JEmacs: The Java/Scheme-based Emacs

Per Bothner
`<per@bothner.com>`

Abstract

JEmacs is a re-implementation of the Emacs programmable text editor. It is written in Java, and uses the Swing GUI toolkit. Emacs is based on the extension language Emacs Lisp (Elisp), which is a dynamically-scoped member of the Lisp family. JEmacs supports Elisp, as well as the use of Scheme, a more modern statically-scoped Lisp dialect. Both languages get compiled to Java bytecodes, either in advance or on-the-fly, using the Kawa compilation framework.

1 Introduction

Emacs [7] [5] (in various versions) is a popular programmer's text editor. Emacs is programmable using "Emacs Lisp" (Elisp) [6] and many powerful packages are written in Elisp. The Free Software Foundation has a goal to replace Elisp with Scheme while also providing a translator to convert old Elisp files to Scheme. One reason is that Elisp is an ad-hoc, non-standard Lisp variant not used anywhere else, and not consistent with modern programming-language ideas. Another reason is that Guile, the primary GNU dialect of Scheme, is intended to be the standard extension language for GNU software, and so it makes sense for Emacs (the main GNU application with extensive use of a scripting language) to follow suit.

My opinion is that Guile is not the best Scheme implementation to use for Emacs. I happen to be biased, as I am the author of Kawa, a Java-based Scheme implementation. I also think that just replacing the extension language may not go far enough, and perhaps it is time to also replace the low-level code written in C. (One of the XEmacs maintainers told me he would really like to replace the re-display engine of XEmacs.)

Therefore, I have been working on a "next-generation" Emacs, based on Java and Kawa. The design includes:

- An implementation of the Scheme language.
- An implementation of the Elisp (core) syntax and language, such as functions used for creating lists and strings, defining functions, and macros.
- A set of Java classes based on the Swing GUI api that implement the Emacs "types", such as `Buffer`, `Keymap`, `Window`, `Marker`.
- A set of Scheme bindings to the Java methods. These are "similar to" and have the same names as standard Emacs Lisp functions, but are written in Scheme and intended to be called from Scheme.
- The equivalent Elisp functions: Implementations of the high-level Emacs functions as Elisp functions, so existing Elisp applications can (mostly) run without change.

The totality of these features is what I call "JEmacs". Below is a screenshot showing some of what has already been implemented. It includes a frame with a menubar, split into multiple windows, each with a mode line. The top window is a "Scheme interaction window", where Scheme expressions can be typed, and the result displayed. (Notice the user input is automatically boldfaced.) The user has just typed `C-x C-f`, which is bound to the function `find-file`. When `find-file` is called interactively or with no arguments, the `read-dialog` is called, which pops up the simple dialog window we see.

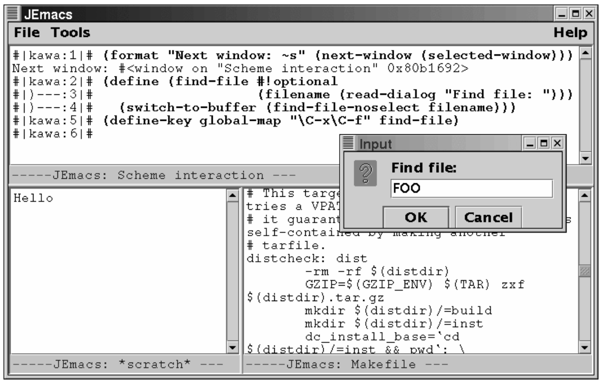

Figure 1: JEmacs in action

2 Motivation

This is a major, perhaps foolhardy, undertaking. Here are some reasons why it might make sense; I expand on these later.

- Swing is a modern GUI toolkit with good support for major Emacs concepts.
- Building on a Java run-time means we benefit from the work being done to run Java bytecodes fast.
- Java is multi-threaded.
- Kawa is a modern object-oriented Scheme, while Emacs is based on rather old design ideas.
- Java is based on Unicode and has good internationalization support.
- Java has lots of neat packages we can use.
- It would be useful to have Scheme (and Elisp) scripting for Swing applications.
- It is a good way to learn Swing!

3 Kawa

Kawa (see [2], [3]) is an implementation of the Scheme programming language written in Java. I have been developing Kawa since 1996. Unlike other such implementations, Kawa compiles Scheme into Java bytecodes, with non-trivial optimizations. It also provides almost all the other features you expect from a production Scheme system (including `eval` and `load`) and convenient interaction between Scheme and Java.

Kawa is also a framework used to implement Scheme, and which can be used for other languages. The package `gnu.bytecode` provides classes to generate Java `.class` bytecode files, including methods to generate the Java virtual machine instructions. It also lets you read, print, and otherwise manipulate the Java `.class` file format.

At a higher level, the `gnu.expr` package works on `Expression` objects. This is basically an abstract syntax tree (AST) representation, and the package has classes to generate and optimize expressions and declarations. It uses the `gnu.bytecode` package to generate bytecodes from the `Expression` representation.

The Kawa framework was originally used to compile Scheme. However, I wrote the beginnings of an EcmaScript (JavaScript) implementation, and others are using Kawa to compile other languages. For the JEmacs project, the framework is being used to compile Emacs Lisp to Java bytecodes, replacing the Emacs byte-compiler: Instead of `.elc` files loaded into the Emacs bytecode interpreter, we use Java bytecode (`.class`) files loaded into a Java Virtual Machine.

4 Performance

A primary advantage of JEmacs is that Kawa is potentially much faster than either Elisp or Guile. Using an optimizing compiler that compiles to bytecode is certainly going to be faster than Guile or Emacs's simple interpreter. The Emacs bytecode-compiler uses the same idea, and produces a bytecode format that is more suitable to Emacs than Java bytecodes. However, there are many projects and companies working very hard on running Java bytecodes fast. The common approach is to use a "Just-in-Time compiler" (JIT), which dynamically compiles a bytecode method into native code inside the runtime. Another approach is to use a traditional "ahead-of-time" compiler (such as the Gcc-based Gcj [1]). It thus seems plausible (though unproven) that JEmacs can achieve better performance than Emacs.

5 The Swing Toolkit

Sun introduced Swing [4] in 1998 as the "next-generation" GUI toolkit for Java. Swing has a lot of functionality and many useful features. It builds on the earlier AWT (Abstract Windowing Toolkit). Of particular interest is that the text support in Swing is both very powerful, and also seems to be inspired by Emacs ideas. Swing has new "widgets" based on separating the "model" (data) and "view+control" (look+feel). For example, Swing distinguishes between a `Document` versus a `JTextComponent` that displays the `Document`, which is essentially the same as the Emacs buffer versus window distinction.

Swing also has a `Keymap` class similar to that of Emacs, and a `Position` that is like an Emacs marker. Unfortunately, neither of these are quite right for Emacs, but it was not difficult to create new classes that implement the Swing interfaces.

Swing has some other nice features, such as "pluggablelook-and-feel" (themeability), a number of flexible "widgets", and support for "structured" documents (i.e. XML/HTML structure in a buffer).

One problem with Swing is that while it is portable and freely redistributable, it does not have a free license, and there are no free re-implementations so far. (A related issue is that the documentation of Swing is very poor.) I'm hoping that by the time JEmacs becomes useable a free re-implementation of (the needed subset of) Swing will be available, and perhaps JEmacs will encourage this to happen. If not, we may re-write JEmacs so it can be built on top of some other free library (such as Gtk/Gnome or Qt).

Another possible problem with Swing is performance. Swing has the reputation for being slow. The first Emacs package ported to JEmacs (Towers-of-Hanoi animation) does run slower than under XEmacs. The cause of this is not clear, but it is quite possible it is due to Swing overheads. Perhaps in practice this may not matter, but it is certainly a possibility we may replace the use of Swing with a faster toolkit. More likely is replacing some of the implementation classes by alternative implementations; Swing is very flexible in this respect, because the API is defined in terms of abstract interfaces, rather than specific classes. (Some of these new implementation classes may also be useful in implementing a free Swing replacement.)

6 Multi-threading

One problem with traditional Emacs is that it is single-threaded. If you start some non-trivial operation (such as getting new mail), your Emacs session will be frozen until the operation completes. Java is designed to be multi-threaded, so it is in theory straightforward to create a multi-threaded Emacs.

One complication is that the Emacs Lisp execution model is inherently single threaded, since any Elisp function can change the current buffer or window to another buffer or window, while in the middle of the function. This means we cannot associate a thread with each buffer or with each window.

A solution to this problem is to use "buffer groups", that is a group of related buffers and their windows which run in the same thread. Typically, there would be one buffer group for each Elisp "application". By default, when an Elisp function creates a new buffer, it is put in the same buffer group as the the current buffer. However, an Elisp function such as `find-file` can create a new buffer group when it creates a new buffer.

Another problem is that Swing is single-threaded. Only one thread (the event thread) can safely modify a buffer that is visible in a window. In the current JEmacs implementation all interactive commands are run by the event thread. Thus effectively, all of JEmacs is running inside the event thread. A solution is for long-running commands to use a "worker" thread. When the worker thread is finished, it lets the event thread know it is done, which can then update the buffers and display.

7 Java classes for Emacs

The folllowing Java classes implement what we might call the Emacs data "model".

- `Buffer`: An Emacs buffer. Contains a Swing `StyledDocument` object that manages the actual text (and styles). Contains a `BufferKeymap`, which manages the actions executed for different keystrokes.
- `BufferContent`: The actual characters of the `Buffer`. Implements the Swing `Content` interface. This class is needed because standard Swing does not support the `Marker` semantics we need.
- `Marker`: A position in a buffer that gets adjusted as needed. Similar to the Swing `Position` class, but also knows the `Buffer` it points to.

The following Java classes implement what we might call the Emacs "view+controller".

- `BufferKeymap`: A data structure in one-to-one association with a `Buffer`. It implements the Swing `Keymap` interface, and manages the primitive `Keymaps`, to give the correct Emacs functionality.
- `Window`: Extends the Swing `JtextPane` class. Includes an assocated `Modeline`, and a scrollbar.
- `Frame`: A top-level window. A `Frame` contains a nested hierarchy of `Windows`, sub-divided using Swing's `JSplitPane`.

8 Editing procedures

JEmacs includes a number of Scheme procedures for operating on the Java classes just mentioned. The Scheme API is designed to be similar to the traditional Elisp functions, but put in Scheme form. Here is the JEmacs definition of the standard Emacs function `beginning-of-line`, written in Scheme.

```
(define (beginning-of-line
         #!optional
         (n  :: <int> 1)
         (buffer :: <buffer>
           (current-buffer)))
  (invoke buffer 'setPoint
    (point-at-bol n buffer)))

(define-key global-map "\C-a"
            beginning-of-line)
```

Note the optional type declarations for the two parameters. Also note the `invoke` “function”. This calls the specified method (in this case `setPoint`) on the specified object (in this case the `buffer`). In a case like this where the receiver class is known, the Kawa compiler can directly generate a `invokevirtual` bytecode instruction.

Once we have defined `beginning-of-line`, it can be used from either Elisp or Scheme code. This makes it easier to mix Scheme and Elisp, convert (if desired) Elisp to Scheme, and lets us re-use Elisp documentation.

Our goal is to be able to run most Elisp package unmodified. Some packages may require minor changes, but no more than say porting to XEmacs. The first Emacs package that runs under JEmacs without requiring a single modification to the Elisp source is `hanoi.el`, an animation of the Towers-of-Hanoi puzzle. However, it may be desirable to rewrite some packages, possibly in Scheme or Java. For example, `dired` needs a more modern interface.

The Scheme and Elisp APIs implemented in JEmacs are based on those of both GNU Emacs and XEmacs. JEmacs will not implement either API exactly, but will try to implement the features that make most sense. For example, GNU Emacs and XEmacs have very different APIs for manipulating the menubar. In this case, JEmacs implemented a menubar API based on the XEmacs API, partly because it was closer to the Swing menubar model.

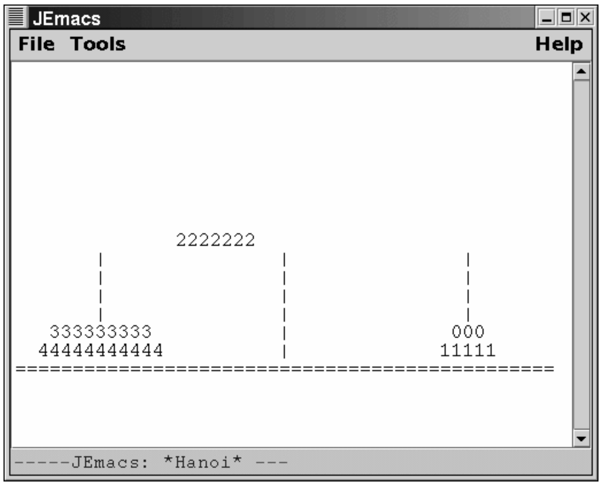

Figure 2: Towers of Hanoi

9 Action

The basic Java event model is of event listeners being registered with objects that generate events. On top of that, Swing has an API especially convenient for keyboard events: You can associate a `Keymap` with each text component, where the `Keymap` maps a `KeyStroke` (high-level keyboard input events) to an `Action`, which is then gets executed. Emacs is similar, except looking up something in a keymap yields a “keymap entry”, of which there are many kinds. So what JEmacs does is to “wrap” the Emacs-style keymap entries using special subclasses of `Action`. For example, looking up a prefix key in Emacs returns another keymap; in JEmacs it returns a `PrefixAction`. Performing the `PrefixAction` modifies the current `BufferKeymap` state so that when the next keystroke appears it will lookup a “key sequence” that is the concatenation of the remembered prefix key(s) and the new keystroke. One slight difference from standard Emacs: JEmacs remembers previous prefix keys on a per-`Buffer` basis, so if you switch to a different buffer with the mouse, the old prefix key is remembered until you switch back.

10 Some issues in implementing Elisp

There are some tricky issues if you want to implement Elisp, especially if you want nice interoperability with Scheme. (The plan for GNU Emacs is to translate Elisp into Scheme, which raises similar issues.)

10.1 Syntax

While both Scheme and Elisp share the fully parenthesized prefix notation common to the Lisp family, there are some differences in the syntax of literals and identifiers. For example, the character `'a'` is written `#\a` in Scheme, and `?a` in Elisp. The part of a Lisp system that converts a stream of characters to a value (usually a linked list) is traditionally called the "reader". We needed to write an Elisp reader to go along with the Scheme reader, and make sure the right one is invoked. This is fairly straightforward, and (except for some obscure features) done.

Once the reader has converted an input line or file to a list, the list needs to be converted input Kawa's internal `Expression` (abstract syntax tree) representation. This is handled similar for Elisp and Scheme. However, Elisp has some new syntax forms (such as `defun`, `save-excursion`) and and some forms that are different than in Scheme (such as `lambda`). For that we need to write new syntax transformers. This is almost done.

10.2 Symbols

The symbol data type in Scheme is very simple: It is an immutable atomic string; you can create a symbol from a (mutable) string, and you can convert the symbol back to a string (for example for printing). Whenever you convert a string to a symbol, you will always get the same identical symbol, as long as the strings have the same characters. This process is called *interning* and is implemented using a global hash-table. Symbols are used for multiple purposes, but the most important one is that identifiers in a Scheme program are represented using symbols.

Java has a similar datatype, the class `String`, which is used all over the place in Java. Java has a method, called `intern`, which returns an interned version of the `String`. This functionality is exactly what is needed for Scheme, so Kawa uses `String` for Scheme symbols. This has the side benefit of increasing interoperabilty between Scheme and Java.

On the other hand, an Elisp symbol has extra properties: value and function bindings, and a property list. The traditional implementation is that a symbol value is a pointer to a structure containing the necessary fields. This makes extracting the value and function bindings cheap, but it requires extra space in all symbols. JEmacs uses an alternative implementation: an Elisp symbol value is a reference to a `String` instance, just as in Kawa. To get the value or function binding of a symbol, you lookup the symbol in the current `Environment`. This yields a `Binding`, which contains the needed fields. For symbols that are used as identifiers in a function, the compiler generates code to get the `Binding` when the function is loaded. Since we don't have to do a hashtable lookup when the function is executed, symbol lookup is about as fast as in the traditional implementation.

10.3 Nil - the empty list

In Elisp, the empty list and the symbol `'nil` are the same object, but in Scheme they are different. There are various ways to deal with the problem, none particularly elegant. In Kawa, lists inherit from the abstract `Sequence` class. I feel it is important that the empty list also be a `Sequence`, even for Elisp, and it is important to be able to pass lists between Scheme and Elisp code. I decided that `'nil` would have to be a special case: While the Elisp symbol `t` is represented using the `String "t"`, the symbol `nil` is represented by the special `LList` value `LList.Empty` that Kawa uses for empty lists. Thus the predicate `(symbolp x)` is implemented as `(x == List.Empty || x instanceof java.lang.String)`.

10.4 Standard Elisp Functions

Elisp has many builtin functions and macros which are different from Scheme. There is no fundamental difficulty with this; just a lot of porting/conversion work. Many of the basic editing functions are already implemented, but many (such as those involving searching) are not.

11 Variables

Variable lookup is different in Scheme and Elisp in two main ways: Elisp uses dynamic scoping, while Scheme uses lexical scoping; and Elisp has different namespaces for function names and variables names, while Scheme has a single namespace for both. The latter is an easy matter of the compiler emitting the code to look for the name in the correct namespace. Handling dynamic scoping is done using Kawa's support for the `fluid-name` form, which provides dynamic binding using a very flexible name-binding mechanism.

11.1 Constrained Variables

In Kawa, each global variable is a `Binding` object. A `Binding` has an optional name, a value field, and a `Constraint`. The constraint contains the actual methods that get/set the value of the `Binding`. For example, setting `b` to `x` does `b.constraint.set(b, x)`.

The default action for `get` retrieves the `Binding` value field. Different sub-classes of `Constraint` have different implementations of `get` and `set`. If there is a thread-local dynamic or buffer-local binding, we just put the appropriate constraint in the binding.

This framework can handle indirection, unbound variables (`get` throws an exception), and constraint propagation. Changing a value can trigger arbitrary checks or notification messages.

12 Modes

In Emacs, a mode is a set of keybindings, functions, and variables local to a buffer. There is no object corresponding to a "mode", but there is a set of conventions to follow. In an object-oriented environment it seems better to define a separate *mode class* for each mode. Each buffer that has a mode enabled should have a corresponding *mode instance*.

Each buffer has a linked list of mode instances, one for each major/minor mode that is active for the buffer. Mode functions are compiled to virtual methods of the mode object. Instead of a buffer-local variable, use a field of the mode object. This provides fast access to variables in compiled code, without run-time symbol lookup. A derived mode can use mode class inheritance.

As an example, the abstract class `ProcessMode` inherits from the generic `Mode` class. A `ProcessMode` represents some kind of *process* which (usually) generates output that gets inserted into the buffer. Partial implementations exist of the sub-classes `InfProcessMode`, which displays the output of an external (possibly-interactive) program, and `TelnetMode`. Both of these provide minimal terminal emulation. A full terminal emulator would be desirable, though line wrapping is a complication. (Swing handles line wrapping on word boundaries, as expected by people used to word processor, but a normal terminal emulator wraps on character boundaries. The best solution is probably to write a custom `View` class.)

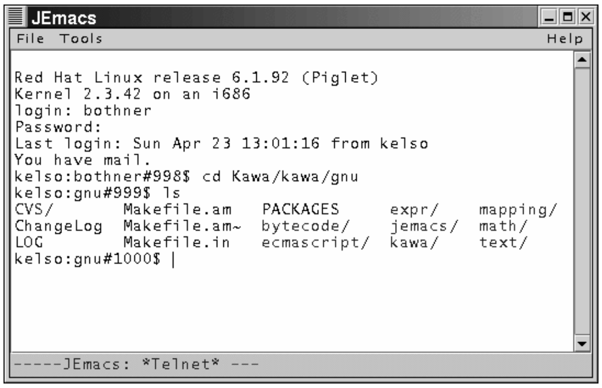

Figure 3: Telnet mode

The `Mode` framework has been used for modes written directly in Java. It is not yet clear how to write a mode using Scheme, nor whether legacy Elisp code can be automatically compiled into classes that inherit from `Mode`.

13 Unicode and Internationalization

The Java `char` is a 16-bit Unicode character. Internal strings and JEmacs buffers use these Unicode `chars`. On the other hand, external files consist of 8-bit bytes. So Java provides named encodings that map byte streams and character streams.

Java with Swing handles much of the work needed for complex text processing. For example, bi-directional text (as needed for Hebrew and Arabic) is taken care of. In the screenshot below, we have a file encoded in UTF-8. It contains a string of four Hebrew characters, stored in the buffer in logical (reading) order. When they displayed, they are shown right-to-left. Notice that if you make a text selection that includes both English and Hebrew text, the selected characters form a contiguous region in the memory, but because different segments are displayed in different order, in the window we see the selection as two non-contiguous pieces.

Also note that characters not present in the font are displayed as hollow rectangles. I.e. I need to install a more complete font!

Current releases of Emacs and XEmacs support some internationalization, using the Mule framework which supports text in many character sets. However, current releases of Mule do not yet support Unicode, which is where most of the world is heading. Mule is based on

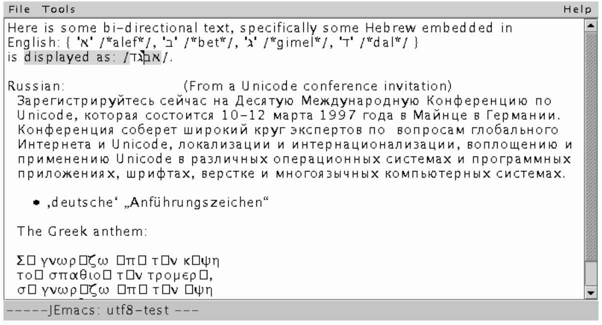

Figure 4: Multiple scripts

some rather dated design decisions. JEmacs will not support Mule; it won't need to.

14 Word Processing and XML

Documents will be increasingly represented using XML externally and DOM (Document Object Model) internally. Most of the programs and libraries for manipulating and formatting XML are written in Java. For example, the Apache group has some major Java-based XML projects.

If Emacs is to support word processing features, it should build on XML standards. It is easier to use third-party Java libraries if the Emacs core is Java-based.

The plan is for JEmacs to implement a class that implements the XEmacs "extent" API, and also implements the Swing `Element` interface, as well as the DOM `Node` interface.

JEmacs will use some form of "virtual document" that implements the Swing `Document` interface, but gets its content indirectly from other documents. The class `AbstractString` is an abstract class that generalizes strings, buffers, shared substrings, buffer regions, and general indirection. This is part of a design to support editable documents which are defined using transformations from other documents (rather like a database "view").

15 Status and Conclusion

A "proof-of-concept" prototype is working, including partial Elisp implementation. There is still a lot of work before you want to use JEmacs for day-to-day general editing.

Using the libraries of Java and Swing takes care of many problems.

JEmacs has a mailing list and a home page (http://JEmacs.SourceForge.net/).

JEmacs is currently distributed together with Kawa (http://www.gnu.org/software/kawa/).

References

[1] Per Bothner. *A Gcc-based Java Implementation*. IEEE Compcon '97, 1997.

[2] Per Bothner. *Kawa - Compiling Dynamic Languages to the Java VM*. Usenix Annual Technical Conference, 1998.

[3] Per Bothner. *Kawa: Compiling Scheme to Java*. Lisp Users Conference, 1998.

[4] Kathy Walrath and Mary Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley, ISBN 0-201-43321-4, 1999.

[5] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, 1985.

[6] Bil Lewis, Daniel LaLiberte, and GNU manual group. *GNU Emacs Lisp reference manual*. Free Software Foundation, 1990.

[7] Richard Stallman. *EMACS: The Extensible, Customizable, Self-Documenting Display Editor*. In Text manipulation: Proceedings of the ACM SIGPLAN/SIGOA symposium (Portland, OR, June, 1981), June, 1981.