MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. TR-1451 September 1993

Emacs Lisp in Edwin Scheme

by

Matthew Birkholz

This report is a revised version of a thesis submitted to the Department of Electrical Engineering and Computer Science in September, 1993, in partial fulfillment of the requirements for the degree of Master of Science.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-4097 and by the National Science Foundation under grant number MIP-9001651.

Abstract

The MIT-Scheme program development environment includes a general-purpose text editor, Edwin. Edwin provides an integrated platform for a number of tools useful to a software engineer. Such tools are easily written in Edwin's extension language, Edwin Scheme — Scheme augmented with editor data types.

Edwin is very similar in appearance and behavior to another general-purpose text editor, GNU Emacs. Like Edwin, GNU Emacs provides a number of useful tools, written in its extension language, Emacs Lisp. The popularity of GNU Emacs, combined with its easy extensibility, has lead to a large and growing library of tools and enhancements written by GNU Emacs users worldwide. The goal of this thesis is to allow Edwin users to take advantage of this enormous library of Emacs Lisp code.

The size and complexity of the Emacs and Edwin systems makes realization of this goal impossible given the resources available to this project. Instead, a useful compromise was sought. From the beginning, this project took as a concrete goal the emulation of a particularly valuable GNU Emacs tool, the GNUS news reading program (written by Masanobu UMEDA [umerin@mse.kyutech.ac.jp]).

To achieve this goal, an Emacs Lisp interpreter was written in Edwin Scheme. This interpreter implements approximately 70% of the 734 primitives of the Emacs Lisp language. It also integrates the Emacs and Edwin user interfaces and environments to such an extent that the casual user will not notice the differences between the emulated Emacs Lisp windows, buffers, and commands, and those of a normal Edwin Scheme program. The result is that the unmodified source code for the GNUS news reading program can be loaded into Edwin, and the program's commands for marking, reading and filing news articles can be used as though they were normal Edwin commands.

Acknowledgments

I could not have done this alone.

On Jim Miller's recommendation I made Edwin my home, and through his contacts I learned of the MIT Scheme group's interest in a project of this sort. His enthusiasm for the project was crucial in bringing Chris Hanson and me together. As my thesis advisor, Chris deserves more than a little credit for the goal-oriented approach to this project. Without his direction, I am sure I would still be working on it.

In addition to these principle characters, there are many friends, relatives, and even thoughtful acquaintances who have contributed in small yet important ways to the successful completion of this work. I will not try to list them all, but there are a few who deserve special mention. Jinx, BAL, Stephen, Arthur, and Becky are just a few of the MIT Schemers that have made me feel welcome. Before I left DEC's Cambridge Research Lab, I relied on the support of Victor Vyssotsky and Phill Apley, and benefited from the concern and well-wishes of many others. Earlier, at DEC's Artificial Intelligence Laboratory, I was constantly encouraged to "finish your thesis!" by Dave Buffo, Tom Cooper, Eva Hudlicka, and others.

Finally, I could hardly have survived this long, certainly not long enough to have finished this thesis, without the super-human love and patience of my mother and father, and the extraordinary Lori Birkholz.

Contents

1 Introduction

The MIT-Scheme[3] program development environment includes a general-purpose text editor, Edwin. Edwin provides language-specific support for editing Scheme source code, plus convenient interfaces to other tools in the MIT-Scheme environment, e.g. the debugger and the interpreter. In addition to these software development functions, Edwin provides tools for manipulating directories of files, running Unix programs as subprocesses, reading Unix mail messages, and editing source files written in other languages. Consequently, Edwin fills the same role as the general-purpose editor GNU Emacs[9], providing an integrated platform for many of the tools used by a software engineer.

GNU Emacs is a general-purpose text editor in widespread use. It is easily extended via its powerful extension language, *Emacs Lisp*[5]. This ease of extensibility has encouraged the development of a large library of tools and enhancements written by GNU Emacs users worldwide. This library continues to grow as users experiment with new tools for a variety of tasks. The goal of this thesis is to show that Edwin can be extended so that it can take advantage of this enormous library of Emacs Lisp code.

While Edwin is very similar in appearance and behavior to GNU Emacs, it is nonetheless very different internally. Edwin's extension language is *Edwin Scheme* — Scheme[2] augmented with editor data types. Thus, Edwin Scheme inherits Scheme's lexical scoping of variables, whereas Emacs Lisp uses dynamic scoping. Edwin Scheme's internal data structures also differ from those of Emacs Lisp in many details. As a result, there is no straightforward translation from Emacs Lisp code to Edwin Scheme code.

A few valuable GNU Emacs tools have been translated by hand into Edwin Scheme code, and the non-trivial translation has offered opportunities to improve on the behavior of the Emacs tools. However, the significant translation effort discourages experimenting with interesting new Emacs tools. Also, the translated source code is radically different from the original Emacs Lisp making it difficult to track later improvements to the Emacs tools and make corresponding improvements to the Edwin tools. To avoid both problems, Emacs Lisp programs need to be loaded and evaluated directly, without modifications to the Emacs Lisp source code.

To solve this problem, an Emacs Lisp interpreter was written in Edwin Scheme. This interpreter does its best to faithfully implement the behavior of the original Emacs Lisp interpreter and runtime system as provided by GNU Emacs version 18.59, while also integrating this runtime system with the existing Edwin runtime system. Both of these systems are fairly large and complex. It is beyond the scope of this thesis to describe the many data structures and routines used in both Emacs Lisp and Edwin Scheme. Instead, it is assumed that the reader is well acquainted with both systems, or at least is familiar with Lisp environments and equipped with the Emacs User Manual[9], the Emacs Lisp Reference Manual[5], and the MIT-Scheme Reference Manual[4].

The rest of this introduction will orient the well-equipped reader in the direction used to approach the problem. It describes the motivation for choosing to implement an Emacs Lisp interpreter and to emulate its runtime environment. It also describes the advantages sought when attempting to integrate the emulated environment's state with Edwin's, and notes the issues that arise when melding the two.

The problem of using *unmodified* Emacs Lisp programs could be solved by automatic translation into Edwin Scheme by an Emacs Lisp compiler. The compiler would take over the job currently done by hand. However, the compiler would have to be rather sophisticated in order to produce the non-trivial translation. It would have to do extensive global analysis in order to determine the actual scope of the indeterminately scoped Emacs Lisp variables. This is necessary if they are to be implemented by Edwin Scheme's efficient local variables. The compiler would also have to do extensive type inference when operations on Emacs Lisp data types have to be translated into operations on dissimilar Edwin Scheme data types. These analyses would be particularly difficult because of the widespread use in Emacs Lisp programs of data structures storing arbitrary functions. Without extensive analysis, the compiler will have to produce code that does little more than mimic the original Emacs Lisp interpreter. Finally, Emacs Lisp includes an `eval` primitive that gives programmatic access to the Emacs Lisp interpreter. It is used frequently, even in some of the more desirable Emacs Lisp programs. All of these things argue strongly for the implementation of an interpreter.

The Emacs Lisp interpreter and runtime system could be emulated in the MIT-Scheme environment by a program entirely separate from Edwin. The emulator could manage its own windows and buffers, use its own variables and keymaps, and dispatch on its own command key input to execute Emacs Lisp commands. The program could emulate Emacs with complete accuracy, but it would offer few advantages over running Emacs itself as a separate process. The program would run Emacs Lisp programs in MIT-Scheme environment, making them more accessible for integration with Edwin, but the separate emulator alone would not actually provide any integration. For example, there would still be both Emacs Lisp and Edwin Scheme variables named `fill-column`, each one having the same meaning to the user. Text would still have to be explicitly extracted from Emacs buffers and inserted into Edwin buffers before Edwin Scheme programs could manipulate it.

Emulating Emacs Lisp in the MIT-Scheme environment will be *most* advantageous when the Emacs Lisp emulation can reflect its state in Edwin's state and vice versa. This will integrate multiple representations of what are conceptually the same things. When Emacs Lisp code sets its `fill-column` variable, the information will be reflected in the value of the Edwin Scheme variable; and, when Edwin Scheme code sets its `fill-column` variable, the new value will be reflected in the Emacs Lisp variable. Whether Edwin's state and the emulator's state are automatically kept consistent or actually shared, the result is the same: the user does not have to remember to set both versions of the default right margin.

Integration of the two runtime systems will also allow either kind of program to cooperate with the other. When an Emacs Lisp program like GNUS displays a buffer containing the text of a Usenet article, the Emacs buffer will be reflected by an Edwin buffer with the same content. Normal Edwin commands can then manipulate the article text. The GNUS program could even be extended with Edwin commands written using the powerful facilities of the MIT-Scheme environment.

Integration of the user interfaces will have especially compelling advantages. The user will only have to interact with one interface and will not have to distinguish and switch between two, slightly different interfaces. To do this, the Emacs Lisp emulator will have to reflect Emacs windows as Edwin windows, and Emacs key bindings as Edwin key bindings. If the command key dispatching mechanisms of Emacs and Edwin are significantly different, the necessary reflections may be difficult to implement.

Accurately reflecting each system's state in the other's state can present a number of problems. Conceptually identical information may be represented in very different ways. An atomic value could simply be translated from one format to another depending on how it is accessed. An aggregate value, however, would have to be translated into an analogous data structure in the other runtime system, and the two copies would have to be kept consistent. Consistency is particularly difficult to implement and expensive to maintain when side-effects to the data structures are possible.

Fortunately, a completely accurate reflection is not necessary. Side-effects to some Emacs Lisp runtime data structures may not be found in the majority of interesting programs, while others might simply be ignored. Some data structures are not typically modified by Emacs Lisp programs because they are reserved to the user. Others are modified only to form new values which are then re-installed in the runtime system. The reflection in the other runtime system could simply be a new copy translated from the value that was re-installed. Side-effects to these data structures would not have to be detected in order to make many programs work. While side-effects to other types of runtime system data structures might be important enough to warrant their detection and reflection, so far this has not been necessary.

Some operations of Emacs Lisp can make accurate reflection more difficult than if they were not allowed. If these operations are rarely used, the emulator may be able to restrict such usage so as to simplify the translation or sharing of state information. A good example is the use of vector operations to access and modify bindings in some Emacs Lisp keymaps. This is possible because some Emacs Lisp keymaps are represented by vectors. However, most Emacs Lisp programs only use the more abstract operations for accessing and modifying keymap bindings (e.g. `lookup-key` and `define-key`). Restricting the emulation of Emacs Lisp keymaps so that only the abstract operations are supported allows Emacs Lisp keymaps to be represented by Edwin command tables. By sharing this representation, Emacs Lisp's command dispatch mechanism is much more easily reflected in Edwin Scheme's.

While restrictions on Emacs Lisp programs can ease the difficulty of accurate emulation, so can generalizations of Edwin Scheme. Edwin already allows arbitrary procedures to be specified to perform certain tasks. The interface to Edwin's minibuffer completion commands can be so specified, allowing the emulator to provide special procedures that compute completions from Emacs Lisp data structures. In contrast, the Emacs minibuffer completion commands only work with a couple specific data structures and have very few escapes into procedural code.

However, there are still areas where Edwin could be further generalized. Edwin could even be changed to directly support Emacs Lisp data structures. For example, Edwin's command dispatch mechanism could be generalized to recognize and handle Emacs Lisp keymaps.

This project did not pursue most of these opportunities, preferring to limit modifications to Edwin and to avoid adding Emacs Lisp specific code to it. This has made some parts of the emulator more difficult to implement or imposed restrictions on its accuracy or completeness. Future work may abandon the complete separation of the Emacs Lisp emulator from Edwin. The advantages of doing this are considered in later sections.

It is worthwhile to note that complete emulation and integration of the standard Emacs Lisp runtime system is difficult simply because of the system's size. The standard Emacs Lisp runtime system is based on 587 primitive functions manipulating 147 global variables and 11 data types, all implemented in C. It also includes the functions and variables declared in 22 essential Emacs Lisp files pre-loaded into every Emacs, 50 packages and modes autoloaded by the standard configuration, and some 70 additional files.

Complete integration would unify all the redundant functionality, such as the Emacs Lisp and Edwin Scheme interfaces to sendmail, but integrating all of the functionality provided by Emacs with all of the functionality provided by Edwin would take more time than this project allows. Less than complete integration would still accrue benefits where it is pursued, as described above. Where it is not, there may continue to be redundancies such as the separate `fill-column` variables. There may also continue to be gaps in cooperation such as Edwin Scheme commands that do not have their intended effect because Emacs Lisp data was not available in an analogous Edwin Scheme form. However, these are just inconveniences. It is the accuracy of the emulation that allows Emacs Lisp programs to run.

A complete emulation would only have to implement the Emacs Lisp primitives. Unfortunately, some of these primitives require functionality not supported by Edwin. Implementing them would have been very time-consuming and would have enabled few additional programs to run. This project focuses instead on emulating the essential primitives that are required for the operation of the majority of Emacs Lisp programs.

The essential Emacs Lisp primitives and the essential integration of Emacs and Edwin were hard to define at the beginning of this project. The large number of tradeoffs between accurate emulation, tight integration, and cost of implementation would have also made it hard to determine whether the resulting emulation was a success. Instead, a concrete goal was chosen that would demonstrate a useful emulation of Emacs Lisp and a usable integration with Edwin's user interface. That goal was the rudimentary operation of a sophisticated and valuable Emacs Lisp program, the GNUS news reader.

1.1 Related Work

The techniques used in this thesis project were largely inspired by generic interpreter implementation techniques. The Edwin Scheme code for the Emacs Lisp interpreter is a straightforward interpreter implementation similar to the meta-circular evaluator described in [1] and closely follows the implementation in C that is distributed with GNU Emacs. The emulation of the Emacs Lisp data types and functions had to conform to the detailed behavior of GNU Emacs's implementation, and were written to take advantage of the most abstract functionality of Edwin Scheme. Thus, the source code and documentation of these two implementations have had the largest impact on the nature of this work.

It is a peculiarity of this project that programs written in two different languages to manipulate two different runtime environments must nevertheless operate in one shared environment. A similar situation is often addressed by compatibility libraries. For example, C applications written for BSD Unix can be run in a DOS environment provided all the required functionality is implemented by an available BSD-compatible C library.

Compatibility libraries may address the same goal as this project — emulating a different runtime environment within the native one. However, they communicate information to their client programs through very simple data structures and without sharing. This sharing is a ubiquitous feature of both Emacs Lisp and Edwin Scheme. Procedures written in both languages interface via *call by sharing* where the information passed between procedures, e.g. from a runtime procedure exposing information about the runtime environment to an application program, is identified by its implementation in memory, and where modifications to this memory can communicate information back to the runtime environment. The issues raised by data structures that are shared between the runtime environment and an application program are not typically addressed by compatibility libraries.

Related work on heterogeneous systems also avoids these issues. Most of the work in heterogeneous systems involves the implementation of distributed services made up of cooperating applications residing on different machines with only a network for communication. Applications pass information via a Remote Procedure Call abstraction that only provides *call by copying* behavior.[7] Thus, the issues of shared data structures never arise.

In mixed-language programming systems, the different modules of an application can be written in different languages and can use shared memory to share data structures. In some of these systems, information is still passed between modules using call by copying[8][6]. Other systems agree in advance on the implementation of any data structures that will be shared between languages (e.g. the VAX calling standard). Finally, many modern Lisp systems provide a bridge between their native data types and those of other languages via foreign function interfaces. These interfaces either use call by copying semantics for data structures that are transparently converted between native data types, or they provide operations for declaring and manipulating *alien* data structures as new types. This project must support shared data that can be manipulated as though native data structures in two languages.

1.2 Summary

The motivation for this work was the desire to run Emacs Lisp programs as if they were Edwin Scheme programs, without having to translate them into Edwin Scheme by hand. The approach that was taken was to emulate the Emacs Lisp interpreter and runtime environment and to integrate the emulated environment with Edwin's. In particular, the emulated Emacs user interface (including its windows and command key dispatch mechanism) were to be integrated with Edwin's, so that Edwin users could continue to use one, familiar interface. Complete and accurate emulation and integration was impractical for a variety of reasons. However, there seemed to be acceptable compromises in both accuracy and completeness that would allow a minimally useful system be built with the available resources. As a concrete goal, the minimally useful system was expected to be able to execute the GNUS news reading program well enough that articles could be retrieved and read using the normal Edwin user interface.

Many compromises were made in the accuracy of the Emacs Lisp emulation, and many aspects of Emacs and Edwin have not been integrated. The following sections document the implementation of the Emacs Lisp emulator and these deficiencies, in anticipation of further work. Section 2 describes the implementation of the Emacs Lisp interpreter. In particular, the implementation of Emacs

Lisp's symbols and subrs is discussed, showing how they support the reflection of the emulated Emacs Lisp runtime environment in the Edwin Scheme environment. Section 3 tackles the details of implementing that reflection. First, some of the guiding principles and useful techniques are presented. Then, the emulation of specific Emacs Lisp data structures and functionality is described. Section 4 discusses the specific compatibility issues raised by the less than complete and accurate Emacs Lisp emulation. The conclusion summarizes the qualified success of the project and the work left to be done.

2 The Interpreter

An Emacs Lisp program starts out as a text file describing a sequence of Emacs Lisp expressions, which the `load` primitive can read and evaluate. The evaluation of these expressions typically installs the commands, hooks, and command key bindings of the Emacs Lisp program in the runtime system. The program can then be invoked by the user directly via command key input, or indirectly through commands that run its hooks.

This section describes the process of reading an Emacs Lisp program – converting the text describing Emacs Lisp expressions into data structures that can be evaluated by the interpreter. It also describes the operation of the interpreter proper – how the `eval` primitive interprets expressions as instructions for installing and later executing the Emacs Lisp program. In this discussion, it is assumed that the reader is familiar with the general organization and operation of Lisp interpreters and runtime environments. This section will focus on the peculiarities of reading and evaluating Emacs Lisp programs. In so doing, it will describe in detail the representation and interpretation of two important Emacs Lisp data types. These data types provide the hooks that lead the interpreter into the Edwin Scheme code that will emulate the Emacs Lisp runtime system and reflect its state in Edwin.

The first stage of interpreting Emacs Lisp code is to read it, to produce a representation of the Emacs Lisp data structures denoted by the input text. The syntax of Emacs Lisp code is similar to that of Edwin Scheme, but there are differences in the syntax of character, string, and vector literals that required that a new reader be written. The new reader could have produced instances of special Emacs Lisp types distinct from the Scheme data types, but few of the Emacs Lisp types differed from an analogous Scheme data type. The implementation actually started out making this distinction, using distinct operators to manipulate the Emacs Lisp data types. There was no performance penalty because the operators were substituted for equivalent Scheme operators at compile-time. However, the constant conversions, particularly from literal Scheme constants to their Emacs Lisp equivalents, caused the clarity of the resulting code to suffer. Eventually, the reader was revised to produce native Scheme data types, and the rest of the source code was changed to manipulate them using the normal Scheme operators. The reader now produces Scheme integers to represent Emacs Lisp numbers and characters. Scheme strings and vectors represent Emacs Lisp strings and vectors.

A Scheme symbol is *not* used to represent an Emacs Lisp symbol because the two objects are so different. When interned into the global Scheme obarray, a Scheme symbol is unique to a case-insensitive name, but has no inherent state other than its name. In contrast, an Emacs Lisp symbol is interned into one of many, possibly user-defined obarrays. (In Emacs, obarrays are vectors whose elements are initially zero. The emulator uses the same representation so that

Emacs Lisp code that creates such a vector and uses it as an obarray will be emulated correctly.) An interned Emacs Lisp symbol is unique to a case-*sensitive* name *and* an obarray, and it has additional inherent state: a value, a function, a property list. A Scheme structure [1] is used to represent an Emacs Lisp symbol and hold this state, just as a C struct is used in Emacs. This maximizes the performance of the interpreter, which must access a symbol's function and current value quickly. The Scheme structure is also useful to hold information about how the Emacs Lisp symbol is being reflected into Edwin. (This is described in more detail in Section 2.2.)

Like Emacs Lisp strings and vectors, Emacs Lisp lists are represented by native Scheme lists, implying that Emacs Lisp conses are represented by Scheme pairs and Emacs Lisp's empty list is represented by Scheme's empty list. This takes some care because Emacs Lisp represents its empty list with a symbol named `nil`. Like any other symbol, this symbol can have a function and property list, and so ought to be represented by a symbol object. In fact, the emulator does create a symbol object to emulate the Emacs Lisp symbol `nil`. It never appears at the ends of lists or anywhere else in Emacs Lisp data structures, but it is substituted for Scheme's empty list whenever a symbol operation is being applied. The fields of this symbol structure hold the name, function, property list, and any other information about `nil`.

These are all of the Emacs Lisp data structures that can be denoted by input text. Instances of the other Emacs Lisp data types, such as windows, buffers, and markers, are parts of the Emacs Lisp runtime system and are represented according to how they are reflected in Edwin's runtime system. This is discussed in Section 3.

Once Emacs Lisp code has been read, it can be evaluated by the `eval` primitive. As in the meta-circular evaluator of [1], this primitive, along with the `funcall` primitive, forms the core of the interpreter. The `eval` primitive is applied to an expression to start the process of computing the expression's value. The nature of this process depends on the type of the expression. If the expression is a simple symbol, the value of the symbol is returned. The value is computed by the `%symbol-value` procedure, which is described in Section 2.2. Any other type of object is either a list representing a function call (of which special forms are a special case) or a self-evaluating object.

Normal function call expressions are evaluated by applying a function, described by the first element of the expression, to the values of the argument expressions (the rest of the elements of the expression). However, Emacs Lisp, like all Lisps, uses the same list syntax to represent special forms and macro calls.

[1] A *Scheme structure* is an object that contains a number of named fields. The type of a Scheme structure is declared by a `define-structure` expression whose syntax and semantics are similar to Common Lisp `defstruct` expressions. `define-structure` is an MIT-Scheme extension to standard Scheme and is described in [4].

These other kinds of expressions must be handled differently; so, when applied to a function call expression, a Lisp evaluator must first determine, based on the first element of the expression, whether it is a normal function call, a macro call, or a special form.

In Emacs Lisp, the first element of a function call expression can be a symbol. The symbol can be thought of as the name of a function, and the function it names is the value of the function field of the symbol structure (mentioned above, and described in more detail in Section 2.2). This field can be empty, causing the `eval` primitive to signal `void-function`. The field can also refer to another symbol. This is *function aliasing*. The function named by the first symbol is defined to be the same function named by the other symbol. An arbitrarily long chain of function aliasing is possible, with the function field of each symbol pointing to the next symbol in the chain. To find the ultimate function named by a symbol or chain of symbols, the `eval` primitive uses the `%function*` procedure. This procedure is actually applied to the first element of a function call expression regardless of its type. If the first element was a symbol, any function aliasing is resolved and the named function is returned. If the function aliasing is circular, `%function*` will end up in an infinite loop trying to find the end of the chain. A circular chain could be detected and an error raised without much added expense, but the implementation was left as it is since this is exactly how the original Emacs Lisp interpreter works.

Once the `%function*` procedure has found the function described by the first element of a function call expression, the `eval` primitive can determine how to evaluate the expression. The function can be any of several things: a primitive subroutine (a *subr*), a lambda expression, a macro expression, or an autoload description. If it is not one of these things, `eval` signals an `invalid-function` error. Otherwise, `eval` proceeds as follows.

If the function is a subr and the subr is marked as a special form, then the evaluator applies the subr to the argument expressions. The subr can then evaluate the appropriate expressions, depending on the required behavior of the special form. For example, the first argument expression to the `if` subr is evaluated. If the value of the expression is not `nil`, the second expression is evaluated and its value is returned by the `if` subr. If the value of the first expression was `nil`, the `progn` subr is applied to the expressions following the second expression (if any). The value returned by the `progn` subr (or `nil`) is returned by the `if` subr.

If the function is a subr that is *not* marked as a special form, the argument expressions are evaluated and the subr is applied to the results. When applying either type of subr, the Scheme predicate `procedure-arity-valid?` is used to detect subrs applied to the wrong number of arguments. This allows the emulator to signal the appropriate Emacs Lisp error immediately, rather than wait to catch the Scheme `wrong-number-of-arguments` condition and then signal the Emacs

Lisp error.

If the function is a lambda expression (a list starting with a particular Emacs Lisp symbol named `lambda`), the argument expressions are each evaluated, producing a list of argument values. The `funcall-lambda` procedure takes the lambda expression and the argument values and executes the function. The `funcall-lambda` procedure first examines the lambda expression's parameter list (its second element) for any `&optional` or `&rest` keywords indicating optional or rest parameters. During this examination, it rewrites the list of argument values to include a value of `nil` for any optional parameters that would otherwise not have a value, and to collect multiple arguments into a list as the value of a rest parameter. The parameter list is also rewritten to produce a new list of the parameter symbols (without `&optional` or `&rest` keywords). If the rewrites are not successful, an Emacs Lisp `wrong-number-of-arguments` error is signaled. Otherwise, the `%specbind` procedure is applied to the parameters, the arguments, and the thunk during which the bindings should be active. The thunk applies the `progn` primitive to the body of the lambda expression (the elements following the second element).

The `%specbind` procedure is a straightforward implementation of Emacs's shallow binding mechanism. When a symbol is bound to a value, `%specbind` first saves the symbol's old value (if any) and then sets the symbol to the new value. This is accomplished using the normal symbol operations `%symbol-bound?`, `%symbol-value`, and `%set-symbol-value!`. As control passes out of the dynamic extent of the call to `%specbind`, the symbol's original state is restored, again using the normal symbol operations `%set-symbol-unbound!` or `%set-symbol-value!`. `%specbind` was originally implemented by a call to Scheme's `dynamic-wind` procedure, which ensured that the code restoring the symbol to its original state would be run before control passed out of the dynamic extent of the call to `%specbind`, whether because of a normal return or a non-local exit.

Unfortunately, Scheme's `dynamic-wind` mechanism was too expensive to use every time a symbol was lambda-bound. Instead, `%specbind` just pushes a record onto the list assigned to the Scheme variable `*specpdl*`. The record includes the symbols that were bound and the states they had before they were bound. If control exits through `%specbind` via a normal return, `%specbind` pops the record off `*specpdl*` and uses its information to restore the parameter symbols to their previous states. Handling non-local exits, however, requires the cooperation of all Emacs Lisp primitives to which a non-local exit could return. These primitives must remember the value `*specpdl*` had when control first passed through them, and restore that value if control returns to them via a non-local exit. These primitives must restore `*specpdl*`'s value by popping records and using their information to restore symbols to the states they had before they were bound. When enough records have been popped, `*specpdl*` will regain its former value and all symbols that had been subsequently bound will be restored to their original states.

If the function is a macro expression (a list whose first element is a particular Emacs Lisp symbol named `macro`), then the rest of the list should be a lambda expression. The lambda expression is applied to the unevaluated argument expressions using `funcall-lambda`. The value returned by `funcall-lambda` should be another expression that `eval` can evaluate. The value of this other expression is finally returned as the value of the original function call expression.

If the function is an autoload description, the `do-autoload` procedure is applied to it. This procedure loads the Emacs Lisp file named in the description, and then checks that the function of the function call expression is no longer an autoload description. If it *is* still an autoload description, an Emacs Lisp error is signaled. All of this has to be done inside a thunk passed to `protect-with-autoload-queue`, which ensures that some of the modifications made during the thunk will be undone if an error is signaled. This is one of the defined behaviors of autoloading or requiring (loading a file as a result of calling the `require` subr) in Emacs. The effects to be undone are collected as a list, the value of the Emacs Lisp `autoload-queue` variable. When no autoload is in progress, the value of this variable is `nil` and no effects are recorded. During autoloading, the value of the variable is set to a non-`nil` value (initially, the symbol `t`). Any Emacs Lisp primitives that change the function values of Emacs Lisp symbols, or add features to the feature list, will notice this and record the original function value or feature list. `protect-with-autoload-queue` uses Scheme's `dynamic-wind` procedure to ensure that the recorded effects are undone if an error is signaled. Once the specified file is successfully loaded and the function of the function call expression is no longer an autoload description, the `eval` primitive can again try to evaluate the expression.

2.1 Subrs

In Emacs Lisp, subrs are the primitive funcallable objects. In the emulator, subrs are application hooks [2] and thus Scheme funcallable objects. The application hook data for each subr is a Scheme structure whose fields are assigned to most of the information associated with Emacs Lisp subrs: a name, a documentation string, an interactive specification, and a flag indicating whether the subr is a special form. The arity of the Emacs Lisp subr is represented by the arity of the apply hook procedure.

As described above, the interpreter takes care to pass the unevaluated argument expressions from the function call of a subr which is a special form. This allows the subr's procedure to interpret the syntax of the special form as required and to call the interpreter to evaluate the appropriate subforms in the appropriate order.

[2] An *application hook* is an object that can contain arbitrary user data and also be applied like a procedure. Application hooks are an MIT-Scheme extension to standard Scheme and are described in [4].

A few subrs have to implement non-local control flow, which they do using MIT-Scheme's condition system and `call-with-current-continuation`. The `catch` subr captures its continuation and establishes a handler for a special type of Scheme condition only signaled by the `throw` subr. This type of condition has fields assigned to the throw tag and value. The handler is active while the body of the `catch` special form is evaluated. If the handler is invoked and the signaled condition's tag matches the one to be caught, then the thrown value is passed to the captured continuation, which returns that value from the `catch` subr call. Otherwise, the value of the `catch` special form's body is returned normally. The `throw` subr just needs to make the special type of Scheme condition using its tag and value arguments, and signal it. If the condition is not caught, the default handler for the condition tries signaling an Emacs Lisp `no-catch` error, just as would happen in Emacs. If the error condition is not caught, its default condition handler will call Edwin's `editor-error` procedure with message strings like the ones displayed by Emacs.

Similar to `catch`, the `condition-case` subr captures its continuation and establishes a handler for a second special type of Scheme condition representing an Emacs Lisp error. This type of condition has fields assigned to the Emacs Lisp error's name and associated data. The handler is active while the body of the `condition-case` special form is evaluated. If the handler is invoked and the signaled condition matches one of the special form's handlers, then that handler's body is evaluated and its value is passed to the captured continuation, which returns that value from the `condition-case` subr call. Otherwise, the value of the `condition-case` special form's body is returned normally. The `signal` subr just needs to make the special type of Scheme condition using the error name and data, and signal it. If the error condition is not caught, its default condition handler will call Edwin's `editor-error` procedure with message strings like the ones displayed by Emacs.

The bulk of the remaining implementation details have less to do with the behavior of the interpreter and more to do with the behavior of specific editor data structures. These details are discussed in the context of reflecting the Emacs Lisp runtime system into the Edwin Scheme runtime system. There are a few general observations about the implementation that can be made here and that should be useful to keep in mind when reading or modifying the source code.

The `DEFUN` syntax turns an Emacs-style subr declaration into a subr object bound as the function of a symbol object. Both the symbol and subr objects are assigned to Scheme variables according to a naming convention. The subr object is assigned to a Scheme variable with the same name except that `el:` is prepended. The prefix avoids shadowing useful Edwin variables with the same names as some of the Emacs primitives. The symbol object is assigned to a Scheme variable with the same name except that `Q` is prepended. This convention is reminiscent of a naming convention followed in Emacs's C code. A number of other symbol objects are used as constants by the emulation code, and are also assigned to variables according to the same convention. An example is the constant symbol named `t`, which is bound to the Scheme variable `Qt`.

The arguments to subrs are typically the values of Emacs Lisp expressions, and must be type-checked and coerced to Scheme equivalents. Again, following a convention in the Emacs C code, a number of procedures are provided named, e.g. `CHECK-NUMBER`, `CHECK-CHAR`, `CHECK-MARKER-COERCE-INT`, etc. These procedures implement the same type-checking done by Emacs's C code and signal the same Emacs Lisp errors when necessary. They also help ensure that the correct conversions are performed. For example, `CHECK-CHAR` checks that its argument is an Emacs character (an integer) and converts it to the corresponding Scheme character object. `CHECK-POSITION-COERCE-MARKER` accepts a position expressed as an Emacs buffer position (using 1-based indexing) or a marker (an Edwin mark) and converts the given position to a Scheme buffer position (0-based index), signaling the appropriate Emacs Lisp error if a marker was given that points nowhere.

2.2 Symbols

As mentioned at the beginning of this section, an Emacs Lisp symbol is represented by a Scheme structure whose fields contain the symbol's value, function, and property list. The symbol structure also holds hooks (procedures) called by some operations on the symbol. These specialized procedures implement the operations and arrange to reflect the state of the symbol in the Edwin runtime system, or to access the Edwin runtime state in order to reflect it in the value of the symbol. The hooks are assigned to the following fields of the symbol structure and are used to implement the corresponding symbol operations described here:

`bound?` (`%symbol-bound?` *symbol*)
Returns `#t` if *symbol* currently has a value; else, it returns `#f`.

`unbound!` (`%set-symbol-unbound!` *symbol*)
Causes *symbol* to have no current value.

`get-value` (`%symbol-value` *symbol*)
Returns the current value of *symbol*. If *symbol* does not have a value, a `void-variable` error is signaled.

`set-value!` (`%set-symbol-value!` *symbol value*)
Sets the current value of *symbol* to *value*.

`get-default` (`%symbol-default` *symbol*)
Returns the default value of *symbol*.

`set-default!` (`%set-symbol-default!` *symbol value*)
Sets the default value of *symbol* to *value*.

`make-local!` (`%make-variable-buffer-local!` *symbol*)
Makes the symbol have a local value in the current buffer.

`make-all-local!` (`%make-local-variable!` *symbol*)
Makes the symbol have local values in all buffers.

`kill-local!` (`%kill-local-variable!` *symbol*)
Makes the symbol have no local values.

`set-docstring!` (`%put!` *symbol property value*)
The procedure assigned to the `set-docstring!` field is called whenever the `%put!` operation adds a `variable-documentation` property to *symbol*. This allows the documentation string for an Emacs Lisp variable to be reflected in the corresponding Edwin editor variable, if any.

There are a few other fields of the symbol structure. The `name`, `value`, `function`, and `plist` slots should be familiar as the normal attributes of Lisp symbols. (Why there is a `value` slot while there are also `get-value` and `set-value!` hooks is explained below.) The `next` field is used to chain interned symbols that have hashed to the same bucket in the Emacs Lisp obarray. The `command` field points to an Edwin command created to reflect the Emacs Lisp command named by the symbol.

There are a number of other operations on symbols too. The `%symbol-name`, `%symbol-function`, and `%symbol-plist` operations are straightforward, accessing the corresponding fields of a symbol structure. These operations are peculiar to Emacs Lisp and are not reflected in Edwin, so they do not need any hooks.

2.2.1 Simple Symbols

Most Emacs Lisp symbols are used as local variables — i.e. they are bound for the duration of a function call. They have one global value, not multiple, buffer-local values, and their values do not have to be reflected in the Edwin runtime system. The operations on these simple symbols are straightforward, and do not require individual hooks that escape into arbitrary reflection code. In this common case, the overhead of calling hooks (closures) can be avoided. The symbol operations listed above first access the `value` field of a symbol, since this field will typically contain the current global value of the simple symbol, or a special constant that indicates that the symbol is an unbound simple symbol. However, the `value` field may contain a second special constant that indicates that the symbol is *not* a simple symbol. In that case, the fields named above will contain hooks implementing the corresponding operations on the symbol.

2.2.2 Variable Symbols

Some Emacs Lisp symbols have buffer-local values — values that are current only when a certain buffer is current. The Emacs Lisp emulator implements such a symbol by finding or creating an Edwin editor variable with the same name. The editor variable can have buffer-local values, and already implements the behavior required of the Emacs Lisp symbol. The Emacs Lisp symbol is labeled as a non-simple symbol and provided with hooks that implement each of the operations listed above in terms of analogous operations on the Edwin editor variable. By default, the editor variable is maintained with identical state — its values are always identical to the emulated Emacs Lisp symbol's values. The only data type conversion is from an Emacs Lisp value of `Qt` (the symbol named `t`) to the analogous Edwin Scheme constant `#t`. (This satisfies pre-existing editor variables that insist on having only boolean values.)

2.2.3 Generic Symbols

Some Emacs Lisp symbols need to run specialized code to access or modify their values. This may be because the value of the symbol is intended to reflect (and be reflected in) the state of the Edwin Scheme runtime system. Emacs has similar types of symbols, whose values reflect values in C data structures. These kinds of symbols cannot be made to have buffer-local values, using the `make-variable-buffer-local` or `make-local-variable` subrs (though some of these symbols are *defined* to have buffer-specific values). Similarly, the `%make-local!` and `%make-all-local!` operations on generic symbols have no effect.

The details of how variable and generic symbols are used to integrate the emulated Emacs Lisp runtime system and the Edwin Scheme runtime system are addressed in Section 3.

3 Reflecting Emacs's State in Edwin

Many of Emacs's data structures are represented by Edwin data structures, e.g. Emacs buffers are represented by Edwin buffers. This reduces the implementation effort by taking advantage of existing Edwin functionality, and also helps integrate the runtime systems by allowing them to share state. To implement this sharing, the emulated Emacs Lisp primitives must be able to realize new states of the emulated Emacs Lisp runtime system by modifying the shared data structures, and also must be able to infer the state of the Emacs Lisp runtime system from the state of the Edwin data structures. For example, an Emacs primitive that sets an Emacs marker to a particular point in an Emacs buffer is implemented by a procedure that sets the corresponding Edwin mark to the same point in the corresponding Edwin buffer. An Emacs primitive that returns the position of an Emacs marker will infer the correct number from the index of the corresponding Edwin mark.

A number of techniques are used to share as much state between Emacs Lisp and Edwin Scheme as possible. Sometimes this sharing is assisted by additional information in separate data structures maintained by the emulation code. Sometimes it is made possible by hacking Edwin data structures below their abstract interfaces. However, there are some aspects of the emulated Emacs Lisp runtime state that cannot be easily or efficiently realized in terms of Edwin's runtime state. In these cases, the Emacs Lisp emulation has to implement this state separately. The rest of this section discusses the techniques used to emulate the significant aspects of Emacs Lisp's runtime state, and to reflect this state in Edwin Scheme.

3.1 User Variables

An Emacs Lisp program is often parameterized by a number of *user variables* – symbols whose values are intended to be modified by users to customize the behavior of the program. These symbols have a `variable-documentation` property whose value is a string starting with an asterisk. Only these symbols satisfy the `user-variable-p` predicate and can be modified using the `set-variable` or `edit-options` commands. Edwin editor variables can play a similar role, and are modified by the Edwin `set-variable` command. The Edwin command is more general, allowing any editor variable to be modified, but it is typically used in the same way as the Emacs command.

To reflect Emacs's user variables as Edwin editor variables, all symbols declared by the `defvar` and `defconst` subrs to have documentation strings starting with an asterisk are made into variable symbols (using the `%make-symbol-variable!` procedure). The variable symbols store their values as the values of Edwin editor variables with the same names. They also reflect their `variable-documentation` properties in the documentation strings of the editor variables. Thus, the user can get the variable documentation, and inspect and set the values of these symbols using Edwin's normal variable commands.

3.2 The Current Buffer

Emacs has the notion of a *current buffer* — the buffer acted upon by most of its primitives either implicitly, or by default. The current buffer is set to the buffer associated with the selected window before each interactive command is invoked. It can be temporarily changed (until the next interactive command is invoked) by the `set-buffer` subr. This primitive just changes the notion of the current buffer; it does not change the selected window or the buffer associated with the selected window.

Edwin also has a notion of a current buffer, which serves the same role as Emacs's. Edwin's current buffer is defined as the buffer of the selected window. It cannot be changed without changing either the selected window or the buffer associated with the selected window.

Edwin's notion of the current buffer, therefore, is incompatible with Emacs's. The Emacs Lisp emulation must maintain and use a separate notion of the current buffer. This state variable is initialized by the code that emulates Emacs's command dispatch mechanism, and it is set by the `set-buffer` subr. Fortunately, most Edwin procedures take an explicit buffer or mark argument, so the emulator can easily specify the buffer on which Edwin procedures should operate.

3.3 Markers

Edwin's *permanent* marks move as characters are inserted or deleted behind them in their buffer so that they continue to point to the same character, just as Emacs markers do. However, Edwin actually has two flavors of marks: left inserting and right inserting. The right inserting marks do not move when characters are inserted at the position of the mark. Thus, permanent *right inserting* marks are used to represent Emacs markers. They implement the correct behavior except for two minor differences. The first difference is between a mark's index and a marker's position. A permanent mark's index is the zero-based index of the buffer character following the mark. A marker's position is the one-based index of the buffer character following the marker. The second difference is that a marker can point nowhere, so that it does not slow down inserts and deletes in a buffer.

The first difference is relatively easy to reconcile. The `marker-position` subr must return an integer one greater than the index of the Edwin mark that is representing the Emacs mark. If the marker is pointing nowhere, `marker-position` signals the appropriate Emacs Lisp error.

The second difference is accommodated by violating the mark abstraction slightly. There is normally no way to make an Edwin mark point nowhere. However, the emulation code uses low-level structure modifiers to set a mark's `index` and `group` fields to `#f`. This is not a legal value for these fields, but Emacs markers should not be finding their way into Edwin code. Before setting the mark's fields to `#f`, the emulator turns the mark into a *temporary* mark, using the Edwin procedure `mark-temporary!`. A temporary mark does not appear on a buffer's list of marks to move because of insertions or deletions. Thus, the strange values of the mark's fields will not be observed by a buffer while moving its marks, and the mark will no longer slow down a buffer's inserts and deletes, just like a real Emacs marker. Once a marker is given a buffer and position again, it can be turned into a permanent mark using Edwin's `mark-permanent!` procedure.

Some care must be taken to distinguish between marker *positions*, which are one-based indexes, and mark *indexes*, which are zero-based indexes. Arguments to Emacs Lisp subr's may specify positions using integers (positions) or markers, and these are usually converted to the Edwin equivalents required by Edwin procedures — buffer positions (indexes) or marks. Other arguments to subr's may specify numbers using integers or markers (e.g. the arguments to the `+` subr), with the markers being converted to their one-based positions. The emulator uses three coercion procedure to help make the correct distinctions and implement the correct conversions:

`CHECK-MARKER-COERCE-INT`
Returns an Edwin mark when given an Emacs marker or integer. Otherwise, signals a `wrong-type-argument` error.

`CHECK-NUMBER-COERCE-MARKER`
Returns an integer when given an Emacs integer or marker. The integer is to be used as a number, and so is a one-based position. Otherwise, signals a `wrong-type-argument` error.

`CHECK-POSITION-COERCE-MARKER`
Returns an integer when given an Emacs integer or marker. The integer is to be used as a buffer position, and so is a zero-based index. Otherwise, signals a `wrong-type-argument` error.

3.4 Window Points

In both Emacs and Edwin, a window keeps track of its own point into its buffer. This allows multiple windows to view and edit different locations in the same buffer. In Emacs, a window's marker is right-inserting — insertions at the marker's position do not advance the marker. In Edwin, a window's mark is left-inserting — insertions at the mark position *do* advance the marker. If a buffer displayed in a window is erased and new text inserted, the point in an Edwin window will be left at the end of the buffer, whereas the point in an Emacs window will be left at the start of the buffer.

This is a drastic change in the way a program like GNUS behaves. To eliminate this incompatibility, all buffer insertions done by the emulated Emacs Lisp subrs are performed inside the dynamic extent of a call to `%fixup-window-point-movement`. This procedure arranges to find all window points that may be pointing at the place of insertion. After the insertions, it restores the window points to their original position.

3.5 Named Commands

A typical Emacs Lisp program will define a number of *commands* using a `defun` expression with a top-level `interactive` expression in its body. The `interactive` expression may provide code or a string specifying how to prompt the user for the arguments to the command. The program may also declare a command using an `autoload` expression with a non-nil fourth argument (`from_kbd`). Although there are many ways to create functions that can be called interactively, these are the two most common techniques. These two techniques intern symbols that can be bound to command key sequences and executed by the command dispatch mechanism. The commands can also be executed by giving their names to the `M-x` command (`execute-extended-command`). Although there are other ways to invoke Emacs Lisp programs, these are the two most common techniques — especially the latter. For example, the RMAIL program is run by typing `M-x rmail`; the GNUS program is run by typing `M-x gnus`.

A typical Edwin program will define a number of commands using `define-command` syntax. This syntax builds a command object that includes an interactive specification — a procedure or string specifying how to prompt the user for the arguments to the command. Edwin commands can be bound to command key sequences or executed by Edwin's `M-x` command (`execute-extended-command`). As in Emacs, the latter is the usual way to run an Edwin program.

To allow Edwin to run Emacs programs as though they were native Edwin programs, Edwin commands must be created to reflect the Emacs commands. Since the interactive specification of an Edwin command can be an arbitrary procedure, that procedure can get the interactive specification of the Emacs command and interpret it, returning the command arguments. The procedure that actually executes the Edwin command would accept the command arguments and apply the Emacs command to them using the `%apply-interactively` procedure, which does necessary initializations (e.g. setting Emacs's current buffer and `this-command` variable).

3.6 Keymaps

Edwin's command key dispatching mechanism is controlled by comtabs. A comtab associates a command key with various types of data: a command to be executed, a secondary comtab (used by the `prefix-key` command to dispatch additional command keys), a pair providing both a comtab for dispatching additional command keys *and* a command to do the dispatching; a comtab alias (a pair of a comtab and a command key, which is interpreted as equivalent to the datum associated with that command key in that comtab), and `#f` (indicating that the command key sequence is not bound).

Emacs's command key dispatching mechanism is similar and is controlled by keymaps. A keymap associates a command key with a number of different types of data: an interactive function, a keyboard macro (string), a keymap alias (a pair of a keymap and a command key), a secondary keymap, or a symbol whose function is any of these.

An Edwin comtab can be used to implement the same association as does an Emacs keymap, assuming that Emacs keymap data can be converted to Edwin comtab data. This is trivial for secondary keymaps and keymap aliases, which are already valid comtab data. The other types of keymap data will have to be converted to Edwin commands. However, Edwin commands are uniquely named, while most of the keymap data (the keyboard macros and interactive functions) are anonymous.

The solution used in this emulation is to create *anonymous commands* using Edwin's internal procedure `%make-command`. `%make-command` creates a command structure, but does not register it in Edwin's table of named commands. The name of an anonymous command, which must be a string, is a particular string constant that serves to identify it as an anonymous command. The interactive specification of an anonymous command is a procedure that returns the original Emacs Lisp keymap datum that was converted into this anonymous command. This serves two purposes. First, the `lookup-key` subr, recognizing an anonymous command by its name, can convert the command back into the original keymap datum simply by invoking the interactive specification procedure. Second, Edwin's command dispatch mechanism will invoke the interactive specification procedure and apply the command's procedure to the returned value — the keymap datum. The anonymous command's procedure will be `%keymap-dispatch`, a procedure which, when applied to a keymap datum, emulates Emacs's dispatching behavior.

3.7 Modes

In Emacs, a buffer's mode is defined implicitly by mode-related state variables — the buffer-local keymap and the buffer-local values of symbols like `major-mode` and `mode-name`. In Edwin, a buffer has an explicit major mode — a structure that includes fields called `comtabs`, `name`, and `display-name`. Edwin uses the values of these fields in much the same way as Emacs uses its mode-related state. Therefore, the mode-related state of an Emacs buffer needs to be reflected in the major mode of the corresponding Edwin buffer.

All procedures that emulate Emacs's manipulation of mode-related state first apply `guarantee-elisp-mode!` to the current buffer. This procedure ensures that the buffer's major mode is an Emacs Lisp mode — a major mode reflecting Emacs Lisp's notion of the buffer's mode. These major mode objects are only used by one buffer, so that the fields of the structure are buffer-specific. `guarantee-elisp-mode!` can tell that a buffer's major mode is an Emacs Lisp mode by looking for a particular property in the major mode's `alist` field. If the property is missing, the major mode is a normal Edwin major mode and must be replaced by a new Emacs Lisp mode.

The fields of a new Emacs Lisp mode are initialized to reflect a buffer in Emacs's `fundamental-mode` with no local keymap. The `comtabs` field of the major mode structure is initialized to a list of one comtab — the comtab of Edwin's `Fundamental` mode. This comtab is used to represent Emacs's global keymap since it is inherited by most of Edwin's major modes and has many of the same bindings.

Edwin major modes are normally uniquely named, making them easy to reference. However, Emacs Lisp modes are never referenced except implicitly through their associated buffer, and inventing unique names for them would be difficult. Like the anonymous commands wrapping Emacs keymap data, Emacs Lisp modes are anonymous major modes; they are not entered into Edwin's table of major modes.

The `major-mode` and `mode-name` variables are implemented by generic symbols whose `set-value` hooks apply `guarantee-elisp-mode!` to the current buffer and then modify the fields of the Emacs Lisp mode. Similarly, the procedures implementing Emacs's `use-local-map` and `kill-all-local-variables` subrs apply `guarantee-elisp-mode!` to the current buffer and then modify the list of comtabs of the Emacs Lisp mode. Thus, these procedures force the current buffer to become an Emacs-style buffer with an Emacs Lisp mode. This seems reasonable because most Emacs Lisp programs do not take these actions outside of functions intended to initialize the mode of a buffer created by the programs.

The `get-value` hooks of the `major-mode` and `mode-name` symbols can handle a current buffer that is either a normal Edwin buffer or an Emacs-style buffer. They simply access the appropriate fields of the current buffer's major mode regardless of its type. The `current-local-map` primitive just returns `nil` for Edwin buffers. Thus, Emacs Lisp code can inspect the modes of normal Edwin buffers without forcing them to become Emacs-style buffers.

It is interesting to note that Edwin minor modes can still be used in Emacs-style buffers. The comtabs of the minor modes are still prepended to the list of the buffer's applicable comtabs. Newly enabled Edwin minor modes will still override all other command key bindings, even those of Emacs's local keymap. Whether this feature will prove useful remains to be seen.

3.8 Prompting and Completion

The basis for Emacs's minibuffer input and completing behavior is a function and the values of a few symbols. The C function `read_minibuf` is used by all of the prompting subrs to implement the kernel functionality, such as saving the window configuration, activating the minibuffer window with a prompt and keymap, and starting the recursive edit loop. Completion is provided by command key dispatch in the minibuffer, which uses the minibuffer's keymap to map certain command keys to completion functions. The completion functions get the information they need from the values of three symbols:

- The value of `minibuffer-completion-table` should be an obarray (which associates strings with symbols) or an alist (specifically, an alist that associates strings with arbitrary objects). The keys of either kind of association include the possible completions of the minibuffer input.
- The value of `minibuffer-completion-predicate` should be a predicate used to filter the keys of `minibuffer-completion-table`. Together, the values of these two symbols determine the possible completions.
- The value of `minibuffer-completion-confirm` should indicate whether completions must be displayed and confirmed before being accepted. If the value of this symbol is `nil`, the unique prefix of a possible completion will be accepted without displaying the entire string and asking for confirmation.

Edwin's minibuffer input and completion behavior is implemented in a similar fashion. Its internal procedure, `%prompt-for-string`, does essentially the same work as `read_minibuf`, activating the minibuffer with a specialized major mode. The comtabs of the major mode provide the normal editing commands plus a few completion commands. The completion commands get the information they need from the values of three Scheme variables:

- The value of `completion-procedure/complete-string` should be a procedure that can complete a string.
- The value of `completion-procedure/list-completions` should be a procedure that can list the completions of a string.
- The value of `completion-procedure/verify-final-value?` should be a predicate indicating whether a string is an acceptable final value even if it is not unique.

- The value of `*completion-confirm?*` indicates whether completions should be displayed and confirmed before being accepted.

The emulator can easily find the appropriate values to fluid-bind to the Scheme variables. The values of the completion procedure variables can be simple procedures that use the `try-completion` or `all-completions` subrs together with the values of the Emacs Lisp symbols to complete the input string or to find a list of all its completions. The value of the `*completion-confirm?*` variable is also easily computed, using the value of the Emacs Lisp symbol `minibuffer-completion-confirm`.

The mode to be used by `%prompt-for-string` is a little more difficult to produce. Most of the minibuffer input subrs call `read_minibuf` with the values of certain Emacs Lisp symbols, which are typically standard keymaps. However, the symbols' values could be set to anything by the user, or even temporarily lambda-bound by Emacs Lisp code. Also, the `read-from-minibuffer` subr calls `read_minibuf` with a keymap provided as an argument. The minibuffer input subrs, therefore, must be emulated by procedures calling `%prompt-for-string` with a mode whose comtabs are: an arbitrary comtab representing a keymap that was a symbol value or direct argument, and the comtab representing Emacs's global keymap.

The required mode is easy to construct on the fly and, like the anonymous major modes of Emacs-style buffers, will not be registered as a named mode, allowing it to be garbage collected after its brief use during the call to `%prompt-for-string`. However, the required mode is often already represented by existing Edwin major modes. The values of the Emacs Lisp symbols used by the minibuffer input subrs are typically standard keymaps. These standard keymaps are represented by the comtabs of standard Edwin minibuffer modes. Rather than construct an anonymous mode on the fly, these comtabs are recognized and the standard modes are used. The `keymap->mode` procedure implements this mapping, returning either a standard Edwin minibuffer mode or a newly constructed anonymous mode.

4 Compatibility Issues

The techniques described in Section 3 address the two goals of accurately emulating the Emacs Lisp primitives, and tightly integrating the Emacs Lisp and Edwin Scheme environments. However, complete and accurate emulation and integration were not possible given the available resources. The size and complexity of the Emacs Lisp runtime system required that a number of Emacs Lisp primitives be left unimplemented. Of the primitives that *have* been implemented, many have capabilities that are rarely used and would not have been worth the effort to accurately implement. Some capabilities are simply not implemented, but others were replaced with useful approximations.

Complete and accurate integration is also difficult to achieve, both because of the size of the Emacs Lisp runtime system and because some aspects of the runtime system's state are difficult to reflect in analogous Edwin Scheme state. In any case, tight integration is more a convenience than a necessity; it does not increase the number of Emacs Lisp programs that can be emulated.

Users who expect all of the functionality of Emacs Lisp to be available or all analogous aspects of Emacs Lisp and Edwin Scheme state to be integrated will be disappointed. To set realistic expectations, this section describes the limitations of the Emacs Lisp emulator. It discusses both the general principles that guided the selection of compromises, and specific deficiencies that can lead to errors during emulation. This will help users understand why an Emacs Lisp program may be encountering errors, and will also point out the areas where the current system could be improved.

4.1 Signaling Errors

A couple caveats about the Emacs Lisp emulator concern how error situations are handled. Edwin Scheme code uses the MIT-Scheme condition system to signal error conditions, whereas Emacs Lisp uses its own mechanism. The emulated subrs endeavor to indicate error situations by signaling the same Emacs Lisp errors that would have been signaled by the original Emacs Lisp subrs in an analogous situation. This allows for the correct operation of Emacs Lisp programs that were designed to catch and handle these errors.

The emulated subrs accomplish this signaling behavior in a variety of ways. They can usually adequately detect error situations via the same type checking as that found in the original subrs. The type checking code can then signal the correct Emacs error. Adequate type checking often eliminates the possibility of Scheme errors being signaled by the Edwin Scheme procedures used to implement the subrs. In other cases, the emulated subrs may use Scheme condition handlers to catch Scheme conditions signaled by the Edwin Scheme procedures. The handlers can then signal the appropriate Emacs errors instead. However, it is difficult to anticipate all of the conditions that could be signaled by Edwin Scheme procedures, so it is still possible for Scheme conditions to be signaled during the emulation of an Emacs Lisp program. The obvious cases that correspond directly to the Emacs errors signaled by the original Emacs Lisp subrs should be handled correctly.

One obvious case that currently is *not* handled is the `quit` error that is signaled by Emacs when it gets the interrupt character (`?\C-g`). This character is bound to the `keyboard-quit` command in the global keymap, but Emacs's handling of `?\C-g` is more complicated than simple command key dispatch, and involves at least the values of two symbols. Edwin's own handling of `?\C-g` is no less complicated, using a mechanism entirely separate from the normal Scheme condition system. While it is probably possible to more accurately emulate the interruption of Emacs Lisp code by `?\C-g`, this has not been done. There are likely to be few Emacs Lisp programs that rely on this behavior anyway.

The emulator will correctly implement the error signaling and handling behavior of Emacs Lisp in most situations. However, there are a few situations in which Emacs Lisp errors will be signaled by the emulator though they would not have been signaled in Emacs. These errors reveal deficiencies in the emulator. A simple example is an Emacs Lisp program that attempts to call a subr that has not been implemented. The attempt will cause a `void-function` Emacs error to be signaled. This error will indicate to the user what went wrong in a fairly straightforward fashion. A less straightforward error is generated when an Emacs Lisp program violates one of the restrictions imposed by the emulator. For example, an Emacs Lisp program might attempt to cdr down a sparse keymap (i.e. traverse the keymap using `cdr`). This violates the restriction that only the abstract keymap operations be applied to keymaps. The result is that the `cdr` subr will signal an Emacs Lisp `wrong-type-argument` error because the sparse keymap is not a cons, but actually an Edwin comtab.

An Emacs Lisp program can also violate restrictions of the emulator and cause a Scheme error condition to be signaled. This may be a consequence of using Edwin editor variables to store the values of certain Emacs Lisp symbols. In Emacs Lisp, the values of most symbols can be any object. This includes symbols for whom only a few values make sense, such as symbols that are conceptually boolean valued. An Emacs Lisp program is free to set the value of such a symbol to be `nil` or any non-`nil` value. For example, an Emacs Lisp program could set the buffer-local value of `buffer-read-only` to a string describing why the buffer should not be modified.

Setting some Emacs Lisp symbols to unusual values can cause Scheme errors to be signaled. Symbols with buffer-local values are emulated by *variable symbols* which store their values as the values of similarly named Edwin editor variables. Edwin editor variables can restrict the values to which they can be set and will signal a Scheme condition if a given value is outside their expected range. Thus, an Emacs Lisp program that uses a buffer-local symbol in an unusual way may cause an editor variable to signal a Scheme condition. This should be rare because only a few standard Edwin editor variables have restrictions on their values, and these standard names are unlikely to be used in unusual ways by Emacs Lisp programs. However, a few cases already exist among the standard Emacs Lisp symbols that were implemented. Any restrictions imposed on these symbols by the emulator have been documented in the standard symbols' documentation strings. Section A.0.7 includes all documentation strings that were augmented with these kinds of notes.

4.2 Ignoring Deficiencies

The Emacs Lisp emulator can signal errors whenever it is unable to correctly emulate Emacs Lisp code, but this is often not what is desired. There are many cases where a deficiency of the emulator can be ignored without grossly affecting the behavior of an Emacs Lisp program.

For example, there is no analogue for Emacs's `help-form` symbol in Edwin, so the emulator does not provide equivalent functionality. The emulator could signal an error any time the symbol's value is referenced or set. This would help the user understand why an Emacs Lisp program is not behaving exactly as expected, but also makes it impossible to run the program even if it would run correctly in all other respects. Instead, the emulator gives `help-form` an initial value and ignores any attempt to change it. As a result, the program may be a little less helpful than it could have been, but it won't halt with an error just because it references or sets the value of `help-form`.

The same strategy applies when implementing subrs. The Emacs Lisp subr `interactive-p` is supposed to return `t` only if it is called from the initial, interactive invocation of an Emacs Lisp command. This is easy to implement in Emacs because the subr can examine the interpreter stack. Doing the same thing in Edwin Scheme is probably possible, but probably difficult. However, in the entire GNU Emacs distribution, `interactive-p` is invoked in only 5 function calls, all of which produce acceptable behavior if they always return `t`. Rather than leave this subr undefined and halt any programs using it, the `interactive-p` subr is emulated by always returning `t`.

When deficiencies in the emulation are not signaled, it becomes harder to track down the source of discrepancies between a program running in Emacs and the same program being emulated in Edwin. An error resulting from a deficiency may be far removed from the point where the deficiency first appeared. Backtracking from the error to that point is likely to be difficult. Therefore, the emulator will only attempt to proceed with partially correct behavior where it is not likely to cause further trouble. Otherwise, an error will be signaled. In either case, the known deficiencies of the emulated symbols and subrs are noted in their documentation strings. This will at least make the user aware of the possible sources of discrepancies.

4.3 Being Innocuous

Another general principle followed by the Emacs Lisp emulator is to be conservative when reflecting the emulated Emacs Lisp system's state into Edwin. A reflection that breaks Edwin is considered worse than the lack of a reflection that defeats integration or even introduces a discrepancy in the emulation. It is difficult to guarantee that Edwin cannot be broken, but the following techniques have been used to eliminate obvious problems.

This project emulates a large number of the Emacs Lisp primitives, but it does not attempt to emulate the many functions provided in essential Emacs Lisp files. These files define functions (e.g. `not`) needed by nearly all Emacs Lisp programs. The files also define the global command key bindings. Since these essential files are necessary to make most Emacs Lisp programs work, they must be loaded into the emulation. However, if they were loaded in as they are, without modification, they would override most of Edwin's global command key bindings. Many of its editor variable values would be overridden too. To avoid the problem of overriding the global command key bindings without patching (and maintaining patches to) the essential Emacs Lisp files, a switch is provided that controls whether *any* comtab bindings can be overridden. This switch, the Scheme variable `allow-elisp-define-key-overrides?`, is fluid-bound to `#f` while the essential files are loaded.

The `defvar` primitive by definition will not override existing variable values, but the `defconst` primitive is defined to do just that. However, if `defconst` expressions in the essential Emacs Lisp files override existing Edwin editor variable values, many of them will break Edwin. Fortunately, the defined behavior can be compromised so as not to break Edwin. Variable symbols (symbols reflected by Edwin editor variables) remember whether their corresponding editor variable existed before they were created. If it did, the variable symbol assumes that the editor variable is being used by Edwin and that its value should not be overridden by `defconst`. `defconst` detects this situation when the `%symbol-bound?` operation returns the `edwin` symbol. If `%symbol-bound?` returns any other value, then the symbol's value can be set; the symbol was either unbound or its editor variable was created by the emulation code.

4.4 Handling Side-Effects

The Emacs Lisp emulator currently does not go to great lengths to handle side-effects to Emacs Lisp data structures. These side-effects need to be handled specially when they cause changes in state of an Emacs Lisp data structure and that state is being reflected in the state of an Edwin Scheme data structure. To a large extent, detecting and reflecting these state changes has been unnecessary because of the large amount of data that is actually shared between the two systems. Any change in the state of shared data structures already affects both systems.

However, there are a number of Emacs Lisp data structures that are reflected in Edwin Scheme via translation into a corresponding Edwin Scheme data structure. The translation is only applied when a data structure is registered with the Emacs Lisp runtime system by calling a subr or setting the value of a symbol. The translation, a newly constructed Edwin Scheme data structure, is similarly registered with the Edwin Scheme runtime system. Side-effects to the Emacs Lisp data structure will not invoke this translation process and register a new reflection in the Edwin Scheme runtime system.

An example of this is the value of the `exec-path` symbol. The value of this symbol should be a list of pathnames (strings) or `nil`. It is analogous to the value of the Edwin editor variable `exec-path`, which is a list of pathname objects, strings, or `#f`. The `get-value` method for the Emacs variable can easily coerce the value of the Edwin variable to the correct type, mapping pathnames to strings and `#f` to `nil`. However, the resulting list will be distinct (unless we want to force the Edwin variable to share the new value). Any side-effects to it will not change the state of the editor variable.

So far, this deficiency in the emulation has not been a problem. Side-effects to many data structures, while possible, are unusual in Emacs Lisp programs. For example, most programs will change a symbol's `variable-documentation` property via the `put` subr, rather than via direct modification of the symbol's property list. The `put` subr will detect the change and can reflect it as a new documentation string for a corresponding Edwin editor variable. The direct modification of the property list will not be detected and the documentation string of the editor variable will not reflect the new state.

As a possible source of bugs, unhandled side-effects are noted in appropriate documentation strings. In the future, Emacs Lisp lists might be built out of special cons cells that accept notification hooks. Then, any side-effects to an Emacs Lisp list could trigger code that reflects the new state of the list into a corresponding Edwin Scheme data structure. Practical experience with the current emulator can focus such efforts to handle side-effects that are crucial to useful Emacs Lisp programs.

4.5 Naming Commands and Variables

The Emacs Lisp emulator does its best to reflect the commands defined by Emacs Lisp programs in corresponding Edwin commands with the same name. Users can then easily invoke an Emacs Lisp command simply by invoking the corresponding Edwin command. The Emacs Lisp command name can be used to bind command keys to the corresponding Edwin command, and the same name can be provided to the `M-x` command (`execute-extended-command`). The Emacs Lisp commands will even appear in the completion lists displayed by `execute-extended-command`.

The same close integration is intended for Emacs Lisp variables, the symbols whose global or buffer-local values parameterize the behavior of Emacs Lisp programs. The names of corresponding Edwin editor variables should be the same so that users can easily inspect and set the values of the Emacs Lisp variables via operations on the Edwin editor variables.

Unfortunately, Emacs Lisp symbol names are case-sensitive and they are unique to a particular symbol only among symbols in the same obarray. An uninterned symbol, or a symbol interned in another obarray could have the exact same name as another symbol. The names of Edwin commands and editor variables, however, are case-*in*sensitive and must be unique among all commands or variables (respectively).

A straightforward mapping of Emacs command names to Edwin command names would just lowercase the Emacs command names, but this can cause conflicts when distinct symbols naming different commands are mapped to the same Edwin command name. The same problem exists when mapping Emacs variable names to Edwin editor variables names. Two distinct symbols may end up trying to store their global or buffer-local values as the values of the same editor variable.

A more complex mapping of distinct symbols to *unique* Edwin command or variables names is not likely to be easy to use. Luckily, the practice of naming Emacs commands and variables with uppercase letters is rare, and the name conflicts that would be caused by the straightforward mapping are even rarer. For now, the emulator uses the straightforward mapping and risks conflicts between Emacs variables. Name conflicts between distinct Emacs commands are arbitrarily decided in favor of the command first defined — the Emacs symbol naming the command will be associated with an Edwin command that invokes that symbol as an Emacs command. The decision in favor of the first command defined is actually a consequence of being innocuous. The emulator will not re-define an existing Edwin command to invoke an Emacs command.

The mapping between Emacs commands and Edwin commands is not one-to-one for another reason. Function aliasing makes it hard to find new named commands that aught to be reflected as an Edwin command. If the function of an Emacs symbol is set to be another symbol, then the first symbol serves an an alias for the function named by the second symbol. If the second symbol becomes the name of a command, then the first symbol becomes an alias for the named command, but there is currently no way to find that first symbol again and to reflect it as an Edwin command. As with the name conflicts described above, the named command aliases that would be lost by this particular sequence of definitions are thought to be rare in practice, and no effort has been made to eliminate the problem.

4.6 Using Comtabs For Keymaps

As described in Section 3.6, Emacs Lisp keymaps are represented by Edwin Scheme comtabs. At an abstract level, this representation is a good fit because the behaviors of keymaps and comtabs are very similar. Implementing the abstract keymap operations like `define-key` and `lookup-key` in terms of operations on comtabs was fairly straightforward, and integrating keymaps and comtabs helped to integrate Emacs's and Edwin's command key dispatching mechanisms.

At a more concrete level, a *full keymap* is supposed to be represented by a vector, and a *sparse keymap* by a pair whose car is the symbol `keymap` and whose cdr is an association list. Emacs Lisp programs are supposed to be able to apply vector operations to full keymaps and list operations to sparse keymaps. However, emulating these operations, when applied to an Edwin comtab representing a keymap, would be difficult. Instead, the emulator imposes the restriction that only the abstract keymap operations be applied to the emulated keymaps. This restriction was thought to be acceptable to many useful Emacs Lisp programs, and has proven acceptable to the benchmark GNUS news reader.

One notable exception to the restriction is the standard Emacs Lisp function `suppress-keymap`, which uses `aset` to modify a full keymap (vector). This function is widely used, so to accommodate it, the `aset` subr is implemented so that it can be applied to a comtab to produce the same key binding as it would have produced when applied to a full keymap in Emacs.

4.6.1 special keys

Even when restricted to the supported, abstract operation on keymaps, there are some fundamental differences between Edwin comtabs and Emacs's keymaps that can cause errors in Emacs Lisp code. Edwin command keys are lists of arbitrary objects, which are usually Scheme characters and sometimes other objects representing special keys (e.g. function keys). Emacs command keys are strings of ascii characters. Special keys are encoded as escape sequences via a mapping that is terminal-dependent. Emacs Lisp code that tries to manipulate command key bindings involving function keys (e.g. by using the escape sequence produced by a commonly used terminal) will not work.

This difference between comtabs and keymaps also affects the emulation of the `this-command-keys` subr. In the simple case, `this-command-keys` uses Edwin's `current-command-key` procedure to get the command key sequence that most recently invoked a command. The Edwin command key sequence (a list of characters) is easily converted into an Emacs command key sequence (a string) which is returned. If the command key sequence includes a special key, the special key is be converted to a string containing the name of the special key with a prefix of `\e[` and suffix of `~`. This will be meaningless to a program that really understands escape sequences, but will work with programs that just compare them. If the command key sequence includes anything else, the emulation gives up and signals an error.

4.6.2 `esc-map`

Another difference between comtabs and keymaps concerns how they handle command keys with the meta-bit set. Emacs converts a command key with the meta-bit set into two command keys — the `ESC` key and the original command key with the meta-bit clear. Edwin does just the opposite. The `ESC` command key is bound to a command that sets the meta-bit on the next command key. The result is that Edwin does not have a comtab that corresponds to Emacs's `esc-map`. Emacs Lisp code that tries to modify the `esc-map` or build a keymap to bind to `ESC` will not work.

4.7 Integrating Essential Emacs Lisp Facilities

This project has implemented a number of Emacs Lisp subrs and variables, concentrating on those required by the benchmark GNUS program. This program also requires a number of standard Emacs Lisp functions and variables defined in the essential Emacs Lisp files that are pre-loaded into every Emacs. These files are loaded into the emulated Emacs Lisp environment, but the functions and variables they define have not been integrated with the Edwin Scheme environment.

This can be a problem when Emacs Lisp variables have the same name as existing Edwin editor variables. By default, the values of the Emacs Lisp variables are reflected as the values of the Edwin editor variables and vice versa. No translation is currently done between Emacs Lisp and Edwin Scheme data types. When the values expected of the Emacs Lisp variables are not compatible with the values expected of the Edwin editor variables, errors can result. For example, `find-file-hooks` is both an Emacs Lisp variable and an Edwin editor variable, but its value is a list in Emacs Lisp and an event distributor in Edwin Scheme. Emacs Lisp code that tries to examine the value of `find-file-hooks` using the `car` subr will halt with an error.

Eventually, the Emacs Lisp variables like `find-file-hooks` can be emulated by generic symbols, just as other primitive Emacs Lisp variables are emulated now. Alternatively, Edwin data types like event distributors might be manipulated (in limited ways) by Emacs Lisp subrs as though they were native data types, just as comtabs are manipulated in limited ways as though keymaps.

4.8 Miscellaneous Other Discrepancies

The preceding sections describe a number of caveats about the emulated Emacs Lisp runtime system, and general issues about the compatibility of the emulated environment with the original environment provided by Emacs. The compatibility issues have been illustrated by a few examples of the specific incompatibilities can arise, but these examples were not exhaustive. There are additional incompatibilities that may be instances of the general issues, special-purpose compromises, or combinations of both. For completeness, the appendix lists all of the primitives, indicating their implementation status. It also includes all documentation strings that note any restrictions or other unusual behaviors anticipated in specific Emacs Lisp primitives. This section deals with the special-purpose compromises in the implementation of specific Emacs Lisp primitives.

4.8.1 `mode-line-format`

Edwin's `mode-line-format` editor variable can be bound to values that will be interpreted in almost exactly the same way that Emacs interprets the values of its own `mode-line-format` symbol. The effect of setting the Emacs symbol to certain values can therefore be emulated by setting the Edwin editor variable to similar values. The translation from an Emacs value to a similar Edwin value would have to take into account not only a few minor differences in the meanings of the `%`-*constructs*, but also a more troublesome difference — the fact that Emacs symbols can appear in the Emacs value where Edwin editor variables would be expected in the Edwin value. This difference is troublesome because the values of the editor variables or Emacs symbols are in turn interpreted as though values of `mode-line-format` (with an egregious exception). Thus, a faithful emulation must not only translate Emacs Lisp values of `mode-line-format` substituting editor variables for Emacs symbols or vice versa, it must also arrange that the values of any of the referenced editor variables or Emacs symbols be similarly translated. Even if this is accomplished, there is the additional problem of side-effects to the values of any of the symbols or editor variables involved.

A partially correct emulation was found to be sufficient to get GNUS working, and should be acceptable to many other Emacs Lisp programs. Emacs's `mode-line-format` symbol is emulated by a generic symbol. The `get-value` hook of the generic symbol returns a copy of the value of the Edwin editor variable `mode-line-format`. The copy substitutes Edwin editor variables with freshly interned Emacs symbols. The `set-value` hook of the variable symbol is similar. It sets the Edwin editor variable to a copy of the given value in which any Emacs symbols were replaced with corresponding Edwin editor variables. Also, the Emacs symbols are made into variable symbols so that changes in their values will be reflected in changes to the values of the corresponding editor variables, though no special translations are done on the values of these symbols.

This is a good example of a situation in which a small change to Edwin could eliminate a compatibility issue. If Edwin allowed a buffer's mode line to be computed by an arbitrary procedure, the emulator could specify a procedure that correctly interprets the value of the Emacs Lisp `mode-line-format` symbol. Whether or not the Edwin Scheme `mode-line-format` editor variable was eliminated, there would be no need to reflect the value of the Emacs Lisp `mode-line-format` symbol in Edwin.

4.8.2 `this-command` and `last-command`

The Emacs Lisp symbol `this-command` is represented by a generic symbol whose operation hooks use Edwin's *command message* facility. If an appropriate message was not left by the `set-value` hook since the current command was dispatched, then the `get-value` hook returns Edwin's `(current-command)` (an Edwin command object) as a last resort. However, the emulation arranges to set the value of `this-command` whenever an Emacs command is called interactively. Thus, the unusual value of `this-command` will only be observable outside the dynamic scope of an Emacs command invocation, e.g. while a hook is being run, such as a process filter function or a `find-file-hook`.

The Emacs Lisp symbol `last-command` is emulated in a similar fashion. Using Edwin's command message facility, the `get-value` hook will return the value to which `this-command` was set during the previous command, or else an Edwin command object. This is sufficient to allow Emacs commands to recognize that a related command was invoked just previously, or that they have been invoked multiple times in succession. Thus, commands that work like Emacs's `yank` and `yank-pop` commands are supported.

4.8.3 `local-set-key`

The Emacs primitive `local-set-key` is supposed to find or create a local keymap for the current buffer and add a command key binding to it. However, an Edwin buffer does not have a local keymap unless it is an Emacs-style buffer. For simplicity, the emulation of `local-set-key` changes the current buffer into an Emacs-style buffer (if necessary), creates a local keymap for the buffer (if necessary), and creates a comtab binding that reflects the intended keymap binding. This is a rather drastic action. Changing an Edwin buffer into an Emacs-style buffer leaves it in a mode comparable to Emacs's `Fundamental` mode. The Emacs Lisp code may not have intended to make any major change in the current buffer's mode, but simply to enhance the existing mode with a convenient binding.

4.8.4 process-status-message

Edwin's `process-status-message` is a reasonable facsimile of the Emacs subr with the same name. However, to describe a signal that interrupted or killed a subprocess, Emacs uses the corresponding string from the C array `sys_siglist`. This is a descriptive string like `"bus error"`. The Edwin procedure just gives the signal number (e.g. `10` for a bus error) and notes when core has been dumped.

4.8.5 substitute-command-keys

When Emacs commands are defined, they are often given documentation strings intended to be displayed to the user only after the `substitute-command-keys` subr has been applied to them. This subr will recognize certain substrings of the documentation string as directives that need to be replaced. The most common directive names a related command and is replaced by the command key sequence that invokes that command. The resulting string shows the user how to invoke related commands using the short command key sequences instead of the much longer command names.

The same feature is implemented in Edwin, which applies its own `substitute-command-keys` procedure to substitute command names with command key sequences. Unfortunately, the Edwin procedure is not compatible with the Emacs subr. Not all of the directives recognized by the Emacs subr are recognized by the Edwin procedure. The peculiarities of the comtabs constructed by Emacs Lisp code also make it likely that command key sequences that invoke some of the related commands will not be found. When this happens, the directives that could not be replaced by command key sequences will be replaced instead by a default string indicating that the `execute-extended-command` (`M-x`) command can be used instead. This is only true when the Emacs command is accurately reflected in an Edwin command with the same name.

The emulated `substitute-command-keys` subr will interpret all of the directives found in the documentation strings of Emacs commands, and will return a string with the intended substitutions. Unfortunately, there is no hook in Edwin that would allow the emulated subr to be used instead of the Edwin procedure. Thus, the Edwin commands that are created to reflect Emacs commands are given the documentation strings that result from applying the subr to the original Emacs documentation string. The command key sequences described in the resulting string may become out-of-date, but they are still more likely to be correct. Again, this is a case where an appropriate hook added to Edwin could eliminate a compatibility issue.

5 Conclusion

This project was intended as a small step towards the ultimate goal of running any Emacs Lisp program in the Edwin Scheme environment. This ultimate goal requires that all of the hundreds of functions and variables provided by the Emacs Lisp runtime environment be emulated and integrated with the analogous facilities of the Edwin Scheme environment. It was obvious that the ultimate goal could not be realized given the limited resources of this project, but a useful step toward this goal could be made.

To ensure that the actual goal of this project would be useful, a valuable Emacs Lisp program, the GNUS news reader, was chosen as a benchmark. The actual goal of the project would be to emulate enough of the Emacs Lisp runtime system, and to integrate enough of Edwin Scheme with the emulated system, that the basic facilities of the GNUS news reader could be used as though the program were a native Edwin program. By focusing on this specific program, only a fraction of the Emacs Lisp runtime facilities would have to be implemented. By choosing such a sophisticated program, enough of the essential runtime facilities would need to be implemented that it would then be likely that other Emacs Lisp programs could be made to work given small, incremental improvements to the emulated environment.

In spite of the ultimate goal, and the efforts made to integrate the emulated Emacs Lisp runtime environment with the Edwin Scheme environment, these are still really two separate worlds. The buffers being used by Emacs Lisp code are unusual in a number of ways. The major modes of these buffers are anonymous. The comtabs of these major modes contain anonymous commands. Because of this, the normal Edwin commands for examining a buffer's state may yield strange results. For example: the Edwin command `describe-bindings`, when invoked in an Emacs-style buffer, may show many command key sequences bound to commands named `??` — the anonymous commands of the Emacs Lisp emulator. This kind of output is less informative than it could be. In the Emacs Lisp world, the `describe-bindings` subr would recognize the anonymous commands of the Emacs Lisp emulator and know how to find the original name of the Emacs Lisp command. Its output would reveal the names of the bound commands as they are known in the emulated Emacs Lisp environment, and these names could be used when creating bindings in Emacs Lisp keymaps or when using `execute-extended-elisp-command`, the Emacs Lisp version of Edwin's `execute-extended-command` command.

Integration of the Emacs Lisp and Edwin Scheme environments is really just a convenience. By integrating variables like `fill-column`, the user is saved the chore of maintaining the consistency of two representations of what are conceptually the same thing. By integrating the user interfaces (the windows, mode lines, and command dispatch mechanisms) of the emulated Emacs Lisp and native Edwin

Scheme systems, the user is saved from the disruption of switching between two slightly different interfaces. Where integration breaks down, the user will have to recognize that there are really two worlds, and will have to operate on the emulated Emacs Lisp world using the commands and functions native to that world.

Some of the incremental improvements that may be necessary in order to run other Emacs Lisp programs have already been demonstrated by the current system. There are many aspects of the Emacs Lisp runtime system that have not been integrated into Edwin. In particular, the functions and variables defined by the essential Emacs Lisp files (see Section 4.7) have not been addressed by this project. By default, Emacs Lisp user variables will share state with a similarly named Edwin editor variable. Where this default behavior causes problems, the user may be able to integrate incompatible representations of similar state following the example of the current emulator code. Generic symbols have been a powerful tool for arranging arbitrary translations between representations, and Edwin often supports equally powerful hooks.

Many of the incompatibilities discussed in Section 4 are reasonable restrictions for the Emacs Lisp system. A program that sets `fill-column` to a negative or non-integer is probably in error anyway. Other incompatibilities may be eliminated by more sophisticated use of the existing techniques. Such efforts may simply have been deemed impractical given the resources of this project. Some incompatibilities may require new techniques, including those anticipated by the discussion in Section 4. Emacs Lisp data types might be replaced by data structures that react to side-effects by invoking hooks. Edwin Scheme data structures might also be extended so that they react to side-effects. An Edwin Scheme process, such as the displaying of a command's documentation string, might also support hooks that customize that process. Finally, there are a number of features of Emacs Lisp that Edwin does not currently support. Eliminating these incompatibilities might involve implementing mode abbrev, horizontal scrolling, selective display, and perhaps others.

Despite its limitations, this project has succeeded in emulating a large fraction of Emacs Lisp's primitive functions and variables. Of the 587 primitive functions, 430 have been implemented. Of the 147 primitive variables, 88 have been implemented. The actual goal of running the GNUS news reader has been accomplished to the extent that articles can be fetched from the news server, displayed in an Edwin window, and saved in files. The database recording the articles that have been read is correctly maintained and can be edited with the usual GNUS commands. All of this can be done by an average Edwin user who knows little more than how to load the Emacs Lisp emulator into the Edwin Scheme system. The same Emacs Lisp code used to load and configure the GNUS program in Emacs can be used to load and configure it in Edwin. In terms of its basic functionality (i.e. except for the more sophisticated commands such as those calling the Emacs

Lisp sendmail program), the GNUS news reader, its windows, buffers, and simple commands, can be used as though GNUS were a native Edwin Scheme program. The performance of the emulator has received only scant attention. However, the GNUS program is currently only about 5 times slower than it is when run in its native GNU Emacs environment. Chris Hanson, the author of Edwin, believes that further performance testing and enhancement could allow the Emacs Lisp emulator to rival the speed of the original GNU Emacs implementation.

The purpose of this document was to describe the techniques used to emulated the Emacs Lisp world, and to integrate it with the Edwin Scheme world. The purpose was also to enumerate the limitations of the current emulator, and by so doing indicate opportunities for future improvement. Hopefully, it has served to set the expectations of users of the Emacs Lisp emulator. When they try loading and executing Emacs Lisp programs, they should have the information they need to recognize errors or other unusual behavior resulting from the limitations of the current system. Such experiments can then focus efforts to eliminate these limitations and move even closer to the ultimate goal.

A The Primitives

This appendix contains an index of the Emacs Lisp primitive functions and variables. Each entry in the list starts with a letter indicating:

- I The primitive has been implemented.
- R The primitive has been implemented, but its behavior is restricted, or even unsupported. Section A.0.7 shows the documentation strings of these primitives, which include a note describing how (and *if*) the primitive is supported by Edwin.
- U The primitive is unimplemented.

The rest of the entry gives the name and type of the primitive, and the name of the file in which it was implemented in the GNU Emacs distribution.

A.0.6 Index

I `%` (function in data.c)
I `*` (function in data.c)
I `+` (function in data.c)
I `-` (function in data.c)
I `/` (function in data.c)
I `/=` (function in data.c)
I `1+` (function in data.c)
I `1-` (function in data.c)
I `<` (function in data.c)
I `<=` (function in data.c)
I `=` (function in data.c)
I `>` (function in data.c)
I `>=` (function in data.c)
U `Control-X-prefix` (function in keymap.c)
I `abbrev-all-caps` (variable in abbrev.c)
I `abbrev-expansion` (function in abbrev.c)
R `abbrev-mode` (variable in buffer.c)
I `abbrev-start-location` (variable in abbrev.c)
I `abbrev-start-location-buffer` (variable in abbrev.c)
I `abbrev-symbol` (function in abbrev.c)
I `abbrev-table-name-list` (variable in abbrev.c)
I `abbrevs-changed` (variable in abbrev.c)
U `abort-recursive-edit` (function in keyboard.c)
I `accept-process-output` (function in process.c)

I `accessible-keymaps` (function in keymap.c)
I `add-name-to-file` (function in fileio.c)
I `all-completions` (function in minibuf.c)
I `and` (function in eval.c)
I `append` (function in fns.c)
I `apply` (function in eval.c)
U `apropos` (function in keymap.c)
I `aref` (function in data.c)
I `arrayp` (function in data.c)
I `aset` (function in data.c)
I `ash` (function in data.c)
I `assoc` (function in fns.c)
I `assq` (function in fns.c)
I `atom` (function in data.c)
R `auto-fill-hook` (variable in buffer.c)
U `auto-save-interval` (variable in keyboard.c)
I `autoload` (function in eval.c)
U `backtrace` (function in eval.c)
U `backtrace-debug` (function in eval.c)
I `backward-char` (function in cmds.c)
I `backward-prefix-chars` (function in syntax.c)
I `barf-if-buffer-read-only` (function in buffer.c)
U `baud-rate` (function in dispnew.c)
I `beginning-of-line` (function in cmds.c)
R `blink-paren-hook` (variable in cmds.c)
I `bobp` (function in editfns.c)
I `bolp` (function in editfns.c)
I `boundp` (function in data.c)
R `buffer-auto-save-file-name` (variable in buffer.c)
R `buffer-backed-up` (variable in buffer.c)
I `buffer-enable-undo` (function in buffer.c)
I `buffer-file-name` (variable in buffer.c)
R `buffer-file-name` (function in buffer.c)
I `buffer-flush-undo` (function in buffer.c)
I `buffer-list` (function in buffer.c)
I `buffer-local-variables` (function in buffer.c)
I `buffer-modified-p` (function in buffer.c)
I `buffer-name` (function in buffer.c)
R `buffer-read-only` (variable in buffer.c)
I `buffer-saved-size` (variable in buffer.c)
I `buffer-size` (function in editfns.c)

I `buffer-string` (function in editfns.c)
I `buffer-substring` (function in editfns.c)
R `buffer-undo-list` (variable in buffer.c)
I `bufferp` (function in data.c)
I `bury-buffer` (function in buffer.c)
I `byte-code` (function in bytecode.c)
I `call-interactively` (function in callint.c)
U `call-last-kbd-macro` (function in macros.c)
U `call-process` (function in callproc.c)
U `call-process-region` (function in callproc.c)
U `capitalize` (function in casefiddle.c)
U `capitalize-region` (function in casefiddle.c)
U `capitalize-word` (function in casefiddle.c)
I `car` (function in data.c)
I `car-safe` (function in data.c)
R `case-fold-search` (variable in buffer.c)
I `catch` (function in eval.c)
I `cdr` (function in data.c)
I `cdr-safe` (function in data.c)
I `char-after` (function in editfns.c)
I `char-equal` (function in editfns.c)
I `char-or-string-p` (function in data.c)
I `char-syntax` (function in syntax.c)
I `char-to-string` (function in editfns.c)
I `clear-abbrev-table` (function in abbrev.c)
I `clear-visited-file-modtime` (function in fileio.c)
U `command-execute` (function in keyboard.c)
U `command-history` (variable in callint.c)
U `command-line-args` (variable in emacs.c)
I `commandp` (function in eval.c)
I `completing-read` (function in minibuf.c)
R `completion-auto-help` (variable in minibuf.c)
I `completion-ignore-case` (variable in minibuf.c)
I `completion-ignored-extensions` (variable in dired.c)
I `concat` (function in fns.c)
I `cond` (function in eval.c)
I `condition-case` (function in eval.c)
I `cons` (function in alloc.c)
I `consp` (function in data.c)
I `continue-process` (function in process.c)
U `coordinates-in-window-p` (function in x11fns.c)

U `coordinates-in-window-p` (function in xfns.c)
I `copy-alist` (function in fns.c)
I `copy-file` (function in fileio.c)
I `copy-keymap` (function in keymap.c)
I `copy-marker` (function in marker.c)
I `copy-sequence` (function in fns.c)
I `copy-syntax-table` (function in syntax.c)
R `ctl-arrow` (variable in buffer.c)
U `ctl-x-map` (variable in keymap.c)
I `current-buffer` (function in buffer.c)
I `current-column` (function in indent.c)
I `current-global-map` (function in keymap.c)
I `current-indentation` (function in indent.c)
I `current-local-map` (function in keymap.c)
I `current-prefix-arg` (variable in callint.c)
I `current-time-string` (function in editfns.c)
I `current-window-configuration` (function in window.c)
U `cursor-in-echo-area` (variable in dispnew.c)
U `debug-end-pos` (variable in xdisp.c)
U `debug-on-error` (variable in eval.c)
U `debug-on-next-call` (variable in eval.c)
U `debug-on-quit` (variable in eval.c)
U `debugger` (variable in eval.c)
U `default-abbrev-mode` (variable in buffer.c)
R `default-case-fold-search` (variable in buffer.c)
R `default-ctl-arrow` (variable in buffer.c)
R `default-directory` (variable in buffer.c)
R `default-fill-column` (variable in buffer.c)
R `default-left-margin` (variable in buffer.c)
R `default-major-mode` (variable in buffer.c)
I `default-mode-line-format` (variable in buffer.c)
R `default-tab-width` (variable in buffer.c)
R `default-truncate-lines` (variable in buffer.c)
I `default-value` (function in data.c)
I `defconst` (function in eval.c)
I `define-abbrev` (function in abbrev.c)
I `define-abbrev-table` (function in abbrev.c)
I `define-global-abbrev` (function in abbrev.c)
I `define-key` (function in keymap.c)
I `define-mode-abbrev` (function in abbrev.c)
I `define-prefix-command` (function in keymap.c)

U `defining-kbd-macro` (variable in macros.c)
I `defmacro` (function in eval.c)
I `defun` (function in eval.c)
I `defvar` (function in eval.c)
I `delete-backward-char` (function in cmds.c)
I `delete-char` (function in cmds.c)
R `delete-exited-processes` (variable in process.c)
I `delete-file` (function in fileio.c)
I `delete-other-windows` (function in window.c)
I `delete-process` (function in process.c)
I `delete-region` (function in editfns.c)
I `delete-window` (function in window.c)
I `delete-windows-on` (function in window.c)
I `delq` (function in fns.c)
I `describe-bindings` (function in keymap.c)
I `describe-syntax` (function in syntax.c)
I `ding` (function in dispnew.c)
I `directory-file-name` (function in fileio.c)
U `directory-files` (function in dired.c)
U `disabled-command-hook` (variable in keyboard.c)
U `discard-input` (function in keyboard.c)
I `display-buffer` (function in window.c)
I `display-completion-list` (function in minibuf.c)
R `do-auto-save` (function in fileio.c)
U `documentation` (function in doc.c)
U `documentation-property` (function in doc.c)
U `downcase` (function in casefiddle.c)
U `downcase-region` (function in casefiddle.c)
U `downcase-word` (function in casefiddle.c)
U `dump-emacs` (function in emacs.c)
U `dump-emacs-data` (function in emacs.c)
U `echo-keystrokes` (variable in keyboard.c)
I `elt` (function in fns.c)
R `enable-recursive-minibuffers` (variable in minibuf.c)
U `end-kbd-macro` (function in macros.c)
I `end-of-line` (function in cmds.c)
I `enlarge-window` (function in window.c)
I `eobp` (function in editfns.c)
I `eolp` (function in editfns.c)
I `eq` (function in data.c)
I `equal` (function in fns.c)

I `erase-buffer` (function in buffer.c)
U `esc-map` (variable in keymap.c)
I `eval` (function in eval.c)
I `eval-current-buffer` (function in lread.c)
I `eval-minibuffer` (function in minibuf.c)
I `eval-region` (function in lread.c)
U `exec-directory` (variable in callproc.c)
R `exec-path` (variable in callproc.c)
U `execute-extended-command` (function in keyboard.c)
U `execute-kbd-macro` (function in macros.c)
U `executing-kbd-macro` (variable in macros.c)
U `executing-macro` (variable in macros.c)
I `exit-minibuffer` (function in minibuf.c)
U `exit-recursive-edit` (function in keyboard.c)
U `expand-abbrev` (function in abbrev.c)
I `expand-file-name` (function in fileio.c)
I `fboundp` (function in data.c)
I `featurep` (function in fns.c)
I `features` (variable in fns.c)
I `file-attributes` (function in dired.c)
I `file-directory-p` (function in fileio.c)
I `file-exists-p` (function in fileio.c)
U `file-locked-p` (function in filelock.c)
I `file-modes` (function in fileio.c)
I `file-name-absolute-p` (function in fileio.c)
I `file-name-all-completions` (function in dired.c)
I `file-name-as-directory` (function in fileio.c)
U `file-name-completion` (function in dired.c)
I `file-name-directory` (function in fileio.c)
I `file-name-nondirectory` (function in fileio.c)
I `file-newer-than-file-p` (function in fileio.c)
I `file-readable-p` (function in fileio.c)
I `file-symlink-p` (function in fileio.c)
I `file-writable-p` (function in fileio.c)
R `fill-column` (variable in buffer.c)
I `fillarray` (function in fns.c)
I `fmakunbound` (function in data.c)
I `following-char` (function in editfns.c)
I `format` (function in editfns.c)
I `forward-char` (function in cmds.c)
I `forward-line` (function in cmds.c)

I `forward-word` (function in syntax.c)
I `fset` (function in data.c)
I `funcall` (function in eval.c)
I `function` (function in eval.c)
I `fundamental-mode-abbrev-table` (variable in abbrev.c)
R `garbage-collect` (function in alloc.c)
R `gc-cons-threshold` (variable in alloc.c)
I `generate-new-buffer` (function in buffer.c)
I `get` (function in fns.c)
I `get-buffer` (function in buffer.c)
I `get-buffer-create` (function in buffer.c)
I `get-buffer-process` (function in process.c)
I `get-buffer-window` (function in window.c)
I `get-file-buffer` (function in buffer.c)
I `get-largest-window` (function in window.c)
I `get-lru-window` (function in window.c)
I `get-process` (function in process.c)
U `getenv` (function in environ.c)
I `getenv` (function in editfns.c)
I `global-abbrev-table` (variable in abbrev.c)
I `global-key-binding` (function in keymap.c)
U `global-map` (variable in keymap.c)
U `global-mode-string` (variable in xdisp.c)
I `global-set-key` (function in keymap.c)
I `global-unset-key` (function in keymap.c)
I `goto-char` (function in editfns.c)
U `help-char` (variable in keyboard.c)
R `help-form` (variable in keyboard.c)
I `identity` (function in fns.c)
I `if` (function in eval.c)
R `indent-tabs-mode` (variable in indent.c)
I `indent-to` (function in indent.c)
U `inhibit-quit` (variable in eval.c)
I `input-pending-p` (function in keyboard.c)
I `insert` (function in editfns.c)
U `insert-abbrev-table-description` (function in abbrev.c)
I `insert-before-markers` (function in editfns.c)
I `insert-buffer-substring` (function in editfns.c)
I `insert-char` (function in editfns.c)
I `insert-default-directory` (variable in fileio.c)
I `insert-file-contents` (function in fileio.c)

U `insert-string` (function in mocklisp.c)
I `int-to-string` (function in data.c)
I `integer-or-marker-p` (function in data.c)
I `integerp` (function in data.c)
I `interactive` (function in callint.c)
R `interactive-p` (function in eval.c)
I `intern` (function in lread.c)
I `intern-soft` (function in lread.c)
I `interrupt-process` (function in process.c)
U `inverse-video` (variable in dispnew.c)
I `key-binding` (function in keymap.c)
I `key-description` (function in keymap.c)
U `keyboard-translate-table` (variable in keyboard.c)
I `keymapp` (function in keymap.c)
I `kill-all-local-variables` (function in buffer.c)
I `kill-buffer` (function in buffer.c)
U `kill-emacs` (function in emacs.c)
U `kill-emacs-hook` (variable in emacs.c)
I `kill-local-variable` (function in data.c)
I `kill-process` (function in process.c)
I `last-abbrev` (variable in abbrev.c)
I `last-abbrev-location` (variable in abbrev.c)
I `last-abbrev-text` (variable in abbrev.c)
R `last-command` (variable in keyboard.c)
U `last-command-char` (variable in keyboard.c)
U `last-input-char` (variable in keyboard.c)
U `last-kbd-macro` (variable in macros.c)
R `left-margin` (variable in buffer.c)
I `length` (function in fns.c)
I `let` (function in eval.c)
I `let*` (function in eval.c)
I `list` (function in alloc.c)
I `list-buffers` (function in buffer.c)
I `list-processes` (function in process.c)
I `listp` (function in data.c)
I `load` (function in lread.c)
U `load-average` (function in fns.c)
I `load-in-progress` (variable in lread.c)
I `load-path` (variable in lread.c)
I `local-abbrev-table` (variable in abbrev.c)
I `local-key-binding` (function in keymap.c)

U `meta-prefix-char` (variable in keyboard.c)
I `min` (function in data.c)
I `minibuffer-complete` (function in minibuf.c)
I `minibuffer-complete-and-exit` (function in minibuf.c)
I `minibuffer-complete-word` (function in minibuf.c)
I `minibuffer-completion-confirm` (variable in minibuf.c)
I `minibuffer-completion-help` (function in minibuf.c)
I `minibuffer-completion-predicate` (variable in minibuf.c)
I `minibuffer-completion-table` (variable in minibuf.c)
I `minibuffer-depth` (function in minibuf.c)
R `minibuffer-help-form` (variable in minibuf.c)
I `minibuffer-local-completion-map` (variable in keymap.c)
I `minibuffer-local-map` (variable in keymap.c)
I `minibuffer-local-must-match-map` (variable in keymap.c)
I `minibuffer-local-ns-map` (variable in keymap.c)
U `minibuffer-prompt-width` (variable in window.c)
I `minibuffer-scroll-window` (variable in window.c)
I `minibuffer-window` (function in window.c)
U `ml-arg` (function in mocklisp.c)
U `ml-if` (function in mocklisp.c)
U `ml-interactive` (function in mocklisp.c)
U `ml-nargs` (function in mocklisp.c)
U `ml-prefix-argument-loop` (function in mocklisp.c)
U `ml-provide-prefix-argument` (function in mocklisp.c)
U `mocklisp-arguments` (variable in eval.c)
R `mode-line-format` (variable in buffer.c)
U `mode-line-inverse-video` (variable in xdisp.c)
R `mode-name` (variable in buffer.c)
I `modify-syntax-entry` (function in syntax.c)
I `move-to-column` (function in indent.c)
I `move-to-window-line` (function in window.c)
I `narrow-to-region` (function in editfns.c)
I `natnump` (function in data.c)
I `nconc` (function in fns.c)
R `newline` (function in cmds.c)
R `next-screen-context-lines` (variable in window.c)
I `next-window` (function in window.c)
I `nlistp` (function in data.c)
U `no-redraw-on-reenter` (variable in dispnew.c)
I `noninteractive` (variable in emacs.c)
I `nreverse` (function in fns.c)

I nth (function in fns.c)
I nthcdr (function in fns.c)
I null (function in data.c)
I obarray (variable in lread.c)
U open-dribble-file (function in keyboard.c)
U open-network-stream (function in process.c)
U open-termscript (function in dispnew.c)
I or (function in eval.c)
I other-buffer (function in buffer.c)
I other-window (function in window.c)
U overlay-arrow-position (variable in xdisp.c)
U overlay-arrow-string (variable in xdisp.c)
R overwrite-mode (variable in buffer.c)
I parse-partial-sexp (function in syntax.c)
I parse-sexp-ignore-comments (variable in syntax.c)
I point (function in editfns.c)
I point-marker (function in editfns.c)
I point-max (function in editfns.c)
I point-max-marker (function in editfns.c)
I point-min (function in editfns.c)
I point-min-marker (function in editfns.c)
U polling-period (variable in keyboard.c)
I pop-to-buffer (function in buffer.c)
R pop-up-windows (variable in window.c)
I pos-visible-in-window-p (function in window.c)
I preceding-char (function in editfns.c)
I prefix-arg (variable in callint.c)
I prefix-numeric-value (function in callint.c)
I previous-window (function in window.c)
U primitive-undo (function in undo.c)
I prin1 (function in print.c)
I prin1-to-string (function in print.c)
I princ (function in print.c)
I print (function in print.c)
I print-escape-newlines (variable in print.c)
I print-length (variable in print.c)
I process-buffer (function in process.c)
I process-command (function in process.c)
R process-connection-type (variable in process.c)
U process-environment (variable in callproc.c)
I process-exit-status (function in process.c)

I `process-filter` (function in process.c)
I `process-id` (function in process.c)
I `process-kill-without-query` (function in process.c)
I `process-list` (function in process.c)
I `process-mark` (function in process.c)
I `process-name` (function in process.c)
I `process-send-eof` (function in process.c)
I `process-send-region` (function in process.c)
I `process-send-string` (function in process.c)
I `process-sentinel` (function in process.c)
I `process-status` (function in process.c)
I `processp` (function in process.c)
I `prog1` (function in eval.c)
I `prog2` (function in eval.c)
I `progn` (function in eval.c)
I `provide` (function in fns.c)
R `pure-bytes-used` (variable in alloc.c)
R `purecopy` (function in alloc.c)
R `purify-flag` (variable in alloc.c)
I `put` (function in fns.c)
U `quit-flag` (variable in eval.c)
I `quit-process` (function in process.c)
I `quote` (function in eval.c)
R `random` (function in fns.c)
I `rassq` (function in fns.c)
I `re-search-backward` (function in search.c)
I `re-search-forward` (function in search.c)
I `read` (function in lread.c)
I `read-buffer` (function in minibuf.c)
I `read-char` (function in lread.c)
I `read-command` (function in minibuf.c)
I `read-file-name` (function in fileio.c)
I `read-from-minibuffer` (function in minibuf.c)
I `read-from-string` (function in lread.c)
U `read-key-sequence` (function in keyboard.c)
I `read-minibuffer` (function in minibuf.c)
I `read-no-blanks-input` (function in minibuf.c)
I `read-string` (function in minibuf.c)
I `read-variable` (function in minibuf.c)
I `recent-auto-save-p` (function in fileio.c)
U `recent-keys` (function in keyboard.c)

I `set-default` (function in data.c)
I `set-file-modes` (function in fileio.c)
U `set-input-mode` (function in keyboard.c)
I `set-marker` (function in marker.c)
I `set-process-buffer` (function in process.c)
I `set-process-filter` (function in process.c)
I `set-process-sentinel` (function in process.c)
U `set-screen-height` (function in dispnew.c)
U `set-screen-width` (function in dispnew.c)
I `set-syntax-table` (function in syntax.c)
I `set-window-buffer` (function in window.c)
I `set-window-configuration` (function in window.c)
U `set-window-hscroll` (function in window.c)
I `set-window-point` (function in window.c)
I `set-window-start` (function in window.c)
I `setcar` (function in data.c)
I `setcdr` (function in data.c)
U `setenv` (function in environ.c)
I `setplist` (function in data.c)
I `setq` (function in eval.c)
I `setq-default` (function in data.c)
U `shell-file-name` (variable in callproc.c)
I `shrink-window` (function in window.c)
I `signal` (function in eval.c)
I `single-key-description` (function in keymap.c)
I `sit-for` (function in dispnew.c)
U `sit-for-millisecs` (function in sunfns.c)
I `skip-chars-backward` (function in search.c)
I `skip-chars-forward` (function in search.c)
I `sleep-for` (function in dispnew.c)
U `sleep-for-millisecs` (function in sunfns.c)
I `sort` (function in fns.c)
R `split-height-threshold` (variable in window.c)
I `split-window` (function in window.c)
U `stack-trace-on-error` (variable in eval.c)
I `standard-input` (variable in lread.c)
I `standard-output` (variable in print.c)
I `standard-syntax-table` (function in syntax.c)
U `start-kbd-macro` (function in macros.c)
I `start-process` (function in process.c)
I `stop-process` (function in process.c)

I `store-match-data` (function in search.c)
I `string-equal` (function in fns.c)
I `string-lessp` (function in fns.c)
I `string-match` (function in search.c)
I `string-to-char` (function in editfns.c)
I `string-to-int` (function in data.c)
I `stringp` (function in data.c)
I `subrp` (function in data.c)
I `subst-char-in-region` (function in editfns.c)
I `substitute-command-keys` (function in doc.c)
I `substitute-in-file-name` (function in fileio.c)
I `substring` (function in fns.c)
U `sun-change-cursor-icon` (function in sunfns.c)
U `sun-get-selection` (function in sunfns.c)
U `sun-menu-internal` (function in sunfns.c)
U `sun-set-selection` (function in sunfns.c)
U `sun-window-init` (function in sunfns.c)
U `suspend-emacs` (function in keyboard.c)
I `switch-to-buffer` (function in buffer.c)
I `symbol-function` (function in data.c)
I `symbol-name` (function in data.c)
I `symbol-plist` (function in data.c)
I `symbol-value` (function in data.c)
I `symbolp` (function in data.c)
I `syntax-table` (function in syntax.c)
I `syntax-table-p` (function in syntax.c)
I `system-name` (function in editfns.c)
I `system-type` (variable in emacs.c)
R `tab-width` (variable in buffer.c)
I `temp-buffer-show-hook` (variable in window.c)
I `terpri` (function in print.c)
I `text-char-description` (function in keymap.c)
R `this-command` (variable in keyboard.c)
R `this-command-keys` (function in keyboard.c)
I `throw` (function in eval.c)
U `top-level` (variable in keyboard.c)
U `top-level` (function in keyboard.c)
R `truncate-lines` (variable in buffer.c)
U `truncate-partial-width-windows` (variable in xdisp.c)
I `try-completion` (function in minibuf.c)
U `undo-boundary` (function in undo.c)

R `undo-high-threshold` (variable in alloc.c)
R `undo-threshold` (variable in alloc.c)
U `unexpand-abbrev` (function in abbrev.c)
U `unlock-buffer` (function in filelock.c)
U `unread-command-char` (variable in keyboard.c)
I `unwind-protect` (function in eval.c)
U `upcase` (function in casefiddle.c)
U `upcase-region` (function in casefiddle.c)
U `upcase-word` (function in casefiddle.c)
U `update-display` (function in sunfns.c)
I `use-global-map` (function in keymap.c)
I `use-local-map` (function in keymap.c)
R `user-full-name` (function in editfns.c)
I `user-login-name` (function in editfns.c)
I `user-real-login-name` (function in editfns.c)
I `user-real-uid` (function in editfns.c)
I `user-uid` (function in editfns.c)
I `user-variable-p` (function in eval.c)
I `values` (variable in lread.c)
I `vconcat` (function in fns.c)
I `vector` (function in alloc.c)
I `vectorp` (function in data.c)
I `verify-visited-file-modtime` (function in fileio.c)
U `vertical-motion` (function in indent.c)
U `visible-bell` (variable in dispnew.c)
I `vms-stmlf-recfm` (variable in fileio.c)
I `waiting-for-user-input-p` (function in process.c)
I `where-is` (function in keymap.c)
I `where-is-internal` (function in keymap.c)
I `while` (function in eval.c)
I `widen` (function in editfns.c)
I `window-buffer` (function in window.c)
I `window-edges` (function in window.c)
I `window-height` (function in window.c)
I `window-hscroll` (function in window.c)
R `window-min-height` (variable in window.c)
R `window-min-width` (variable in window.c)
I `window-point` (function in window.c)
I `window-start` (function in window.c)
U `window-system` (variable in dispnew.c)
U `window-system-version` (variable in dispnew.c)

U `x-proc-mouse-event` (function in x11fns.c)
U `x-proc-mouse-event` (function in xfns.c)
U `x-rebind-key` (function in xfns.c)
U `x-rebind-key` (function in x11fns.c)
U `x-rebind-keys` (function in xfns.c)
U `x-rebind-keys` (function in x11fns.c)
U `x-rubber-band` (function in xfns.c)
U `x-set-background-color` (function in xfns.c)
U `x-set-background-color` (function in x11fns.c)
U `x-set-bell` (function in xfns.c)
U `x-set-bell` (function in x11fns.c)
U `x-set-border-color` (function in x11fns.c)
U `x-set-border-color` (function in xfns.c)
U `x-set-border-width` (function in x11fns.c)
U `x-set-border-width` (function in xfns.c)
U `x-set-cursor-color` (function in xfns.c)
U `x-set-cursor-color` (function in x11fns.c)
U `x-set-font` (function in xfns.c)
U `x-set-font` (function in x11fns.c)
U `x-set-foreground-color` (function in x11fns.c)
U `x-set-foreground-color` (function in xfns.c)
U `x-set-icon` (function in xfns.c)
U `x-set-internal-border-width` (function in xfns.c)
U `x-set-internal-border-width` (function in x11fns.c)
U `x-set-keyboard-enable` (function in xfns.c)
U `x-set-mouse-color` (function in x11fns.c)
U `x-set-mouse-color` (function in xfns.c)
U `x-set-mouse-inform-flag` (function in xfns.c)
U `x-set-window-edges` (function in xfns.c)
U `x-store-cut-buffer` (function in x11fns.c)
U `x-store-cut-buffer` (function in xfns.c)
I `y-or-n-p` (function in fns.c)
I `yes-or-no-p` (function in fns.c)
I `zerop` (function in data.c)

A.0.7 Documentation

This section lists the documentation strings of all Emacs Lisp primitives which were implemented, but whose behavior is not completely emulated. The documentation strings include notes explaining any restrictions or other unusual behaviors.

```
this-command

The command now being executed.
The command can set this variable; whatever is put here
will be in  last-command  during the following command.

NOTE: In Edwin, this-command is sometimes an Edwin command object.


last-command

The last command executed.  Normally a symbol with a function definition,
but can be whatever was found in the keymap, or whatever the variable
`this-command' was set to by that command.

NOTE: In Edwin, last-command is often an Edwin command object.


this-command-keys

Return string of the keystrokes that invoked this command.

NOTE: Commands invoked by function keys will not get the usual
terminal-specific escape sequences.


help-form

Form to execute when character help-char is read.
If the form returns a string, that string is displayed.
If help-form is nil, the help char is not recognized.

NOTE: This variable is not supported by Edwin.
```

```
exec-path

*List of directories to search programs to run in subprocesses.
Each element is a string (directory name) or nil (try default directory).

NOTE: In Edwin, each element is a pathname or false.  The get/set-value
methods of exec-path will translate from/to the Edwin exec-path.  The
exec-path should not be side-effected without re-setting the symbol value
afterwards.


purecopy

Make a copy of OBJECT in pure storage.
Recursively copies contents of vectors and cons cells.
Does not copy symbols.

NOTE: In Edwin, this just does a deep copy of lists, strings, and vectors.


garbage-collect

Reclaim storage for Lisp objects no longer needed.
Returns info on amount of space in use:
 ((USED-CONSES . FREE-CONSES) (USED-SYMS . FREE-SYMS)
  (USED-MARKERS . FREE-MARKERS) USED-STRING-CHARS USED-VECTOR-SLOTS)
Garbage collection happens automatically if you cons more than
gc-cons-threshold  bytes of Lisp data since previous garbage collection.

NOTE: In Edwin, returns the number of free words in the heap.  This
number is incompatible with the expected association list, so Emacs
programs examining the return value will signal an error.


gc-cons-threshold

*Number of bytes of consing between garbage collections.

NOTE: This variable is meaningless in Edwin.
```

pure-bytes-used

Number of bytes of sharable Lisp data allocated so far.

NOTE: This variable is meaningless in Edwin.

purify-flag

Non-nil means loading Lisp code in order to dump an executable.

NOTE: This variable is meaningless in Edwin.

undo-threshold

Keep no more undo information once it exceeds this size.
This threshold is applied when garbage collection happens.
The size is counted as the number of bytes occupied,
which includes both saved text and other data.

NOTE: This variable cannot be set in Edwin.

undo-high-threshold

Don't keep more than this much size of undo information.
A command which pushes past this size is itself forgotten.
This threshold is applied when garbage collection happens.
The size is counted as the number of bytes occupied,
which includes both saved text and other data.

NOTE: This variable cannot be set in Edwin.

default-ctl-arrow

Default ctl-arrow for buffers that do not override it.
This is the same as (default-value 'ctl-arrow).

NOTE: This variable can only be t in Edwin.

```
default-truncate-lines

Default truncate-lines for buffers that do not override it.
This is the same as (default-value 'truncate-lines).

NOTE: This variable can only be a boolean in Edwin.


default-fill-column

Default fill-column for buffers that do not override it.
This is the same as (default-value 'fill-column).

NOTE: This variable can only be an exact nonnegative integer in Edwin.


default-left-margin

Default left-margin for buffers that do not override it.
This is the same as (default-value 'left-margin).

NOTE: This variable can only be an exact nonnegative integer in Edwin.


default-tab-width

Default tab-width for buffers that do not override it.
This is the same as (default-value 'tab-width).

NOTE: This variable can only be an exact nonnegative integer in Edwin.


default-case-fold-search

Default case-fold-search for buffers that do not override it.
This is the same as (default-value 'case-fold-search).

NOTE: This variable can only be a boolean in Edwin.
```

mode-line-format

```
Template for displaying mode line for current buffer.
Each buffer has its own value of this variable.
Value may be a string, a symbol or a list or cons cell.
For a symbol, its value is used (but it is ignored if t or nil).
 A string appearing directly as the value of a symbol is processed verbatim
 in that the %-constructs below are not recognized.
For a list whose car is a symbol, the symbol's value is taken,
 and if that is non-nil, the cadr of the list is processed recursively.
 Otherwise, the caddr of the list (if there is one) is processed.
For a list whose car is a string or list, each element is processed
 recursively and the results are effectively concatenated.
For a list whose car is an integer, the cdr of the list is processed
  and padded (if the number is positive) or truncated (if negative)
  to the width specified by that number.
A string is printed verbatim in the mode line except for %-constructs:
  (%-constructs are allowed when the string is the entire mode-line-format
   or when it is found in a cons-cell or a list)
  %b -- print buffer name.      %f -- print visited file name.
  %* -- print *, % or hyphen.   %m -- print value of mode-name (obsolete).
  %s -- print process status.   %M -- print value of global-mode-string. (obs)
  %p -- print percent of buffer above top of window, or top, bot or all.
  %n -- print Narrow if appropriate.
  %[ -- print one [ for each recursive editing level.  %] similar.
  %% -- print %.   %- -- print infinitely many dashes.
Decimal digits after the % specify field width to which to pad.
```

NOTE: The set-value method for mode-line-format sets the Edwin variable mode-line-format to a _copy_ of the new value. Thus, you can't modify the buffer's mode-line by side-effecting the new value. Also, the Emacs symbols in the new value are replaced with Edwin variables. Setting the Emacs symbols to new values will cause the Edwin variables to be updated, but the new values cannot contain Emacs symbols. Edwin variables won't be substituted for the symbols and Edwin will signal an error.

default-major-mode

```
*Major mode for new buffers.  Defaults to fundamental-mode.
nil here means use current buffer's major mode.

NOTE: This variable can only be 'fundamental-mode in Edwin.


major-mode

Symbol for current buffer's major mode.

NOTE: This variable can only be a symbol in Edwin.


abbrev-mode

Non-nil turns on automatic expansion of abbrevs when inserted.
Automatically becomes local when set in any fashion.

NOTE: This variable can only be nil in Edwin.


case-fold-search

*Non-nil if searches should ignore case.
Automatically becomes local when set in any fashion.

NOTE: This variable can only be a boolean in Edwin.


mode-name

Pretty name of current buffer's major mode (a string).

NOTE: This variable can only be a string in Edwin.


fill-column

*Column beyond which automatic line-wrapping should happen.
Automatically becomes local when set in any fashion.
```

NOTE: This variable can only be an exact nonnegative integer in Edwin.

left-margin

*Column for the default indent-line-function to indent to.
Linefeed indents to this column in Fundamental mode.
Automatically becomes local when set in any fashion.

NOTE: This variable can only be an exact nonnegative integer in Edwin.

tab-width

*Distance between tab stops (for display of tab characters), in columns.
Automatically becomes local when set in any fashion.

NOTE: This variable can only be an exact nonnegative integer in Edwin.

ctl-arrow

*Non-nil means display control chars with uparrow.
Nil means use backslash and octal digits.
Automatically becomes local when set in any fashion.

NOTE: This variable can only be t in Edwin.

truncate-lines

*Non-nil means do not display continuation lines;
give each line of text one screen line.
Automatically becomes local when set in any fashion.

Note that this is overridden by the variable
truncate-partial-width-windows if that variable is non-nil
and this buffer is not full-screen width.

NOTE: This variable can only be a boolean in Edwin.

```
default-directory

Name of default directory of current buffer.  Should end with slash.

NOTE: This variable can only be a string in Edwin.


auto-fill-hook

Function called (if non-nil) after self-inserting a space at column
beyond fill-column

NOTE: This variable can only be nil in Edwin.


buffer-file-name

Name of file visited in current buffer, or nil if not visiting a file.

NOTE: This variable can only be a string or nil in Edwin.


buffer-auto-save-file-name

Name of file for auto-saving current buffer,
or nil if buffer should not be auto-saved.

NOTE: This variable can only be a string or nil in Edwin.


buffer-read-only

Non-nil if this buffer is read-only.

NOTE: This variable will only evaluate to a boolean in Edwin.


buffer-backed-up

Non-nil if this buffer's file has been backed up.
```

Backing up is done before the first time the file is saved.

NOTE: This variable can only be a boolean in Edwin.

selective-display

t enables selective display:
 after a ^M, all the rest of the line is invisible.
 ^M's in the file are written into files as newlines.
Integer n as value means display only lines
 that start with less than n columns of space.
Automatically becomes local when set in any fashion.

NOTE: This variable can only be nil in Edwin.

selective-display-ellipses

t means display ... on previous line when a line is invisible.
Automatically becomes local when set in any fashion.

NOTE: This variable can only by nil in Edwin.

overwrite-mode

Non-nil if self-insertion should replace existing text.
Automatically becomes local when set in any fashion.

NOTE: This variable can only be nil in Edwin.

buffer-undo-list

List of undo entries in current buffer.

NOTE: This variable is not supported by Edwin.

newline

```
Insert a newline.  With arg, insert that many newlines.
In Auto Fill mode, can break the preceding line if no numeric arg.

NOTE: Doesn't do anything special in Auto Fill mode in Edwin.
```

```
blink-paren-hook
```

```
Function called, if non-nil, whenever a char with closeparen syntax is self-
inserted.

NOTE: This variable is not supported in Edwin.
```

```
user-full-name
```

```
Return the full name of the user logged in, as a string.

NOTE: In Edwin, this is the current login name as given in utmp, NOT
the pw_gecos field from the /etc/passwd entry.
```

```
interactive-p
```

```
Return t if function in which this appears was called interactively.
This means that the function was called with call-interactively (which
includes being called as the binding of a key)
and input is currently coming from the keyboard (not in keyboard macro).

NOTE: This function always returns t in Edwin.
```

```
do-auto-save
```

```
Auto-save all buffers that need it.
This is all buffers that have auto-saving enabled
and are changed since last auto-saved.
Auto-saving writes the buffer into a file
so that your editing is not lost if the system crashes.
This file is not the file you visited; that changes only when you save.
```

Non-nil argument means do not print any message if successful.

NOTE: The nomsg argument is not supported by Edwin.

random

Return a pseudo-random number.
On most systems all integers representable in Lisp are equally likely.
  This is 24 bits' worth.
On some systems, absolute value of result never exceeds 2 to the 14.
If optional argument is supplied as t,
 the random number seed is set based on the current time and pid.

NOTE: The random number seed is set based on the current real and
process times only.

indent-tabs-mode

*Indentation can insert tabs if this is non-nil.
Setting this variable automatically makes it local to the current buffer.

NOTE: This variable can only be a boolean in Edwin.

make-keymap

Construct and return a new keymap, a vector of length 128.
All entries in it are nil, meaning "command undefined".

NOTE: Edwin requires that this be a comtab.

make-sparse-keymap

Construct and return a new sparse-keymap list.
Its car is 'keymap and its cdr is an alist of (CHAR . DEFINITION).
Initially the alist is nil.

NOTE: Edwin requires that this be a comtab.

esc-map

Default keymap for ESC (meta) commands.
The normal global definition of the character ESC indirects to this keymap.

NOTE: This variable is not supported by Edwin.

ctl-x-map

Default keymap for C-x commands.
The normal global definition of the character C-x indirects to this keymap.

NOTE: This variable can only be a comtab in Edwin.

completion-auto-help

*Non-nil means automatically provide help for invalid completion input.

NOTE: This variable can only be a boolean in Edwin.

enable-recursive-minibuffers

*Non-nil means to allow minibuffers to invoke commands which use
recursive minibuffers.

NOTE: This variable can only be a boolean in Edwin.

minibuffer-help-form

Value that help-form takes on inside the minibuffer.

NOTE: help-form is not supported by Edwin.

delete-exited-processes

*Non-nil means delete processes immediately when they exit.
nil means don't delete them until 'list-processes' is run.

NOTE: This variable can only be a boolean in Edwin.

process-connection-type

Control type of device used to communicate with subprocesses.
Values are nil to use a pipe, t for a pty (or pipe if ptys not supported).
Value takes effect when 'start-process' is called.

NOTE: This variable can only be a boolean in Edwin.

pop-up-windows

*Non-nil means display-buffer should make new windows.

NOTE: This variable can only be a boolean in Edwin.

next-screen-context-lines

*Number of lines of continuity when scrolling by screenfuls.

NOTE: This variable can only be an exact nonnegative integer in Edwin.

split-height-threshold

*display-buffer would prefer to split the largest window if this large.
If there is only one window, it is split regardless of this value.

NOTE: This variable can only be an exact nonnegative integer in Edwin.

window-min-height

```
*Delete any window less than this tall (including its mode line).

NOTE: This variable can only be an exact integer greater than 1 in Edwin.


window-min-width

*Delete any window less than this wide.

NOTE: This variable can only be an exact integer greater than 2 in Edwin.
```

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] William Clinger (*editor*). Revised^4 report on the algorithmic language Scheme. Technical Report AI Memo 848b, Mass. Inst. of Technology, Artificial Intelligence Laboratory, 1992. Forthcoming. Simultaneous publication expected with Lisp Pointers and University of Oregon.

[3] Chris Hanson. *MIT Scheme User's Manual, for Scheme Release 7.1.* Mass. Inst. of Technology, Cambridge, MA, 0.9 edition, 1991. DRAFT.

[4] Chris Hanson, the MIT Scheme Team, and a cast of thousands. *MIT Scheme Reference Manual, for Scheme Release 7.1.3.* Mass. Inst. of Technology, Cambridge, MA, 1.1 edition, November 1991.

[5] Bil Lewis, Dan LaLiberte, and the GNU Manual Group. *GNU Emacs Lisp Reference Manual.* Free Software Foundation, Cambridge, MA, first edition, March 1990.

[6] David Notkin and William G. Griswold. Enhancement through extension: The extension interpreter. *Proceedings of the SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques (ACM SIGPLAN Notices)*, 22(7), July 1987.

[7] David Notkin, Norman Hutchinson, Jan Sanislo, and Michael Schwartz. Heterogeneous computing environments: Report on the acm sigops workshop on accomodating heterogeneity. *Communications of the ACM*, 30(2):132–140, February 1987.

[8] James M. Purtilo. The polylith software bus. Technical Report UMIACS–TR–90–65, University of Maryland, Institute for Advanced Computer Studies, May 1990.

[9] Richard Stallman. *GNU Emacs Manual, Version 18.* Free Software Foundation, Cambridge, MA, sixth edition, March 1987.