

Cloning the Emacs Lisp Programming Language in R7RS Scheme

ANONYMOUS AUTHOR(S)

“Schemacs” is a recent work-in-progress software project with the goal of cloning the Emacs Lisp programming language and GNU Emacs user interface in portable R7RS Scheme such that the software can be run on any R7RS-compliant Scheme implementation. Prior art such as MIT/GNU Edwin only work on a single Scheme implementation, and do not evaluate Emacs Lisp code, though there have been attempts to implement Emacs Lisp for Edwin. For “Schemacs,” constructing a useful Emacs-like programming environment in Scheme that is backward compatible with GNU Emacs is one goal, but another is to answer two questions: how practical is the R7RS Scheme language for software engineering, and how portable is it? GNU Emacs is a well-known Lisp interactive programming environment primarily used for text editing and computer programming which originated at MIT in the mid 1970s that is still relatively widely-used today, especially among the Lisp programmers community. Emacs is described by many as an environment that allows end users to quickly and easily adapt the software to their personal needs through Lisp scripting. This paper is an exposition of the opinions of the authors informed by the experience of cloning Emacs in strictly R7RS-compliant Scheme. This paper discusses briefly what motivated this work, prior art, and discusses in depth some of the the practical engineering problems that have been encountered thus far. The authors hope the experience gleaned in this project can begin to answer the questions of practicality and portability of the R7RS standard, and we hope we might encourage more people to make better use of the language specified by the R7RS standard.

Additional Key Words and Phrases: GNU Emacs, Elisp, Scheme, R7RS, portability

ACM Reference Format:

Anonymous Author(s). 2025. Cloning the Emacs Lisp Programming Language in R7RS Scheme. 1, 1 (July 2025), 10 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION AND MOTIVATION

Schemacs is a work-in-progress clone of the GNU Emacs application, including a clone of the Emacs Lisp programming language, written in strictly R7RS-compliant Scheme code.

The Scheme programming language has been evolving for nearly half a century now, and in spite of it never becoming a widely-used language in industry, it still has interesting properties which seem to attract certain individuals who appreciate minimalism and the mathematical beauty of Lambda Calculus, for some definition of “beauty” and “minimalism.”

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Association for Computing Machinery.
XXXX-XXXX/2025/7-ART \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

Being a Lisp dialect, it is perhaps not surprising that there appears to be significant overlap between the community of Scheme users and the community of GNU Emacs Lisp users. And so it seems to be a natural consequence of this overlap that there have been more than a few attempts over the years to clone Emacs in the Scheme language.

The idea of replacing Emacs Lisp with Guile Scheme may be as old as the Guile implementation itself, which according to the history of Guile written in it’s manual [Developers 2024] dates back to the mid 1990s when Richard M. Stallman first announced that Guile would be the GNU project’s official extension language. The intention to replace Emacs Lisp with Guile was mentioned explicitly in the paper titled “Down with Emacs Lisp” [Neubauer and Sperber 2001] which elucidated various perceived problems with using Emacs Lisp as an extension language, especially that of dynamically scoped variables. But over the years GNU Emacs Lisp has gradually improved upon some of these perceived problems by, for example, providing lexical scoped variables, and native compilation [Corallo et al. 2020]. As a result, there is seemingly no longer a very pressing need for porting the Emacs Lisp runtime to Guile. Emacs Lisp seems to be evolving into more than just a scripting language, and is becoming a more practical and efficient programming language in it’s own right, one that resembles Common Lisp more than Scheme with the `cl-lib` providing macros such as `cl-loop`, and the EIEIO implementation of the Meta-Object Protocol inspired by CLOS.

Nonetheless work on various Scheme-based Emacs projects continue for somewhat enigmatic reasons that seem to go beyond mere practical utility or pedagogy. What motivates the authors of this paper to clone GNU Emacs and Emacs Lisp in Scheme is certainly curiosity, and the desire to use Scheme more often in an editor software which we use every day, but also our personal convictions that the R7RS standard [Shinn et al. 2021] itself is a useful programming language, and not just for academic research, but for practical software engineering projects. We feel the need to challenge conventional wisdom (or what we understand to be conventional wisdom) that Scheme programmers should simply choose a suitable implementation for their task and not bother with portability; we would like to choose the R7RS language standard itself as our Scheme, and find out what is required to make software written in this language work on real Scheme implementations.

Although the authors spend most of our time on Schemacs doing the ordinary software development work of implementing Emacs Lisp features, the experiences we have had so far which we believe will be of interest to readers of these proceedings is the question of how practical it is to engineer software applications, such as an editor that clones Emacs Lisp, while restricting ourselves to only using the R7RS standard programming language, including both the “Small”, and the yet

only partially completed R7RS “Large” standards. Another question which is likely to be interesting: how might software applications such as Schemacs be designed such that they may run on the largest number of R7RS-compliant Scheme implementations, especially for those which one might refer to as “industrial strength” Scheme implementations designed for general purpose software engineering applications, as opposed to smaller Schemes designed for niche applications.

Therefore, the rest of this paper will discuss prior art, an overview of technical problems encountered, approaches taken to solve these problems, recommendations for Scheme implementers which might make it easier for users of R7RS Scheme to solve these problems, as well as some of the techniques used in the software architecture such as monads and lenses. We conclude that R7RS is indeed a powerful language that is certainly useful for practical software engineering, but as a relatively new language (11 years old), it suffers from the lack of a mature software ecosystem, and we call on Scheme programmers to write more software targeting the R7RS standard in favor of a specific Scheme software platform, to help improve the software ecosystem.

2 PRIOR ART

There are *many* Emacs-like text editors, a few are even written in Scheme and provide Scheme as a scripting language. Most Emacs-like editors focus on cloning the “user experience” of Emacs, particularly the key bindings used to trigger updates to the state of the file being edited, but there have been a handful of attempts at cloning the Emacs Lisp programming language.

Scheme, Emacs Lisp, and Guile Scheme all have a multiple-decades long history, and it has been interesting reviewing the literature from the mid-1990s to early 2010s. There has long been talk of replacing Emacs Lisp with Scheme, and making Emacs Lisp a language implemented on top of Scheme. One of the reasons was because Emacs Lisp was originally a dynamically scoped language and there was a desire for using an extension language with lexical scoping [Neubauer and Sperber 2001]. This is no longer an issue with modern Emacs Lisp, as lexical scoping is now a feature of the language. When reading through the Guile mailing list of decades past, some people have noted that Guile’s bytecode interpreter and JIT compiler might have made Emacs faster. This is also no longer an issue with modern GNU Emacs, as Emacs provides ahead-of-time (AOT) compilation through `libgccjit` [Corallo et al. 2020]. Furthermore, Emacs is not typically used in performance-critical applications, and modern computers are fast enough that making Emacs Lisp more efficient is no longer likely to be a pressing concern for most Emacs users.

But much of the prior art for Schemacs emerged from this era, and so it is worth mentioning that although Schemacs may be similar to this prior art, it does not necessarily share all of the same goals. The reasons for implementing a Scheme-based Emacs today are very different from what the reasons may have been in the 1990s, or even as recently as the 2010s.

2.1 Guile-Based Emacs

Prior art that most closely resembles the goals of the “Schemacs” project is Ken Raeburn’s Guile-Based Emacs [Raeburn 2009]. The stated goal of the project was:

converting GNU Emacs to use Guile as its programming language. Support for Emacs Lisp will continue to exist, of course, but it may be through translation and/or interpretation; the Lisp engine itself may no longer be the core of the program.

Unfortunately, this software project appears to have been abandoned at the time of this writing. One big difference between this project and Schemacs is that there is no intention of actually replacing the GNU Emacs Lisp interpreter.

2.2 Guile Emacs Lisp

The Guile platform provides alternative language implementations, one of which is a small clone of the Emacs Lisp programming language [Developers 2024]. It first shipped with Guile in 2011 [Courtès 2011], and each new release of Guile continues to ship this Emacs Lisp implementation, although it seems Guile Emacs Lisp has not been updated much since it was first released, and the implementation in its current form is still incomplete and unable to run most existing Emacs Lisp programs. It would be trivial to replace this implementation with Schemacs in the Guile code base, if the Guile maintainers are interested.

2.3 Edwin

Another such project is the Edwin editor, a clone of Emacs version 18 [Adams et al. 2021] (the version of Emacs released in 1987) [manual editors 2025], and part of the MIT/GNU Scheme compiler. It is a complete, Emacs-like editor written entirely in Scheme. For reasons unknown to us, the ecosystem of software packages written to extend the functionality of Edwin is much smaller than that of GNU Emacs. Emacs users may likely find the lack of functionality of Edwin (relative to GNU Emacs) makes it somewhat restrictive for practical use. Although Edwin is portable to many operating systems and runs well on modern computing hardware, it seems that the maintenance of the Edwin editor has been kept to a bare minimum and has introduced relatively few new features since it was first released. The Schemacs Emacs Lisp interpreter can be ported to MIT/GNU Scheme, and could be made to allow Emacs Lisp code to run in Edwin.

In 1993, Matthew Birkholz wrote a paper, now designated *AI Memo TR-1451* [Birkholz 1993], describing an implementation of Emacs Lisp written for MIT Scheme and Edwin. Schemacs has the same goal, although Schemacs has the further goal of being portable to multiple Scheme implementations by closely adhering to the R7RS standard and the SRFIs.

2.4 The Guile-Emacs Project

Guile-Emacs [Templeton 2024] is a project in which the Guile Scheme compiler in the form of `libguile` is linked into GNU Emacs so that Guile Scheme can be used to write software that controls the state of the Emacs process directly. This creates one process that includes two separate programming languages running side-by-side, each of which can be used to write scripts that extend the functionality of the text editor. Since Guile-Emacs is specific to the Guile implementation of Scheme and is built specifically to work on top of the existing GNU Emacs code base, the direction of this work is orthogonal to that of Schemacs which aims to be an alternative to Emacs Lisp that works on any R7RS Scheme platform.

3 EXPERIENCES USING R7RS SCHEME TO IMPLEMENT SCHEMACS

In this section we discuss some of the difficulties encountered while implementing Schemacs while being restricted to using only features provided by the R7RS Small and the SRFIs.

The R7RS Small Scheme programming language provides a minimal set of features, arguably not enough features for industrial software engineering. The “Scheme Request for Implementation” (SRFI) process provides a way of standardizing new features and extensions to the Scheme language [Gleckler 2018] in a way that should encourage Scheme implementer to do so consistently, which theoretically can make code more portable. The R7RS Large Scheme programming language incorporates many of the SRFI documents into the language standard. Although *Large* is not yet ratified at the time of this writing, in our experience the SRFIs which have been incorporated into the *Large* standard still provide enough functionality that it is still a useful language for software engineering projects such as Schemacs. Beyond that, whether R7RS Scheme is suitable to write software applications is simply a matter of finding Scheme platforms which implement enough SRFIs to satisfy one’s engineering goals.

To people familiar with Scheme it is perhaps no surprise that the biggest difficulty encountered by the authors in implementing Schemacs using only portable R7RS Scheme code is that there is not much consistency in the set of features provided across various Scheme implementations. Though many Scheme implementations are compliant with R7RS Small, it is difficult to find any two implementations which implement the same subset of SRFIs. Though the Scheme programming language may have a half-century-long history, R7RS Scheme is still just barely over a decade old, and many of the SRFIs are even newer still. From our point of view, the R7RS software ecosystem has not yet matured, and this is the primary cause of most of the problems we have encountered.

3.1 Limitations of `cond-expand`

The `define-library` system and the `cond-expand` macro does make writing portable code possible, though writing `cond-expand` statements to fine-tune library definitions to

run across various Scheme implementations can be a fairly labor-intensive task.

One issue we encountered was that `#:keyword` syntax is allowed by several Scheme implementations, and used extensively throughout the Guile libraries, while other standards-compliant Scheme implementations (such as Chibi), throw a reader exception on these keywords, even when guarded by `cond-expand` expressions. So when defining a wrapper procedure around an implementation-specific API, if the API requires arguments be passed by keyword, it is not enough to use `cond-expand` to provide alternative procedure definitions in the same file. The Guile implementation for SRFI-69 `make-hash-table` is one example where arguments can be passed by keyword, namely where the programmer can specify whether to use weak key/value references. The following example which works for Guile will cause reader errors in some other Scheme implementations:

```
(define hashtable
  (cond-expand
    (guile
      (make-hash-table
        string=? string-hash #:weak 'keys))
      ;; Some readers fail here on " #:weak"
    (else
      (make-hash-table string=? string-hash))))
```

Because the `make-hash-table` API in Guile is defined to use non-standard syntax, `cond-expand` cannot prevent reader exceptions when porting that code to other Scheme implementations.

To work around this limitation, procedure definitions with non-standard syntax need to be moved into their own separate files and called through wrapper procedures which do use standard-compliant syntax. These wrapper procedures can be written into libraries such that Scheme implementations which are unable to read the non-standard syntax never include that code. This is accomplished through the use of the `include` form of the `define-library` syntax, where the `include` form is written within a `cond-expand` form.

The SRFI-69 specification states that `make-hash-table` can take arbitrary implementation-specific arguments after the first two optional arguments, so Guile using the `#:weak` keyword to specify implementation specific arguments is still compliant with the specification. However to improve portability, we recommend Scheme implementers avoid using non-standard syntax for APIs that are a part of SRFI implementations, even for optional procedure arguments.

3.2 Semantics of `include` is under-specified

According to section 4.1.7 of the R7RS Small document [Shinn et al. 2021]:

Both `include` and `include-ci` take one or more filenames expressed as string literals, apply an implementation-specific algorithm to find corresponding files. . .

Implementations are encouraged to search for files in the directory which contains the including file, and to provide a way for users to specify other directories to search.

Though R7RS explicitly encourages searching for included files “in the directory which contains the including file,” not all Scheme implementations follow this suggestion. An alternative approach sometimes used, which is still compliant with the R7RS Small specification, is to search for `include` files using the same mechanism for mapping logical library names to files. This typically means inspecting a “load library path” parameter configured before loading or compiling the program. The problem arises when there are two or more library definition files which have the same filenames, but are in placed in different directories, such as:

```
(set-library-path! '("./a/b" "./a/c"))
(import
  (a b x) ;maps to ./a/b/x.sld,
           ; includes x.scm from ./a/b
  (a c x) ;maps to ./a/c/x.sld,
           ; includes x.scm from ./a/c
```

In the above example, a hypothetical Scheme implementation using the “library path” method to find include files, rather than the recommended “same directory” method, results in a problem where no matter what library path search order is specified, one of the `x.scm` files will always shadow the other.

This problem can be partially mitigated by wrapping every `include` expression inside of a `cond-expand` expression with a path specifically tailored to each Scheme implementation according to it’s method of finding include files. Another mitigating approach is making all Scheme program files unique for the set of all library declarations used in the software. Yet another approach is changing the logical names of libraries from, for example, `(a b x)` to something more like `(a-b-x)`, but this restricts programmers to using the most conservative file naming conventions, rejecting valid library names in order to fit the code base to a particular Scheme implementation.

It has been tempting to write a portable, R7RS-compliant `define-library` system which is more strictly in compliance with R7RS recommendations, that would translate Scheme libraries to whatever module or library system might be in place on the various Scheme implementations to which Schemacs is ported, in order to avoid these kinds of problems. But the solution to this problem should be the responsibility of Scheme maintainers.

The wording in the R7RS report has good reason for giving leeway to implementer on how to find included files, after all, not all Scheme implementations may necessarily run on a computer which has a notion of a directory tree, or even have a filesystem. However, since all POSIX systems do use directory trees to organize filesystems, it could be beneficial for errata to be written into the R7RS Small Scheme report, or to include wording in the not-yet-ratified Large report, which *requires* any Scheme implementations that are specifically designed

to run on POSIX systems to follow the rule of searching for `include` files in the same directory as the including file, since these Scheme implementations are already constrained to use the features of the POSIX standard.

3.3 Mitigating inconsistencies in SRFI support

SRFIs provide a way to compromise between the equally important but mutually exclusive goals of keeping the Scheme language as small as possible, while also addressing the needs of software developers for more features. Some SRFIs define an abstraction for platform-specific computing features, such as hardware parallel threads, or features which extend the garbage collector, such as weak key/value dictionaries. But apart from these examples, many SRFIs can be implemented entirely using the R7RS Small standard alone. In fact, many SRFIs have portable, permissively licensed reference implementations which can be copied into any software project which needs them.

SRFIs are a very pragmatic compromise because they provide useful features to software developers without placing an unnecessary burden on maintainers of Scheme implementations, while Scheme implementation may still choose to implement a SRFI internally in order to improve performance or security over the reference implementation. Furthermore, a Scheme implementation may choose to *support* a SRFI library without necessarily always *shipping* that library in the deliverable software package containing the Scheme implementation, so the footprint of the Scheme software shipped can be tailored to the use case. Also there are third-party Scheme software package repositories such as “Snow Fort” [anonymous 2025b] and “Akku” [anonymous 2025a] which can make it easier to extend the functionality of a production Scheme software installation.

However, in our experience in developing the “Schemacs” software, this compromise does place one additional burden on software developers who are trying to write portable R7RS Scheme code (at this point in history prior to the ratification of the complete R7RS Large standard), which is that you must somehow track the list of SRFI dependencies used throughout your project, ideally to a symbol-by-symbol granularity, so that your code can import each symbol from the correct library based on which Scheme implementation you are using. The following simplified example shows how we must mitigate the problem for the hash tables data type.

```
(define-library (schemacs hash-tables)
  (export make-hash-table
          hash-table-ref
          hash-table-set!)
  (cond-expand
    ((library (srfi 69))
     (import (only (srfi 69)
                   make-hash-table
                   hash-table-ref
                   hash-table-set!)))
    ((library (srfi 125))
```

```

(import (only (srfi 125)
              make-hash-table
              hash-table-ref
              hash-table-set!)))

(other-scheme
 (import (only (other-scheme dictionary)
              new-dictionary
              key-lookup
              key-set!))))

(begin
 (cond-expand
  (other-scheme
   (define hash-table-ref key-lookup)
   (define hash-table-set! key-set!)
   (define (make-hash-table equal hash . ignore)
    (new-dictionary equal))))))

```

This rather verbose block of code is an example of what we must do to smooth-over the differences in the features provided between the various Scheme implementations to which Schemacs has been ported, short of submitting bug reports and patches to Scheme implementations that have not yet implemented the SRFIs we require. With the above library definition, the rest of the code base can import the required hash table APIs from the (`schemacs hash-table`) interface library, and the code in the interface library takes care of interfacing those symbols to the underlying Scheme implementation. We also show in the example a hypothetical “Other Scheme” implementation in which we might use a built-in data structure similar enough to hash tables that we can implement the hash table APIs we required.

This issue will hopefully someday soon become a non-issue as more Scheme implementations gradually make improvements toward R7RS compliance. But until the R7RS software ecosystem matures, anyone trying to write strictly R7RS compliant software will have to maintain code such as in the above example.

We strongly recommend Scheme implementation maintainers place the highest priority on providing support for more SRFI implementations, especially for data structures such as hash tables which are so commonly used in software engineering, so much so that many other popular programming languages provide them as a built-in primitive data structure. We also recommend that Scheme implementations provide feature to help warn programmers when non-standard features of the Scheme language are used.

3.4 Non-standard pattern matching

As noted in SRFI-262 [Preston-Kendal 2025], a lack of pattern matching in Scheme has been a problem since roughly 1987 [Wadler 1987]. However as noted in SRFI-204 [Thibault 2022], the “Wright-Cartwright Shinn Pattern Matcher,” (WCS) is supported in some form or another across a wide range of Scheme implementations. Unfortunately, since SRFI-204 is withdrawn, most Scheme implementations do not attempt to comply with it. In spite of SRFI-204 being withdrawn,

we have found that using some subset of the WCS pattern matcher specified in SRFI-204 is at least partially portable across various Scheme implementations.

But our approach to pattern matching is much more reliable than restricting the code to some vague subset of WCS and hoping it will be portable: we have noticed that the version of WCS written by Alex Shinn, which is provided in the Chibi Scheme implementation, is itself very easily portable, and works consistently across many Scheme implementations. As such, the Chibi Scheme pattern matcher is currently included in the Schemacs source code base and is used to implement the Emacs Lisp interpreter.

Also available to us is a version of SRFI-241 [Nieper-Wißkirchen 2023] is a finalized SRFI which provides a slightly different but equally useful pattern matcher. This pattern matcher was evaluated for fitness of purpose in Schemacs, but the syntax of patterns specified in SRFI-241 is somewhat different from WCS (using unquote syntax for variable binding), and so was ultimately not used.

At the time of this writing, SRFI-257 [Egorov 2024] and SRFI-262 [Preston-Kendal 2025] are draft SRFIs which may be able to replace SRFI-204. It is possible in the future that Schemacs may also replace the Chibi WCS pattern matcher that is currently being used with one of these new standardized pattern matchers as the syntax appears not to be too different from WCS, but whether we do decide to rewrite the Emacs Lisp interpreter to use SRFI-262 will depend on the engineering trade-offs, namely whether it will make Schemacs more portable than it is right now, and how much work would be required to rewrite the interpreter to use a slightly different pattern matcher.

4 SOFTWARE ARCHITECTURE AND APPLIED THEORY

In this section we would like to give a brief overview of the software architecture of Schemacs as it has been structured thus far, especially with regard to portability. Although R7RS Small Scheme is a relatively small language, it has served our work on Schemacs fairly well as a solid foundation for implementing features of our software. We will provide examples in this section of how we implement a monadic parser [Liang et al. 1995] and “lenses” [O’Connor 2011] using only features provided by the R7RS Small language, which makes our code very portable. We also demonstrate how we can create abstractions around platform-specific APIs for tasks such as GUI programming by using the parameter objects feature specified in R7RS Small.

Unless otherwise noted, the code written so far currently works on several major R7RS-compliant Scheme implementations, including (in no particular order) Guile, Gambit, Stklos, MIT Scheme, Gauche, Chibi, and also Chez Scheme by way of Gwen Weinholt’s R7RS-to-R6RS translator [Weinholt 2018] from the “Akku” [anonymous 2025a] package manager.

4.1 Test Suites

Writing tests into the test-suite for each new feature is important for any software project, but especially for software which may run on multiple platforms. Tests check whether the assumptions of the programmers hold true on each new platform. Also the test suite can serve as a way to gauge how much work will be necessary to complete a port the software whenever the code is run on a yet-untried Scheme implementation.

GNU Emacs provides its own test suite for testing itself, which is programmed using a system known as “Emacs Regression Testing” (ERT) [editors 2025], which is itself implemented in Emacs Lisp. The first phase of the Schemacs project has focused on implementing enough of Emacs Lisp interpreter to be able “bootstrap” ERT, i.e. to be able to load and run the ERT source code such that the GNU Emacs test suite included in the Emacs source code base can be run. At the time of this writing, ERT is not yet fully working in Schemacs.

For testing the core Emacs Lisp implementation that will be used to bootstrap ERT, SRFI-64 [Bothner 2006] provides a standardized API for test suites, although unfortunately a few major R7RS-compliant Scheme implementations do not implement SRFI-64. More portable alternative Scheme unit testing frameworks are being evaluated at this time.

4.2 Monadic parsing and pretty printing

Schemacs currently defines an Emacs Lisp AST, where each node of the AST is a record type. We implemented our own monadic lexing and parsing library for the Schemacs project. Lexing is performed on input ports constructed by `open-input-file` or `open-input-string`. The tokenizer is designed as a kind of state table, using the R7RS vector data type to map characters to monadic procedures, where each monadic procedure then greedily consumes the characters from the input port to produce a token.

Each token has an integer value, and a parser which again uses vectors to map tokens values to monadic procedures consumes the tokens to constructs the nodes of the AST. The tokenizer keeps track of source locations and each AST node has one or two slots for a starting and (optionally) an ending location.

Because Scheme is an untyped language the concept of “monads” is more of a design pattern, in contrast with typed languages such as Haskell where the type system rigidly enforces how monads are used [Liang et al. 1995]. The design pattern can be demonstrated with a simplified example in the following code:

```
(define-record-type <lexer-monad-type>
  (make<lexer-monad> lex)
  lexer-monad-type?
  (lex lex-monad-proc)
  ;; Must be a procedure constructed
  ;; by one of the combinators in this
  ;; library.
```

```
)

(define-record-type <lexer-state-type>
  (make<lexer-state>
    line column buffer ahead1 path port cont)
  lex-st-type?
  ; .... slot definitions ....
)

(define (consume-char lex-st)
  ; Updates the character and line counters, collect
  ; characters in output string port if necessary.
  #t
)

(define (char predicate)
  (make<lexer-monad>
    (lambda (lex-st)
      (if (predicate (ahead1 lex-st))
          (consume-char (ahead1 lex-st))
          #f))))

(define (run-lexer in-port . lex-monad-procs)
  (let ((lex-st
        (make<lexer-state>
          0 0 #f #f #f in-port #f)))
    (let loop ((procs lex-monad-procs))
      (cond
        ((null? procs) #t)
        (((lex-monad-proc (car procs)) lex-st)
         (loop (cdr monad-procs)))
        (else #f)))))
```

A lexer monad must be constructed using the combinators in the library. All monadic procedures are wrapped in the record type `<lexer-monad-type>` so that the monad evaluator `run-lexer` can raise exceptions whenever it is given a procedure that has not been constructed by one of the combinators. For example, to parse character using a character predicate, a `char` combinator is defined which can be used to construct a lexer procedure that inspects the current character and returns the result of applying that character to a given predicate. Several other primitive combinators besides `char` are defined which allow a programmer to define a monad that can lex any Emacs Lisp token. The `run-lexer` procedure can then be used to evaluate whichever lexer monads have been constructed. Evaluation returns `#f` if lexing fails, otherwise the result of lexing is returned. Finally, the `<lexer-state-type>` record type is the lexer state, which maintains the input port, a single look-ahead character, the source location, and provides an optional buffer for collecting consumed characters.

An advantage of our approach to monadic parsing is that it becomes easier to define a parser without requiring intermediate data structures to track each token produced by the lexer, rather each token is produced on demand by the parser.

Another advantage is that monads can be implemented entirely using record types, procedures, and input ports, which are features universal to all compliant R7RS Small Scheme platforms. This means when porting Schemacs to another platform, there is no need to concern ourselves with whether any given Scheme implements SRFI-115 “Scheme Regular Expressions,” [Shinn 2014], we can be sure that at the very least lexing and parsing will work no matter what. If in some future versions of Schemacs performance becomes an issue, it may be possible to provide an alternative parser which does use SRFI-115 on Scheme implementations which provide more efficient regular expressions.

It should perhaps be noted that the original lexer and parser used for the proof-of-concept Emacs Lisp interpreter was provided by the Emacs Lisp implementation that ships with Guile Scheme, which made use of a Guile-specific implementation of regular expressions. Apart from not being portable, this implementation only constructed lists, rather than an AST, and so it was not possible to find the source location of the code which raised an error, which made implementing the Emacs Lisp interpreter prohibitively difficult.

4.2.1 Monadic pretty printing. We have also implemented our own monadic pretty-printer for the Schemacs project, using the same design pattern as with monadic parsing described above, of course using output ports rather than input ports, and providing a different set of primitive combinators such as `indent` and `line-break`. Although SRFI-166 “Monad Formatting” [Shinn 2020] defines a standard API, the Schemacs pretty printer does not yet comply with this API. Time permitting, it should be made to do so eventually, in order to satisfy the project requirements of maximum portability.

4.3 “Lenses” inspired by Haskell

The Emacs user interface responds to chains of keyboard input events through the use of a data structure called a “keymap.” Each keymap looks up a keyboard input event and may produce a procedure that triggers an update to the editor state, or may produce a nested keymap in which the end user is expected to produce another keyboard input event to be looked up in the nested keymap. Keymap data structures can be constructed and updated using lists of key events mapped to command procedures.

To implement Emacs keymaps, it is necessary to have some mechanism which can lookup or update some deeply nested structure from a list of keys, where these keys are some representation of keyboard input events. There are a few approaches we could have chosen. SRFI-17 “Generalized `set!`” [Bothner 2000], for example, provides a macro system for simplifying the syntax of mutating data. Unfortunately, SRFI-17 is not widely supported, and furthermore does not provide an easy way to compose setters for deeply-nested data structures.

The approach we chose was to define our own “lens” framework inspired by the well-known Haskell Lens library, which is based on the work of Russel O’Connor [O’Connor 2011]. The

Haskell Lens library demonstrates a pragmatic way of defining composable lenses that conform elegantly with Haskell’s type system. Scheme being an untyped language greatly simplifies our *Lens* implementation as it need only define a record type that combines a “getter” and “setter” procedure, defines the higher-order procedures used to compose these “lens” structures together, and defines the primitive operations to “view” and “set” a part of a data structure using a lens. Other primitive operations such as traversals or folds are not necessary for Schemacs, though our current lens library design does not preclude the possibility of implementing these features, (that said, we acknowledge that the lack of these primitives arguably means our lenses implementation does not satisfy the definition of the term “lenses”).

The following code demonstrates our approach with a simplified example:

```
(define-record-type <unit-lens-type>
  (make-unit-lens getter setter)
  %unit-lens-type?
  (getter ulens-getter)
  (setter ulens-setter))

(define (lens-view struct . lenses)
  (cond
    ((null? lenses) struct)
    (else
     (apply lens-view
              ((ulens-getter (car lenses)) struct)
              (cdr lenses)))))

(define (lens-set! new struct lens0 . lenses)
  (cond
    ((null? lenses)
     ((ulens-setter lens0) struct new))
    (else
     (apply lens-set!
              new-value
              ((ulens-getter lens0) struct)
              (car lenses)
              (cdr lenses)))))

;;---- Example lenses ----;;

(define =>car (make-unit-lens car setcar!))
(define =>cdr (make-unit-lens cdr setcdr!))

(define =>imm-car ;; immutable car
  (make-unit-lens car
    (lambda (cell new-value)
      (cons new-value (cdr cell)))))

(define (set-cadr! cell value)
  (lens-set! value cell =>cdr =>car))
;; Note: given this simplified example, an
;; immutable ‘set-cadr’ is not possible.
```

One advantage of using lenses is that when defining Scheme libraries, it is easier to export a single symbol for each record field using lenses, rather than exporting two symbols for each record field. For example, rather than exporting two symbols, one for the setter, we need only export a single symbol =>`keymap-keyboard-event` defined as the lens that combines both the getter and setter.

We are also free to implement different kinds of lenses, such as immutable lenses which replace the whole structure rather than mutate it, (although immutable lenses are not possible with the simplified example above, please refer to our source code which is publicly available). It is also possible to define what we call “canonical” lenses which will construct a new keymap node and then update it if the keymap node mapped to a given key does not yet exist.

Our lens implementation could be made more ergonomic with macros, as of right now our lenses must be hand-written for every record field.

4.4 Parameterizing system calls

R7RS Small defines “parameters” as a mechanism for dynamic bindings. We have taken care to define the editor state and editor logic in a way that is not platform-specific, and separating the “low-level” platform-specific system calls into their own libraries. Every record type that forms part of the editor state, such as “buffers,” “windows,” and “frames,” has an additional field for platform specific data, such as a GUI window handle that can be associated with an Emacs “frame.”

Furthermore, procedures which modify the editor state must define Emacs commands which can select the correct low-level procedure implementations that modify the low-level data in the editor state. These Emacs commands may even provide multiple different low-level implementations within the same Scheme platform, for example, one for a GUI view and one for a CLI view of the editor state. Emacs commands such as `split-window-below`, which splits a parent frame into an upper and lower half each presenting a different Emacs “window” view, must be programmable such that low-level procedures are parameterized according to which low-level implementation is currently being used by the end user.

To accomplish this, we define libraries which are reminiscent of “abstract interfaces” in object oriented programming languages such as Java or C#, each exported symbol is a parameter object. The following code demonstrates the technique with a simplified example:

```
(define-record-type <schemacs-frame>
  (make<schemacs-frame>
    title rectangle lowlevel-data)
  schemacs-frame-type?
  (title      get-title      set-title!)
  (rectangle  get-rectangle  set-rectangle!)
  (lowlevel   get-frame-lowlevel set-frame-lowlevel!))
;; 'lowlevel' refers to platform-specific data.

(define make-frame-impl
```

```
;; so-called "abstract interface"
(make-parameter
  (lambda (editor-state)
    (display "'make-frame' not implemented\n")
    #f)))
```

The library defining the low-level implementation imports the symbols of the so-called “abstract interface” and parameterizes them with the various platform-specific procedure implementations that make system calls. Once the dynamic extent of this abstract interface has been parameterized, a procedure is evaluated that responds to input events, any updates to the editor state use the parameterized APIs to update the low-level data structures as necessary. The following example shows how this might work if there were a low-level GUI API called `gtk3-split-window-horizontal`.

```
;;-- in library (schemacs lowlevel linux gtk3) --;;
(define (gtk3-make-frame editor-state)
  ;; Low-level implementation of 'make-frame'.
  (let* ((schemacs-frame
          (get-current-schemacs-frame editor-state))
        ;; Split the window,
        ;; get the new lowlevel data.
        (new-gtk3-pane
         (gtk3-split-window-horizontal <-- sys call
          (get-frame-lowlevel schemacs-frame)))
        ;; Construct a new Schemacs window as
        ;; a child of the current frame and
        ;; containing the new lowlevel data.
        (new-schemacs-frame
         (make<schemacs-window>
          schemacs-frame new-gtk3-pane)))
    ;; Use lenses to store the new frame
    (lens-set!
     (cons new-schemacs-frame
           (lens-view
            schemacs-frame
            =>schemacs-window-list))
     schemacs-frame
     =>schemacs-window-list)
    new-schemacs-frame))

(define (gtk3-init-editor)
  (gtk3-set-window-event-handler
   *gtk3-main-window*
   (lambda (gtk3-input-event)
     (parameterize
      ((make-frame-impl gtk3-make-frame)
       ;; ... many other parameters ... ;;
      )
      (schemacs-handle-input-event
       (gtk3->schemacs gtk3-input-event)))))
  (gtk3-start-event-loop))
```

Currently a Linux Gtk3 GUI low-level implementation is the only one provided in Schemacs. GUI or CLI implementations

for other platforms could be defined using the same technique described here.

Note: In this paper we refer to platform-specific data and procedures as “low-level,” although in the Schemacs source code we call these “back-ends.” Our thinking is that the “back-end” of our application is any code calling directly into the platform-specific system APIs, while the “front-end” of Schemacs consists of the Scheme APIs that command the user-facing Emacs user interface. This may be confusing because our “back-end” (low-level) code is calling into the platform-specific GUI APIs, and GUI-related code is typically referred to as “front-end” code. For the sake of clarity, in this paper we refer to any system call to update the GUI as a “low-level” implementation, rather than a “back-end” so as not to confuse anyone who thinks of the GUI as a “front-end” for the application. Please be aware of this when reading the Schemacs source code.

5 CONCLUSION

As a result of the experience of working on this project, the authors feel strongly that the R7RS Small standard language provides an excellent foundation for defining other programming languages such as R7RS Large or Emacs Lisp, and that when the Large standard is finally ratified it will almost certainly have a feature set that will be very competitive with many more popular languages used in industry nowadays.

We acknowledge that the Scheme language emphasizes minimalism, and that this minimalism grants more freedom to implement a Scheme in whatever way an implementation author sees fit. However for Scheme implementations which have a goal of providing a general-purpose programming platform which can be used to engineer production software applications, what we have been calling “industrial strength” Scheme implementations, we hope these implementations are quick to adopt the new language standard, to emphasize strict compliance with the language standard, and to provide support for more SRFIs sooner. We also recommend avoiding non-standard syntax such as keywords objects in the library interfaces for SRFI implementations. It would also probably be very helpful for industrial strength Scheme implementations to provide features that help programmers write standard-compliant code, such as optionally warning programmers who use non-standard features of the implementation.

As we have worked on the Schemacs project, we have learned through conversation that the Scheme community has very diverse opinions about the strengths and practical uses of the language. Some people seem to think of it as a purely pedagogical or research language. Some seem to think the fragmentation of the Scheme software ecosystem caused by the lack of features and abundance of Scheme implementations, is an endearing feature of the language that should be embraced — that programmers should choose the correct Scheme implementation for their task and not worry about portability. However through our experience in implementing Schemacs we have come to believe that in

spite of the the difficulties arising from the minimalism of the R7RS Small standard, this need *not necessarily* result in fragmentation or relegate the Scheme language to the niche of research or pedagogy. It seems to us to be possible to target our software to the R7RS standard itself and still reasonably expect that it can be ported to several compliant implementations.

But R7RS Scheme is a relatively new language, 11 years old at the time of this writing, and so the software ecosystem does not yet seem to us to be very mature. This could improve if more Scheme programmers made the conscious choice to target R7RS Scheme rather than a specific scheme implementation when writing code. And since R7RS is a super-set of R5RS, it should be easier to port R5RS Scheme programs to R7RS to help mature the software ecosystem sooner.

We also discussed in this paper our use of a portable implementation of the WCS pattern matcher, and also the example of our own portable monadic lexer and parser libraries, and we believe these examples show that by simply writing Scheme libraries more carefully to use only the features provided by R7RS Small along with the necessary SRFIs, these libraries can very easily be ported to run on any R7RS-compliant Scheme platform, and therefore expand the software ecosystem of the R7RS language. Logically, having a larger ecosystem of portable, strictly R7RS-compliant Scheme libraries should then make it easier to write portable Scheme programs.

So in closing, we would like to recommend to the Scheme community at large to more seriously consider that it may indeed be a worthwhile endeavor to write code targeting the R7RS standard rather than a specific Scheme implementation, and we would like the Schemacs project to serve as an example of this endeavor in the hopes that it will help to grow the Scheme software ecosystem, and expand the community of people who use it.

REFERENCES

- Stephen Adams, Chris Hanson, and the MIT Scheme Team. 2021. *MIT/GNU Scheme User's Manual for release 12.1*. MIT Press, Massachusetts Institute of Technology, Cambridge, MA.
- anonymous. 2025a. *Akku.scm*. Scheme Community. Retrieved 2025-07-17 from <https://akkuscm.org/>
- anonymous. 2025b. *Snow Fort*. Scheme Community. Retrieved 2025-07-17 from <https://snow-fort.org/>
- Matthew Birkholz. 1993. *Emacs Lisp in Edwin Scheme*. Master's thesis. Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory (CSAIL). <http://hdl.handle.net/1721.1/6787>
- Per Bothner. 2000. *Generalized set!* Scheme Community. Retrieved 2025-07-17 from <https://srfi.schemers.org/srfi-17/srfi-17.html>
- Per Bothner. 2006. *A Scheme API for test suites*. Scheme Community. Retrieved 2025-07-17 from <https://srfi.schemers.org/srfi-64/srfi-64.html>
- Andrea Corallo, Luca Nassi, and Nicola Manca. 2020. Bringing GNU Emacs to Native Code. <https://doi.org/10.5281/ZENODO.3736363>
- Ludovic Courtès. 2011. GNU Guile 2.0.0 released. <https://lists.gnu.org/archive/html/guile-devel/2011-02/msg00173.html>
- The Guile Developers. 2024. *Guile Reference Manual* (3.0.10 ed.). Free Software Foundation. <https://www.gnu.org/software/guile/manual/guile.pdf>
- GNU Emacs Manual editors. 2025. *ERT: Emacs Lisp Regression Testing*. Free Software Foundation. Retrieved 2025-07-17 from https://www.gnu.org/software/emacs/manual/html_node/ert/index.html

- 1027 Sergei Egorov. 2024. *Simple extendable pattern matcher with back-*
 1028 *tracking*. Scheme Community. Retrieved 2025-07-17 from <https://srfi.schemers.org/srfi-257/srfi-257.html>
 1029 Arthur A. Gleckler. 2018. *Growing Schemes*. Association for Comput-
 1030 *ing Machinery*. [https://www.schemeworkshop.org/2018/Gleckler.](https://www.schemeworkshop.org/2018/Gleckler.pdf)
 1031 *pdf*
 1032 Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad transform-
 1033 *ers and modular interpreters*. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming*
 1034 *Languages* (San Francisco, California, USA) (*POPL '95*). Asso-
 1035 *ciation for Computing Machinery*, New York, NY, USA, 333–343.
 1036 <https://doi.org/10.1145/199448.199528>
 1037 GNU Emacs manual editors. 2025. *GNU Emacs Release History*. Free
 1038 *Software Foundation*. Retrieved 2025-07-17 from [https://www.gnu.](https://www.gnu.org/software/emacs/history.html)
 1039 *org/software/emacs/history.html*
 1040 Matthias Neubauer and Michael Sperber. 2001. Down with Emacs Lisp:
 1041 *dynamic scope analysis*. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming* (Florence,
 1042 *Italy*) (*ICFP '01*). Association for Computing Machinery, New York,
 1043 *NY, USA*, 38–49. <https://doi.org/10.1145/507635.507642>
 1044 Marc Nieper-Wißkirchen. 2023. *Match — Simple Pattern-Matching*
 1045 *Syntax to Express Catamorphisms on Scheme Data*. Scheme Com-
 1046 *munity*. Retrieved 2025-07-17 from [https://srfi.schemers.org/srfi-](https://srfi.schemers.org/srfi-241/srfi-241.html)
 1047 *241/srfi-241.html*
 1048 Russell O'Connor. 2011. Functor is to Lens as Applicative is to Biplate:
 1049 *Introducing Multiplate*. arXiv:1103.2841 [cs.PL]
 1050
 1051 Daphne Preston-Kendal. 2025. *Extensible pattern matcher*. Scheme
 1052 *Community*. Retrieved 2025-07-17 from [https://srfi.schemers.org/srfi-](https://srfi.schemers.org/srfi-262/srfi-262.html)
 1053 *262/srfi-262.html*
 1054 Ken Raeburn. 2009. *Guile-Based Emacs*. Massachusetts Institute
 1055 *of Technology*. Retrieved 2025-07-17 from [https://www.mit.edu/](https://www.mit.edu/~raeburn/guilemacs/)
 1056 *~raeburn/guilemacs/*
 1057 Alex Shinn. 2014. *Scheme Regular Expressions*. Scheme Community.
 1058 *Retrieved 2025-07-17 from https://srfi.schemers.org/srfi-115/srfi-*
 1059 *115.html*
 1060 Alex Shinn. 2020. *Monadic Formatting*. Scheme Community. Retrieved
 1061 *2025-07-17 from https://srfi.schemers.org/srfi-166/srfi-166.html*
 1062 Alex Shinn, John Cowan, and Arthur A. Gleckler. 2021. *Revised7*
 1063 *Report on the Algorithmic Language Scheme*. Scheme Community.
 1064 *https://standards.scheme.org/official/r7rs.pdf*
 1065 Robin Templeton. 2024. *The Guile-Emacs Project*. The Guile-Emacs
 1066 *Project*. Retrieved 2025-07-17 from <https://guile-emacs.org/>
 1067 Felix Thibault. 2022. *Wright-Cartwright-Shinn Pattern Matcher*.
 1068 *Scheme Community*. Retrieved 2025-07-17 from [https://srfi.](https://srfi.schemers.org/srfi-204/srfi-204.html)
 1069 *schemers.org/srfi-204/srfi-204.html*
 1070 Philip Wadler. 1987. A critique of Abelson and Sussman or why
 1071 *calculating is better than scheming*. *SIGPLAN Not.* 22, 3 (3 1987),
 1072 *83–94*. <https://doi.org/10.1145/24697.24706>
 1073 Gwen Weinholt. 2018. *R7RS versus R6RS*. (homepage). Retrieved
 1074 *2025-07-17 from https://weinholt.se/articles/r7rs-vs-r6rs/*
 1075
 1076
 1077
 1078
 1079
 1080
 1081
 1082
 1083