Emacs and eev, or: How to Automate Almost Everything

Eduardo Ochs

[1]http://angg.twu.net/

edrx@mat.puc-rio.br
Not currently affiliated to any institution.
Snail-mail address: R. Jardim Botânico 622/103B, Jardim Botânico, Rio de Janeiro, RJ, Brazil, CEP 22461-000.

Abstract. *Interacting with programs with command-line interfaces always involve a bit of line editing, and each CLI program tends to implement independently its own minimalistic editing features. We show a way of centralizing these editing tasks by making these programs receive commands that are prepared, and sent from, Emacs. The resulting system is a kind of Emacs- and Emacs Lisp-based "universal scripting language" in which commands can be sent to both external programs and to Emacs itself either in blocks or step-by-step under very fine control from the user.*

Note: this is a working draft that has many pieces missing and needs urgent revision on the pieces it has. Current version: 2005jun01 0:42. Newer versions are being uploaded to ⟨http://angg.twu.net/FISL/⟩, and two animations (in Flash) showing eev at work can be found at: ⟨http://angg.twu.net/eev-current/anim/channels.anim.html⟩ and ⟨http://angg.twu.net/eev-current/anim/gdb.anim.html⟩.

1. Three kinds of interfaces

Interactive programs in a Un*x system[1] can have basically three kinds of interfaces: they can be mouse-oriented, like most programs with graphical interfaces nowadays, in which commands are given by clicking with the mouse; they can be character-oriented, like most editors and mail readers, in which most commands are single keys or short sequences of keys; and they can be line-oriented, as, for example, shells are: in a shell commands are given by editing a full line and then typing "enter" to process that line.

It is commonplace to classify computer users in a spectrum where the "users" are in one extreme and the "programmers" are in the other; the "users" tend to use only mouse-oriented and character-oriented programs, and the "programmers" only character-oriented and line-oriented programs.

In this paper we will show a way to "automate" interactions with line-oriented programs, and, but not so well, to character-oriented programs; more precisely, it is a way to edit commands for these programs in a single central place — Emacs — and then send them to the programs; re-sending the same commands afterwards, with or without modifications, then becomes very easy.

This way ("e-scripts") can not be used to send commands to mouse-oriented programs — at least not without introducing several new tricks. But "programmers" using Un*x systems usually see most mouse-oriented programs — except for a few that are *intrinsically* mouse-oriented, like drawing programs — as being just wrappers around line-oriented programs than perform the same tasks with different interfaces; and so, most mouse-oriented programs "do not matter", and our method of automating interactions using e-scripts can be used to automate "almost everything"; hence the title of the paper.

[1]Actually we are more interested in GNU systems than in "real" Unix systems; the reasons will become clear in the section nnn. By the way: the term "Unix" is Copyright (C) Bell Labs).

2. "Make each program do one thing well"

One of the tenets of the Unix philosophy is that each program should do one thing, and do it well; this is a good design rule for Unix programs because the system makes it easy to invoke external programs to perform tasks, and to connect programs.

Some of parts of a Unix system are more like "meta-programs" or "sub-programs" than like self-contained programs that do some clearly useful task by themselves. Shells, for example, are meta-programs: their main function is to allow users to invoke "real programs" and to connect these programs using pipes, redirections, control structures (if, for, etc) and Unix "signals". On the other hand, libraries are sub-programs: for example, on GNU systems there's a library called GNU readline that line-oriented programs can use to get input; if a program, say, bc (a calculator) gets its input by calling readline(...) instead of using the more basic function fgets(...) then its line-oriented interface will have a little more functionality: it will allow the user to do some minimal editing in the current line, and also to recall, edit and issue again some of the latest commands given.

3. Making programs receive commands

Many line-oriented programs allow "scripting", which means executing commands from a file. For example, in most shells we can say "source ~/ee.sh", and the shell will then execute the commands in the file ~/ee.sh. There are other ways of executing commands from a file — like "sh ~/ee.sh" — but the one with "source" is the one that we'll be more interested in, because it is closer to running the commands in ~/ee.sh one by one by hand: for example, with "source ~/ee.sh" the commands that change parameters of the shell — like the current directory and the environment variables — will work in the obvious way, while with "sh ~/ee.sh" they would only change the parameters of a temporary subshell; the current directory and the environment variables of the present shell would be "protected".

So, it is possible to prepare commands for a shell (or for scriptable line-oriented programs; for arbitrary line-oriented programs see the section nnn) in several ways: by typing them at the shell's interface — and if the shell uses readline its interface can be reasonably friendly — or, alternatively, by using a text editor to edit a file, say, ~/ee.sh, and by then "executing" that file with "source ~/ee.sh". "source ~/ee.sh" is a lot of keystrokes, but that can be shortened if we can define a shell function: by putting

```
function ee () { source ~/ee.sh; }
```

in the shell's initialization file (~/.bashrc, ~/.zshrc, ...) we can reduce "source ~/ee.sh" to just "ee": e, e, enter — three keystrokes.

We just saw how a shell — or, by the way, any line-oriented program in which we can define an 'ee' function like we did for the shell — can receive commands prepared in an external editor and stored in a certain file; let's refer to that file, ~/ee.sh, as a "temporary script file". Now it remains to see how an external text editor can "send commands to the shell", i.e., how to make the editor save some commands in a temporary script file in a convenient way, that is, without using too many keystrokes...

4. Sending commands

GNU Emacs, "the extensible, self-documenting text-editor" ([S79]), does at least two things very well: one is to edit text, and so it can be used to edit temporary scripts, and thus to send commands to shells and to line-oriented programs with 'ee' functions; and the other one is to run Lisp. Lisp is a powerful programming language, and (at least in principle!) any action or series of actions can be expressed as a program in Lisp; the first thing that we want to do is a way to mark a region of a text and "send it as commands to a shell", by saving it in a temporary script file. We implement that in two ways:

```
 1: (defun ee (s e)
 2:   "Save the region in a temporary script"
 3:   (interactive "r")
 4:   (write-region s e "~/ee.sh"))
 5:
 6: (defun eev (s e)
 7:   "Like 'ee', but the script executes in verbose mode"
 8:   (interactive "r")
 9:   (write-region
10:    (concat "set -v\n" (buffer-substring s e)
11:            "\nset+v")
12:    nil "~/ee.sh"))
```

'ee' (the name stands for something like 'emacs-execute') just saves the currently-marked region of text to ~/ee.sh; 'eev' (for something like 'emacs-execute-verbose') does the same but adding to the beginning of the temporary script a command to put the shell in "verbose mode", where each command is displayed before being executed, and also adding at the end an command to leave verbose mode.

We can now use 'ee' and 'eev' to send a block of commands to a shell: just select a region and then run 'ee' or 'eev'. More precisely: mark a region, that is, put the cursor at one of the extremities of the region, then type C-SPC to set Emacs's "mark" to that position, then go to other extremity of the region and type M-x eev (C-SPC and M-x are Emacs's notations for Control-Space and Alt-x, a.k.a. "Meta-x"). After doing that, go to a shell and make it "receive these commands", by typing 'ee'.

5. Hyperlinks

When we are learning Un*x — and until the point where we become a genius or an *idiot savant* who knows all the documentation files by heart; but I've been using GNU systems for 10 years and I'm neither of these yet — we spend a lot of the time when we are not issuing commands (which are mostly to shells) accessing and reading documentation files; and we can use Emacs Lisp "one-liners" to create "hyperlinks" to files:

```
1: (info "(emacs)Lisp Eval")
2: (find-file "~/usrc/busybox-1.00/shell/ash.c")
3: (find-file "/usr/share/emacs/21.4/lisp/info.el")
```

These expressions, when executed — which is done by placing the cursor after them and then typing C-x C-e, or, equivalently, M-x eval-last-sexp — will open a page of Emacs manual (the manual is in "Info" format), open the source file 'shell/ash.c' of a program called busybox, and open the file 'info.el' from the Emacs sources, respectively. As some of these files and pages can be very big, these hyperlinks are not yet very satisfactory: we want ways to not only open these files and pages but also to "point to specific positions", i.e., to make the cursor go to these positions automatically. We can do that by defining some new hyperlink functions, that are invoked like this:

```
1': (find-node "(emacs)Lisp Eval" "C-x C-e")
2': (find-fline "~/usrc/busybox-1.00/shell/ash.c"
                "void\nevalpipe")
3': (find-fline "/usr/share/emacs/21.4/lisp/info.el"
                "defun info")

1'': (find-enode "Lisp Eval" "C-x C-e")
2'': (find-busyboxfile "shell/ash.c" "void\nevalpipe")
3'': (find-efile "info.el" "defun info")
```

The convention is that these "extended hyperlink functions" have names like 'find-*xxx*node', 'find-*xxx*file', or 'find-*xxxyyy*'; as the name 'find-file' was already taken by a standard Emacs function we had to use 'find-fline' for ours.

Here is the definition of 'find-node' and 'find-fline':

```
14: (defun ee-goto-position (&optional pos-spec)
15:   "If POS-SPEC is a string search for its first
16:    occurrence in the file; if it is a number go to the
17:    POS-SPECth line; if it is nil, don't move."
18:   (cond ((null pos-spec))
19:         ((numberp pos-spec)
20:          (goto-char (point-min))
21:          (forward-line (1- pos-spec)))
22:         ((stringp pos-spec)
23:          (goto-char (point-min))
24:          (search-forward pos-spec))
25:         (t (error "Invalid pos-spec: %S" pos-spec))))
26:
27: (defun find-fline (fname &optional pos-spec)
28:   "Like (find-file FNAME), but accepts a POS-SPEC"
29:   (find-file fname)
20:   (ee-goto-position pos-spec))
31:
32: (defun find-node (node &optional pos-spec)
33:   "Like (info NODE), but accepts a POS-SPEC"
34:   (info node)
35:   (ee-goto-position pos-spec)))
```

The hyperlinks in lines $1''$, $2''$ and $3''$ are equivalent to the ones in $1'$, $2''$, $3''$ but are a bit shorter, and they hide details like Emacs's path and the version of BusyBox; if we switch to newer versions of Emacs and BusyBox we only need to change the definitions of 'find-busyboxfile' and 'find-efile' to update the hyperlinks. Usually not many things change from one version of a package to another, so most hyperlinks continue to work after the update.

Eev defines a function called 'code-c-d' that makes defining functions like 'find-enode', 'find-busyboxfile' and 'find-efile' much easier:

```
(code-c-d "busybox" "~/usrc/busybox-1.00/")
(code-c-d "e" ""/usr/share/emacs/21.4/lisp/" "emacs")
```

The arguments for ‘code-c-d’ are a code (the “*xxx*” in a “find-*xxx*file”), a directory, and optionally the name of a manual in Info format. The definition of code-c-d is not very interesting, so we won’t show it here.

Note that we can put these Lisp hyperlinks inside “comments” and still execute them (by hand); if we send the following lines to a shell,

```
# (find-node "(gawk)Fields")
seq 4 9 | gawk '{print $1, $1*$1}'
```

the shell will ignore the first line because of the ‘#’; but when we are editing that we can execute the ‘(find-node ...)’ with C-x C-e.

It is so common to have Lisp hyperlinks that extend from some position in a line — usually after a comment sign — to the end of the line that eev implements a special key for executing these hyperlinks: the effect of typing M-e (when eev is installed and “eev mode” is on) is roughly the same of first going to the end of the line and then typing C-x C-e; that is, M-e does the same as the key sequence C-e C-x C-e[2].

[There are many other kinds of hyperlinks. Examples?]

6. Dangerous hyperlinks

Note that these “hyperlinks” can do very dangerous things. If we start to execute blindly every Lisp expression we see just because it can do something interesting or take us to an interesting place then we can end up running something like:

```
(shell-command "rm -Rf ~")
```

which destroy all files in our home directory; not a good idea. Hyperlinks should be a bit safer than that...

The modern approach to safety in hyperlinks — the one found in web browsers, for example — is that following a hyperlink can execute only a few kinds of actions, all known to be safe; the “target” of a hyperlink is something of the form http://..., ftp://..., file://..., info://..., mailto:... or at worst like javascript:...; none of these kinds of actions can even erase our files. That approach limits a lot what hyperlinks can do, but makes it harmless to hide the hyperlink action and display only some descriptive text.

Eev’s approach is the opposite of that. I wrote the first functions of eev in my first weeks after installing GNU/Linux in my home machine and starting using GNU Emacs, in 1994; before that I was using mostly Forth (on MS-DOS), and I hadn’t had a lot of exposure to *nix systems by then — in particular, I had tried to understand *nix’s notions of user IDs and file ownerships and permissions, and I felt that they were a thick layer of complexity that I wasn’t being able to get through.

Forth’s attitude is more like “the user knows what he’s doing”; the system is kept very simple, so that understanding all the consequences of an action is not very hard. If the user wants to change a byte in a critical memory position and crash the machine he can do that, and partly because of that simplicity bringing the machine up again didn’t use to take more than one minute (in the good old days, of course). Forth people developed good backup strategies to cope with the insecurities, and — as strange as that might sound

[2]The main difference between M-e and C-e C-x C-e is how they behave when called with numeric “prefix arguments”: for example, M-0 M-e highlights temporarily the Lisp expression instead of executing it and M-4 M-e executes it with some debugging flags turned on, while C-x C-e when called with any prefix argument inserts the result of the expression at the cursor instead of just showing it at the echo area.

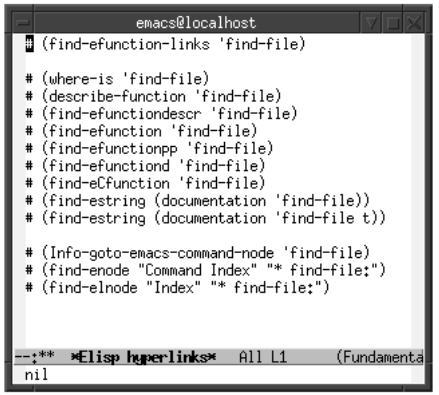

Figure 1: the result of typing M-h M-f find-file

nowadays, where all machines are connected and multi-user and crackers abound — using the system in the Forth way was productive and fun.

*NIX systems are not like Forth, but when I started using them I was accustomed to this idea of achieving simplicity through the lack of safeguards, and eev reflects that. The only thing that keeps eev’s hyperlinks reasonably safe is *transparency*: the code that a hyperlink executes is so visible that it is hard to mistake a dangerous Lisp expression for a “real” hyperlink. Also, all the safe hyperlink functions implemented by eev start with ‘find-’, and all the ‘find-’ functions in eev are safe, except for those with names like ‘find-*xxx*sh’ and ‘find-*xxx*sh0: for example,

```
(find-sh "wget --help" "recursive download")
```

executes “wget --help”, puts the output of that in an Emacs buffer and then jumps to the first occurrence of the string “recursive download” there; other ‘find-*xxx*sh’ functions are variations on that that execute some extra shell commands before executing the first argument — typically either switching to another directory or loading an initialization file, like ~/.bashrc or ~/.zshrc. The ‘find-*xxx*sh0’ functions are similar to their ‘find-*xxx*sh’ counterparts, but instead of creating a buffer with their output they just show it at Emacs’s echo area and they use only the first argument and ignore the others (the pos-spec).

7. Generating Hyperlinks

Do we need to remember the names of all hyperlinks functions, like find-fline and find-node? Do we need to type the code for each hyperlink in full by hand? The answers are “no” and “no”.

Eev implements several functions that create temporary buffers containing hyperlinks, that can then be cut and pasted to other buffers. For example, ‘M-h M-f’ creates links about an Emacs Lisp function: typing ‘M-h M-f’ displays a prompt in a minibuffer asking for the name of an Elisp function; if we type, say, ‘find-file’ there (note: name completion with the TAB key works in that prompt) we get a buffer like the one in figure 1.

The first line of that buffer is a hyperlink to that dynamically-generated page of hyperlinks. Its function — ‘find-efunction-links’ — has a long name that is hard to remember, but there’s a shorter link that will do the same job:

```
(eek "M-h M-f find-file")
```

The argument to 'eek' is a string describing a sequence of keys in a certain verbose format, and the effect of running, say, `(eek "M-h M-f find-file")` is the same as of typing 'M-h M-f find-file'.

[M-h is a prefix; `(eek "M-h C-h")` shows all the sequences with the same prefix.]

[Exceptions: M-h M-c, M-h M-2, M-h M-y. Show examples of how to edit hyperlinks with M-h M-2 and M-h M-y.]

[Mention hyperlinks about a key sequence? `(eek "M-h M-k C-x C-f")`]

[Mention hyperlinks about a Debian package?]

8. Glyphs

Emacs allows redefining how characters are displayed, and one of the modules of eev — eev-glyphs — uses that to make some characters stand out. Character 15, for example, is displayed on the screen by default as '^O' (two characters, suggesting "control-O"), sometimes in a different color from normal text[3].

Eev changes the appearance of char 15 to make it be displayed as a red star. Here is how: Emacs has some structures called "faces" that store font and color information, and 'eeglyphs-face-red' is a face that says "use the default font and the default background color, but a red foreground"; eev's initialization code runs this,

```
(eev-set-glyph 15 ?* 'eev-glyph-face-red)
```

which sets the representation of char 15 to the "glyph" made of a star in the face eeglyphs-face-red.

For this article, as red doesn't print well in black and white, we used this instead:

```
(eev-set-glyph 15 342434)
```

this made occurrences of char 15 appear as the character 342434, '•' (note that this is outside of the ascii range), using the default face, i.e., the default font and color.

Eev also sets a few other glyphs with non-standard faces. The most important of those are '⟪' and '⟫', which are set to appear in green against the default background, with:

```
(eev-set-glyph 171 171 'eev-glyph-face-green)
(eev-set-glyph 187 187 'eev-glyph-face-green)
```

There's a technical point to be raised here. Emacs can use several "encodings" for files and buffers, and '⟪' and '⟫' only have character codes 171 and 187 in a few cases, mainly in the 'raw-text' encoding and in "unibyte" buffers; in most other encodings they have other char codes, usually above 255, and when they have these other codes Emacs considers that they are other characters for which no special glyphs were set and shows them in the default face. This visual distinction between the below-255 '⟪' and '⟫' and the other '⟪' and '⟫'s is deliberate — it helps preventing some subtle bugs involving the anchor functions of section 12.

[3]Determined by the "face" escape-glyph-face, introduced in GNU Emacs in late 2004.

9. Compose Pairs

To insert a '•' in a text we type 'C-q C-o' — C-q "quotes" the next key that Emacs receives, and 'C-q C-o' inserts a "literal C-o", which is a char 15. Typing '⟪' and '⟫'s — and other non-standard glyphs, if we decide to define our own — involves using another module of eev: eev-compose.

Eev-compose defines a few variables that hold tables of "compose pairs", which map pairs of characters that are easy to type into other, weirder characters; for example, 'eev-composes-otheriso' says that the pair `"<<"` is mapped to `"⟪"` and that `">>"` is mapped to `"⟫"`, among others. When we are in "eev mode" the prefix 'M-,' can be used to perform the translation: typing 'M-, < <' enters '⟪', and typing 'M-, > >' enters '⟫'.

The variable 'eev-composes-accents' holds mappings for accented chars, like `"'a"` to `"á"` and `"cc"` to `"ç"`; 'eev-composes-otheriso' takes care of the other mappings that still concern characters found in the ISO8859-1 character set, like '⟪' and '⟫' as above, `"_a"` to `"ª"`, `"xx"` to `"×"`, and a few others; 'eev-composes-globalmath' and 'eev-composes-localmath' are initially empty and are meant to be used for used-defined glyphs. The suffix 'math' in their names is a relic: Emacs implements its own ways to enter special characters, which support several languages and character encodings, but their code is quite complex and they are difficult to extend; the code that implements eev's 'M-,', on the other hand, takes about just 10 lines of Lisp (excluding the tables of compose pairs) and it is trivial to understand and to change its tables of pairs. 'M-,' was created originally to enter special glyphs for editing mathematical texts in TEX, but it turned out to be a convenient hack, and it stuck.

10. Delimited regions

Sometimes it happens that we need to run a certain (long) series of commands over and over again, maybe with some changes from one run to the next; then having to mark the block all the time becomes a hassle.

One alternative to that is using a variaton of M-x eev, eev-bounded, that saves the region around the cursor up to certain delimiters instead of saving what's between Emacs's "point" and "mark". The original definition of eev-bounded was something like

```
(defun eev-bounded ()
  (interactive)
  (eev (ee-search-backwards "\n#•\n")
       (ee-search-forward   "\n#•\n")))
```

where the call to ee-search-backwards would search for the first occurrence of the string `"\n#•\n"` (newline, hash sign, control-O, newline) before the cursor and return the position after the `"\n#•\n"`, without moving the cursor, and the call to ee-search-forward would do something similar with a forward search.

The block below shows a typical application:

```
# (code-c-d "lua5" "/tmp/usrc/lua-5.0.2/")
# (find-lua5file "INSTALL")
# (find-lua5file "config" "support for dynamic loading")
# (find-lua5file "config")
# (find-lua5file "")
#•
```

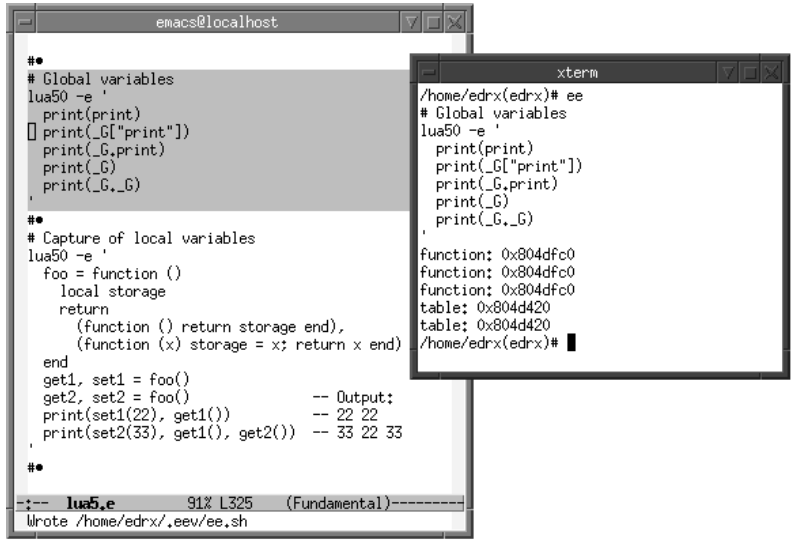

Figure 2: sending a delimited block with F3

```
rm   -Rv ~/usrc/lua-5.0.2/
mkdir -p ~/usrc/lua-5.0.2/
tar   -C ~/usrc/ \
   -xvzf $S/http/www.lua.org/ftp/lua-5.0.2.tar.gz
cd       ~/usrc/lua-5.0.2/
cat >> config <<'---'
LOADLIB= -DUSE_DLOPEN=1
DLLIB= -ldl
MYLDFLAGS= -Wl,-E
EXTRA_LIBS= -lm -ldl
---
make test  2>&1 | tee omt
./bin/lua -e 'print(loadlib)'
#●
```

in unpacks a program (the interpreter for Lua), changes its default configuration slightly, then compiles and tests it.

[about the size: the above code is "too small for being a script", and the hyperlinks are important]

Eev binds F3 to a function that runs the "default bounded function", which is usually eev-bounded; and the real definition of eev-bounded is quite different from the one above, as it includes code that highlightings temporarily the region that was saved (see Figure 2) and it also sets the default bounded function to eev-bounded — eev also implements several other functions that have "bounded versions", for example eelatex, that saves the region (plus certain standard header and footer lines) to a "temporary LaTeX file" and saves into the temporary script file the commands to make 'ee' run LaTeX on that and display the result. The block below is an example of (...)

```
%●
% (eelatex-bounded)
% (ee-once (eelatex-bounded))
\def\myttbox#1{%
  \setbox0=\hbox{\texttt{a}}%
  \hbox to \wd0{\hss#1\hss}%
}
\catcode`●=13 \def●{\myttbox{$\bullet$}}
\begin{verbatim}
abcdefg
   d●fg
\end{verbatim}
%●
```

Actually F3 runs the function eeb-default; eeb-default consults the variable eeb-default (in Emacs Lisp a function and a variable can have the same name: the "value as function" and the "value as variable" of the symbol eeb-default are kept in two different slots)

```
(defun eev-bounded ()
  (interactive)
  (setq eeb-defaults eev-bounded)
  (eeb-default))
```

and eeb-default is a list containing eev, delim1, delim2 and other information.

F3, delimiters, glyphs

gdb (here-documents, gcc, ee-once)

mention TeX (add header and footer)

deal with blocks that are executed several times with small changes

example: compilation

(alternative: here-documents, gcc, gdb, screenshot(s) for gdb)

Flashes for a while

delimiter lines should be comments

different languages have use a different syntax for comments

example: latex

11. Communication channels

The way that we saw to send commands to a shell is in two steps: first we use M-x eev in Emacs to "send" a block of commands, and then we run 'ee' at the shell to make it "receive" these commands. But there is also a way to create shells that "listen" not only to the keyboard for their input, but also to certain "communication channels"; by making Emacs send commands through these communication channels we can skip the step of going to the shell and typing 'ee' — the commands are received immediately.

The screenshot at Figure 3 shows this at work. The user has started with the cursor at the second line from the top of the screen in the Emacs window and then has typed F9 several times. Eev binds F9 to a command that operates on the current line and then moves down to the next line; if the current line starts with '●' then what comes after the '●' is considered as Lisp code and executed immediately, and the current line doesn't start

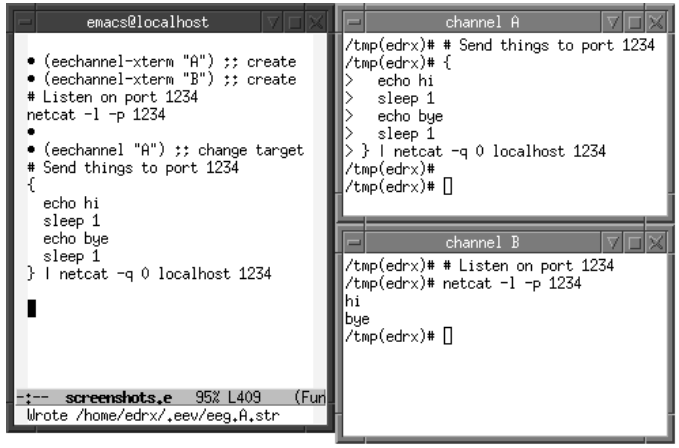

Figure 3: sending commands to two xterms using F9

with '•' then its contents are sent through the default communication channel, or though a dummy communication channel if no default was set.

The first F9 executed `(eechannel-xterm "A")`, which created an xterm with title "channel A", running a shell listening on the communication channel "A", and set the default channel to A; the second F9 created another xterm, now listening to channel B, and set the default channel to B.

The next two 'F9's sent each one one line to channel B. The first line was a shell comment ("`# Listen...`"); the second one started the program netcat, with options to make netcat "listen to the internet port 1234" and dump to standard output what it receives.

The next line had just '•'; executing the rest of it as Lisp did nothing. The following line changed the default channel to A.

In the following lines there is a small shell program that outputs "hi", then waits one second, then outputs "bye", then waits for another second, then finishes; due to the "`| netcat ...`" its output is redirected to the internet port 1234, and so we see it appearing as the output of the netcat running on channel B, with all the expected delays: one second between "hi" and "bye", and one second after "bye"; after that last one-second delay the netcat at channel A finishes receiving input (because the program between '{' and '}' ends) and it finishes its execution, closing the port 1234; the netcat at B notices that the port was closed and finishes its execution too, and both shells return to the shell prompt.

There are also ways to send whole blocks of lines at once through communication channels; see Section 16.

11.1. The Implementation of Communication Channels

Communication channels are implemented using an auxiliary script called 'eegchannel', which is written in Expect ([L90] and [L95]). If we start an xterm in the default way it starts a shell (say, /bin/bash) and interacts with it: the xterm sends to the shell as characters the keystrokes that it receives from the window manager and treats the characters that the shell sends back as being instructions to draw characters, numbers and symbols on the screen. But when we run `(eechannel-xterm "A")` Emacs creates an xterm that interacts with another program — eegchannel — instead of with a shell, and eegchannel in its turn runs a shell and interacts with it.

Eegchannel passes characters back and forth between the xterm and the shell without changing them in any way; it mostly tries to pretend that it is not there and that the xterm is communicating directly with the shell. However, when eegchannel receives a certain signal it sends to the shell a certain sequence of characters that were not sent by the xterm; it "fakes a sequence of keystrokes".

Let's see a concrete example. Suppose than Emacs was running with process id ("pid") 1000, and running `(eechannel-xterm "A")` in it made it create an xterm, which got pid 1001; that xterm ran eegchannel (pid 1002), which ran /bin/bash (pid 1003). Actually Emacs invoked xterm using this command line:

```
xterm -n "channel A" -e eegchannel A /bin/bash
```

and xterm invoked eegchannel with `eegchannel A /bin/bash`; eegchannel saw the 'A', saved its pid (1002) to the file ~/.eev/eeg.A.pid, and watched for SIGUSR1 signals; every time that it (the eegchannel) receives a SIGUSR1 it reads the contents of ~/.eev/eeg.A.str and sends that as fake input to the shell that it is controlling. So, running

```
echo 'echo $[1+2]' > ~/.eev/eeg.A.str
kill -USR1 $(cat ~/.eev/eeg.A.pid)
```

in a shell sends the string "`echo $[1+2]`" (plus a newline) "through the channel A"; what Emacs does when we type F9 on a line that does not start with '•' corresponds exactly to that.

12. Anchors

The function 'to' can be used to create hyperlink to certain positions — called "anchors" — in the current file. For example,

```
# Index:
# ⟪.first_block⟫          (to "first_block")
# ⟪.second_block⟫         (to "second_block")

#•
# ⟪first_block⟫   (to ".first_block")
echo blah
#•
# ⟪second_block⟫   (to ".second_block")
echo blah blah
#•
```

What 'to' does is simply to wrap its argument inside '⟪' and '⟫' characters and then jump to the first occurrence of the resulting string in the current file. In the (toy) example above, the line that starts with "`# ⟪.first_block⟫`" has a link that jumps to the line that starts with "`# ⟪first_block⟫`", which has a link that jumps back — the anchors and "(to ...)"s act like an index for that file.

[find-anchor]

The function 'find-anchor' works like a 'to' that first opens another file:

```
(find-anchor "~/.zshrc" "update-homepage")
```

does roughly the same as:

```
(find-fline "~/.zshrc" "⟪update-homepage⟫")
```

Actually 'find-anchor' consults a variable, 'ee-anchor-format', to see in which strings to wrap the argument. Some functions modify 'ee-anchor-format' temporarily to obtain special effects; for example, a lot of information about the packages installed in a Debian GNU system is kept in a text file called /var/lib/dpkg/info/status; `(find-status "emacs21")` opens this file and searches for the string `"\nPackage: emacs21\n"` there — that string is the header for the block with information about the package emacs21, and it tells the size of the package, description, version, whether it is installed or not, etc, in a format that is both machine-readable and human-readable.

13. E-scripts

The best short definition for eev that I've found involves some cheating, as it is a circular definition: "eev is a library that adds support for e-scripts to Emacs" — and e-scripts are files that contain chunks meant to be processed by eev's functions. Almost any file can contain parts "meant for eev": for example, a HOWTO or README file about some program will usually contain some example shell commands, and we can mark these commands and execute them with M-x eev; and if we have the habit of using eev and we are writing code in, say, C or Lua we will often put elisp hyperlinks inside comment blocks in our code. These two specific languages (and a few others) have a feature that is quite convenient for eev: they have syntactical constructs that allow comment blocks spanning several lines — for example, in Lua, where these comment blocks are delimited by "`--[[`" and "`--]]`"s, we can have a block like

```
--[[
#•
# This file: (find-fline "~/LUA/lstoindexhtml.lua")
# A test:
cd /tmp/
ls -laF | col -x \
  | lua50 ~/LUA/lstoindexhtml.lua tmp/ \
  | lua50 -e 'writefile("index.html", io.read("*a"))'
#•
--]]
```

in a Lua script, and the script will be at the same time a Lua script and an e-script.

When I started using GNU and Emacs the notion of an e-script was something quite precise to me: I was keeping notes on what I was learning and on all that I was trying to do, and I was keeping those notes in a format that was partly English (or Portuguese), partly executable things — not all of them finished, or working — after all, it was much more practical to write

```
rm -Rv ~/usrc/busybox-1.00/
tar -C ~/usrc/ -xvzf \
  $S/http/www.busybox.net/downloads/busybox-1.00.tar.gz
cd ~/usrc/busybox-1.00/
cp -iv ~/BUSYBOX/myconfig .config
make menuconfig
make       2>&1 | tee om
```

than to write

```
Unpack BusyBox's source, then run "make menuconfig"
and "make" on its main directory
```

and then have to translate that from English into machine commands every time... So, those files where I was keeping my notes contained "executable notes", or were "scripts for Emacs", and I was quite sure that everyone else around were also keeping notes in executable formats, possibly using other editors and environments (vi, maybe?) and that if I showed these people my notes and they were about some task that they were also struggling with then they would also show me *their* notes... I ended up making a system that uploaded regularly all my e-scripts (no matter how messy they were) to my home page, and writing a text — "The Eev Manifesto" ([O99]) — about sharing these executable notes.

Actually trying to define an e-script as being "a file containing executable parts, that are picked up and executed interactively" makes the concept of an e-script *very* loose.

Note that we *can* execute the Lua parts in the code above by running the Lua interpreter on it, we *can* execute the elisp one-liner with M-e in Emacs, and we *can* execute the shell commands using F3 or M-x eev; but the code will do nothing by itself — it is passive.

A piece of code containing instructions in English on how to use it is also an e-script, in a sense; but to execute these instructions we need to invoke an external entity — a human, usually ourselves — to interpret them. This is much more flexible, but also much more error-prone and slow, than just pressing a simple sequence of keys like M-e, or F9, or F3, alt-tab, e, e, enter.

14. Splitting eev.el

When I first submittted eev for inclusion in GNU Emacs, in 1999, the people at the FSF requested some changes. One of them was to split eev.el — the code at that point was all in a single Emacs Lisp file, called eev.el — into several separate source files according to functionality; at least the code for saving temporary scripts and the code for hyperlinks should be kept separate.

It turned out that that was the wrong way of splitting eev. The frontier between what is a hyperlink and what is a block of commands is blurry:

```
man foo
man -P 'less +/bar' foo
# (eev "man foo")
# (eev "man -P 'less +/bar' foo")
# (find-man "foo" "bar")
```

The two 'man' commands above can be considered as hyperlinks to a manpage, but we need to send those commands to a shell to actually open the manpage; the option -P 'less +/bar' instructs 'man' to use the program 'less' to display the manpage, and it tells 'less' to jump to the first occurrence of the string "bar" in the text, and so it is a hyperlink to a specific position in a manpage. Each of the two 'eev' lines, when executed, saves one of these 'man' commands to the temporary script file; because they contain Lisp expressions they look much more like hyperlinks than the 'man' lines. The last line, 'find-man', behaves much more like a "real" hyperlink: it opens the manpage *inside Emacs* and searches for the first occurrence of 'bar' there; but Emacs's code for displaying manpages

was tricky, and it took me a few years to figure out how to add support for pos-spec-lists to it...

So, what happens is that often a new kind of hyperlink will begin its life as a series of shell commands (another example: using 'gv –page 14 file.ps' to open a PostScript file and then jump to a certain page) and then it takes some time to make a nice hyperlink function that does the same thing; and often these functions are implemented by executing commands in external programs.

There's a much better way to split conceptually what eev does, though. Most functions in eev take a region of text (for example Emacs's own "selected region", or the extent of Lisp expression coming before the cursor) and "execute" that in some way; the kinds of regions are

Emacs's (selected) region	M-x eev, M-x eelatex (sec. 4)
last-sexp (Lisp expression at the left of the cursor)	C-x C-e, M-E (sec. 5)
sexp-eol (go to end of line, then last-sexp)	C-e C-x C-e, M-e (sec. 5)
bounded region	F3, M-x eev-bounded, M-x eelatex-bounded (sec. 10)
bounded region around *anchor*	(ee-at "*anchor*" ...) (sec. 16)
current line	F9 (sec. 11)
no text (instead use the next item in a list)	F12 (sec. 15)

Actions (can be composed):

- Saving a region or a string into a file
- Sending a signal to a process
- Executing as Lisp
- Executing immediately in a shell
- Start a debugger

other functions, like find-fline are find-node, are auxiliary functions that implement some kinds of hyperlinks

[Emacs terminology: commands]

15. Steps

[Simple examples]

[writing demos]

[hyperlinks for which no short form is known]

[producing animations and screenshots]

16. Big Modular E-scripts

A shell can be run in two modes: either interactively, by expecting lines from the user and executing them as soon as they are received[4], or by scripts: in the later case the shell already has access to the commands, and executes them in sequence as fast as possible, with no pause between one command and the next.

[4] except for multi-line commands.

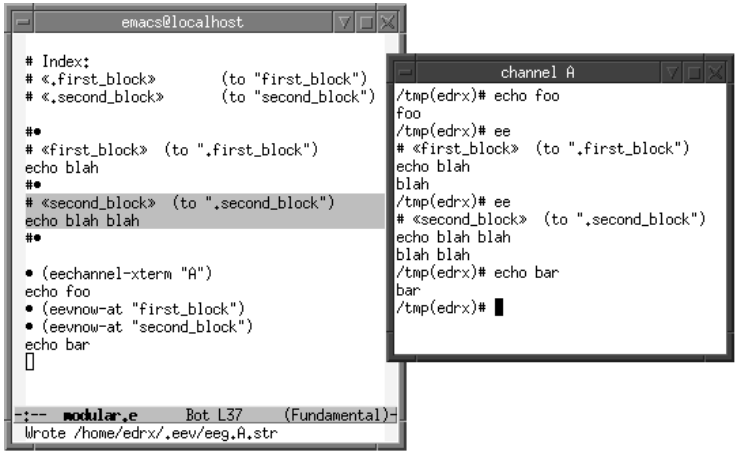

Figure 4: sending a block at once with eevnow-at

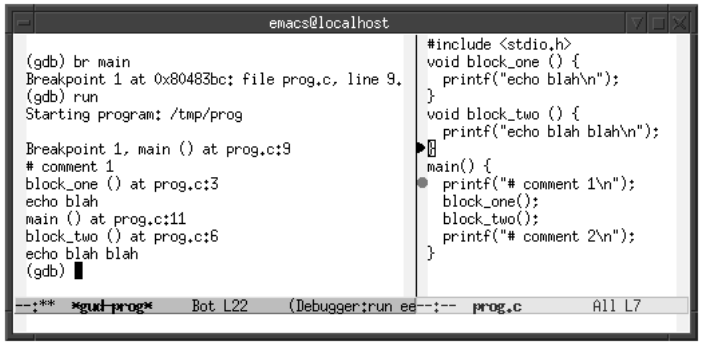

Figure 5: single-stepping through a C program

We will start with a toy example — actually the example from Section 12 with five new lines added at the end — and then and the next section we will see a real-world example that uses these ideas.

[Somewhere between a script and direct user interaction]

[No loops, no conditionals]

[Several xterms]

17. Internet Skills for Disconnected People

Suppose that we have a person P who has learned how to use a computer and now wants to learn how the internet works. That person P knows a bit of programming and can use Emacs, and sure she can use e-mail clients and web browsers by clicking around with the mouse, but she has grown tired of just using those things as black boxes; now she wants to experiment with setting up HTTP and mail servers, to understand how data packets are driven around, how firewalls can block some connections, such things.

The problem is that P has never had access to any machine besides her own, which is connected to the internet only through a modem; and also, she doesn't have any friends who are computer technicians or sysadmins, because from the little contact that she's had with these people she's got the impression that they live lifes that are almost as grey as the ones of factory workers, and she's afraid of them. To add up to all that, P has some hippie job that makes her happy but poor, so she's not going to buy a second computer, and the books she can borrow, for example, Richard Stevens' series on TCP/IP programming, just don't cut.

One of eev's intents isto make life easier for autodidacts. Can it be used to rescue people in positions like P's[5]? It was thinking on that that I created a side-project to eev called "Internet Skills for Disconnected People": it consists of e-scripts about running a second machine, called the "guest", emulated inside the "host", and making the two talk to each other via standard internet protocols, via emulated ethernet cards. Those e-scripts make heavy use of the concepts in the last section [...]

18. Availability and Resources

Eev can be downloaded from the author's homepage, http://angg.twu.net/. That page also contains lots of examples, some animations showing some of eev's features at work, a mailing list, etc.

Eev is in the middle of the process of becoming a standard part of GNU Emacs; I expect it to be integrated just after the release of GNU Emacs 22.1 in mid-2005. Eev's copyright has already been transferred to the FSF; it is distributed under the GPL license.

19. References

[L90] - Libes, D. *Expect: Curing Those Uncontrollable Fits of Interaction.* 1990. Available online from http://expect.nist.gov/ .

[L95] - Libes, D. *Exploring Expect.* O'Reilly, 1995.

[O99] - Ochs, E. *The Eev Manifesto.* http://angg.twu.net/eev-manifesto.html

[S79] - Stallman, R. *EMACS: The Extensible, Customizable Display Editor.* http://www.gnu.org/software/emacs/emacs-paper.html

[5] by the way, I created P inspired on myself; my hippie job is being a mathematician.