User's Guide to Eshell: The Emacs Shell

John Wiegley

Table of Contents

1 What is Eshell?

Eshell is a *command shell* written in Emacs Lisp. Everything it does, it uses Emacs' facilities to do. This means that Eshell is as portable as Emacs itself. It also means that cooperation with Lisp code is natural and seamless.

What is a command shell? To properly understand the role of a shell, it's necessary to visualize what a computer does for you. Basically, a computer is a tool; in order to use that tool, you must tell it what to do—or give it "commands." These commands take many forms, such as clicking with a mouse on certain parts of the screen. But that is only one form of command input.

By far the most versatile way to express what you want the computer to do is by using an abbreviated language called *script*. In script, instead of telling the computer, "list my files, please", one writes a standard abbreviated command word—'`ls`'. Typing '`ls`' in a command shell is a script way of telling the computer to list your files.[1]

The real flexibility of this approach is apparent only when you realize that there are many, many different ways to list files. Perhaps you want them sorted by name, sorted by date, in reverse order, or grouped by type. Most graphical browsers have simple ways to express this. But what about showing only a few files, or only files that meet a certain criteria? In very complex and specific situations, the request becomes too difficult to express using a mouse or pointing device. It is just these kinds of requests that are easily solved using a command shell.

For example, what if you want to list every Word file on your hard drive, larger than 100 kilobytes in size, and which hasn't been looked at in over six months? That is a good candidate list for deletion, when you go to clean up your hard drive. But have you ever tried asking your computer for such a list? There is no way to do it! At least, not without using a command shell.

The role of a command shell is to give you more control over what your computer does for you. Not everyone needs this amount of control, and it does come at a cost: Learning the necessary script commands to express what you want done. A complicated query, such as the example above, takes time to learn. But if you find yourself using your computer frequently enough, it is more than worthwhile in the long run. Any tool you use often deserves the time spent learning to master it.[2]

As of Emacs 21, Eshell is part of the standard Emacs distribution.

1.1 Contributors to Eshell

Contributions to Eshell are welcome. I have limited time to work on this project, but I will gladly add any code you contribute to me to this package.

The following persons have made contributions to Eshell.

- Eli Zaretskii made it possible for Eshell to run without requiring asynchronous subprocess support. This is important for MS-DOS, which does not have such support.

[1] This is comparable to viewing the contents of a folder using a graphical display.

[2] For the understandably curious, here is what that command looks like: But don't let it fool you; once you know what's going on, it's easier than it looks: `ls -lt **/*.doc(Lk+50aM+5)`.

- Miles Bader contributed many fixes during the port to Emacs 21.
- Stefan Monnier fixed the things which bothered him, which of course made things better for all.
- Gerd Moellmann also helped to contribute bug fixes during the initial integration with Emacs 21.
- Alex Schroeder contributed code for interactively querying the user before overwriting files.
- Sudish Joseph helped with some XEmacs compatibility issues.

Apart from these, a lot of people have sent suggestions, ideas, requests, bug reports and encouragement. Thanks a lot! Without you there would be no new releases of Eshell.

2 Installation

As mentioned above, Eshell comes preinstalled as of Emacs 21. If you're using Emacs 20.4 or later, or XEmacs 21, you can download the most recent version of Eshell from `http://www.gci-net.com/users/j/johnw/Emacs/packages/eshell.tar.gz`.

However, if you are using Emacs 21, you may skip this section.

2.1 Short Form

Here's exactly what to do, with no explanation why:

1. '`M-x load-file RET eshell-auto.el RET`'.
2. '`ESC : (add-to-list 'load-path "<path where Eshell resides>") RET`'.
3. '`ESC : (add-to-list 'load-path "<path where Pcomplete resides>") RET`'.
4. '`M-x eshell RET`'.

   You should see a version banner displayed.
5. '`ls RET`'.

   Confirm that you see a file listing.
6. '`eshell-test RET`'.

   Confirm that everything runs correctly. Use *`M-x eshell-report-bug`* if not.
7. '`cd ${dirname (locate-library "eshell-auto")} RET`'.
8. '`find-file Makefile RET`'.
9. Edit the Makefile to reflect your site.
10. '`M-x eshell RET`'.
11. '`make install RET`'.
12. '`find-file $user-init-file RET`'.
13. Add the following lines to your '`.emacs`' file:

    ```
    (add-to-list 'load-path "<directory where you install Eshell>")
    (load "eshell-auto")
    ```
14. '`M-x eshell RET`'.
15. '`customize-option #'eshell-modules-list RET`'.
16. Select the extension modules you prefer.
17. Restart Emacs!
18. '`M-x info RET m Eshell RET`'.

    Read the manual and enjoy!

2.2 Long Form

1. Before building and installing Eshell, it is important to test that it will work properly on your system. To do this, first load the file '`eshell-auto`', which will define certain autoloads required to run Eshell. This can be done using the command *`M-x load-file`*, and then selecting the file '`eshell-auto.el`'.

2. In order for Emacs to find Eshell's files, the Eshell directory must be added to the `load-path` variable. This can be done within Emacs by typing:

   ```
   ESC : (add-to-list 'load-path "<path where Eshell resides>") RET
   ESC : (add-to-list 'load-path "<path where Pcomplete resides>") RET
   ```

3. Start Eshell from the distributed sources, using default settings, by typing *`M-x eshell`*.
4. Verify that Eshell is functional by typing `ls` followed by ⟨RET⟩. You should have already seen a version banner announcing the version number of this release, followed by a prompt.
5. Run the test suite by typing `eshell-test` followed by ⟨RET⟩ in the Eshell buffer. It is important that Emacs be left alone while the tests are running, since extraneous command input may cause some of the tests to fail (they were never intended to run in the background). If all of the tests pass, Eshell should work just fine on your system. If any of the tests fail, please send e-mail to the Eshell maintainer using the command *`M-x eshell-report-bug`*.
6. Edit the file '`Makefile`' in the directory containing the Eshell sources to reflect the location of certain Emacs directories at your site. The only things you really have to change are the definitions of `lispdir` and `infodir`. The elisp files will be copied to `lispdir`, and the info file to `infodir`.
7. Type *`make install`* in the directory containing the Eshell sources. This will byte-compile all of the '`*.el`' files and copy both the source and compiled versions to the directories specified in the previous step. It will also copy the info file, and add a corresponding entry to your '`dir`' file—-if the program `install-info` can be found on your system.

   If you only want to create the compiled elisp files, but don't want to install them, you can type just *`make`* instead.
8. Add the directory into which Eshell was installed to your `load-path` variable. This can be done by adding the following line to your '`.emacs`' file:

   ```
   (add-to-list 'load-path "/usr/local/share/emacs/site-lisp/eshell")
   ```

   The actual directory on your system may differ.
9. To install Eshell privately, edit your '`.emacs`' file; to install Eshell site-wide, edit the file '`site-start.el`' in your '`site-lisp`' directory (usually '`/usr/local/share/emacs/site-lisp`' or something similar). In either case enter the following line into the appropriate file:

   ```
   (load "eshell-auto")
   ```

10. Restart Emacs. After restarting, customize the variable `eshell-modules-list`. This variable selects which Eshell extension modules you want to use. You will find documentation on each of those modules in the Info manual.

If you have TEX installed at your site, you can make a typeset manual from '`eshell.texi`'.

1. Run TEX by typing *`texi2dvi eshell.texi`*. (With Emacs 21.1 or later, typing *`make eshell.dvi`* in the '`man/`' subdirectory of the Emacs source distribution will do that.)
2. Convert the resulting device independent file '`eshell.dvi`' to a form which your printer can output and print it. If you have a postscript printer, there is a program, `dvi2ps`,

which does that; there is also a program which comes together with TEX, `dvips`, which you can use. For other printers, use a suitable DVI driver, e.g., `dvilj4` for LaserJet-compatible printers.

3 Basic overview

A command shell is a means of entering verbally-formed commands. This is really all that it does, and every feature described in this manual is a means to that end. Therefore, it's important to take firm hold on exactly what a command is, and how it fits in the overall picture of things.

3.1 Commands verbs

Commands are expressed using *script*, a special shorthand language computers can understand with no trouble. Script is an extremely simple language; oddly enough, this is what makes it look so complicated! Whereas normal languages use a variety of embellishments, the form of a script command is always:

verb [*arguments*]

The verb expresses what you want your computer to do. There are a fixed number of verbs, although this number is usually quite large. On the author's computer, it reaches almost 1400 in number. But of course, only a handful of these are really necessary.

Sometimes, the verb is all that's written. A verb is always a single word, usually related to the task it performs. `reboot` is a good example. Entering that on GNU/Linux will reboot the computer—assuming you have sufficient privileges.

Other verbs require more information. These are usually very capable verbs, and must be told specifically what to do. The extra information is given in the form of *arguments*. For example, the `echo` verb prints back whatever arguments you type. It requires these arguments to know what to echo. A proper use of `echo` looks like this:

```
echo This is an example of using echo!
```

This script command causes the computer to echo back: "This is an example of using echo!"

Although command verbs are always simple words, like `reboot` or `echo`, arguments may have a wide variety of forms. There are textual arguments, numerical arguments—even Lisp arguments. Distinguishing these different types of arguments requires special typing, for the computer to know exactly what you mean.

3.2 Command arguments

Eshell recognizes several different kinds of command arguments:

1. Strings (also called textual arguments)
2. Numbers (floating point or integer)
3. Lisp lists
4. Lisp symbols
5. Emacs buffers
6. Emacs process handles

Most users need to worry only about the first two. The third, Lisp lists, occur very frequently, but almost always behind the scenes.

Strings are the most common type of argument, and consist of nearly any character. Special characters—those used by Eshell specifically—must be preceded by a backslash (‘\’). When in doubt, it is safe to add backslashes anywhere and everywhere.

Here is a more complicated `echo` example:

```
echo A\ Multi-word\ Argument\ With\ A\ \$\ dollar
```

Beyond this, things get a bit more complicated. While not beyond the reach of someone wishing to learn, it is definitely beyond the scope of this manual to present it all in a simplistic manner. Get comfortable with Eshell as a basic command invocation tool, and learn more about the commands on your system; then come back when it all sits more familiarly on your mind. Have fun!

4 Commands

Essentially, a command shell is all about invoking commands—and everything that entails. So understanding how Eshell invokes commands is the key to comprehending how it all works.

4.1 Invocation

Unlike regular system shells, Eshell never invokes kernel functions directly, such as `exec(3)`. Instead, it uses the Lisp functions available in the Emacs Lisp library. It does this by transforming the command you specify into a callable Lisp form.[1]

This transformation, from the string of text typed at the command prompt, to the ultimate invocation of either a Lisp function or external command, follows these steps:

1. Parse the command string into separate arguments.
2.

4.2 Completion

4.3 Aliases

4.4 History

4.5 Scripts

[1] To see the Lisp form that will be invoked, type: ‘`eshell-parse-command "echo hello"`’

5 Arguments

5.1 The Parser

5.2 Variables

5.3 Substitution

5.4 Globbing

5.5 Predicates

6 Input/Output

7 Process control

8 Extension modules

8.1 Writing a module

8.2 Module testing

8.3 Directory handling

8.4 Key rebinding

8.5 Smart scrolling

8.6 Terminal emulation

8.7 Built-in UNIX commands

9 Extras and Goodies

10 Bugs and ideas

If you find a bug or misfeature, don't hesitate to let me know! Send email to `johnw@gnu.org`. Feature requests should also be sent there. I prefer discussing one thing at a time. If you find several unrelated bugs, please report them separately.

If you have ideas for improvements, or if you have written some extensions to this package, I would like to hear from you. I hope you find this package useful!

10.1 Known problems

Below is complete list of known problems with Eshell version 2.4.2, which is the version included with Emacs 21.2.

Differentiate between aliases and functions
: Allow for a bash-compatible syntax, such as:

```
alias arg=blah
function arg () { blah $* }
```

'`for i in 1 2 3 { grep -q a b && *echo has it } | wc -l`' outputs result after prompt
: In fact, piping to a process from a looping construct doesn't work in general. If I change the call to `eshell-copy-handles` in `eshell-rewrite-for-command` to use `eshell-protect`, it seems to work, but the output occurs after the prompt is displayed. The whole structured command thing is too complicated at present.

Error with `bc` in `eshell-test`
: On some XEmacs system, the subprocess interaction test fails inexplicably, although `bc` works fine at the command prompt.

Eshell does not delete '`*Help*`' buffers in XEmacs 21.1.8+
: In XEmacs 21.1.8, the '`*Help*`' buffer has been renamed such that multiple instances of the '`*Help*`' buffer can exist.

Pcomplete sometimes gets stuck
: You press ⟨TAB⟩, but no completions appear, even though the directory has matching files. This behavior is rare.

'`grep python $<rpm -qa>`' doesn't work, but using '`*grep`' does
: This happens because the `grep` Lisp function returns immediately, and then the asynchronous `grep` process expects to examine the temporary file, which has since been deleted.

Problem with C-r repeating text
: If the text *before point* reads `"./run"`, and you type *`C-r r u n`*, it will repeat the line for every character typed.

Backspace doesn't scroll back after continuing (in smart mode)
: Hitting space during a process invocation, such as `make`, will cause it to track the bottom of the output; but backspace no longer scrolls back.

It's not possible to fully `unload-feature` Eshell

Menu support was removed, but never put back

Using C-p and C-n with rebind gets into a locked state
: This happened a few times in Emacs 21, but has been unreproducable since.

If an interactive process is currently running, *M-!* doesn't work

Use a timer instead of `sleep-for` when killing child processes

Piping to a Lisp function is not supported
: Make it so that the Lisp command on the right of the pipe is repeatedly called with the input strings as arguments. This will require changing `eshell-do-pipeline` to handle non-process targets.

Input redirection is not supported
: See the above entry.

Problem running `less` without arguments on Windows
: The result in the Eshell buffer is:

```
Spawning child process: invalid argument
```

: Also a new `less` buffer was created with nothing in it... (presumably this holds the output of `less`).

: If `less.exe` is invoked from the Eshell command line, the expected output is written to the buffer.

: Note that this happens on NT-Emacs 20.6.1 on Windows 2000. The term.el package and the supplied shell both use the `cmdproxy` program for running shells.

Implement '`-r`', '`-n`' and '`-s`' switches for `cp`

Make *M-5 M-x eshell* switch to "*eshell<5>*", creating if need be

'`mv` *dir file*`.tar`' does not remove directories
: This is because the tar option –remove-files doesn't do so. Should it be Eshell's job?

Bind `standard-output` and `standard-error`
: This would be so that if a Lisp function calls `print`, everything will happen as it should (albeit slowly).

When an extension module fails to load, '`cd /`' gives a Lisp error

If a globbing pattern returns one match, should it be a list?

Make sure syntax table is correct in Eshell mode
: So that *M-DEL* acts in a predictable manner, etc.

Allow all Eshell buffers to share the same history and list-dir

There is a problem with script commands that output to '`/dev/null`'
: If a script file, somewhere in the middle, uses '`> /dev/null`', output from all subsequent commands is swallowed.

Split up parsing of text after '`$`' in '`esh-var.el`'
: Make it similar to the way that '`esh-arg.el`' is structured. Then add parsing of '`$[?\n]`'.

After pressing *M-RET*, redisplay before running the next command

Argument predicates and modifiers should work anywhere in a path

```
/usr/local/src/editors/vim $ vi **/CVS(/)/Root(.)
Invalid regexp: "Unmatched ( or \\("
```

With `zsh`, the glob above expands to all files named '`Root`' in directories named '`CVS`'.

Typing '`echo ${locate locate}/bin<TAB>`' results in a Lisp error

Perhaps it should interpolate all permutations, and make that the globbing result, since otherwise hitting return here will result in "(list of filenames)/bin", which is never valuable. Thus, one could `cat` only C backup files by using '`ls ${identity *.c}~`'. In that case, having an alias command name `glob` for `identity` would be useful.

Once symbolic mode is supported for `umask`, implement `chmod` in Lisp

Create `eshell-expand-file-name`

This would use a data table to transform things such as '`~+`', '`...`', etc.

Abstract '`em-smart.el`' into '`smart-scroll.el`'

It only really needs: to be hooked onto the output filter and the pre-command hook, and to have the input-end and input-start markers. And to know whether the last output group was "successful."

Allow for fully persisting the state of Eshell

This would include: variables, history, buffer, input, dir stack, etc.

Implement D as an argument predicate

It means that files beginning with a dot should be included in the glob match.

A comma in a predicate list should mean OR

At the moment, this is not supported.

Error if a glob doesn't expand due to a predicate

An error should be generated only if `eshell-error-if-no-glob` is non-nil.

'`(+ RET SPC TAB`' does not cause `indent-according-to-mode` to occur

Create `eshell-auto-accumulate-list`

This is a list of commands for which, if the user presses *RET*, the text is staged as the next Eshell command, rather than being sent to the current interactive process.

Display file and line number if an error occurs in a script

`wait` doesn't work with process ids at the moment

Enable the direct-to-process input code in '`em-term.el`'

Problem with repeating '`echo ${find /tmp}`'

With smart display active, if *RET* is held down, after a while it can't keep up anymore and starts outputting blank lines. It only happens if an asynchronous process is involved. . .

I think the problem is that `eshell-send-input` is resetting the input target location, so that if the asynchronous process is not done by the time the next *RET* is received, the input processor thinks that the input is meant for the process;

which, when smart display is enabled, will be the text of the last command line! That is a bug in itself.

In holding down *RET* while an asynchronous process is running, there will be a point in between termination of the process, and the running of `eshell-post-command-hook`, which would cause `eshell-send-input` to call `eshell-copy-old-input`, and then process that text as a command to be run after the process. Perhaps there should be a way of killing pending input between the death of the process, and the `post-command-hook`.

Allow for a more aggressive smart display mode
: Perhaps toggled by a command, that makes each output block a smart display block.

Create more meta variables

'`$!`' The reason for the failure of the last disk command, or the text of the last Lisp error.

'`$=`' A special associate array, which can take references of the form '`$=[REGEXP]`'. It indexes into the directory ring.

Eshell scripts can't execute in the background

Support zsh's "Parameter Expansion" syntax, i.e. '`${`*name*`:-`*val*`}`'

Write an `info` alias that can take arguments
: So that the user can enter '`info chmod`', for example.

Create a mode `eshell-browse`
: It would treat the Eshell buffer as a outline. Collapsing the outline hides all of the output text. Collapsing again would show only the first command run in each directory

Allow other revisions of a file to be referenced using '`file{rev}`'
: This would be expanded by `eshell-expand-file-name` (see above).

Print "You have new mail" when the "Mail" icon is turned on

Implement *M*`-|` for Eshell

Implement input redirection
: If it's a Lisp function, input redirection implies `xargs` (in a way...). If input redirection is added, also update the `file-name-quote-list`, and the delimiter list.

Allow '`#<`*word arg*`>`' as a generic syntax
: With the handling of *word* specified by an `eshell-special-alist`.

In `eshell-veal-using-options`, allow a `:complete` tag
: It would be used to provide completion rules for that command. Then the macro will automagically define the completion function.

For `eshell-command-on-region`, apply redirections to the result
: So that '`+ > 'blah`' would cause the result of the `+` (using input from the current region) to be inserting into the symbol `blah`.

If an external command is being invoked, the input is sent as standard input, as if a '`cat <region> |`' had been invoked.

If a Lisp command, or an alias, is invoked, then if the line has no newline characters, it is divided by whitespace and passed as arguments to the Lisp function. Otherwise, it is divided at the newline characters. Thus, invoking `+` on a series of numbers will add them; `min` would display the smallest figure, etc.

Write `eshell-script-mode` as a minor mode
: It would provide syntax, abbrev, highlighting and indenting support like `emacs-lisp-mode` and `shell-mode`.

In the history mechanism, finish the `bash`-style support
: This means '`!n`', '`!#`', '`!:%`', and '`!:1-`' as separate from '`!:1*`'.

Support the -n command line option for `history`

Implement `fc` in Lisp

Specifying a frame as a redirection target should imply the currently active window's buffer

Implement '`>`*func-or-func-list*'
: This would allow for an "output translators", that take a function to modify output with, and a target. Devise a syntax that works well with pipes, and can accomodate multiple functions (i.e., '`>'(upcase regexp-quote)`' or '`>'upcase`').

Allow Eshell to read/write to/from standard input and output
: This would be optional, rather than always using the Eshell buffer. This would allow it to be run from the command line (perhaps).

Write a `help` command
: It would call subcommands with '`--help`', or '`-h`' or '`/?`', as appropriate.

Implement `stty` in Lisp

Support `rc`'s matching operator, e.g. '`~ (`*list*`)` *regexp*'

Implement `bg` and `fg` as editors of `eshell-process-list`
: Using `bg` on a process that is already in the background does nothing. Specifying redirection targets replaces (or adds) to the list current being used.

Have `jobs` print only the processes for the current shell

How can Eshell learn if a background process has requested input?

Support '`2>&1`' and '`>&`' and '`2>`' and '`|&`'
: The syntax table for parsing these should be customizable, such that the user could change it to use rc syntax: '`>[2=1]`'.

Allow '`$_[-1]`', which would indicate the last element of the array

Make '`$x[*]`' equal to listing out the full contents of '`x`'
: Return them as a list, so that '`$_[*]`' is all the arguments of the last command.

Copy ANSI code handling from '`term.el`' into '`em-term.el`'
: Make it possible for the user to send char-by-char to the underlying process. Ultimately, I should be able to move away from using term.el altogether, since everything but the ANSI code handling is already part of Eshell. Then, things

would work correctly on MS-Windows as well (which doesn't have '`/bin/sh`', although '`term.el`' tries to use it).

Make the shell spawning commands be visual
: That is, make (`su`, `bash`, `telnet`, `rlogin`, `rsh`, etc.) be part of `eshell-visual-commands`. The only exception is if the shell is being used to invoke a single command. Then, the behavior should be based on what that command is.

Create a smart viewing command named `open`
: This would search for some way to open its argument (similar to opening a file in the Windows Explorer).

Alias `read` to be the same as `open`, only read-only

Write a `tail` command which uses `view-file`
: It would move point to the end of the buffer, and then turns on auto-revert mode in that buffer at frequent intervals—and a `head` alias which assums an upper limit of `eshell-maximum-line-length` characters per line.

Make `dgrep` load `dired`, mark everything, then invoke `dired-do-search`

Write mesh.c
: This would run Emacs with the appropriate arguments to invoke Eshell only. That way, it could be listed as a login shell.

Use an intangible `PS2` string for multi-line input prompts

Auto-detect when a command is visual, by checking `TERMCAP` usage

The first keypress after *`M-x watson`* triggers 'eshell-send-input'

Make `/` electric
: So that it automatically expands and corrects pathnames. Or make pathname completion for Pcomplete auto-expand '`/u/i/std<TAB>`' to '`/usr/include/std<TAB>`'.

Write the `pushd` stack to disk along with `last-dir-ring`

Add options to `eshell/cat` which would allow it to sort and uniq

Implement `wc` in Lisp
: Add support for counting sentences, paragraphs, pages, etc.

Once piping is added, implement `sort` and `uniq` in Lisp

Implement `touch` in Lisp

Implement `comm` in Lisp

Implement an `epatch` command in Lisp
: This would call `ediff-patch-file`, or `ediff-patch-buffer`, depending on its argument.

Have an option such that '`ls -l`' generates a dired buffer

Write a version of `xargs` based on command rewriting
: That is, '`find X | xargs Y`' would be indicated using '`Y ${find X}`'. Maybe `eshell-do-pipelines` could be changed to perform this on-thy-fly rewriting.

Write an alias for `less` that brings up a `view-mode` buffer
: Such that the user can press ⟨SPC⟩ and ⟨DEL⟩, and then ⟨q⟩ to return to Eshell. It would be equivalent to: '`X > #<buffer Y>; view-buffer #<buffer Y>`'.

Make `eshell-mode` as much a full citizen as `shell-mode`
: Everywhere in Emacs where `shell-mode` is specially noticed, add `eshell-mode` there.

Permit the umask to be selectively set on a `cp` target

Problem using *`M-x eshell`* after using `eshell-command`
: If the first thing that I do after entering Emacs is to run `eshell-command` and invoke `ls`, and then use *`M-x eshell`*, it doesn't display anything.

`M-RET` during a long command (using smart display) doesn't work
: Since it keeps the cursor up where the command was invoked.

Concept Index

Function and Variable Index

(Index is nonexistent)

Key Index

(Index is nonexistent)