Eclector User's Manual

Table of Contents

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 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.

2.1.2 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.

2.1.3 Package for parse result construction features

The package for features related to the creation of client-defined parse results is named `eclector.parse-result`. 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.

2.1.4 Package for CST features

The package for features related to the creation of concrete syntax trees is named `eclector.concrete-syntax-tree`. 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.

2.2 Ordinary reader features

In this section, symbols written without package marker are in the `eclector.reader` package (see Section 2.1.1 [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 2.3 [Readtable features], page 10). These functions are discussed in the section Section 2.2.1 [Common Lisp reader compatible interface], page 3.
- The second category is comprised of the `*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.

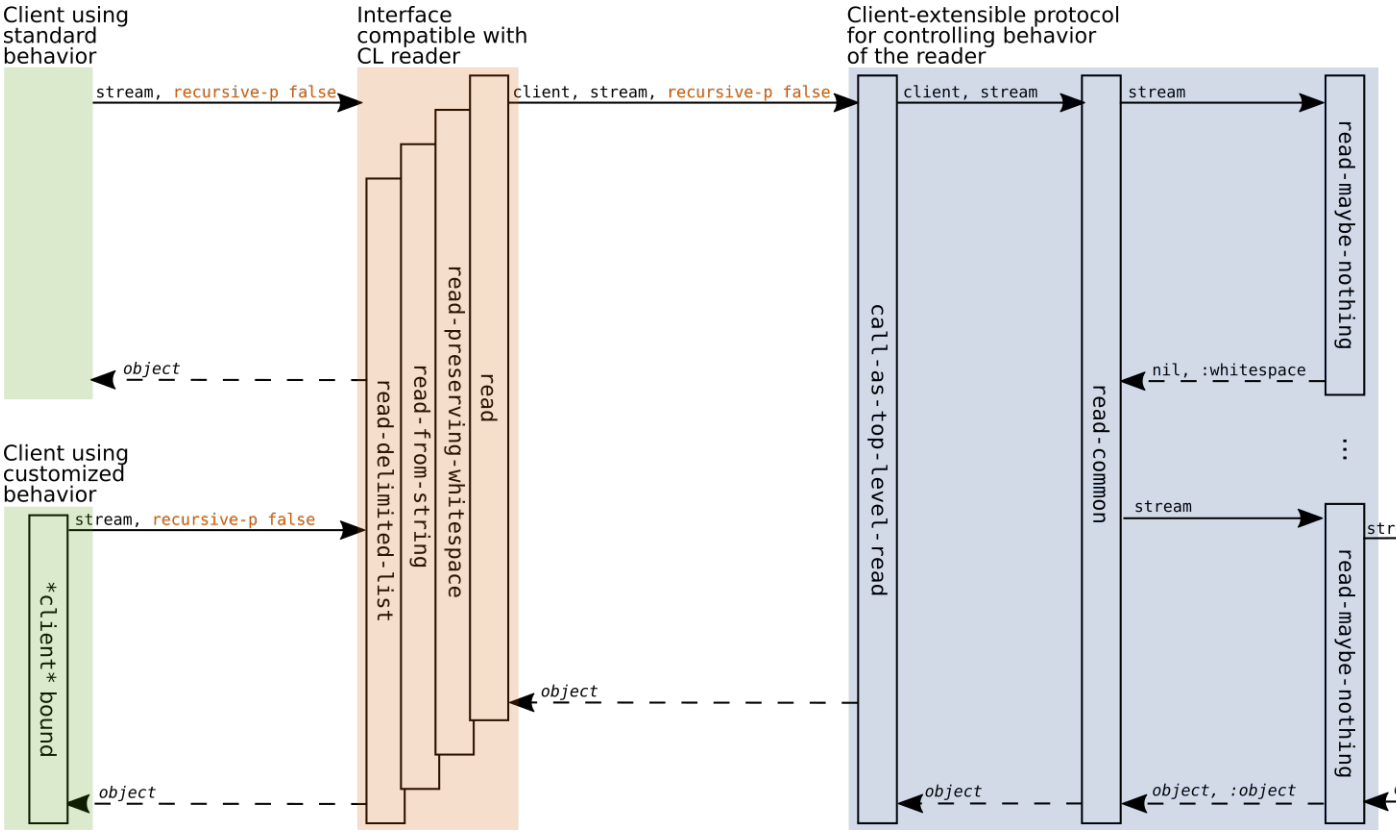

Figure 2.1: Functions and typical function call sequences. Solid arrows represent calls, dashed arrows represent returns from function calls. Labels above arrows represent arguments and return values.

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{,-preserving-whitespace,-from-string,-delimited-list}` are called by client code.

`*client*` [Variable]

This variable is used by several generic functions called by `read`. The default value of the variable is `nil`. Client code that wants 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.

2.2.1 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 `*client*` variable.

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

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

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

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

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

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

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

2.2.2 Reader behavior protocol

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

`call-as-top-level-read` *client thunk input-stream eof-error-p eof-value preserve-whitespace-p* [Defgeneric]

This generic function is called by `read`, if `read` is called with a true 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` *client input-stream eof-error-p eof-value* [Defgeneric]

This generic function is called by `read`, passing it the value of the variable `*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` *client input-stream eof-error-p eof-value* [Defgeneric]

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` *client input-stream reason* [Defgeneric]

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 `*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*` [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 2.4 [Parse result construction features], page 10).

`read-token` *client input-stream eof-error-p eof-value* [Defgeneric]

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` *client input-stream token escape-ranges* [Defgeneric]

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.

`interpret-symbol-token` *client input-stream token position-package-marker-1 position-package-marker-2* [Defgeneric]

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` *client input-stream package-indicator symbol-name internp* [Defgeneric]

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` *client input-stream char readtable* [Defgeneric]

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` *client name* [Defgeneric]
This generic function is called by the default `#\` reader macro function to find a character by name. *name* is the name that has been read converted to upper case. The function has to either return the character designated by *name* or `nil` if no such character exists.

`make-structure-instance` *client name initargs* [Defgeneric]
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` *client thunk package-designator* [Defgeneric]
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` *client expression* [Defgeneric]
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` *client feature-expression* [Defgeneric]
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` *client feature-expression* [Defgeneric]

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 feature expressions like `#+(and my-cl-implementation (special-feature a b))` to succeed 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.

`fixup` *client object seen-objects mapping* [Defgeneric]

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). *mapping* is a hash table of substitutions, mapping marker objects to replacement objects. A method specialized on a class, instances of which consists of parts, should modify *object* by scanning its parts for marker objects, replacing found markers with replacement object and recursively calling `fixup` for all parts. `fixup` is called for side effects – its return value is ignored.

Default methods specializing on the *object* parameter for `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.

2.2.3 Quotation and quasiquotation

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

`wrap-in-quote` *client material* [Defgeneric]

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` *client form* [Defgeneric]

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` *client form* [Defgeneric]

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` *client form* [Defgeneric]
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` *context* `&optional` [Defmacro]
(*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.

2.2.4 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 2.3 [Readtable features], page 10). The following functions are provided for this purpose:

`set-standard-syntax-types` *readtable* [Function]
This function sets the standard syntax types in *readtable* (See HyperSpec section 2.1.4.)

`set-standard-macro-characters` *readtable* [Function]
This function sets the standard macro characters in *readtable* (See HyperSpec section 2.4.)

`set-standard-dispatch-macro-characters` *readtable* [Function]
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` *readtable* [Function]
This function sets the standard syntax types and macro characters in *readtable* by calling the above three functions.

2.3 Readtable features

In this section, symbols written without package marker are in the `eclector.readtable` package (see Section 2.1.2 [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` *object* [Defgeneric]

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

2.4 Parse result construction features

In this section, symbols written without package marker are in the `eclector.parse-result` package (see Section 2.1.3 [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`.

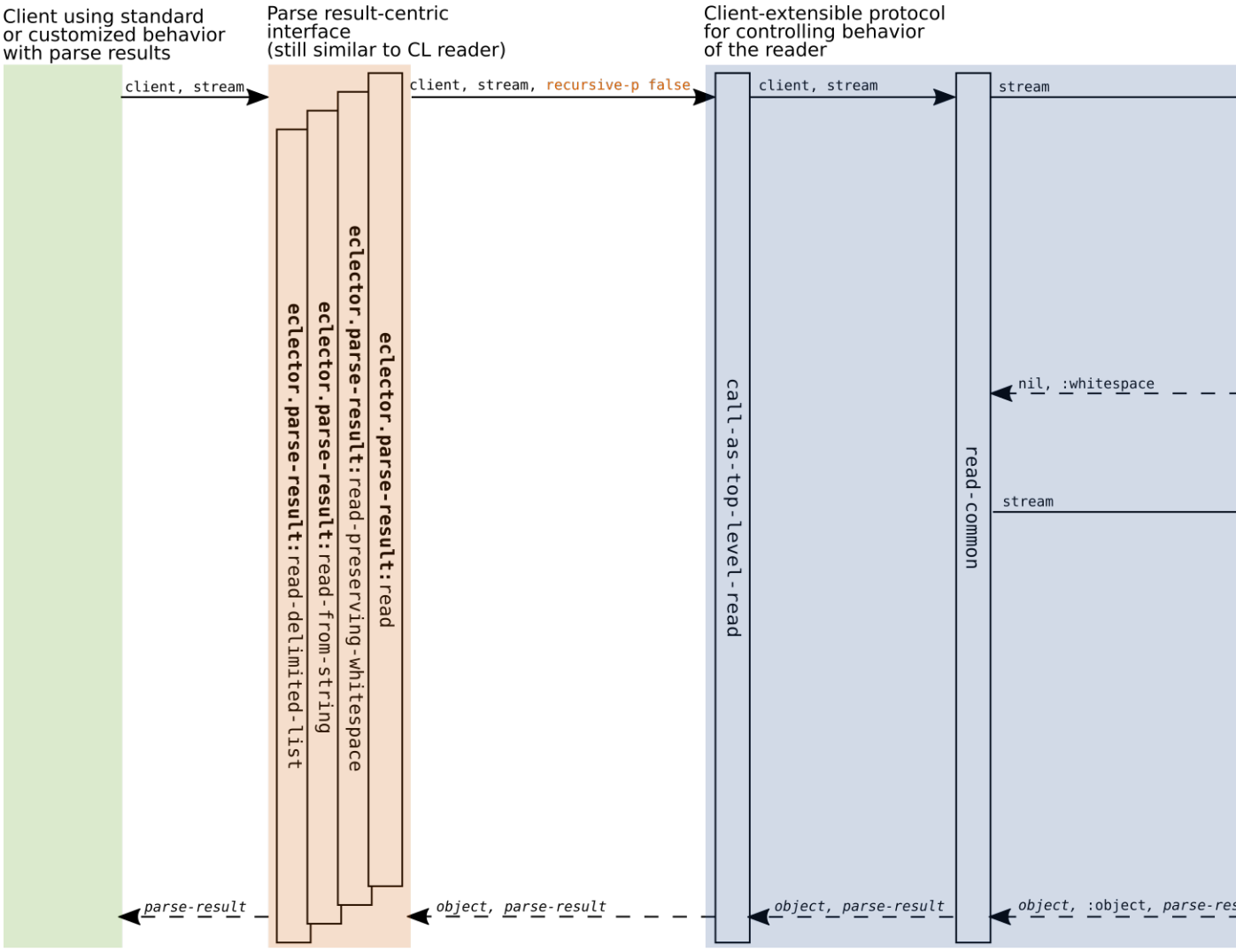

Figure 2.2: Functions and typical function call sequences. Solid arrows represent calls, dashed arrows represent returns from function calls. Labels above arrows represent arguments and return values. Differences from the non-parse result version are highlighted with bold text.

Figure 2.2 shows typical function call patterns that arise when the functions `read{,-preserving-whitespace,-from-string,-delimited-list}` are called by client code.

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

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

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` *client string* `&optional` (*eof-error-p* `t`) (*eof-value* `nil`) `&key` (*start* `0`) (*end* `nil`) (*preserve-whitespace* `nil`) [Function]

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` [Class]

This class should generally be used as a superclass for client classes using this package.

`source-position` *client stream* [Defgeneric]

This generic function is called in order to determine the current position in *stream*. The default method calls `cl:file-position`.

`make-source-range` *client start end* [Defgeneric]

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`*`)`.

`make-expression-result` *client result children source* [Defgeneric]

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 `make-source-range` and `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` *client stream reason source* [Defgeneric]

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.

2.5 CST reader features

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

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

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

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` *string* `&optional` (*eof-error-p* `t`) (*eof-value* `nil`) [Function]
`&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 &optional`*condition* [Function]

> 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:*readtabe*` 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 18.

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 17.

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 17, 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 17, 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 17.
- `#S` as described in Section 4.2.3 [Structure instance creation], page 18.
- `(`, `#(` and `#S` as described in Section 4.2.4 [Circular structure], page 18.
- 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.

Concept index

Function and macro and variable and type index