undefined	–
defined	unbound
final	the object
referenced *(not strictly needed)*	the object
circular	unbound
final (circular)	the object
referenced (circular) *(not strictly needed)*	the object

The distinction between the states `final` and `referenced` on the one hand and `final (circular)`, and `referenced (circular)` on the other hand is not required for implementing labeled objects. Those two pairs of states are therefore collapsed to just `final` and `final (circular)` in the remainder of this section. The following figure and paragraphs describe generic functions and methods which implement the creation, registration, lookup and manipulation of labeled objects according to the reduced set of states:

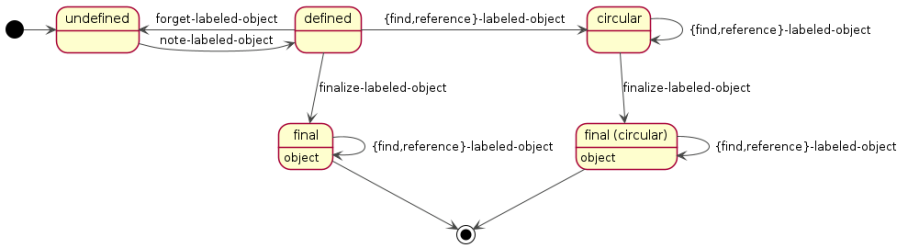

Figure 2.4: Reduced set of states of a labeled object and protocol functions with corresponding state transitions.

In addition to the generic functions referenced in the above diagram, the generic functions `fixup-graph-p`, `fixup-graph` and `fixup` are part of the protocol. Those functions are used to replace labeled objects with their respective final objects within an object that is about

to be returned to the caller of `eclector.reader:read`[2]. To this end, the `#=` reader macro function must inspect and update the state of the labeled object it is processing after reading *`expression`* by calling `finalize-labeled-object`. `finalize-labeled-object` decides whether `fixup-graph` (see below) must be called: If after reading *`expression`* the labeled object is in state `:circular`, *`expression`* must have contained circular references and the result of reading it contains labeled objects that have to be replaced with their respective final objects. `fixup-graph` and `fixup` perform this replacement. This replacement is performed by recursively traversing objects which are reachable from the final object of the labeled objects, for example by visiting the slots of standard objects, and replacing labeled objects with their respective final object.

In certain cases, the computational complexity of this traversal and replacement can be rather high, depending on when and how exactly the traversal is performed: consider an expression of the form `#1=(1 #1# #2=(2 #2# ...))`. The nested labeled objects in this expression are all circular and thus require fixing up. The `read` call for the innermost labeled object, say `#100=...`, returns first and the fixup processing for the labeled object could be performed immediately. The problem is that each of the labeled objects would be processed in the same manner which would lead to a computation complexity of $O(NM)$ where N is the number of labels and M is the number of nodes in the object graph rooted at the object which is returned by the outermost `read` call. One way to avoid this problem would be to perform fixup processing only for the outermost `read` call. The problem with that approach is that only a small sub-graph of the whole object graph may be circular in which case most of the work for traversing the whole graph would be wasted. To address both problems, Eclector allows clients to track the nesting of labeled objects and fix up sub-graphs which contain multiple nested objects in one go (see [Generic-Function eclector.reader|fixup-graph-p], page 16).

`call-with-label-tracking` `[eclector.reader]` [Generic Function]
client thunk

This generic function is called by the default method on `call-as-top-level-read` in order to establish a context for tracking `#=` label definitions and `##` label references around a call to *thunk*.

The default method on this generic function establishes a context in which the default `#=` and `##` reader macro functions can make the appropriate calls to `note-labeled-object`, `forget-labeled-object`, `find-labeled-object`.

`note-labeled-object` `[eclector.reader]` [Generic Function]
client input-stream label parent

This generic function is called by the default `#=` reader macro function to note the definition of a labeled object with label *label* while reading from *input-stream*. The function creates, registers and returns a representation of the labeled object. The returned object is registered in the sense that a subsequent call to `find-labeled-object` with arguments *client* and *label* returns the same object unless `forget-labeled-object` has been called to unregister the object.

[2] This fixup processing has to be delayed under certain circumstances (see Section 5.3 [Circular objects and custom reader macros], page 28).

parent is either `nil` or a (previously created) surrounding labeled object. The parent labeled object is provided to allow the client to potentially defer fixup processing for the new labeled object if the processing for the surrounding labeled object subsumes the processing for the new labeled object.

Note that, when reading an expression of the form `#N=object`, this function is called after reading `#N=` from *input-stream* but before reading `object`. Consequently, the created and returned labeled object is defined but does not have an object associated with it.

The default method on this generic function calls `make-labeled-object` with *client*, *input-stream* and *label* to create an object of an unspecified type. The method registers and returns the created object. Client code should manipulate the object only via the generic functions described in this section and in particular not rely on the object being of a particular type (since methods on `make-labeled-object` specialized to certain client classes could return unexpected objects). The default method requires the context established by the default method on `call-with-label-tracking`.

`forget-labeled-object` `[eclector.reader]` [Generic Function]
client label

This generic function is called by the default `#=` reader macro function when Eclector reads an invalid labeled object of the form `#N=#N#` and the caller chooses to recover from the resulting error (see Chapter 3 [Recovering from errors], page 23). In that situation, the remainder of the input is processed as if there had been no labeled object with label *N*. This function makes the labeled object undefined so that a subsequent `find-labeled-object` call for *label* will return `nil`.

The default method on this generic function requires the context established by the default method on `call-with-label-tracking`.

`find-labeled-object` `[eclector.reader]` [Generic Function]
client label

This generic function is called by the default `##` reader macro function to look up the previously registered representation of a labeled object for *label*. The function returns `nil` if no such object has been registered for *label* and the registered object otherwise.

The default method on this generic function requires the context established by the default method on `call-with-label-tracking`.

`make-labeled-object` `[eclector.reader]` [Generic Function]
client input-stream label parent

This generic function is called by `note-labeled-object` to create and return a representation of a labeled object with label *label*. *parent* is either `nil` or a previously created, surrounding labeled object which allows the client to potentially defer fixup processing for the new labeled object if the processing for the surrounding labeled object subsumes the processing.

The default method on this generic function creates and returns an object of an unspecified type. Client code should manipulate the object only via the generic functions `labeled-object-state`, `finalize-labeled-object` and

`reference-labeled-object` and in particular not rely on the object being of a particular type (since methods on this generic function specialized to certain client classes could return unexpected objects).

`labeled-object-state` `[eclector.reader]` [Generic Function]

client object

This generic function is called by the default `#=` reader macro function to determine the state of *object*. This function returns

- `nil` if *object* is not a labeled object
- two values if *object* is a labeled object: one of the keywords `:defined`, `:circular`, `:final`, `:final/circular` and the final object stored in *object* if the first value is either `:final` or `:final/circular` or `nil` otherwise.

The following table lists all possible return value shapes:

object **is a labeled object**	**First value**	**Second value**
no	`nil`	
yes	`:defined`	`nil`
yes	`:circular`	`nil`
yes	`:final`	*final-object*
yes	`:final/circular`	*final-object*

The default method on this generic function is applicable to labeled object representations returned by the default methods on `note-labeled-object` and `make-labeled-object`.

`finalize-labeled-object` `[eclector.reader]` [Generic Function]

client labeled-object object

This generic function is called by the default `#=` reader macro function after reading a complete labeled object in order to store *object* in *labeled-object* and change the state of *labeled-object* to either `:final` or `:final/circular`. The function returns two values: the finalized *labeled-object* and the new state of *labeled-object*.

The default method on this generic function is applicable to labeled object representations returned by the default methods on `note-labeled-object` and `make-labeled-object`.

`reference-labeled-object` `[eclector.reader]` [Generic Function]

client input-stream labeled-object

This generic function is called by the default `##` reader macro function to process a reference to *labeled-object* while reading from *input-stream*. *labeled-object* must be a representation of a labeled object and has, in the context of the `##` reader macro function, likely been obtained by calling `find-labeled-object`. Depending on the state of *labeled-object*, this function returns either *labeled-object* itself or an object that can be returned to the caller as-is. In case *labeled-object* is returned, it will be replaced by its associated object later, when `fixup-graph` is called.

The default method on this generic function is applicable to labeled object representations returned by the default methods on `note-labeled-object` and `make-labeled-object`.

As briefly mentioned above, the generic functions `fixup-graph` and `fixup` traverse and inspect objects in the object graph reachable from an object that is about to be returned to the caller of `eclector.reader:read`. In order to distinguish ordinary objects from labeled objects that act as placeholders in the object graph and must be replaced with their respective final objects, `fixup` methods call `labeled-object-state` on all encountered objects. `labeled-object-state` returns `nil` for all objects that are not labeled objects and `:final` for labeled objects which must be replaced with their final object.

`fixup-graph-p` `[eclector.reader]` [Generic Function]

client root-labeled-object

This generic function is potentially called by a method on `finalize-labeled-object` to determine whether the object graph reachable from the object of *root-labeled-object* should be fixed up by calling `fixup-graph` with *client* and *labeled-object*.

Multiple default methods on this generic function jointly implement the following behavior:

- If *root-labeled-object* has a parent labeled object, *root-labeled-object* should not be fixup up immediately (since the fixup processing for ancestor labeled objects will subsume the fixup processing for *root-labeled-object*).
- If *root-labeled-object* does not have parent labeled object but has child labeled objects, *root-labeled-object* should be fixed up immediately.
- If *root-labeled-object* does not have parent labeled object and is in state `:final/circular`, *root-labeled-object* should be fixed up immediately.

`fixup-graph` `[eclector.reader]` [Generic Function]

client root-labeled-object `&key` object-key

This generic function is potentially called after the reader has constructed an object graph which is reachable from the object of *root-labeled-object* and noticed circular references within this graph to fix up circular references before the object of *root-labeled-object* is returned to the caller (of `read` or related functions).

object-key is a function that accepts a labeled object and returns the object of the labeled object.

The default method on this generic function creates a hash table for tracking already processed objects and calls `fixup` with *client*, the object of *root-labeled-object* and the hash table to recursively process objects in the object graph which is reachable from the object of *root-labeled-object*.

`fixup` `[eclector.reader]` [Generic Function]

client object seen-objects

This generic function is potentially called to apply circularity-related changes to the object constructed by the reader before it is returned to the caller. *object* is the object that should be modified. *seen-objects* is a `eq`-hash table used to track already processed objects (see below). A method specialized to a class, instances of which consists of parts, should modify *object* by scanning its parts for labeled object markers, replacing found labeled object markers with the respective final object and recursively calling `fixup` for all parts.

To recognize labeled objects which have to be replaced, methods should call `labeled-object-state` on each part of *object* and interpret the returned values as follows: if `nil` is returned, the part should not be replaced but recursively processed. If `:final` is returned as the first value, the part should be replaced with the final object that is returned as the second value. Parts are replaced by mutating *object*.

`fixup` is called for side effects – its return value is ignored.

Default methods specializing the *object* parameter to `cons`, `array`, `standard-object` and `hash-table` process instances of those classes in the obvious way.

An unspecialized `:around` method queries and updates *seen-objects* to ensure that each object is processed exactly once.

2.3.4 S-expression creation and quasiquotation

The following generic functions allow clients to construct representations of quoted and quasiquoted forms.

`wrap-in-quote` `[eclector.reader]` [Generic Function]
client material

This generic function is called by the default `'`-reader macro function to construct a quotation form in which *material* is the quoted material.

The default method on this generic function returns a result equivalent to `(list 'common-lisp:quote` *`material`*`)`.

`wrap-in-quasiquote` `[eclector.reader]` [Generic Function]
client form

This generic function is called by the default `` ` ``-reader macro function to construct a quasiquotation form in which *form* is the quasiquoted material.

The default method on this generic function returns a result equivalent to `(list 'eclector.reader:quasiquote` *`form`*`)`.

`wrap-in-unquote` `[eclector.reader]` [Generic Function]
client form

This generic function is called by the default `,`-reader macro function to construct an unquote form in which *form* is the unquoted material.

The default method on this generic function returns a result equivalent to `(list 'eclector.reader:unquote` *`form`*`)`.

`wrap-in-unquote-splicing` `[eclector.reader]` [Generic Function]
client form

This generic function is called by the default `,@`-reader macro function to construct a splicing unquote form in which *form* is the unquoted material.

The default method on this generic function returns a result equivalent to `(list 'eclector.reader:unquote-splicing` *`form`*`)`.

Backquote and unquote syntax is forbidden in some contexts such as multi-dimensional array literals (`#A`) and structure literals (`#S`) thus Eclector has a mechanism for controlling whether backquote, unquote or both should be allowed in a given context. Since custom reader macros may also have to control this aspect, Eclector provides an external protocol:

`with-forbidden-quasiquotation` `[eclector.reader]` [Macro]
context `&optional` (quasiquote-forbidden-p `t`) (unquote-forbidden-p `t`) `&body`body

Disallow backquote syntax, unquote syntax or both in `read` functions called during the execution of *body*. *context* is a symbol identifying the current context which is used for error reporting. A typical value is the name of the reader macro function in which this macro is used. *quasiquote-forbidden-p* controls whether backquote syntax should be forbidden. The value `:keep` causes the binding to remain unchanged. *unquote-forbidden-p* controls whether unquote syntax should be forbidden. The value `:keep` causes the binding to remain unchanged.

`wrap-in-function` `[eclector.reader]` [Generic Function]
client name

This generic function is called by the default `#'`-reader macro function to construct a form that applies the `function` special operator to the *name* expression.

The default method on this generic function returns a result equivalent to `(list 'common-lisp:function` *form*`)`.

2.3.5 Readtable initialization

The standard syntax types and macro character associations used by the ordinary reader can be set up for any readtable object implementing the readtable protocol (see Section 2.4 [Readtable features], page 19). The following functions are provided for this purpose:

`set-standard-syntax-types` `[eclector.reader]` [Function]
readtable

This function sets the standard syntax types in *readtable* (See HyperSpec section 2.1.4.)

`set-standard-macro-characters` `[eclector.reader]` [Function]
readtable

This function sets the standard macro characters in *readtable* (See HyperSpec section 2.4.)

`set-standard-dispatch-macro-characters` `[eclector.reader]` [Function]
readtable

This function sets the standard dispatch macro characters, that is sharpsign and its sub-characters, in *readtable* (See HyperSpec section 2.4.8.)

`set-standard-syntax-and-macros` `[eclector.reader]` [Function]
readtable

This function sets the standard syntax types and macro characters in *readtable* by calling the above three functions.

2.4 Readtable features

In this section, symbols written without package marker are in the `eclector.readtable` package (see Section 2.1.3 [Package for readtable features], page 2).

This package exports two kinds of symbols:

1. Symbols the names of which correspond to the names of symbols in the `common-lisp` package. The functions bound to these symbols are generic versions of the corresponding standard Common Lisp functions. Clients can define custom readtables by defining methods on these generic functions.
2. Symbols bound to additional functions and condition types.

`readtablep` `[eclector.readtable]` [Generic Function]
object

This function is the generic version of the standard Common Lisp function `cl:readtablep`. The function returns true if *object* can be used as a readtable in Eclector via the protocol functions in the `ecelctor.readtable` package. The default method returns `nil`.

TODO

2.5 Parse result construction features

In this section, symbols written without package marker are in the `eclector.parse-result` package (see Section 2.1.4 [Package for parse result construction features], page 2).

This package provides clients with a reader that behaves similarly to `cl:read` but returns custom parse result objects controlled by the client. Some parse results correspond to things like symbols, numbers and lists that `cl:read` would return, while others, if the client chooses, represent comments and other kinds of input that `cl:read` would discard. Furthermore, clients can associate source location information with parse results.

Clients using this package pass a "client" instance for which methods on the generic functions described below are applicable to `read`, `read-preserving-whitespace` or `read-from-string`. Suitable client classes can be constructed by using `parse-result-client` as a superclass and at least defining a method on the generic function `make-expression-result`.

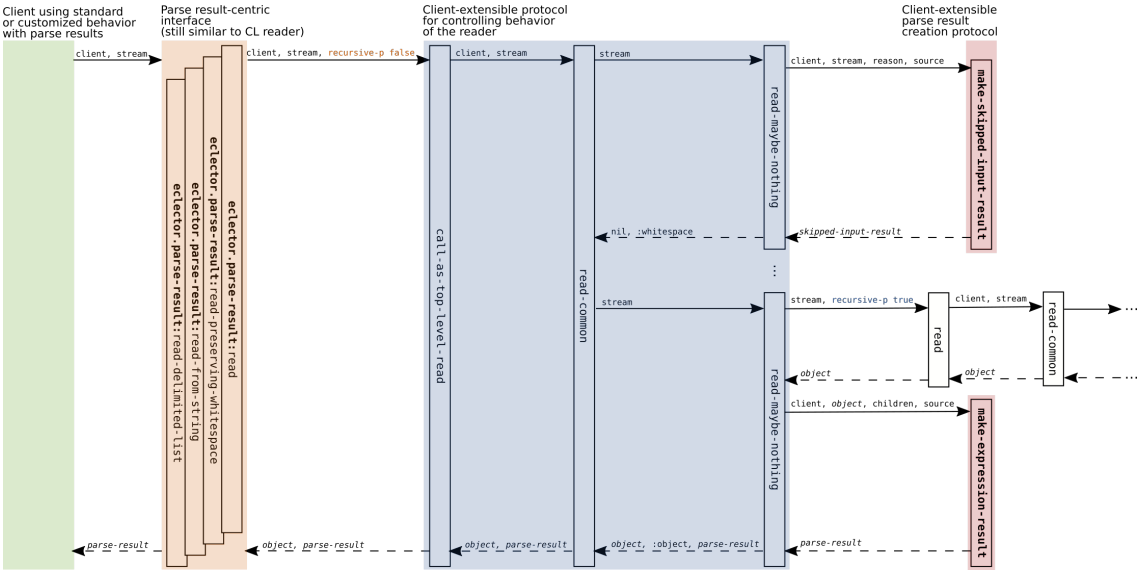

Figure 2.5: Functions and typical function call sequences. Solid arrows represent calls, dashed arrows represent returns from function calls. Labels above arrows represent arguments and return values. Differences from the non-parse result version are highlighted with bold text.

Figure 2.5 shows typical function call patterns that arise when the functions `read`, `read-preserving-whitespace`, `read-from-string` and `read-delimited-list` are called by client code.

`read` `[eclector.parse-result]` [Function]

client `&optional` (input-stream `*standard-input*`) (eof-error-p `t`) (eof-value `nil`)

This function is the main entry point for this variant of the reader. It is in many ways similar to the standard Common Lisp function `cl:read`. The differences are:

- A client instance must be supplied as the first argument.
- The first return value, unless *eof-value* is returned, is an arbitrary parse result object created by the client, not generally the read object.
- The second return value, unless *eof-value* is returned, is a list of "orphan" results. These results are return values of `make-skipped-input-result` and arise when skipping input at the toplevel such as comments which are not lexically contained in lists: `#|orphan|# (#|not orphan|#)`.
- The function does not accept a *recursive* parameter since it sets up a dynamic environment in which calls to `eclector.reader:read` behave suitably.

`read-preserving-whitespace` `[eclector.parse-result]` [Function]

client `&optional`(input-stream `*standard-input*`) (eof-error-p `t`) (eof-value `nil`)

This function is similar to the standard Common Lisp function `cl:read-preserving-whitespace`. The differences are the same as described above for `read` compared to `cl:read`.

`read-from-string` `[eclector.parse-result]` [Function]
client string `&optional` (eof-error-p `t`) (eof-value `nil`) `&key` (start `0`) (end `nil`) (preserve-whitespace `nil`)

This function is similar to the standard Common Lisp function `cl:read-from-string`. The differences are:

- A client instance must be supplied as the first argument.
- The first return value, unless *eof-value* is returned, is an arbitrary parse result object created by the client, not generally the read object.
- The *third* return value, unless *eof-value* is returned, is a list of "orphan" results (Described above).

`parse-result-client` `[eclector.parse-result]` [Class]
This class should generally be used as a superclass for client classes using this package.

`make-expression-result` `[eclector.parse-result]` [Generic Function]
client result children source

This generic function is called in order to construct a parse result object. The value of the *result* parameter is the raw object read. The value of the *children* parameter is a list of already constructed parse result objects representing objects read by recursive `read` calls. The value of the *source* parameter is a source range, as returned by `eclector.base:make-source-range` and `eclector.base:source-position` delimiting the range of characters from which *result* has been read.

This generic function does not have a default method since the purpose of the package is the construction of *custom* parse results. Thus, a client must define a method on this generic function.

`make-skipped-input-result` `[eclector.parse-result]` [Generic Function]
client stream reason source

This generic function is called after the reader skipped over a range of characters in *stream*. It returns either `nil` if the skipped input should not be represented or a client-specific representation of the skipped input. The value of the *source* parameter designates the skipped range using a source range representation obtained via `make-source-range` and `source-position`.

Reasons for skipping input include comments, the `#+` and `#-` reader macros and `*read-suppress*`. The aforementioned reasons are reflected by the value of the *reason* parameter as follows:

Input	**Value of the *reason* parameter**
Comment starting with `;`	`(:line-comment . 1)`
Comment starting with `;;`	`(:line-comment . 2)`
Comment starting with n `;`	`(:line-comment . `n`)`
Comment delimited by `#| |#`	`:block-comment`
`#+`*`false-expression`*	`(:sharpsign-plus . `*`false-expression`*`)`
`#-`*`true-expression`*	`(:sharpsign-minus . `*`true-expression`*`)`

`*read-suppress*` is true	`*read-suppress*`
A reader macro returns no values	`:reader-macro`

The default method returns `nil`, that is the skipped input is not represented as a parse result.

2.6 CST reader features

In this section, symbols written without package marker are in the `eclector.concrete-syntax-tree` package (see Section 2.1.5 [Package for CST features], page 2).

`read` `[eclector.concrete-syntax-tree]` [Function]

`&optional` (input-stream `*standard-input*`) (eof-error-p `t`) (eof-value `nil`)

This function is the main entry point for the CST reader. It is mostly compatible with the standard Common Lisp function `cl:read`. The differences are:

- The return value, unless *eof-value* is returned, is an instance of a subclass of `concrete-syntax-tree:cst`.
- The function does not accept a *recursive* parameter since it sets up a dynamic environment in which calls to `eclector.reader:read` behave suitably.

`read-preserving-whitespace` `[eclector.concrete-syntax-tree]` [Function]

`&optional`(input-stream `*standard-input*`) (eof-error-p `t`) (eof-value `nil`)

This function is similar to the standard Common Lisp function `cl:read-preserving-whitespace`. The differences are the same as described above for `read` compared to `cl:read`.

`read-from-string` `[eclector.concrete-syntax-tree]` [Function]

string `&optional` (eof-error-p `t`) (eof-value `nil`) `&key` (start `0`) (end `nil`) (preserve-whitespace `nil`)

This function is similar to the standard Common Lisp function `cl:read-from-string`. The differences are the same as described above for `read` compared to `cl:read`.

3 Recovering from errors

3.1 Error recovery features

Eclector offers extensive support for recovering from many syntax errors, continuing to read from the input stream and return a result that somewhat resembles what would have been returned in case the syntax had been valid. To this end, a restart named `eclector.reader:recover` is established when recoverable errors are signaled. Like the standard Common Lisp restart `cl:continue`, this restart can be invoked by a function of the same name:

`recover` `[eclector.reader]` [Function]

`&optional`condition

This function recovers from an error by invoking the most recently established applicable restart named `eclector.reader:recover`. If no such restart is currently established, it returns `nil`. If *condition* is non-`nil`, only restarts that are either explicitly associated with *condition*, or not associated with any condition are considered.

When a `read` call during which error recovery has been performed returns, Eclector tries to return an object that is similar in terms of type, numeric value, sequence length, etc. to what would have been returned in case the input had been well-formed. For example, recovering after encountering the invalid digit in `#b11311` returns either the number `#b11011` or the number `#b11111`.

3.2 Recoverable errors

A syntax error and a corresponding recovery strategy are characterized by the type of the signaled condition and the report of the established `eclector.reader:recover` restart respectively. Attempting to list and describe all examples of both would provide little insight. Instead, this section describes different classes of errors and corresponding recovery strategies in broad terms:

- Replace a missing numeric macro parameter or ignore an invalid numeric macro parameter. Examples: `#=1` ⟶ `1`, `#5P"."` ⟶ `#P"."`
- Add a missing closing delimiter. Examples: `"foo` ⟶ `"foo"`, `(1 2` ⟶ `(1 2)`, `#(1 2` ⟶ `#(1 2)`, `#C(1 2` ⟶ `#C(1 2)`
- Replace an invalid digit or an invalid number with a valid one. This includes digits which are invalid for a given base but also things like 0 denominator. Examples: `#12rc` ⟶ `1`, `1/0` ⟶ `1`, `#C(1 :foo)` ⟶ `#C(1 1)`
- Replace an invalid character with a valid one. Example: `#\foo` ⟶ `#\?`
- Invalid constructs can sometimes be ignored. Examples: `(,1)` ⟶ `(1)`, `#S(foo :bar 1 2 3)` ⟶ `#S(foo :bar 1)`
- Excess parts can often be ignored. Examples: `#C(1 2 3)` ⟶ `#C(1 2)`, `#2(1 2 3)` ⟶ `#2(1 2)`
- Replace an entire construct by some fallback value. Example: `#S(5)` ⟶ `nil`, `(#1=)` ⟶ `(nil)`

3.3 Potential problems

Note that attempting to recover from syntax errors may lead to apparent success in the sense that the `read` call returns an object, but this object may not be what the caller wanted. For example, recovering from the missing closing `"` in the following example

```
(defun foo (x y)
  "My documentation string
  (+ x y))
```

results in `(DEFUN FOO (X Y) "My documentation string<newline> (+ x y))")`, not `(DEFUN FOO (X Y) "My documentation string" (+ x y))`.

4 Side effects

This chapter describes potential side effects of calling `eclector.reader:read`, `eclector.reader:read-preserving-whitespace` or `eclector.reader:read-from-string` for different kinds of clients.

4.1 Potential side effects for the default client

The following destructive modifications are considered uninteresting and ignored in the remainder of this section:

- Changes to the state of streams passed to the functions mentioned above.
- Changes to objects within expressions currently being read.

Furthermore, the remainder of this section is written under the following assumptions:

- The stream object passed to `eclector.reader:read` does not cause additional side effects on its own.
- The variable `eclector.reader:*client*` is bound to an object for which there are no custom applicable methods on generic functions belonging to protocols provided by Eclector that introduce additional side effects.
- The variable `eclector.readtable:*readtable*` is bound to an object for which
  - there are no custom applicable methods on generic functions belonging to protocols provided by Eclector that introduce additional side effects
  - no non-default macro functions have been installed

If any of the above assumptions does not hold, "all bets are off" in the sense that arbitrary side effects other than the ones described below are possible. For notes regarding non-default clients, See Section 4.2 [Potential side effects for non-default clients], page 26.

4.1.1 Symbols and packages (default client)

The default method on the generic function `eclector.reader:interpret-symbol` may create and intern symbols, thereby modifying the package system.

4.1.2 Read-time evaluation (default client)

The default method on the generic function `eclector.reader:evaluate-expression` uses `cl:eval` to evaluate arbitrary expressions, potentially causing side effects. With the default readtable, the generic function is only called by the macro function of the `#.` reader macro.

4.1.3 Standard reader macros (default client)

The default method on the generic function `eclector.reader:call-reader-macro` can cause side effects by calling macro functions that cause side effects. The following standard reader macros potentially cause side-effects:

- `#.` as described in Section 4.1.2 [Read-time evaluation (default client)], page 25.

4.2 Potential side effects for non-default clients

4.2.1 Symbols and packages

In addition to the potential side effects described in Section 4.1.1 [Symbols and packages (default client)], page 25, strings passed as the third argument of `eclector.reader:interpret-token` are potentially destructively modified during conversion to the current readtable case.

4.2.2 Read-time evaluation

The same considerations as in Section 4.1.2 [Read-time evaluation (default client)], page 25, apply.

4.2.3 Structure instance creation

Clients defining methods on `eclector.reader:make-structure-instance` which implement the standard behavior of calling the default constructor (if any) of the named structure should consider side effects caused by slot initforms of the structure. The following example illustrates this problem:

```
(defvar *counter* 0)
(defstruct foo (bar (incf *counter*)))
#S(foo)
*counter* ⇒ 1
#S(foo)
*counter* ⇒ 2
```

4.2.4 Circular structure

The `fixup` generic function potentially modifies its second argument destructively. Clients that define methods on `eclector.reader:make-structure-instance` should be aware of this potential modification in cases like `#1=#S(foo :bar #1#)`. Similar considerations apply for other ways of constructing compound objects such as `#1=(t . #1#)`.

4.2.5 Standard reader macros

The following standard reader macros could cause or be affected by side effects when combined with a non-standard client:

- `#.` as described in Section 4.1.2 [Read-time evaluation (default client)], page 25.
- `#S` as described in Section 4.2.3 [Structure instance creation], page 26.
- `(`, `#(` and `#S` as described in Section 4.2.4 [Circular structure], page 26.
- The `,.` (i.e. destructively splicing) variant of the `,` reader macro does not currently destructively modify the surrounding object, but clients should not rely on this fact. This consideration applies to clients that install non-standard macro functions for the `(` and `#(` reader macros.

5 Interpretation of unclear parts of the specification

This chapter describes Eclector's interpretation of passages in the Common Lisp specification that do not describe the behavior of a conforming reader completely unambiguously.

5.1 Interpretation of Sharpsign C and Sharpsign S

At first glance, Sharpsign C and Sharpsign S seem to follow the same syntactic structure: the dispatch macro character followed by the sub-character followed by a list of a specific structure. However, the actual descriptions of the respective syntax is different. For Sharpsign C, the specification states:

> `#C` reads a following object, which must be a list of length two whose elements are both reals.

For Sharpsign S, on the other hand, the specification describes the syntax as:

> `#s(`*`name slot1 value1 slot2 value2 ...`*`)` denotes a structure.

Note how the description for Sharpsign C relies on a recursive `read` invocation while the description for Sharpsign S gives a character-level pattern with meta-syntactic variables. It is possible that this is an oversight and the syntax was intended to be uniform between the two reader macros. Whatever the case may be, in order to provide conforming behavior, Eclector is forced to implement Sharpsign C with a recursive `read` invocation and Sharpsign S with a stricter enforcement of the specified syntax.

More concretely, Eclector behaves as summarized in the following table:

Input	**Behavior**
`#C(1 2)`	Read as `#C(1 2)`
`#C (1 2)`	Read as `#C(1 2)`
`#C#||#(1 2)`	Read as `#C(1 2)`
`#C#.(list 1 (+ 2 3))`	Read as `#C(1 5)`
`#C[1 2]` for left-parenthesis syntax on `[`	Read as `#C(1 2)`
`#S(foo)`	Read as `#S(foo)`
`#S (foo)`	Rejected
`#S#||#(foo)`	Rejected
`#S#.(list 'foo)`	Rejected
`#S[foo]` for left-parenthesis syntax on `[`	Rejected

Eclector provides a strict version of the Sharpsign C macro function under the name `eclector.reader:strict-sharpsign-c` which behaves as follows:

Input	**Behavior**
`#C(1 2)`	Read as `#C(1 2)`
`#C (1 2)`	Rejected
`#C#||#(1 2)`	Rejected
`#C#.(list 1 (+ 2 3))`	Rejected
`#C[1 2]` for left-parenthesis syntax on `[`	Read as `#C(1 2)`

5.2 Interpretation of Backquote and Sharpsign Single Quote

The Common Lisp specification is very specific about the contexts in which the quasiquotation mechanism can be used. Explicit descriptions of the behavior of the quasiquotation mechanism are given for expressions which *are* lists or vectors and it is implied that unquote is not allowed in other expressions. From this description, it is clear that `‘#S(foo :bar ,x)` is not valid syntax, for example. However, whether `‘#’,foo` is valid syntax depends on whether `#’thing` is considered to *be* a list. Since `‘#’,foo` is a relatively common idiom, Eclector accepts it by default.

Eclector provides a strict version of the Sharpsign Single Quote macro function under the name `eclector.reader:strict-sharpsign-single-quote` which does not accept unquote in the function name.

5.3 Circular objects and custom reader macros

The Common Lisp specification describes the behavior of the `##` reader macro as follows:

> `#`*n*`#`, where *n* is a required unsigned decimal integer, provides a reference to some object labeled by `#`*n*`=`; that is, `#`*n*`#` represents a pointer to the same (`eq`) object labeled by `#`*n*`=`.

The vague phrasing “represents a pointer to the same (`eq`) object” is probably chosen to cover the situation in which the object in question is not yet defined when the reader encounters the `#`*n*`#` reference as is the case with input of the form `#`*n*`=(...#`*n*`#...)`. The fact that the object is not yet defined when the reference is encountered is not a problem in general except for one situation: assume `#_` is a custom reader macro in the current readtable which calls `read`. In this situation, reading an expression of the form `#`*n*`=(...#_#`*n*`#...)` causes the reader macro function for `#_` to be called which calls `read` to read the following object which encounters the reference. This chain of calls leads to a potential problem: the `read` call made by the reader macro function has to return some object but it cannot return the object labeled *n* since that object has not been read yet. The reader macro function must therefore receive some sort of implementation-dependent[1] object which stands in for the object labeled *n* and gets replaced at some later time after the object labeled *n* has been read. Since the stand-in object is implementation-dependent, the reader macro function must not make any assumptions regarding the type of the object or operate on it in any way other than returning the object or using the object as a part of a compound object.

The following example violates this principle since the reader macro function in `custom-macro-readtable` calls `cl:second` on the object returned by `eclector.reader:read`:

```
(defun custom-macro-readtable ()
  (let ((readtable (eclector.readtable:copy-readtable
                    eclector.reader:*readtable*)))
    (eclector.readtable:set-dispatch-macro-character
     readtable #\# #\_ (lambda (stream char sub-char)
                         (declare (ignore char sub-char))
                         (second (eclector.reader:read stream t nil t))))
```

[1] We use “implementation-dependent” in the sense defined in the Common Lisp specification except that Eclector is the implementation in question.

```
    readtable))

(let ((eclector.reader:*readtable* (custom-macro-readtable)))
  (eclector.reader:read-from-string "#1=(:a #_#1#)"))
⇒ undefined
```

To handle the problem described above, Eclector imposes the following restriction on custom reader macro functions which call `read`:

> A reader macro function which reads an object by calling `read` must account for the object being of an implementation-dependent type and must not operate on the object in any way other than returning the object or using the object as a part of a compound object.

Concept index

C

client 2, 19
complex literal 27
concrete syntax tree 1, 2, 22

E

error 23

F

function 18, 28

L

labeled object 11

P

parse result 1, 2, 4, 19, 22

Q

quasiquotation 17
quasiquote 28
quotation 17

R

reader macro 27, 28
readtable 2, 18, 19
recovery 23

S

side effects 25
source location 2
source tracking 1, 4
specification interpretation 27
structure literal 27

Function and macro and variable and type index

*

client [eclector.base] 3
skip-reason [eclector.reader] 7

C

call-as-top-level-read [eclector.reader] 6
call-reader-macro [eclector.reader] 9
call-with-current-package
[eclector.reader] 10
call-with-label-tracking
[eclector.reader] 13
check-feature-expression
[eclector.reader] 10
check-symbol-token [eclector.reader] 8

E

evaluate-expression [eclector.reader] 10
evaluate-feature-expression
[eclector.reader] 11

F

finalize-labeled-object [eclector.reader] ... 15
find-character [eclector.reader] 9
find-labeled-object [eclector.reader] 14
fixup [eclector.reader] 16
fixup-graph [eclector.reader] 16
fixup-graph-p [eclector.reader] 16
forget-labeled-object [eclector.reader] 14

I

interpret-symbol [eclector.reader] 9
interpret-symbol-token [eclector.reader] 8
interpret-token [eclector.reader] 8

L

labeled-object-state [eclector.reader] 15

M

make-expression-result
[eclector.parse-result] 21
make-labeled-object [eclector.reader] 14
make-skipped-input-result
[eclector.parse-result] 21
make-source-range [eclector.base] 3
make-structure-instance [eclector.reader] ... 10

N

note-labeled-object [eclector.reader] 13
note-skipped-input [eclector.reader] 7

P

parse-result-client
[eclector.parse-result] 21
position-offset [eclector.base] 3

R

read [eclector.concrete-syntax-tree] 22
read [eclector.parse-result] 20
read [eclector.reader] 5
read-common [eclector.reader] 6
read-delimited-list [eclector.reader] 5
read-from-string
[eclector.concrete-syntax-tree] 22
read-from-string [eclector.parse-result] 21
read-from-string [eclector.reader] 5
read-maybe-nothing [eclector.reader] 7
read-preserving-whitespace
[eclector.concrete-syntax-tree] 22
read-preserving-whitespace
[eclector.parse-result] 20
read-preserving-whitespace
[eclector.reader] 5
read-token [eclector.reader] 7
readtablep [eclector.readtable] 19
recover [eclector.reader] 23
reference-labeled-object
[eclector.reader] 15

S

set-standard-dispatch-macro-
characters [eclector.reader] 18
set-standard-macro-characters
[eclector.reader] 18
set-standard-syntax-and-macros
[eclector.reader] 18
set-standard-syntax-types
[eclector.reader] 18
source-position [eclector.base] 3
stream-position [eclector.base] 3
stream-position-condition [eclector.base] 3

W

with-forbidden-quasiquotation [eclector.reader] 18
wrap-in-function [eclector.reader] 18
wrap-in-quasiquote [eclector.reader] 17
wrap-in-quote [eclector.reader] 17
wrap-in-unquote [eclector.reader] 17
wrap-in-unquote-splicing [eclector.reader] 17