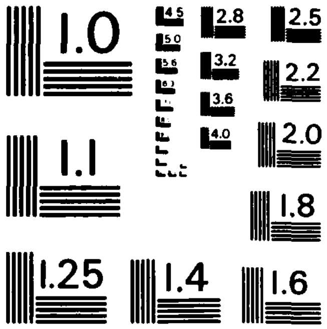
1.0
4.5
5.0
5.6
6.3
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

2

RUNNING ROSS IN AN EMACS ENVIRONMENT

D. J. McArthur

April 1985

P-7088

RUNNING ROSS IN AN EMACS ENVIRONMENT

D. J. McArthur

April 1985

Running ROSS in an Emacs Environment

1. Introduction

This paper describes a software environment in which to run ROSS, an object-oriented simulation language, developed at Rand [MCA82, KLA82]. The goal of the facility is to provide the user with tools that can increase the speed at which the user creates, tests and debugs large ROSS systems. Using ROSS alone, making permanent fixes to code was tedious. The user had to get out of ROSS, into an editor, make the changes, return to ROSS, then re-load the changed file.

The ~~Our~~ approach is to embed ROSS inside Emacs, a powerful, personalizable, screen-oriented, editor. (hence we refer to it as the "ROSS-Emacs" facility). We have tailored Emacs was tailored to provide just the tools the ROSS programmer needs. For example, the user never has to leave ROSS to make code changes, and all the lower-level details of code management—such as finding the file (and location in the file) of behaviors or functions, saving permanent changes onto disk, and loading new definitions back into ROSS—are done automatically. This facility makes ROSS a much more friendly and powerful environment for debugging.

The paper ~~The remaining sections~~ describe various facets of the ROSS-Emacs facility. It ~~We~~ begins with a brief overview, followed by descriptions of commands for editing functions and behaviors, managing Emacs windows, evaluating Lisp forms, compiling Lisp forms, and defining ROSS systems. Each section starts with a brief description, followed by a list of the Emacs commands that implement the described capabilities.

2. Starting and typing ROSS/Lisp forms

In this environment, the user is actually in the Emacs editor. ROSS is invoked automatically as a subprocess of Emacs. The ROSS session has its own buffer, called "lisp" in an Emacs window. The user types forms in the window just as he would to ROSS. When he hits a <CR> Emacs takes the form just typed, submits it to ROSS, and prints the value out on the screen. Thus, in simple interactions, the environment looks just like it did before. However, because you are also in Emacs, many new things are possible. For example, in the Lisp (ROSS) window, you may move the cursor back and forward across the current line, or go back to point to previous lines. In this way you can even re-evaluate expressions you submitted earlier to ROSS. For example, assume that 10 lines earlier in a ROSS session you typed "(setq foo 5)" and now you want foo to have the value 6. Instead of typing a new setq, you can just move the cursor in the Lisp window to the "5" in the previous setq, then change the "5" to "6" in the normal Emacs fashion. Now go to the end of the setq and hit <CR>. Since Emacs submits the expression just prior to the last <CR> to ROSS, this <CR> this will have the desired effect of re-evaluating the form and setting foo to 6. In general, the ability to edit your interactive ROSS session will make it much easier to correct errors and execute repetitive actions.

3. Emacs commands for ROSS and Lisp

A number of new Emacs commands have been defined to make things simpler when interacting with ROSS. Like normal Emacs commands, they have a name and are often associated with special keystroke combinations. But unlike other commands, they know about ROSS and Lisp structures and actions. They not only allow you to reference characters (primitive text structures) but also to reference s-expressions, function-definitions, behavior-definitions, messages and objects (primitive ROSS and Lisp structures). They not only allow you to move or delete things (primitive text operations) but also to evaluate, edit or compile them (primitive ROSS and Lisp operations).

All of the commands, whose operation will be explained below, can either be invoked by the keystrokes indicated in the bindings or by using the Emacs ESC-X prefix. For example, to evaluate the current list you can either say ESC-e or ESC-Xzap-expr; to go to the previous window you can say either ^X-n (hitting the "CTRL" and "x" keys simultaneously, followed by the "n" key), or use ESC-Xprevious-window.

4. Editing function and behavior definitions

The main reason for using ROSS in Emacs is to exploit its powerful editing facilities. To edit functions or behavior definitions use:

Function	Binding	Action
edit-function	ESC-f	Prompts for a function name, then goes to a window with the file where this function is contained, putting the cursor at the start of the definition. If Emacs doesn't know which file the function is in, it will ask you for the file name. It will only have to ask once, since it remembers the name.
edit-pointed-function	ESC-g	Like edit-function, except it will edit the function you are pointing at. You should be pointing at a function defining form (beginning with "(def". If you are not pointing directly at such a form, it will search backward for the nearest such form, and edit that function.
edit-behavior	ESC-b	Like edit-function, only it prompts for an object and a behavior pattern, then goes to the appropriate file. In specifying the behavior pattern the user need only type enough of the initial portion of the pattern to allow unique identification. However, ROSS pattern variables "+" and ">" cannot be used.
edit-pointed behavior	ESC-c	Like edit-pointed function, except it applies to behaviors. You should be pointing to a behavior definition: "(ask xxx when receiving ...)".

ROSS-Emacs commands for editing

The functions *edit-function* or *edit-pointed function* edit Lisp function definitions. The former prompts for a function name, the latter assumes you are pointing at the function definition you intend to edit (which you might be doing if you had just asked Lisp to pretty-print the function definition), so doesn't prompt for anything. In either case, if Emacs can find the function (it may prompt you for a file name if it has trouble) it will open a new window containing the file and will position the cursor at the start of the function definition. You can treat this window as in an ordinary Emacs session. In particular, you can modify the definition, add new functions, use evaluation commands (see Section 6.) and then save the file (^X^S) when you are satisfied with the changes. At this point you will probably want to go back to the Lisp window. You can do this in several ways using standard Emacs commands. These options are discussed in the next section.

To edit behaviors use the Emacs commands *edit-behavior* and *edit-pointed-behavior*. These are perfectly analogous to their counterparts for functions.

In searching through files, these four functions must make assumptions about the syntactic patterns normally associated with function and behavior definitions. Therefore, if you write your functions and behaviors in non-standard ways, these functions may fail to find them. To make your life easier follow these simple rules:

- All function and behavior definitions should be uniformly lower-case.
- Function and behavior definitions should begin in the first column of the file.
- Don't put any extra spaces between definition keywords (e.g., use "when receiving" not "when   receiving".
- Always put the function name on the same line as the word "defun" and always put the message-pattern of a behavior definition on the same line as the words "when receiving", even if this exceeds 80 characters.
- Always terminate the line that begins a function or behavior definition with a <CR> or <LF>

5. Window management

When you've finished editing functions or behaviors in a file you will probably want to get back to the window containing the Lisp session. To do this the following commands are useful:

Function	Binding	Action
switch-to-buffer	^X-b	Prompts for the name of the buffer and associates it with the current window. The old buffer associated with this window merely loses that association: it is not erased or changed in any way.
delete-buffer	^X-d	Removes the current window from the screen and gives it's space to its neighbor below (or above). You do not lose the buffer associated with the window, it just is not visible.
delete-other-windows	^X-1	Deletes all windows but the one you are in, which now takes up the whole screen. The buffers associated with the other windows are not lost.
next-window	^X-n	Switches to the window (and associated buffer) that is below the current window.
previous-window	^X-p	Switches to the window (and associated buffer) that is above the current window.

ROSS-Emacs commands for window maintenance

There are several ways to use these commands to get back to Lisp and ROSS: (i) use ^X-b and respond "lisp" to the prompt, to put Lisp in the window you are now in (thus losing the edited file); (ii) use ^X-n or ^X-p to get to the previous (higher) or next (lower) window on your screen, repeating until you are in the Lisp window; (iii) use ^X-d to delete the current (i.e., edit) window then go to the Lisp window. The latter method is often preferable if you find your screen becoming cluttered with windows associated with previously edited files. It is a good practice to get rid of unneeded windows.

6. Evaluating things

One important thing you'll want to do is evaluate pieces of code you are pointing at. If you are in a window associated with a ROSS session a <CR> can do this, but in any other window <CR> just inserts a new line (the normal Emacs action for <CR>). There are several special functions that allow you to evaluate what you are pointing to in any window.

Function	Binding	Action
zap-thing	ESC-t	Marks the "thing" currently being pointed at, submits it to Lisp where it is evaluated, then returns to the original point in the current buffer. Exactly what "thing" is pointed at is determined by the prefix arguments the user supplies to zap-thing. If no prefix is supplied then the current atom is evaluated; if one argument (^-U) is supplied then the smallest (first-level) list embedding the cursor is evaluated; if two arguments (^-U ^-U) are supplied then the second smallest (second-level) list embedding the cursor is evaluated, and so on. If more arguments are supplied than there are embedding lists, an error is signalled. Zap-thing is the most general evaluation function.
zap-expr	ESC-e	Marks the list s-expression currently being pointed at, submits it to Lisp where it is evaluated, then returns to the original point in the current buffer. Equivalent to ^-U ESC-t.
zap-message	ESC-m	Marks the region around the cursor that begins with any message form "(ask" or "(tell", submits it to Lisp for evaluation, then returns.
zap-defun	ESC-d	Marks the current function defining form, submits it to Lisp, and stays there.
zap-defun-stay	ESC-s	Marks the current function defining form, submits it to Lisp, then returns to the original point in the current buffer.
zap-string	ESC-y	Prompts the user for a Lisp or ROSS form, then submits the form to ROSS, where its value is printed, and finally returns to the original point in the current buffer.

ROSS-Emacs commands for evaluating

In general these functions can be used in any window (not just "lisp"), providing Emacs understands that the code in the window is Lisp or ROSS code.

- It will understand this only if the file has the extention type ".l". So make sure you use this extention if you want to be able evaluate, edit or compile code via Emacs.

When evaluating things just make sure the cursor is in the window containing the code you want to evaluate and is pointing to the exact Lisp or ROSS expression you are interested in. Generally, this means pointing at a top-level element in the target list-structure. If you point to an element in an embedded list, then that list, not the inclusive target list will get evaluated.

Once you are pointing at the right expression, just give the correct keystroke sequence. The result of evaluation should appear in the Lisp window, regardless of whether the evaluated code was in that window or some other one.

7. Compiling code

The current facility enables the user to interactively compile function (not behavior) definitions as well as interactively modify interpreted ones. This can be especially useful when speed is essential.

Function	Binding	Action
compile-function	ESC-i	Takes the function definition currently pointed at, submits it to the Lisp compiler, liszt, then loads the resulting ".o" file into Lisp, finally returning you to where you started.
compile-file	ESC-o	Like compile-function, except the whole file currently being pointed at is written, recompiled, and loaded into Lisp.

ROSS-Emacs commands to compile code

During compilation, a trace of liszt's activity will be appended to the Lisp window so that the user can diagnose errors in compilation.

8. Defining ROSS systems and searching them

As a ROSS system gets larger you may lose track of where various functions, objects and behaviors are. This may make it difficult to rapidly access pieces of code, for example when *edit-function* (see Section 4) asks you where a function is. There is a simple facility that allows Emacs to construct a "map" of your system, obviating the need for you to have one.

Function	Binding	Action
load-ross-system	--	Prompts you for a file name, which should contain the names of each of the files in your ROSS system. The names should be one to a line, beginning in the first column. If you envision loading the system from various directories, each file should be specified by full absolute pathnames. A ROSS system must be loaded using this function before any of the following functions will work.
re-tag-ross-system	--	Goes through each of the files in a loaded ROSS system and remembers the location of each function, macro, variable, object, and behavior in the system. After this, *edit-function* and *edit-behavior* will never have to ask you where a function or behavior is located.
whereis-function	--	Prompts for the name of a function and returns the name of the file where it is located. Will work for defmacros, and defvars as well as defuns. Returns its answer much more rapidly if it has rememebered (*e.g.*, via re-tag-ross-system) where the function is.
whereis-behavior	--	Like *whereis-function*, except it prompts for an object and behavior.

ROSS-Emacs commands for defining and searching a ROSS system

The recommended procedure is to re-tag your ROSS system periodically. This task may take a minute or two, but it greatly speeds up the operation of *edit-function, edit-behavior, whereis-function,* and *whereis-behavior*. Re-tag whenever you have made substantial additions to the system, or have moved things from one file to another. If you don't tag your system, and then ask Emacs where a certain function is, it may have to do a search through all your system files to find it. When you get back from lunch it might be done.

9. Moving around structures

In addition to the normal Emacs commands for moving forward and backward by character or by line, there are some commands that are especially useful for moving around Lisp or ROSS expressions and manipulating them.

Function	Binding	Action
forward-word	--	Move dot forward to the end of a word. If not currently in the middle of a word, skip all intervening punctuation. Then skip over the word, leaving dot positioned after the last character of the word. A word is a sequence of alphanumerics. *Note:* this is normally bound to ESC-f in Emacs. If you envision using it frequently, rebind it to something.
backward-word	--	If in the middle of a word, go to the beginning of that word, otherwise go to the beginning of the preceding word. A word is a sequence of alphanumerics. *Note:* this is normally bound to ESC-b in Emacs. If you envision using it frequently, rebind it to something.
delete-previous-word	--	If not in the middle of a word, delete characters backwards (to the left) until a word is found. Then delete the word to the left of dot. A word is a sequence of alphanumerics. *Note:* this is normally bound to ESC-h in Emacs. If you envision using it frequently, rebind it to something.
delete-next-word	ESC-D	Delete characters forward from dot until the next end of a word. If dot is currently not in a word, all punctuation up to the beginning of the word is deleted as well as the word.
backward-higher-paren	ESC-(	Moves backwards to a higher opening parenthesis that is unmatched by any closing parenthesis before the current position of the cursor. Exactly which such parenthesis is determined by the number of prefix arguments given to the function. If zero or one argument is given (^-U) the cursor goes to the first unmatched parenthesis; that is, the "(" of the first-level list embedding the cursor. If two arguments are given (^-U ^-U) the cursor goes to the opening parenthesis of the second-level list embedding the cursor, and so on. If more prefixes are given than there are lists embedding the cursor, the cursor goes to the opening parenthesis of the top-level list, and an error is signaled.
forward-higher-paren	ESC-)	Analogous to backward-higher-paren, but moves forward in the file.
newline-and-indent	LINEFEED	Insert a newline, just as typing <CR> does, but then insert enough tabs and spaces so that the newly created line has the same indentation as the old one had. This is quite useful when you're typing in a block of program text, all at the same indentation level.
re-indent-line	TAB	Causes the line you are on to shift left or right so that the expressions on it will be aligned correctly with the ones on the previous line. In other words, it pretty-prints the line.
mark-list	--	Causes the list you are pointing at to be marked (*e.g.*, so it can be subsequently killed, copied, etc). Exactly which of the embedded lists gets marked is determined by the number of prefix arguments you supply, according to the logic of ***backward-higher-paren***.

ROSS-Emacs commands for moving around structures

10. Retrieving names of entities

There are a few commands which return the names of entities currently being pointed at. These are mainly used by other functions, hence have no key bindings.

Function	Binding	Action
current-function	--	If pointing at a function defining form (i.e., one beginning with "(def"), returns the function name. If not, returns the name of the nearest function defining form, searching back up through the buffer.
current-object	--	Returns the name of the object sent the message now being pointed at or nearest the cursor.
current-behavior	--	Returns the pattern of the behavior definition ("when receiving") now being pointed at or nearest the cursor.

ROSS-Emacs commands for retrieving names

11. Help and Documentation

There are a couple of functions for on-line help with ROSS in general and the ROSS-Emacs facility in particular.

Function	Binding	Action
ross-emacs-help	ESC-h	Prints information on ROSS-Emacs commands in the Help window.
ross-manual	--	Prompts user for the name of a ROSS command, then prints the ROSS manual documentation for that command in a window. The command should be entered in upper case. The whole command need not be entered, just enough of the beginning to provide unique identification.

ROSS-Emacs commands for help and documentation

12. Getting started

To run the facility you need to have an Emacs initialization file that loads the ROSS-Emacs utilities. A reasonable initialization file to begin with is /a/ftpross/emacs_pro. Copy it into a file named ".emacs_pro" (note the ".") in your home directory. The actual mlisp (mocklisp) code that implements the ROSS-Emacs facility can be found in /a/ftpross/ross-emacs.ml on rand-unix. Finally, /a/ftpross/ross-emacs is a simple shell file for initiating ROSS under Emacs. If you execute this file you will find yourself in an Emacs session, with ROSS running in one window.

To use the ROSS-Emacs facility, the user must have access to an extensible version of Emacs, such as Unix Emacs by James Gosling. Unix Emacs is available from Unipress Software, for a fee. You might also be able to get it for free from CMU, if you're lucky.

13. References

[KLA82] SWIRL: SIMULATING WARFARE IN THE ROSS LANGUAGE, Rand N-1885-AF, September 1982.

[MCA82] THE ROSS LANGUAGE MANUAL, Rand N-1854-AF, September 1982.