Chapter 1. GNU Emacs: Creeping Featurism is a Strength

Table of Contents

The GNU Emacs text editor is unmatched in its notoriety. Writer Neil Stephenson says:

> I use Emacs, which might be thought of as a thermonuclear word processor. ... Emacs outshines all other editing software in approximately the same way that the noonday sun does the stars. It is not just bigger and brighter; it simply makes everything else vanish.
>
> — Neil Stephenson, "In the Beginning There Was The Command Line". http://www.cryptonomicon.com/beginning.html

Emacs's detractors call it obscure, complex, and outdated, compared to more widely used development environments like Microsoft's Visual Studio. Even its fans blame their wrist injuries on Emacs's contorted keyboard command set.

Emacs provokes such strong reactions partly because there's so much of Emacs to react to. The current Emacs sources include 1.1 million lines of code written in Emacs's own programming language, Emacs Lisp. This corpus includes code to help you edit programs in C, Python, and other languages, as you might expect from a programmer's text editor. But it also includes code to help you debug running programs, collaborate with other programmers, read electronic mail and news, browse and search directories, and solve symbolic algebra problems.

At first blush this may seem pretty ridiculous. Why would you want to read your mail using a text editor? Is Emacs actually better at these tasks than programs dedicated to the purpose? Does it even make sense to call Emacs a text editor if one uses it for all these other things? And how does one find one's way through that thicket of features? But Emacs's architecture, founded on its extension language and buttressed by a sound set of editing primitives and coding conventions, allows it to support this bazaar of functionality while also achieving remarkable integration and consistency.

Creeping featurism is the tendency for a program to acquire many features over time that fit poorly with its intended function, clutter its user interface, and complicate its implementation. The term suggests kudzu vines engulfing a building, as if the problem were introduced by something beyond the designers' control.

Does Emacs suffer from creeping featurism? In this essay I want to persuade you that Emacs *has* creeping featurism, but doesn't *suffer* from it. In fact, Emacs's appeal derives directly from its ability to ac-

commodate so many different uses, and make them work well together. In the context of Emacs's architecture, creeping featurism is a strength, not a disease.

Emacs In Use

You can use Emacs like any other text editor. When you invoke Emacs on a file, a window appears displaying the file's contents. You can make your changes, save the revised contents, and exit. However, Emacs is not very effective when used this way: it is slower to start than other popular text editors, and its strengths don't come to bear. When I need to work in this fashion, I don't use Emacs.

Emacs is meant to be started once, when you begin work, and then left running. You can edit as many files as you like within one Emacs session, saving changes as you go. Emacs can keep files in memory without displaying them, so what you see reflects what you're working on at present, but your other tasks are ready to be called up just as you left them. The experienced Emacs user only closes files if the computer seems to be running low on memory, so a long-running Emacs session may have hundreds of files open. This screen shot shows an Emacs session: ???

An Emacs session with two frames. The left frame is split into three windows, showing the Emacs splash screen, a browseable directory listing, and a lisp interaction buffer. The right frame has a single window, displaying a buffer of source code.

There are three essential kinds of objects involved here: frames, windows, and buffers.

Frames are what Emacs calls the windows of your computer's graphical user interface. The screen shot shows two frames side by side. If you use Emacs in a text terminal, perhaps via a **telnet** or **ssh** connection, that terminal is also an Emacs frame. Emacs can manage any number of graphical frames and terminal frames simultaneously.

Windows are subdivisions of frames[1]. New windows are created only by dividing existing windows in two, and deleting a window returns its space to the adjacent windows; as a consequence, a frame's windows (or window) always fill its space completely. There is always a currently selected window, to which keyboard commands apply. Windows are lightweight, and tend to come and go frequently in typical use.

Finally, *buffers* hold editable text. Emacs holds each open file's text in a buffer, but a buffer need not be associated with a file: it might contain the results of a search, on-line documentation, or simply text entered by hand and not yet saved to any file. Each window displays the contents of some buffer.

It's important to understand that, aside from the mode lines at the bottom of each window and other similar decorations, the only way Emacs ever displays text to users is by placing it in a buffer, and then displaying that buffer in some window. Help messages, search results, directory listings, and the like all go into buffers with appropriately chosen names. This may seem like a cheap implementation trick — it does simplify Emacs internally — but it's actually quite valuable, because it means that these different kinds of content are all ordinary editable text: you can use the same commands to navigate, search, organize, trim, and sort these data that are available to you in any other text buffer. Any command's output can serve as any another command's input. This is in contrast with environments like Microsoft Visual Studio, where the results of a search (say) can only be used in the ways Microsoft anticipated would be useful. But Visual Studio is not alone in this; most programs with graphical user interfaces have the same shortcoming.

For example, in the screen shot, note the window of the frame on the left showing a directory listing. Like many directory browsers, this window provides terse keyboard commands for copying, deleting, renaming, and comparing files, selecting groups of files with globbing patterns or regular expressions, and (of course) opening them up in Emacs.

But unlike most directory browsers, the listing itself is plain text, held in a buffer. All the usual Emacs

[1] Note that what most graphical user interfaces call a window, Emacs calls a frame, since Emacs uses the term "window" as described here. This is unfortunate, but Emacs's terminology was established well before the widespread use of graphical user interfaces.

search facilities (including the excellent *incremental search* commands) apply. I can readily cut and paste the listing into a temporary buffer, delete the meta-data on the right to get a plain list of names, winnow out the files I'm not interested in using a regular expression, and end up with a list of file names to pass to some new operation. Once you've gotten used to this, using ordinary directory browsers becomes annoying because the listings feel inaccessible and out of reach. Even composing shell commands can feel like a waste of time because it's not as easy to see the intermediate results of your actions as you go.

Emacs Lisp

The heart of Emacs is its implementation of its own dialect of Lisp. Almost every command you invoke in Emacs, whether from the keyboard, a menu, or by name, is a lisp function. Emacs Lisp plays the key role in Emacs's ability to successfully accommodate the wide range of functionality Emacs has grown to offer.

The Five-Minute Lisp Tutorial

People new to lisp often find the language hard to read; this is mostly because Lisp has fewer syntactic constructs than most languages, but works them harder to get an even richer vocabulary of features. To give you a reader's intuition, consider the following rules for converting code from Python to lisp:

1. Write control constructs as if they were function calls. That is, `while x*y < z: q()` becomes `while (x*y < z, q())`. This is just a change in notation; it's still a while loop.

2. Write uses of infix operators as if they were function calls too, where the functions have odd-looking names. The expression `x*y < z` would become `<(*(x, y), z)`. Now any parentheses present only for grouping (as opposed to those that surround function arguments) are redundant; remove them.

3. Now everything looks like a function call — even control constructs and primitive operations. Move the opening parenthesis of every "call" from after the function name to before the function name, and drop the commas. For example, `f(x, y, z)` becomes `(f x y z)`. To continue the example above, `<(*(x, y), z)` becomes `(< (* x y) z)`.

Taking all three rules together, the Python code `while x*y < z: q()` becomes the lisp code `(while (< (* x y) z) (q))`. This is a proper Emacs Lisp expression, with the same meaning as the original Python statement.

There's more, obviously, but this is the essence of lisp's syntax. The bulk of lisp is simply a vocabulary of functions (like `<` and `*`), and control structures (like `while`), all used as shown here.

Here is the definition of an interactive Emacs command to count words in the selected region of text. If I explain that `"\\<"` is an Emacs regular expression matching the beginning of a word, you can probably read through it:

```
(defun count-region-words (start end)
  "Count the words in the selected region of text."
  (interactive "r")
  (let ((count 0))
    (save-excursion
      (goto-char start)
      (while (re-search-forward "\\<" end t)
        (setq count (+ count 1))
        (forward-char 1))
      (message "Region has %d words." count))))
```

There's an argument to be made that lisp gives its human readers too few visual cues as to what's really going on in the code, and that it should abandon this bland syntax in favor of one that makes more apparent distinctions between primitive operations, function calls, control structures, and the like. However, some of lisp's most important features (which I can't go into here) depend fundamentally on this syntactic uniformity; the many attempts that have been made to leave it behind haven't generally fared well. With experience, many lisp programmers come to feel that the syntax is a reasonable price to pay for the language's power and flexibility.

Emacs and its lisp have some critical characteristics:

- Emacs Lisp is light on bureaucracy. Small customizations and extensions are easy: one-line expressions placed in the `.emacs` file in your home directory, which Emacs loads automatically when it starts, can load existing packages of lisp code, set variables that affect Emacs's behavior, adjust keyboard commands, and so on. You can write a useful command in under a dozen lines — including its on-line documentation.
- Emacs Lisp is interactive. You can enter definitions and expressions into an Emacs buffer and evaluate them immediately. Revising a definition and reevaluating it puts your changes in place; there is no need to recompile or restart Emacs. Emacs is, in effect, an integrated development environment for Emacs Lisp programs.
- Emacs Lisp is safe. Even buggy lisp code won't crash Emacs, and the damage bugs can do is contained in other ways as well. This makes lisp development more pleasant and encourages experimentation.
- Emacs Lisp code is easy to document and browse. A function's definition can include a *documentation string*: text explaining the function's purpose and use. Almost every function provided with Emacs has a docstring, meaning that help on a given facility is never more than a command away. And when Emacs displays a function's docstring, it includes a hyperlink to the function's source code.
- Emacs Lisp is a full programming language; it is comfortable to write reasonably large programs (hundreds of thousands of lines) in Emacs Lisp.
- Emacs Lisp programs you write yourself are a first-class citizens. Every non-trivial editing feature of Emacs itself is written in lisp, so all the primitive functions and libraries Emacs needs are equally available to your own lisp code. Emacs Lisp arrogates no special privileges to itself. (There is a distinction between lisp code and C code, which I'll discuss more below.)
- Emacs Lisp has conventions to help unrelated packages avoid interfering with each other when loaded into the same Emacs session. This allows users to share packages of Emacs Lisp code with each other without the coordination or approval of Emacs's developers.

Grandiose claims require concrete justification, so let's take some time to get into the details, and see how things play out. The best way get a concrete understanding of the relationship between Emacs and its lisp is to walk through the process of defining a simple command. If you would like to try things out as we go, but aren't familiar with how to use Emacs, the Emacs tutorial, linked to from the opening splash screen, can get you started.

The `*scratch*` buffer

The most comfortable way to experiment with Emacs Lisp is to switch to the `*scratch*` buffer, which Emacs creates automatically when it starts up. This buffer is in *Lisp Interaction Mode*, which makes it especially suitable for interactive development of lisp code; we'll say more about modes below.

If you enter a lisp expression into the `*scratch*` buffer and place the cursor after the final parenthesis, you can press **Control**+**j** to evaluate the expression. For example, if you enter:

```
(+ (* 10 10 10) (* 9 9 9))
```

and press **Control**+**j**, Emacs will insert the value of this expression, `1729`, on the line below.

When evaluating short expressions like this, it may be more convenient to press **Meta**+**:**, enter the expression at the prompt in the minibuffer, and press **return**; Emacs will display the value in the echo area. However, **Meta**+**:** doesn't leave you with a record of your interactions as the `*scratch*` buffer does.

"Hello, World" in Emacs Lisp

In Emacs Lisp, a function definition looks like this:

```
(defun hello ()
  (message "Hello, world!"))
```

Enter this text in the `*scratch*` buffer and evaluate it. This defines a function named `hello`. Since all Emacs Lisp expressions must return a value, the `defun` evaluates to the name of the function it has defined; thus, when you evaluate this definition, `hello` appears below it in the `*scratch*` buffer, just as `1729` appeared below the arithmetic expression you entered earlier.

You can now call this function by entering the text:

```
(hello)
```

in the `*scratch*` buffer and evaluating it with **Control**+**j**, or by using **Meta**+**:** and entering the call at the prompt. Calling the function displays the string `Hello, world!` in the echo area, and inserts the value of the call into the `*scratch*` buffer — in this case, the string `"Hello, world!"`.

It's worth noting that the `*scratch*` buffer is an ordinary text buffer: you can go back to an expression or definition you've evaluated before, edit it, and re-evaluate the modified version. Or, you could copy out a subexpression to evaluate it on its own. Like the directory browsing buffer mentioned earlier, it's all just text.

Documentation Strings

Emacs makes it easy to provide on-line documentation for your functions. We can document `hello` with a one-line change:

```
(defun hello ()
  "Greet the world."
  (message "Hello, world!"))
```

The string we've added at the top of the function body is a *documentation string*, or *docstring*. (If you've

used documentation strings in Python, Emacs Lisp's are much the same.) The first line of a docstring should be a self-contained summary of the function's purpose; the rest should provide a complete description of its behavior and use. You can view a function's docstring with the **describe-function** command, bound to **Control**+**h f**.

Since providing on-line documentation takes only marginally more effort than writing comments in the source code, Emacs Lisp programmers generally do write documentation strings: most functions in Emacs of any general utility have them, which greatly facilitates browsing and writing lisp code. In fact, if the cursor is sitting inside a function call, the `describe-function` command will, by default, describe the function being called, so you can see which arguments are expected and what they mean with a few keystrokes.

If you're not sure which function to use, the **apropos** command searches for variables and functions whose names match a given pattern, and displays the summary line of each matching entity's docstring. There are variants of **apropos** that are more specific, like **apropos-command** and **apropos-variable**.

Commands

Now that we have shown how to define functions, we can show the relationship between definitions in the lisp world and the commands available to users. It's simple and direct: an Emacs *command* is simply a lisp function that has been marked as suitable for interactive use. For example, to turn our `hello` function into a command, we edit the definition to read as follows:

```
(defun hello ()
  "Greet the world."
  (interactive)
  (message "Hello, world!"))
```

The `interactive` form isn't a function call or a control structure; this version of the function still behaves exactly as it did before. Rather, the `interactive` form declares that this function is a command. Once you've evaluated this definition, you can invoke `hello` with **Meta**+**x hello**, like any other Emacs command. You could even bind it to a key by evaluating the following expression:

```
(global-set-key "\C-ch" 'hello)
```

After evaluating this expression, you can invoke the **hello** command simply by typing **Control**+**c h**. You can add a menu item that invokes **hello** in a similar fashion.

This is, in fact, how every key press, mouse gesture, or menu selection you make in Emacs is interpreted: each of them is bound to a command, a lisp function marked for interactive use. The arrow keys are bound to commands named `forward-char`, `backward-char`, and so on; the left mouse button is bound to `mouse-drag-region`; and so on. Even the keys for entering plain text are bound to the command `self-insert-command`, which inspects the key that invoked it to decide what to insert.

If a lisp function marked as a command takes arguments, its `interactive` form must tell Emacs how to prompt for those arguments when it is invoked interactively. For example, to make `hello` greet people by name, we could amend its definition as follows:

```
(defun hello (name)
  "Prompt for the name of someone to greet, and greet them."
  (interactive "MName of person to greet: ")
  (message "Hello, %s!" name))
```

If you invoke this version as a command, Emacs prompts you in the minibuffer for the name of a person

to greet, and substitutes that name for the `%s` in the message; ??? shows this in action.

We have defined the command `hello` in the `*scratch*` buffer, bound it to the key sequence **Control+c h**, and then invoked it. Emacs is prompting us for the value of `hello`'s *`name`* argument, and we've entered `Dolly`.

Note that you can still call an Emacs command as you would any other lisp function. In this example, you can display the same message in the echo area by evaluating the expression `(hello "Dolly")`.

So here, in four lines of code, we've written a minimal Emacs command that can be invoked by name or via a key sequence — and documented it as well. This is (or would be, if it did something useful) a unit of work you can share with friends, post on a mailing list, or package up in a file and distribute on the web. Because Emacs imposes so little bureaucracy on extension developers (especially compared to systems like the Eclipse integrated development environment), it takes much less of an initial investment to experiment with new ideas. As a result, it's very common to find yourself wishing for some feature or convenience in Emacs, do a little searching, and discover someone else has already implemented it!

Accessing Text

We've seen how the lisp world connects to Emacs's user interface, by marking functions as commands that the user can invoke by name, by key bindings, or by menu entries. But we haven't seen how lisp might actually get work done. Our `hello` command, albeit well-documented and flexible, doesn't shed much light on how a command might perform Emacs's ostensible purpose: editing text.

To that end, let's look at a function that actually works with the contents of a buffer. Here's a function (not a command; it's only meant to be used by other lisp code) that returns the contents of the current line as a string.

```
(defun jimb-current-line ()
  "Return the contents of the current line, as a string.
This includes the final newline, if any."
  (save-excursion
    (forward-line 0)
    (let ((start (point)))
      (forward-line 1)
      (buffer-substring start (point)))))
```

In Emacs, there is always a *current buffer*, and each buffer has a current editing location called *point*; point, especially, is used and modified by many editing primitives. The body of this function is wrapped up inside a `save-excursion` form, which evaluates the expressions in its body and then restores the current buffer and its point to where they were originally — even if an error occurs and the body exits abruptly. This allows the body code to move point around freely without disturbing the surrounding code.

The call `(forward-line` *`n`*`)` moves point to the start of the *n*th line after the one it's on now. By extension, `(forward-line 0)` moves point to the start of the current line.

The `point` function returns the position of point as an integer. We save the position of the start of the line in the variable `start`, and then move point to the beginning of the next line.

The call `(buffer-substring` *`a b`*`)` returns the contents of the buffer between the positions *a* and *b* as a string. Here, we extract text from `start` (the beginning of the original line) to point (at the beginning of the next line).

As we exit, the `save-excursion` form restores point to its original position, somewhere in the line we extracted, so that our caller is unaware that we've moved point temporarily for our own use. The value of a `save-excursion` form is the value of the last expression in its body. In this case, that is

the string returned by `buffer-substring`: the contents of the line.

This code may look odd to readers familiar with newer bodies of code. Modern programming style avoids global variables, because of their tendency to introduce unexpected interactions between unrelated pieces of code. In Emacs, however, the current buffer is essentially a global variable, and the buffer's point is effectively global to all code operating on that buffer. Wouldn't it be better for commands to take the current buffer as an argument, pass locations to editing primitives as explicit arguments, and use some separate means to move the cursor the user sees?

Emacs takes a different approach, avoiding unexpected interactions while retaining a brevity and clarity that seems to have been abandoned by much modern code. Emacs's approach allows the buffer, and usually the position within the buffer, to remain implicit, and uses `save-excursion` and related constructs to avoid cross-talk. Because the current buffer and its point are almost always the items of interest anyway, the effect of Emacs's approach is to keep text manipulation code terse and clear, free of variables that state the obvious. The risk, of course, is that the programmer might forget to use `save-excursion`.

The name of the function, starting with `jimb-`, may also look a bit strange. Emacs Lisp lacks modules, classes, and namespaces, providing only a single global namespace. To make this approach scale, it is conventional to give lisp identifiers names whose prefix says what package of code the function or variable belongs to. In this case, I'm using the `jimb-` prefix for things that aren't part of any larger package, written just for my personal use. This convention may seem long-winded, and the single global namespace certainly makes it impossible for a package to hide its internal details from other code. But given how badly this system falls short of modern expectations, it works surprisingly well in practice. For whatever reason, violations of abstraction boundaries don't seem to be a major issue in the Emacs code base. And the single namespace seems to make it easier to explore and experiment with code.

Working With Buffers and Windows

Taking this one last step, let's look at a function that modifies text, creates buffers and manipulates windows.

In my work, I frequently find myself trying to make sense of compiler invocations that run on for many lines. The following is a relatively brief example:

```
/home/jimb/gcc/bin/g++ --verbose -o Interpreter.o -DAVMPLUS_IA32
-msse2 -DMMGC_INTERIOR_PTRS -DUNIX -DSOFT_ASSERTS -DLINUX -DAVMPL
US_LINUX -DAVMPLUS_UNIX  -DDEBUG -D_DEBUG   -fno-exceptions -Werr
or -Wall -Wno-reorder -Wno-switch -Wno-invalid-offsetof -Wno-unin
itialized -Wno-strict-aliasing -fmessage-length=0 -finline-functi
ons -finline-limit=65536 -frtti -fexceptions  -g3    -I/home/jimb
/mc/tt-mmgc-update -I/home/jimb/mc/tt-mmgc-update/core -I/home/ji
mb/mc/tt-mmgc-update/nanojit -I/home/jimb/mc/tt-mmgc-update/pcre
 -c Interpreter.ii
```

I've written the following command to break down such commands and pop up a new window to display them in a more legible form. It recognizes the most common compiler flags that expect values, placing those values on the same line as the flag. Since I usually run compilations in Emacs, I can simply place the cursor on the compiler invocation of interest, and use this command to see what's going on.

```
(defun jimb-gcc-cleanup ()
  "Reformat the GCC invocation on the current line for legibility."
  (interactive)
  (let ((case-fold-search nil))
    (save-excursion
      (let ((line (jimb-current-line)))
```

```
(set-buffer (get-buffer-create "*GCC invocation*"))
(erase-buffer)
(insert line)

;; Break the line after each flag (and value, if any).
(goto-char (point-min))
(while (< (point) (point-max))
  ;; If the flag takes a value, find the value's end.
  (if (looking-at "-[IoLTe]\\s-+\\S-+\\s-*")
      (goto-char (match-end 0))
    ;; Otherwise, just skip to the end of the flag.
    (re-search-forward "\\s-+" nil 'at-limit))
  ;; Break the line here.
  (insert "\n"))

;; Pop up the current buffer in a new window,
;; with point at the top.
(goto-char (point-min))
(display-buffer (current-buffer))))))
```

First, we call the `jimb-current-line` function to extract the compiler invocation from the current line. Then we call `get-buffer-create` to create a buffer named `*GCC invocation*` to hold the reformatted command; if there is already a buffer by that name, we just return it. The `set-buffer` function makes it current; `erase-buffer` clears out any old contents; and `insert` places a copy of the command in the buffer for us to work on. We walk the command, inserting line breaks after each flag. And finally, we call `display-buffer` to make the results of our work visible to the user, typically popping it up in a new window. We can see the result of applying **jimb-gcc-cleanup** to the GCC command mentioned above in ???.

Using the newly defined Emacs command `jimb-gcc-cleanup` to reformat a long compiler invocation for legibility.

Now, this command is a small comfort, not a critical service. It wouldn't be worth writing an Eclipse component to pretty-print verbose compiler invocations. But it's certainly worth taking a few minutes to write thirty lines of lisp to have a command that makes one's work a little more pleasant each day.

Buffers and Modes

The last critical piece of the

Now we can see the origins of Emacs's creeping featurism. First, the environment Emacs provides for editing files is also a comfortable environment for developing Emacs itself. When you notice something that would be nice to have, the lack of bureaucracy makes it easy to give it a try. A single command defined in your `.emacs` file can grow into a suite of commands that work together, packaged up in a file you can share with your friends. The most popular packages come to be included in the stock Emacs distribution, completing the process.

Thus, Emacs grows organically, in response to its users' interests; the strange features exist simply because some wrote them, and many others found them useful. The role of Emacs's developers, beyond fixing bugs and adding useful new primitives, is essentially to select the most popular, well-developed packages that the community is already using for inclusion in the official sources.

The low barriers to entry established by Emacs's architecture also encourage the creation of thousands of little tools to make everyday editing a little more comfortable. For long-time Emacs users it's a common experience to be using an Emacs command, notice some obvious refinement that one could make, and then look through the documentation and find that someone else has already implemented it. Given that the lisp-literate cohort of the Emacs user base has been adjusting and adapting Emacs for almost twenty

years now, it's usually a given that many people have already been whereever you are now.

One drawback to this approach is its disorganization: if the issue at hand is something technically formidable, Emacs is more likely to have a handful of incomplete solutions than one unified architecture that solves it thoroughly. For example, Emacs's ability to jump to an identifier's definition doesn't compete with that of Eclipse on Java code, or of Visual Studio when editing programs written in any of Microsoft's favored languages.

Another drawback is the obscurity of the resulting user interface. It's often easy to make a small change to address some irritating detail; to actually rework the design to eliminate the root of the problem takes more effort. Commands and packages tend to grow dozens of customization options and alternative commands before someone with a broader architectural view comes in and cleans things up.

A Component API Is Not An Extension Language

FILL IN SECTION TITLE HERE

BODY TEXT HERE...