Highly configurable Common Lisp reader

Jan Moringen
jan.moringen@posteo.de
Unaffiliated

Robert Strandh
robert.strandh@gmail.com
Unaffiliated

ABSTRACT

The Common Lisp `read` function (usually called the Common Lisp *reader*) is an essential part of every Common Lisp implementation, because it is used to read Common Lisp code for subsequent evaluation. The readers of most existing Common Lisp implementations do this well, but little more. However, there are many situations where it is useful to read Common Lisp code for other purposes as well, and in those situations it can be essential to be able to configure the reader in various ways, for instance so that it does not fail for symbols with incorrect package markers. In this paper, we describe Eclector, a highly configurable Common Lisp reader. Configuration is accomplished by the extensive use of Common Lisp generic functions called by Eclector and that can be specialized by client code.

ACM Reference Format:
Jan Moringen and Robert Strandh. 2026. Highly configurable Common Lisp reader. In *Proceedings of the 19th European Lisp Symposium (ELS'26).* ACM, New York, NY, USA, 8 pages. https://doi.org/xx.xxxx/xxxxxx.xxxxxxx

1 INTRODUCTION

The Common Lisp[1] `read` function, usually referred to as the *reader*, is an essential part of any Common Lisp implementation. It is normally used as the first step of the `read-eval-print` loop or in the file compiler, in both cases reading forms that are then passed to the evaluator. Furthermore, when a Common Lisp system is built (or *bootstrapped*), the reader is needed early since code has to be read before it can be compiled or evaluated. But reading native code for compilation or evaluation is not the only use for a reader. Some situations where a reader might be useful are:

Cross compilation When a Common Lisp system (the *target*) is built from source, a common technique is to compile the source on an existing Common Lisp system (the *host*). The host system can be (perhaps a different version of) the same implementation as the target, or it can be a different implementation altogether. The reader is then executed by the host Common Lisp system. In either case, it is undesirable for the reader to intern symbols in host packages. The SBCL implementation of Common Lisp solves this problem by having the source code defined in a package that cannot clash with the host packages. The main disadvantage of this approach is that all source code that is part of the target system must be specific to that system. External libraries are excluded if such libraries involve packages that may clash with some host packages. With a highly configurable reader, cross compilation can be addressed in a different way.

Parsing an editor buffer As a concrete example of this use case, consider the (bad) Common Lisp program in Listing 1 which a user might type into an editor or integrated development environment (IDE) for Common Lisp code. The IDE could support the user with accurate syntax highlighting[1] and notify them about the syntactic errors[2], semantic errors[3] and style issues[4].

Existing editors for Common Lisp code do not provide these features to the full extent because they cannot analyze the buffer contents faithfully enough. Many assumptions are made that may not hold true. For example, reader macros are not taken into account, and the analysis does not permit the editor to accurately determine the role of symbols in the buffer.

The only technique that can faithfully analyze a buffer of Common Lisp code is the reader. But then, again, it is undesirable for the reader to intern symbols in any package, and the analysis must not fail for errors such as incorrect package names used as package prefixes, incomplete code, or syntax errors in tokens. Consequently, an extended Common Lisp reader that is configurable and can recover from errors is necessary for this approach but not sufficient on its own. Instead, such a reader has to collect enough information about errors and the input text for subsequent stages to process.

```
1  ;; Comment
2  (defun get-name (defun)
3    #+sbcl (cl:::nth
4      #+sbcl #b0102 #-sbcl #b0100
5      defun
6    )
```

Listing 1: A Common Lisp program with multiple errors and style issues that a user could type into an editor buffer.

In this paper, we describe Eclector; a Common Lisp reader that is independent of any particular Common Lisp implementation. Furthermore, Eclector is highly configurable, so that it may be configured not to intern symbols in any host package, and so that syntax errors can be avoided or recovered from.

[1]For example, the second occurrence of `defun` should be highlighted as a parameter or variable, not as the `cl:defun` operator. Ironically, the LaTeX package used for typesetting the listing gets this wrong.
[2]Examples of syntactic errors are three package markers in `cl:::nth`, a digit that is invalid for the chosen radix in `#b0102` and a missing closing parenthesis at the end of the program.
[3]Only one branch of the nested read-time conditional `#+sbcl #b0102 #-sbcl #b0100` is reachable.
[4]Toplevel comments are specified to use three semicolons. Closing parentheses should not be placed on separate lines.

ELS'26, May 11-12, 2026, Kraków, Poland

ACM ISBN 978-2-9557474-3-8.
https://doi.org/xx.xxxx/xxxxxx.xxxxxxx

2 EXISTING COMMON LISP READERS

In this section, we give an overview of the native readers of some common free or open-source existing Common Lisp implementations. We focus on behavior in error situations and features like tracking of source positions that can be essential in various use cases, particularly cases where the reader is not used as the first phase of a `read-eval-print` loop such as processing the contents of editor buffers as illustrated in Listing 1. We list the responses to invalid inputs not as a remark on the quality of the respective implementation but instead to evaluate whether those native readers could be used to read (a heuristic approximation of) invalid inputs as is required in certain use cases.

For the behavior reported here as well as for the benchmarks described in Section 5, the following implementations and versions were used:

Implementation	**Version**
Embeddable Common Lisp (ECL)	commit 3d83b007
Steel Bank Common Lisp (SBCL)	2.6.0.debian
Clozure Common Lisp (CCL)	1.13 (v1.13-21-gda7138ef) LinuxX8664

Like Common Lisp, the Scheme programming language employs a reader for turning textual source code into S-expressions. In modern versions of Scheme, *syntax objects* are used to address some of the challenges we present here. However, we consider syntax objects out of scope due to differences in design between Scheme and Common Lisp.

2.1 Embeddable Common Lisp (ECL)

ECL is implemented as a core written in C. It is built by first compiling this core and then loading Common Lisp code into the system created form the core. For that reason, the reader must be written largely in C.

When an attempt is made to read an incomplete expression, so that the `end-of-file` condition is signaled, only the restart `restart-toplevel` is available. Similarly when an attempt is made to read a symbol with an explicit package marker where the corresponding package does not exist, an error of type `simple-error` is signaled, and only the restart `restart-toplevel` is available. When an attempt is made to read an incorrect token in the form of a malformed symbol with three package markers, an error of type `simple-error` is signaled, and the message is cryptic. For example, an attempt to read the malformed symbol `cl:::car`, results in the error message "There is no package with the name `CL:::C`".[5]

One might think that the reason ECL proposes so few restarts is because it might be difficult to do so for a function written in C but that is not the case; ECL has full support for functionality analogous to `restart-case` for C code.

The ECL reader is not involved in source tracking. The compiler tracks the source only of top-level forms, so recursive calls to the reader need not be handled.

[5]We will call these three invalid inputs "*incomplete* input", "*missing package* input" and "*triple colon* input" respectively in the following sections.

2.2 Steel Bank Common Lisp (SBCL)

The SBCL reader is written in highly optimized Common Lisp, with strategies like sophisticated management of buffers, elision of type-checks and bounds-checks, and without the use of the specific features of the Common Lisp Object System (CLOS) such as standard classes and generic functions (which are relatively slow compared to ordinary functions in SBCL).

For the *incomplete* input, the `end-of-file` condition is signaled and no restart other than `retry` and `abort` is available. For the *missing package* input, an error of type `simple-reader-package-error` is signaled. Several restarts are then available:

- using the current package instead,
- retry finding the package (presumably after it has been created by the user),
- specify a different package,
- return an uninterned symbol, and
- specify a symbol to return.

For the *triple colon* input, an error of type `simple-reader-error` is signaled and no restart other than `retry` and `abort` is available.

The SBCL reader provides several extensions. One such extension is that a package prefix with two colon characters can precede any expression, and not just a symbol. The effect is that the reader then uses the package named by the prefix as the default package to intern symbols in the expression. Another extension is that the exponent marker `r` or `R` can be used to return a rational number that is the exact value of the floating-point syntax used. The names of symbols created by the SBCL reader can be normalized according the "Normalization Form KC" which is specified for Unicode. Whether symbol names are normalized in this way is controlled by the accessor `readtable-normalization` which operates on a readtable.

The SBCL reader can track source positions of each object returned. Source tracking is handled through a special type of stream. The function `%read-preserving-whitespace` checks for the stream argument being of this type and, if so, registers the form and the position of the form. Two parallel adjustable vectors are used for this purpose.

2.3 Clozure Common Lisp (CCL)

The CCL reader is written in Common Lisp, without the use of the specific CLOS features such as generic functions and standard classes.

For the *incomplete* input, the `end-of-file` condition is signaled and only restarts that return to previous break levels are available. For the *missing package* input, several additional restarts are available:

- supplying a different name for the package,
- retry finding the package (presumably after it has been created by the user),
- making the supplied package name a nickname for the package `common-lisp-user`.

For the *triple colon* input, no error is signaled at all, and instead the third package marker is treated as escaped, so that it is part of the symbol.

The CCL reader can track source positions. It registers the stream position before and after an expression has been read, and creates an instance of a `struct` named `source-note` and stores this instance in a hash table. Source positions are recorded only if the stream is an element of the list `*recording-source-streams*`, which is the case when the reader is called for the purpose of evaluating the resulting expression.

2.4 Clasp

Clasp is a Common Lisp implementation that was specifically designed to interoperate with C++ code, in order to take advantage of existing C++ libraries. Clasp was originally based on ECL with significant changes for C++ interoperability. As such, its reader was initially a C++ conversion of the ECL reader. However, in later bootstrapping stages, Clasp loads Eclector, the subject of this paper, and makes it the default reader.

There are plans to modify the bootstrapping procedure of Clasp so that the initial C++ reader is no longer needed. For these reasons, we do not review the C++ reader of Clasp.

3 ECLECTOR: A HIGHLY CONFIGURABLE READER

In this section, we discuss features and other characteristics of Eclector. Eclector was initially created as the native reader for the SICL[6] project, still under development. From the start, it was designed, in addition to providing the reader for SICL, to make it possible to parse an editor buffer as described in a 2018 ELS paper [2]. Early on, the code was extracted from SICL into a separate project and renamed to Eclector[7]. From then on, it evolved entirely independently of SICL, through the work of the first author of this paper and the project goals became more clearly defined as well as broader in scope:

(1) Eclector should be written in portable Common Lisp.
(2) Eclector should provide a reader that strictly conforms to the Common Lisp specification.
(3) Eclector should provide distinct condition types with precise and localizable error messages for all possible syntax errors.
(4) For invalid inputs, Eclector should be able to recover from any syntax error and produce a result that represents the invalid input as closely as possible if the corresponding restart is invoked.
(5) All aspects of Eclector should be extensible and customizable beyond what Common Lisp offers by means of special variables and readtables.
(6) Eclector should provide source location tracking and parse result construction, including the representation of *skipped material* (See Section 3.1.1.).
(7) Eclector should achieve acceptable performance in general (See Section 5.1.) and non-catastrophic performance degradation for unusual and pathological inputs such as deeply nested or highly circular expressions where possible (See Section 5.2.).

[6] https://github.com/robert-strandh/SICL
[7] Available at https://github.com/s-expressionists/eclector

After about a decade of slow-paced development in total, Eclector is now a relatively mature project which mostly meets the above goals and is made available under the "2-clause" BSD license with extensive documentation, many releases and several users. The codebase consists of around 5.000 lines of implementation code and around 5.000 lines of test code. The test suite achieves almost full line- and branch-coverage of the codebase and applies random testing techniques for certain aspects of the implementation.

The following sections discuss features that distinguish Eclector from other Common Lisp readers (Section 3.1) and describe some of the protocols that Eclector provides for customization (Section 3.2).

3.1 Features

3.1.1 Source location information. Eclector provides source location information to modules that can benefit from it such as error conditions and parse results (see below). By default, the source location information consists of the stream position, but clients can define a method on the relevant generic function to implement some other representation. For example, in SICL, the source information consists of the entire input file in the form of a vector of lines, and the line and column of the start and the end of the S-expression.

3.1.2 Parse results and skipped input. In addition to the return values of reader functions defined by the Common Lisp specification, Eclector can produce *parse results* which can capture more information about the input: each input expression, even atoms, can be represented as unique objects so that source location information can be attached unambiguously. Furthermore, *skipped material* can be represented as parse results. By skipped material we mean input which a standard reader skips, such as comments and inapplicable reader conditionals as well as input that a standard reader processes automatically without providing access to the original input such as read-time evaluation, labeled object definitions and references, and the feature expressions in reader conditionals. Clients can configure if and how parse results are represented.

While parse results and the representation of skipped material serve little purpose when a reader is used as the first step of a `read-eval-print` loop, they are essential when the reader is used to parse the contents of an editor buffer.

3.1.3 Specific error condition types. For each error situation, Eclector signals a condition of a type that is specific to that situation. That way, client code can handle each such situation differently. Eclector uses the acclimation[8] library to enable error messages in multiple languages but messages for languages other than English have not been added at this time.

3.1.4 Recovery from errors. For every error situation, restarts are available that allow client code to indicate to Eclector that it should recover from the error. For example, if a (specific subtype of) `end-of-file` condition is signaled, recovery consists of acting as if the input is complete, so that string literals, lists, vectors, etc., are closed. If the name of a non-existing package is used as a package prefix for a symbol, a restart can be invoked that returns an uninterned symbol instead.

[8] https://github.com/robert-strandh/acclimation

3.1.5 Intrinsic and extrinsic use. Eclector can be loaded into an existing Common Lisp implementation without affecting the native reader, or any other system-specific code. It will then define several Eclector-specific packages that a client must refer to in order to use Eclector. We call this way of using Eclector "extrinsic".

But Eclector can also be used as the native reader of a new Common Lisp implementation, or to replace the existing reader of an existing Common Lisp implementation. Its main functions will then be defined in the `common-lisp` package. We call this way of using Eclector "intrinsic". Intrinsic use in a new Common Lisp implementation will require some specific bootstrapping consideration, since CLOS is used extensively both in the implementation of Eclector and for configuring it.

3.1.6 Unusual inputs and robustness. While Eclector will likely be used for reading source code in most applications, the possibility of using Eclector *intrinsically* requires it to handle any kind of input the user of a Common Lisp implementation could apply `read` to. As one example, for data serialized as S-expressions, the lengths of lists and vectors, the nesting depths of expression, the frequency of circular structure and the overall size of the input can be vastly different from source code.

To account for this requirement, Eclector is designed to avoid unbounded recursion where possible. Instead, functions that could cause unbounded recursion if expressed with naive recursion, initially process the input recursively but track the recursion depth and employ iteration and a worklist in case the recursion depth exceeds a threshold. However, not all recursion in Eclector is handled using this approach at the moment. In particular, `read` calls cannot nest to an arbitrary depth with the current design. It may be possible to leverage the reader state protocol (See Section 3.2.3.) to break up these nested calls into chunks manageable by recursive processing but we have not attempted this technique.

3.2 Protocols provided by Eclector

The possibilities for customization that Eclector provides are realized in the form of a number of protocols. This section presents some of the noteworthy protocols[9].

3.2.1 The client parameter design pattern. For providing an optional and unintrusive customization mechanism to clients of the library, Eclector uses the "client parameter" design pattern which is employed in the same way in other libraries related to the SICL project as well[4, Section 3.1]. In short, the pattern consists of adding an additional parameter, usually called `client` and positioned as the first element of the lambda list, to generic functions that should be customizable. The library is structured such that a client (user) of the library can easily arrange for the relevant generic functions to be called with a particular value for the `client` parameter. This organization allows clients to define methods specialized to the class of the client-supplied value on the customizable generic functions and have those methods be the most specific and thus in charge of the behavior of the respective generic function. Eclector provides a special variable named `eclector.base:*client*` which users of the library can set or bind to a client object. Functions like `eclector.reader:read` pass the value of the variable to customizable generic function. As a result, the high-level reader interface is unchanged compared to the Common Lisp specification but aspects of the reader behavior can be customized by defining methods and binding a special variable around calls of the high-level reader function. Listing 2 shows a very simple example of this customization technique.

[9]A complete description of the protocols Eclector provides for customization can be found in the "External protocols" section of the manual.

```
(defclass my-client () ())
(defmethod eclector.reader:interpret-token
    ((client my-client) (stream t) (token t) (escape-ranges t))
  (let ((result (call-next-method)))
    (if (typep result 'number) (- result) result)))
(let ((eclector.base:*client* (make-instance 'my-client)))
  (eclector.reader:read-from-string "(5 #C(6 7))"))
⇒ (-5 #C(-6 -7)), 11
```

Listing 2: A simple customization that makes the reader negate any number tokens (`#C` is a reader macro, so only the parts are affected) before returning them to the caller.

3.2.2 Reader behavior protocol. The Common Lisp specification describes the core of the reader, i.e. excluding reader macros, in terms of a "reader algorithm" which employs concepts such as toplevel and recursive `read` calls, skipping and preserving whitespace, interpretation of tokens, invocation of reader macros and more. However, only a subset of these concepts and algorithmic aspects are reflected in the programming interface of the Common Lisp reader, namely the functions `read`, `read-from-string`, `read-preserving-whitespace`, and `read-delimited-list`. Eclector tries to promote the mentioned concepts and algorithmic aspects to first-class status by providing protocol functions such as `call-as-top-level-read`, `read-common`, `read-maybe-nothing`, and `interpret-token` as well as more specialized protocols, some of which will be discussed in subsequent sections. As demonstrated in Listing 2, these generic functions allow clients to customize the basic behavior of the reader.

3.2.3 Reader state protocol. Any native Common Lisp reader (and the intrinsic version of Eclector does this as well) consults several special variables in order to determine actions in various situations. These special variables are set by user code, by other features of the Common Lisp system, and (rarely) by the reader itself. For example, the reader uses the special variable `*readtable*` to determine which readtable to use to determine the *syntax type* associated with a character. Another example of such a special variable is `*read-base*` that determines the numeric base for reading numbers.

When Eclector is used extrinsically, it is clearly not possible to use the host versions of these variables. For that reason, among others, Eclector defines a *reader state protocol* that removes the direct use of these special variables. Instead, the reader state protocol uses *symbols* (often the same symbols that name the host special variables) called *aspects* to denote the variables.

The reader state which Eclector queries and manipulates through the reader state protocol also contains aspects that do not denote any of the standard special variables: one aspect controls whether

the consing dot is legal in the current state. Another aspect tracks the quasiquotation nesting level. User code can in principle add additional, custom aspects. The intention of the reader state protocol is to represent the entire state of the reader as one or more first-class objects. Such a first-class representation allows clients to capture and restore the state, for example in order to re-`read` certain parts of a longer input or to track dependencies between expressions. Both capabilities are required for incremental parsing use cases (See Section 4.1.6.).

The reader state protocol consists of a small number of generic functions like `state-value`, `(setf state-value)` and `call-with-state-value` that are called by Eclector and client code to query or modify the aspects of the reader state. Each of these generic functions will have an aspect as one of its arguments. These functions also have a `client` parameter that can be used by client code to create specific methods as explained in Section 3.2.1.

3.2.4 Labeled objects. The Common Lisp standard specifies the reader macros `#N=expression` and `#N#` for defining and referencing labeled objects respectively. We call *N* the *label*. The standard describes the processing of these labeled objects in terms of the observable results and some constraints. Eclector implements this description by processing definitions and references for each label *N* according to the state machine illustrated in Figure 1 and a corresponding protocol of first-class objects and operations (discussed below).

Each label starts out as *undefined* and potentially transitions to *defined*, *circular*, *final* or *referenced*. Based on this state machine, Eclector provides a customizable protocol for processing labeled objects with generic functions like `note-labeled-object`, `forget-labeled-object`, `reference-labeled-object`.

The second part of handling labeled objects is processing the objects produced by the reader before they are returned to the caller such that labeled object references[10] are replaced by the respective referenced object. We refer to this process as *fixup*. Eclector uses a *fixup graph* data structure to manage the algorithmic complexity of pathological inputs (See Section 5.2.) as well as another protocol to make the fixup processes extensible and customizable. In particular, clients can add support for fixing up objects such as hash-tables that don't have a literal syntax in the standard, or implementation-specific objects. The fixup protocol consists of generic functions for controlling the traversal of the fixup graph like `walk-fixup-tree` as well as functions for controlling the substitution in individual objects such as `fixup` and `new-value-for-fixup`.

4 APPLICATIONS

In this section, we describe use cases that benefit from features of Eclector like extensibility, source position tracking and error recovery. We also list applications of Eclector in existing Common Lisp projects that are concrete examples of these use cases.

4.1 Use cases

4.1.1 Intrinsic use: using Eclector as the reader of a Common Lisp implementation. The most obvious application for Eclector is as a native reader of some Common Lisp implementation. Currently, Eclector is used this way only in Clasp, but it will also be used in the SICL Common Lisp implementation, once it is finished.

Currently, SICL uses Eclector extrinsically during bootstrapping, in order to read forms to be evaluated in a first-class global environment [3]. Since the host global environment must not be affected as a result of reading these forms, we take full advantage of the ability to configure Eclector to avoid such side effects.

4.1.2 Sandboxed evaluation. The goal of *sandboxed evaluation* is to (read and) evaluate untrusted code, for example code received from possibly adversarial third parties, in a way that makes it impossible to affect the surrounding environment except in limited, explicitly permitted ways. If a standard Common Lisp reader were used in the `read` phase of sandboxed evaluation, certain potentially dangerous behaviors could be restricted, for example via `*read-eval*`. However, malicious parties could still exploit the unmodifable behavior of a standard reader to impair the system: `read` can cause interning of symbols which permanently occupy memory so that the system could run out of memory. Deeply nested expressions could cause the system to signal an error that may be hard to recover from or such expressions could simply crash the system. A configurable reader can be configured such that these potentially dangerous default behaviors are restricted or prohibited.

4.1.3 Safe data serialization. Similar to untrusted *code* in the sandboxed evaluation use case, processing of untrusted *data* can be a concern, for example data is serialized in S-expression form and later deserialized. As a concrete example, the Lichat communication protocol[11] uses an S-expression-based wire-format. Such a format seems ideal for processing with the Common Lisp printing and reading facilities but as for sandboxed evaluation, the standard Common Lisp reader cannot be configured such that it is safe to apply to untrusted input. Another similar example is Gsharp[12] which serializes documents for storage on disk using S-expressions.

One important aspect of data serialization and deserialization is the fact that the S-expression structure of serialized data can be very different from typical code. For example, typical code contains few instances the reader macros for labeled objects which encode circularity and repeated occurrences of expressions. Serialized data on the other hand may be highly circular and contain many expressions which occur repeatedly. See Section 5.2 for a brief discussion of the potential performance impact of such unusual inputs.

4.1.4 Static analysis. For some applications doing *static analysis* of Common Lisp code, it is not possible, or at least undesirable, to use the native reader. A typical example would be a so-called *linter*, i.e., a tool for analyzing code with the purpose of giving the user more feedback about the code than a typical compiler will do. Such feedback can include violations of conventional style and suggestions for improving the readability of the code. Other examples include

- spelling mistakes in comments or symbol names,
- unnecessary package prefix in case the symbol in question is accessible in the current package or a package-local nickname allows a shorter spelling,

[10] This delayed substitution is required only in case of circular structure. In cases like `(#1=:a #1#)`, the referenced object can be substituted immediately.

[11] https://shirakumo.github.io/lichat/lichat.html

[12] https://github.com/robert-strandh/Gsharp

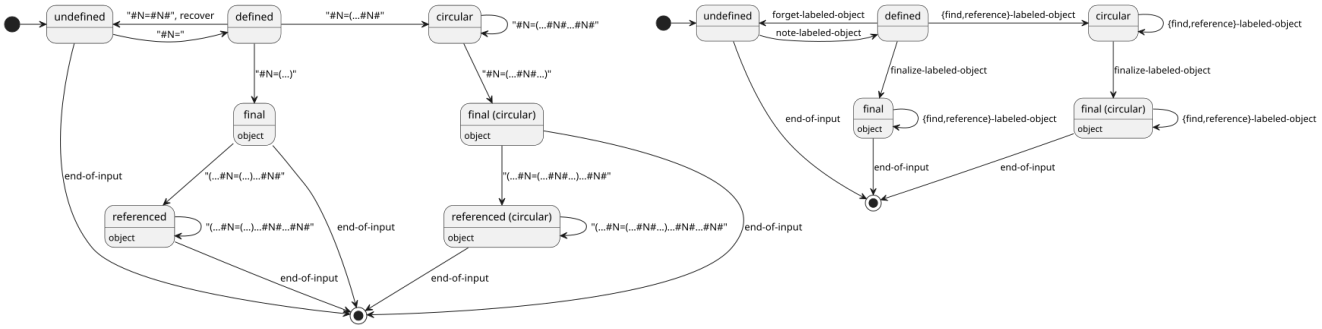

Figure 1: Labeled object state machine. The boxes with rounded edges depict the possible states of a label N. A Lisp object associated with the label in a given state is shown in the lower compartment. The state machine for each label starts in the `undefined` state before any input has been consumed and terminates at the end of input (since new references could appear at any point until then). *Left*: Full set of states with associated inputs. The edges represent state transitions that occur when input which resembles the label next to the edge is consumed. Note that the edge labels show all input consumed so far for the respective transition, not the new input compared to the origin state. *Right*: Reduced set of states used in the protocol with edges labeled by corresponding protocol functions.

- use of two-package markers when spelling an external symbol,
- use of the wrong number of semicolons for a comment at a given level (for example two semicolons for a comment at the toplevel),
- logical errors in reader conditionals such as `#+win32(... #+(and thread (not win32))(...) ...)`.

Such tools should preferably not have any side effects on the host Common Lisp image such as: interning symbols in some host package, altering the package system, or evaluating code.

The host reader can then not be used because of the possibility of read-time evaluation, or simply the use of double package markers for some symbol that will then be interned in some host package. And the code being analyzed may contain package markers for packages that have not been created in the host Common Lisp system.

An important feature of a reader to be used with such a tool is the desire to recover from errors such as syntax errors in tokens. The tool would then report the error and then continue the analysis.

In some cases, "invisible" objects such as feature conditionals, labeled objects and skipped material like comments have to be retained and represented explicitly when those things are the subject of the analysis.

Some features of the Common Lisp reader like reader macro and read-time evaluation remain challenging in the context of this use case, even for a highly configurable reader implementation with error recovery and source tracking.

4.1.5 Syntax highlighting. Coloring or otherwise classifying elements of Common Lisp source code according to different syntactic or semantic criteria can be useful for applications such as editors, documentation preparation systems and browsers for code repositories. As mentioned before, the only practical way of evaluating such criteria is to parse the code using `cl:read` but typical Common Lisp readers are unsuitable. However, Eclector's ability to read incomplete, incorrect and untrusted code while avoiding side effects

```
0  (cl:defpackage #:foo
1    (:use #:cl)
2    (:export #:name)
3    (:size 1)) ; a comment
4
5  (in-package #:cl-user)
6
7  (defconstant name "abc") (defvar *foo* #P"path" "docstring")
8
9  (defclass foo (superclass)
10   ((%foo :initarg :foo :type foo :initform :foo))
11   (:documentation "docstring")
12   (:default-initargs :foo #b01021 :bar '#1=#(a b #1# c d)))
13
14 (deftype my-integer (lo hi)
15   "docstring"
16   (list 'integer lo hi))
17
18 (defun foo (foo &key (list (list (random 1)) list?))
19   "another docstring"
20   (declare (type list list)
21            (ftype (function (t t) (values list)) list)
22            (inline cl:list)
23            (optimize (speed 3)))
24   (loop :for value :from -5 :below pi :by 1/3
25         :when (plusp value) :collect value))
26
27 cl:car  #| the errors |# ml:boat cl::plane (not closed
```

Figure 2: Common Lisp code with syntax highlighting. Color indicates namespaces: `dark-orange`: package, `blue`: function, `dark-red`: variable, `forest-green`: type, `yellow4`: declaration, etc. Bold text indicates definitions. Gray background indicates quoted material. Errors are marked by red underlines.

on the Lisp environment, tracking source locations, and retaining material like comments that would be otherwise be skipped, makes it suitable for this use case. Figure 2 shows Common Lisp code with syntax highlighting in a CLIM-based application based on Eclector parse results and further analysis of the S-expression syntax.

4.1.6 Incremental parsing for editor-like applications. Eclector's ability to recover from errors, represent the state of the reader as a first class object, produce parse results for expressions as well as skipped material with source information and operate without side-effects enable incremental parsing. Incremental parsing is useful in interactive, editor-like applications like REPLs, Listeners, IDEs,

Language Server Protocol servers and of course source code editors. For example, the incremental parsing technique first presented at ELS 2018[2] has been extended to use Eclector. One of the notable extensions consists in tracking of dependencies between expressions such as `(in-package ...)` and subsequent expressions that arise due to the effects on the reader state of the former expression.

4.2 Projects using Eclector

A quick survey of publicly available projects using Eclector revealed about a dozen project. The following section discusses a few of the interesting cases.

4.2.1 formgrep. The library `formgrep`[13] provides functionalities similar to those of the UNIX `grep` utility, except that it works on S-expressions rather than text. This library takes a collection of files and a regular expression as arguments. It then uses Eclector to parse each file. It defines a method on the Eclector generic function `interpret-symbol` that uses the regular expression on the token representing a symbol that Eclector recognizes, to determine whether there is a match. If so, it accumulates the token in a list of matches that is ultimately returned. For reader conditionals, `formgrep` configures Eclector to always read the guarded expression regardless of the condition so that no part of the code is skipped and thus excluded from the search. `formgrep` uses the idiom `(handler-bind ((error #'eclector.base:recover)) ...(read ...) ...)` to deal with parts of the input that are either invalid or cannot be read with the employed Eclector configuration.

4.2.2 mallet. The purpose of the library `mallet`[14] is to read Common Lisp code and analyze it according to configurable rules in order to detect mistakes and deviations from a desired coding style. The library configures Eclector so that it behaves differently compared to the standard reader in multiple ways:

(1) Eclector is configured to produce parse results with source information.
(2) `mallet` recovers from many errors by specifically handling a number of the specialized conditions which Eclector signals.
(3) `mallet` handles elements of the code that cannot be read in the usual way without affecting the Lisp image such as symbols that would require interning or read-time evaluation, by configuring Eclector to produce placeholder objects instead.

4.2.3 Second Climacs editor. As mentioned in Section 3, Eclector was designed from the start to be able to parse the contents of an editor buffer. We use the ability to do incremental parsing in the Second Climacs text editor, still under development. The objective is to have an editor specifically for writing Common Lisp code that can analyze the buffer contents more faithfully on the level of character syntax, S-expression syntax and semantics than existing editors can do. The character syntax is handled by a combination of Eclector and the incrementalist library[15] which uses an extension of the incremental parsing described in Section 4.1.6. The S-expression syntax is handled by a separate library that accepts the concrete syntax trees produced by Eclector and applies grammar rules which describe the Common Lisp operators.

[13] https://github.com/death/formgrep
[14] https://github.com/fukamachi/mallet
[15] http://github.com/s-expressionists/incrementalist

4.2.4 Coalton language. The Coalton language[16] uses Eclector as part of its reader which is in turn used in both the compiler and the included IDE of the language implementation. The main reason for using Eclector seems to be its source tracking capability.

5 PERFORMANCE

5.1 General benchmarks

As a rough indication of the Eclector's efficiency when reading Common Lisp code, we measured the time it took the native readers of different Common Lisp implementations as well as extrinsic Eclector and Eclector configured to produce parse results in those same implementations to read the contents of Common Lisp files taken from real-world open source libraries. Code similar to the following was used to perform the benchmarking:

```
(let ((content (alexandria:read-file-into-string filename)))
  #+sbcl (sb-ext:gc :full t)
  #+ccl  (gc)
  (the-cost-of-nothing:benchmark
   (with-input-from-string (stream content)
     (loop for form = (<<read>> stream nil stream)
           until (eq form stream)))))
```

where «read» is either `cl:read`, `eclector.reader:read` or `eclector.concrete-syntax-tree:read` depending on the benchmark.

We performed the benchmark described above for SBCL, CCL and ECL with default optimization settings. We arbitrarily selected the following large Common Lisp files as inputs for the benchmark runs: from the library `flexi-streams-20241012-git` the file `enc-cn-tbl.lisp` with 48,418 lines and from the library screamer-20210807-git the file `screamer.lisp` with 7,233 lines. With Eclector commit `b2b22cdf` we obtained the following results on an `Intel(R) Core(TM) i7-4610M CPU @ 3.00GHz` machine with 15 GB of memory:

Reader	enc-cn-tbl.lisp	screamer.lisp
SBCL native	0.034 s	0.011 s
SBCL Eclector	0.161 s	0.052 s
SBCL Eclector CST	0.475 s	0.096 s
CCL native	0.186 s	0.047 s
CCL Eclector	1.249 s	0.607 s
CCL Eclector CST	4.108 s	0.961 s
ECL native	0.118 s	0.045 s
ECL Eclector	1.500 s	0.450 s
ECL Eclector CST	7.000 s	1.000 s

We observe that, in a given Common Lisp implementation, the respective native reader takes 8 % to 20 % of the time taken by Eclector to read the same input. On the other hand, Eclector on SBCL is about on par with the native reader of CCL. When producing parse results, Eclector takes significantly more time, so that the native readers read the same input in 1.7 % to 11 % of the time.

5.2 Pathological inputs

As a second indication of performance characteristics, we have chosen two pathological inputs, namely `#1=(#2=(#3= ...#N#=(#1#`

[16] https://github.com/coalton-lang/coalton

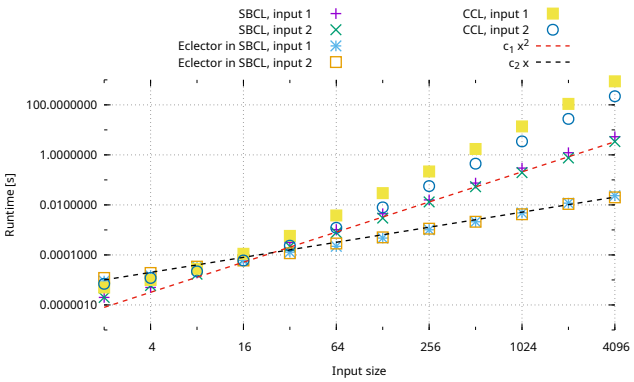

Figure 3: Runtimes in seconds for SBCL's reader and Eclector on the two input shapes and different input sizes. Both axes are log-scaled. Linear and quadratic relations with suitable constant factors have been added for reference.

`#2# #3# ...#N#)))` which we call "input 1 of size N" and `#1=(#2=(#3= ...#N=(1 2 3 #N#) ...#3#) #2#) #1#)` which we call "input 2 of size N". The benchmark consists of reading the two inputs for different sizes N with SBCL's reader and with Eclector, and measuring the respective runtimes. Figure 3 shows the runtimes for the different cases. For SBCL, the runtime is quadratic in the size N of the input for both input shapes. For Eclector, the runtime is linear in the size of the inputs.

The reason for the difference is that SBCL performs the fixup operation immediately on the return value of each finished `read` call. Since each result contains the previous results, this strategy leads to an amount of work proportional to $\sum_{1 \leq i \leq N} i \propto N^2$. Eclector, on the other hand, delays the fixup operation until the outermost `read` call returns which leads to an amount of work proportional to just N.

6 CONCLUSIONS

6.1 Challenges

As remarked at the end of Section 5.1, when executed in a given Common Lisp implementation, Eclector tends to run 4 (SBCL) to 10 (CCL) times slower than the respective native reader which is a significant slowdown. We suspect that two major factors cause the difference in performance:

(1) Eclector's portable code cannot use implementation specific optimizations that are available to native readers.
(2) The numerous possible ways in which Eclector can be configured via generic functions slow down the execution, in particular when the native reader does not use generic functions because ordinary functions perform better in the implementation in question.

We believe that Eclector's performance characteristics are still acceptable for most use cases: When a reader is used as part of file compilation or a `read-eval-print` loop, the time taken by the reader is usually dwarfed by the time taken by the compiler or evaluator. Conversely, for interactive uses like parsing an editor buffer at typing speed, no reader with subsequent analysis steps will be fast enough to process an entire buffer, with arbitrarily sized contents, quickly enough. In such a situation, Eclector can be used *incrementally* as described in Section 4.1.6 so that the amount of parsing work after each change is approximately constant.

Another challenging area is forward and backward compatibility, particularly in two areas: customization possibilities provided to clients and reader macros. For clients, in particular since Eclector already has some users, it is important to not introduce breaking changes in customization protocols. On the other hand, when Eclector's customization possibilities prove insufficient for some new use case, the protocols have to be changed or extended. Reader macros, which are typically not aware of Eclector and its facilities, can be at odds with many of the goals mentioned above such as avoiding effects, always being able to recover, etc. As a secondary issue, more customizability and more sophisticated mechanisms lead to a widening gap between reader macros that make use of Eclector's facilities and fully portable reader macros.

6.2 Future work

We are working on the following new customization protocols for Eclector:

The Common Lisp reader associates *constituent traits* with characters that occur within symbols and numbers. The traits associated with each character are fixed and listed in the Common Lisp specification. For Eclector, we plan to implement a "traits table" that allows clients to change the constituent traits associated with each character.

We are working on a customizable and optimizing quasiquotation processor for Eclector. Customizable here means, that quasiquotation support can be added for new literals such as hash-table literals. Optimizing means that quasiquoted material will be turned into forms that can be evaluated efficiently.

We plan to generalize the protocol for literal construction which currently consists of functions such as `find-character` and `make-structure-instance` to a more general signature like `make-literal` that would also handle numbers, symbols, arrays and other literals. This change will also incorporate the Quaviver library[17] for building floating point numbers as well as better handling of float overflows.

7 ACKNOWLEDGMENTS

We would like to thank John Cowan, Cristoph Keßler, Michael Babich and Michał Herda for reading a draft of this paper and for suggesting improvements.

REFERENCES

[1] *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp.* American National Standards Institute, 1994.
[2] I. A. Durand and R. Strandh. Incremental Parsing of Common Lisp Code. In D. Cooper, editor, *Proceedings of the 11th European Lisp Symposium*, Proceedings of the 11th European Lisp Symposium, pages 16–22, Marbella, Spain, Apr. 2018.
[3] R. Strandh. First-class Global Environments in Common Lisp. In *Proceedings of the 8th European Lisp Symposium*, ELS 2015, pages 79 – 86, April 2015.
[4] R. Strandh and I. Durand. A CLOS protocol for lexical environments. In J. E. Newton, editor, *Proceedings of the 15th European Lisp Symposium, ELS 2022, Porto, Portugal, April 21-22, 2022*, pages 20–26. ELSAA, 2022.

[17] https://github.com/s-expressionists/Quaviver/