Programming in Qed

A Tutorial

Robert Pike, Sean Jensen (ed.)

1. INTRODUCTION

Qed is a programmable text editor, intended for use primarily by programmers. For the average user, **Qed**'s power will likely be unnecessary, even troublesome, and its use is discouraged. For a knowledgeable user who is willing to learn how to use it properly, however, **Qed** is a powerful tool, both when used as an editor or as a (rather idiosyncratic and low-level) programming language.

Qed is very nearly a superset of the University of Toronto **UNIX** version 6 editor, **Ed**, which is itself a strict superset of standard **Ed**. This document assumes considerable familiarity with the standard editor, and some acquaintance with the new features of the U. of T. version. The features of U. of T. **Ed** not found in regular version 6 **Ed** include: the `*` address separator, two character error messages (a query followed by a character which indicates the error, e.g. `?s` for a failed substitution), a simple undo `u` command, and a join `j` command of somewhat greater generality than the *PWB* version. (The new features used in this tutorial will be briefly explained as they turn up.) There is an add-on document for U. of T. **Ed** which describes the enhancements and modifications, at user level, made at U. of T.

This tutorial is not a complete description of the capabilities of **Qed**. Rather, it is an attempt to familiarize the programmer with the general ways in which **Qed** operates, emphasizing the programming techniques which clarify and simplify its use. A full description of the commands and features of **Qed** may be found in the manual section, `qed(1)`.

Before beginning the discussion of **Qed**, some warning should be given. **UNIX Ed** is closely based on a version of **Qed**, running under the **GCOS** operating system, which was written by Dennis Ritchie and Ken Thompson. When Dennis Ritchie wrote **Ed**, he removed many of the features, including most of the programming capabilities, but left in most of the text editing power. Although the **Qed** described here is significantly more complex and powerful than **Ed** (and quite unrelated to the **GCOS Qed**), its increase in power is not proportionate to its increase in complexity. In short, **Ed** is a very powerful editor, and for general editing jobs quite sufficient. **Qed** simplifies some more complicated tasks, and its multifile capability and programmability make many things possible which cannot be done in **Ed**, but to use it well requires a fairly thorough understanding of its operation, which is fairly intricate.

Qed has several drawbacks that should be admitted early. **Qed** programs can be difficult to read, even if done carefully, since it operates at about the level of a particularly cryptic assembler. A careless user can easily damage files using **Qed** incorrectly, but it is much harder to accidentally cause trouble with the U. of T. **Qed** than earlier ones, as the implementors worked very hard at safeguarding. The safeguards are strong enough that occasional users who desire a particular **Qed** feature for one editing job need not feel in danger of making a serious mistake. **Qed**'s power can lead the user astray; it has far more power than is needed for most editing jobs. As an illustrative but rather artificial example, consider the problem of reversing the lines of a file, so that the last line appears at the top, and the first at the bottom, with the contents of the lines unchanged. At first, this may seem a problem for **Qed** if it is only to be done once or twice (if it is to be done often, we would certainly write a C program!), but it is very easy to do in **Ed**:

```
g/^/m0
```

(It will be the convention in this tutorial to show user input in **bold**, and editor output in regular font, offset from the left margin.)

A second, slightly more complicated example is the problem of placing two columnar files, say files generated by `ls`, alongside each other in a buffer. **Ed** can again do the job quite well:

```
$ ed file1
    142
.=
    15
r file2
    153
.=
    30
1ka
16,30 g/^/m'a\
+ka
g/^/ .,+j/       /
```

How does it work? After reading in the two files, each of 15 lines, the first line is marked (`1ka`). Then each line in the second file is moved to the line marked `a`, setting `dot` to the moved line, and the mark is transferred to the next line in the first file (`+ka`). The backslash in the global command line is necessary to terminate the scan of the target address for the move. The last command joins each line to the next, separated by a tab (the white space between the slashes on the join `j` command, a facility in U. of T. **Ed**). Because the global command marks all lines for execution before running the command list on any of the lines, after the join the second of each pair of lines has effectively disappeared from the range of lines of the global, and each execution of the global command list joins a line of the first file with its corresponding line in the second.

The above example demonstrates several things. First, **Ed** is considerably more powerful than most of its users realize. When first given the problem, most users (including the author!) assume it is a task best handled by **Qed**. Second, editor programming often requires the subtle interaction of many commands: the global join is an excellent example. Therefore, debugging editor programs can be difficult. (**Qed** has a tracing feature which greatly simplifies debugging in some cases.) Third, editor programs are usually difficult to read and understand.

This preamble may sound discouraging, but it is only promoting realism. When used well and carefully, **Qed** can be a great time saver, fun to work with, and sometimes even elegant. The original `troff` version of this document was written using **Qed**, storing sequences like `\fBQed\fP` in registers to save typing.

Well, if you've read this far, you must be determined, so you're ready to learn about buffers.

2. Buffers

Ed has one buffer: one scratch area in which to keep text. **Qed** has 56, labeled by lower case alphabetics `a` to `z`, upper case alphabetics `A` to `Z`, and, for reasons worth ignoring at this point, the characters `{`, `|`, `}`, and `~`. Each buffer has its own associated `.` (*dot*), `$` (*dollar*), and filename. The easiest way to see how they work is to `cd` to a directory with about ten `C` source files and type:

```
$ qed *.c
```

(we've already accomplished something which is impossible in **Ed**!) If you were in the **Qed** source directory, you would see something like

```
    a .82   address.c
    b .121  blkio.c
    c .702  com.c
    d .466  getchar.c
    e .284  getfile.c
    f .119  glob.c
    g .725  main.c
    h .209  misc.c
    i .135  move.c
    j .474  pattern.c
    k .156  putchar.c
    l .38   setaddr.c
    m .128  string.c
    n .418  subs.c
    o .220  utf.c
    p .153  utfio.c
    q .36   utfstr.c
```

The first column is the buffer name, the second column is the value of `.` (*dot*) in the buffer, which, on first reading in of a file is set to the value of `$` (*dollar*) in the buffer, that is, the number of lines, and the last column is the file name local to that buffer.

Qed is now waiting for a command. Type `n`

```
n
    a .82   address.c
    b  121  blkio.c
    c  702  com.c
    d  466  getchar.c
    e  284  getfile.c
    f  119  glob.c
    g  725  main.c
    h  209  misc.c
    i  135  move.c
    j  474  pattern.c
    k  156  putchar.c
    l  38   setaddr.c
    m  128  string.c
    n  418  subs.c
    o  220  utf.c
    p  153  utfio.c
    q  36   utfstr.c
```

and look at the output. It should be nearly identical to the above output **Qed** printed when it loaded the files on start up, but now there is only one `.`, indicating that buffer `a` is the current buffer. The `n` (for *names*) command is **Qed**'s equivalent of `ls -l`.

Now do an `f` command. You'll see

```
f
    a .82   address.c
```

In **Qed**, the `f` command tells you more than just the file name. Now change something in the file, say substitute out a tab or delete an empty line, and do another `f`:

```
f
    address.c
```

The prime tells you that the contents of the buffer are known to differ from the named file. Now try

```
bb f
    b .121  blkio.c
```

The `bb` and `f` can be placed on the same line, as **Qed** does not require a newline after most commands. The `bb` says *change to buffer* `b`. Buffer `b` is now the current buffer, as indicated by the dot. If you browse around the buffer for a while, you will see that it is really a world unto itself, but changing back to buffer `a` by a `ba` command will reset you back to the original file, with *dot* still at whatever line it was when you typed `bb`.

Why have multiple buffers? For one thing, we can copy or move text between buffers. Go back to buffer a and isolate a subroutine, marking its beginning with ka and its last line with kb. Then type

```
'a,'b tz0
```

This is a regular copy command, but the z after the t tells **Qed** that the text is to be copied to buffer z. The 0 is the usual address, but is interpreted in buffer z rather than the current buffer. Of course, if there is no buffer name character after the t then **Qed** performs the copy command within the current buffer. Do another n: you will see that you are currently in buffer z, and *dot* is set to the last line copied. The m (*move*) command behaves similarly.

3. Special Characters (1)

Change to buffer z and clear it:

```
bz 0,$d
```

Note that line 0 is a valid address for deletion. ,d also would work here, and neither idiom generates an error if the buffer is empty. As well, you could have typed

```
bz Z
```

The Z (*zero*) command unequivocally clears the buffer, even its remembered file name. Now do the following:

```
ap g/^[a-zA-Z_].(/p*
    g/^[a-zA-Z_].*(/p
```

The append commands (a, i and c) all accept a single line of input typed on the command line, with a space or tab separator between the command and its input. As always, a p suffix causes the command to print its result. Buffer z now contains a possibly useful command for **Qed** (do you know what it does?) which we can call up when desired.

Now read some C source into buffer b if there isn't already some there, and try out the buffer like this:

```
bb
\bz
    initio(void)
    getline(addr_t tl, int *lbuf)
    putline(void)
    blkio_r(int b, int *buf)
    blkio_w(int b, int *buf)
```

The sequence `\bz` means *insert the contents of buffer `z` into my input stream here*. The final newline in the buffer is replaced by the one typed after the `z`, so that if you decided later that you wanted to know the line numbers as well, you could tag a `.=` command on the end:

```
\bz .=
    initio(void)
    43
    getline(addr_t tl, int *lbuf)
    51
    putline(void)
    81
    blkio_r(int b, int *buf)
    108
    blkio_w(int b, int *buf)
    116
```

Although most **Qed** commands can be arbitrarily grouped on a line, the global `g` command, as in **Ed**, still reads the full line for its command list, which in this case is `p .=`.

The above example is very important, as it uses a mixture of buffer input and terminal input to run a command, an all-pervading concept in **Qed** programming.

`\bz` is called a *special character*, although in some sense it isn't really a character at all, as it gets completely replaced with the contents of buffer `z`. The `\bz` is interpreted **whenever input is expected**, not just when commands are being read. Try the following examples:

```
by a
\bz
.
p
    g/^[a-zA-Z_].(/p
*ap \bz
    g/^[a-zA-Z_].(/p
*!echo "\bz"
    g/^[a-zA-Z_].*(/p
    !
```

The buffer could contain multiple lines, which would be handled as usual. We could, for example, save in a buffer our example from the introduction, which merged two columnar files alongside each other, and invoke it when desired just as we invoked the global search above. But care must be exercised here, as the newlines in the buffer, except for the last, are also placed in the input stream. If we were to type, with that multiline buffer in `z`, the command

```
s/x/\bz/p
```

mistakenly expecting that buffer `z` had just a single line of text, say a frequently typed word, we would really be saying:

```
s/x/1ka
16,30 g/^/m'a\
*    +ka*
g/^/ .,+j/       //p
```

This would, of course, cause an immediate error, and since **Qed** always returns to terminal input when an error occurs, no damage would be done. Sometimes, though, such mistakes can cause strange results!

If you did try the above command, the error message would be

```
    ?bz2.0 ?x
```

Qed gives a traceback on errors. The elements of the traceback are of the form

error code format

```
?bXM.N
```

where `X` is the buffer name, `M` the line number, and `N` the character number of the character at which the error was recognized. In the above example, the *substitute* command found a syntax error (`?x`) when it read the newline, so the error occurred at the beginning (`.0`) of line `2` of buffer `z`. If input is nested, the deepest-called buffer is printed first.

It is a good idea to pause here and look carefully over what has been covered so far, as the concept of using a buffer to store regular files or command input interchangeably is really the heart of **Qed**. Before reading on, use **Qed** for a while to familiarize yourself with the system of buffers, and try out a few simple buffers for repetitive editing tasks.

Qed has a fair number of special characters for various purposes. In the rest of this section we will look briefly at some of the simpler ones to give you some insight into how they behave. First, enter buffer `z` again and append:

```
bz
a
\Fa
\Fb
.
```

and then look at what **Qed** has appended to the buffer:

```
-,p
    address.c
    blkio.c
```

The special character `\Fa` means *the file name for buffer* `a`, and, like all special characters, is interpreted whenever input is expected. The special character `\f` is a shorthand for *the saved file name in the current buffer*. Try

```
f junk
    z'.2    junk
w
    18
!ls \f
    junk
    !
```

Idioms such as

```
!cc \f
```

are very common. If your file name is long, `\f` can save much typing. If the file name is changed, through an `f` or `e` command, the name actually associated with `\f' is only changed when the new name is completely read in. Thus, you can type

```
e \f
```

to reinitialize a buffer, or

```
e /sys/src/cmd/\f
```

to edit the system version of a program. There is another special character like `\f`, but it is more useful for programming. `\B` means *the current buffer name*. Try

```
!echo \B
    z
    !
```

4. Special Characters (2)

The easiest way to gain familiarity with the more abstruse characters is to use them in messages, which are a special case of comments. A comment starts with a double quote `"`, and continues until the first following double quote, or the end of the line, whichever is first. The line is ignored by **Qed**, except that dot is set to the addressed line, if there is one:

```
4 " This comment sets dot to line 4
```

Messages are just like comments, except that the first character after the double quote is another double quote. If the message ends with a double quote rather than a newline, no newline is printed:

```
bx
" hi
"" hi
     hi
"" hi there "
     hi there |<- cursor is left on this line
""Current buffer: b\B
    Current buffer: bx
```

This last example is mildly interesting. Can we save the command in, say, buffer `x` and call it back, from any buffer, when desired?

```
bx Z
ap ""Current buffer: b\B
    ""Current buffer: bx
bA \bx
    Current buffer: bx
```

Oops! In principle, it can be done, since the current buffer is the one we are working on, not the one being read for input. But, to put the characters `\B` in a buffer, we must delay their interpretation so that they are not replaced with the buffer name until read back as command input. In most systems on **UNIX**, this is done by typing an extra backslash, but things are more civilized in **Qed**. In **Qed**, special characters are *delayed,* not quoted. Perhaps it's simplest just to state the rules:

special characters

- `\X` is a special character if `X` is one of `b`, `B`, `c`, `f`, `F`, `l`, `N`, `p`, `r`, `u`, `U`, `z`, or `"`, sometimes (as with `\b`) followed by a buffer name. It is interpreted *immediately*. (We will see what all these special characters are in due course.)
- `\Z`, where `Z` is **not** one of the above, undergoes no interpretation at all. In particular, the backslash is not stripped away.
- `\c` is reduced on scanning to `\`, but not re-scanned.
- `\'X` is equivalent to `\X`, but special characters embedded in `\X` are not interpreted.

Things are a little different in regular expressions, but let's ignore them for the moment. These four rules, simple though they are, define the interpretation of backslashes in **Qed**. Note that `\\Z`, where `Z` is again *not* one of the above characters, remains `\\Z`, but if `Z` *is* special, say `f` when the saved file name is `junk.c`, then `\\f` becomes `\junk.c`.

Now we know how to install a `\B` in our buffer: we delay its interpretation by putting a `c` between the backslash and the `B`. (The `c` is for *character*, or (it is rumoured), for *Mr. E. S. Cape*, inventor of the backslash.) The `\cB` will reduce to a literal `\B` when typed in:

```
bx Z
ap ""Current buffer: b\cB
    ""Current buffer: b\B

bA \bx
    Current buffer: bA
bB \bx
    Current buffer: bB
```

That's better! Since `\cc` will reduce to `\c`, the number of `c`-s present is always just the number of times the interpretation is to be delayed.

To decide how many delays are necessary, here is the list of input forms that cause characters to be interpreted:

> - terminal input
> - commands or text (such as that saved in buffers) invoked using a special character
> - command lines for the `g`, `v`, `G`, `V`, or `h` commands (`g` and `v` are the same as in **Ed**; we'll see the others a little later)

Note that characters are *not* interpreted when buffers are read from or written to files, or moved or copied with the `m` or `t` commands. Experience is a great help here, so let's look at some examples:

delay and substitution

```
bx          "switch to buffer x
s/$/\B/         "appends x to current line
s/$/\cB/        "appends \B to current line
s/$/\ccB/       "appends \cB to current line
```

but:

delay and global

```
g/xxxx/ s/$/\ccB/   "appends \B to all lines matching /xxxx/
```

appends `\B` to all lines with `xxxx`; the extra `c` is because the command is in a global command string. Let's say we want to change all the `\n`-s to be `\n\t`. There are two ways:

```
,s/\n/\n\t/
" equivalent to
,s/\cn/\cn\ct/
" or
g/\n/ s/\n/\n\t/
```

No delays are necessary because `\n` and `\t` are not special characters, but delaying them once makes no difference: the `\cn` just becomes `\n`, anyway. (Warning: `\n` has special meaning in the replacement text of a substitution in U. of T. **Ed**.)

While we're dealing with globals, it is a good time to introduce the `\N` special character. It means, simply, a *newline*, and is useful primarily because we can delay it in the usual way. Commands, such as `r`, which deal with filenames, must often be followed by a newline, but can be dealt with using `\N` in globals. The **Ed** sequence

```
g/xxxx/ r\
.=
```

can be put all on one line in **Qed**:

```
g/xxxx/ r\cN .=
```

The newline is delayed. In original version 6 **Ed**, it is impossible to globally substitute a newline into lines, but it's straightforward (by **Qed** standards!) in **Qed**:

```
g/xxxx/ s//\\cN/p
```

The `\\cN` is a backslash followed by a delayed newline. The `\cN` becomes `\N` when scanned by the global `g` command, and then becomes a newline when (re-)read for each `s` substitution. In **Qed** (and U. of T. **Ed**) we could also do this by the functionally slightly different

```
g/xxxx/ s//\\
/p
```

[Do you see the difference?].

Backslashes in general are handled more reasonably in **Qed** than in other **UNIX** programs. Because special characters are *delayed* rather than *quoted*, the number of characters required to insert a special character, with interpretation delayed `n` times, is just `n+2` or `n+3`, rather than exponential in `n`. A **troff** line with 31 backslashes, a not-unheard-of occurrence, would in **Qed** have a single backslash followed by five `c` characters. (And would be much easier to understand, text edit, and debug!)

In particular, **Qed** handles backslashes differently from **Ed**. As mentioned earlier, the **Ed** command

```
s2/"/\\n"/p
```

is simply

```
s2/"/\n"/p
```

in **Qed**, because `\n` is *not* a special character. There are, however, characters which are not ‘special’ in the sense we are using here, but are ‘magic’ in that they have non-literal meaning. The most obvious are characters such as `.` and `$` in regular expressions, which must be *quoted* with a backslash to remove their special meaning and make them literal. (It becomes clear after using **Qed**, or even **Ed**, for a while that *all* the magic characters in regular expressions and the like should require a backslash to become `magic'', rather than literal, but the current choice is too wired-in to the minds of most **Ed** users to be changed now.) Because they are not special characters, their interpretation need not be delayed: they only mean something to the `s *substitute* command. None of the magic characters in the substitution

```
s/\(\.\)xxx$/\1/*
```

require delaying when typed in or run from a global command:

```
g/xyz/ s/\(\.\)xxx$/\1/*
```

Exercise

Is the following command the same as the above substitution?

```
s/\c(\c.\c)xxx$/\c1/*
```

Why or why not? Is the following the same as the global substitution?

```
g/xyz/ s/\c(\c.\c)xxx$/\c1/*
```

Try it to test your answers.

Because of these magic characters, two backslashes in a row `\\` mean a single backslash `\` in regular expressions; otherwise it would be impossible to substitute in a real backslash before a magic character:

```
a abc xyz def
s/xyz/\\&/p
    abc \xyz def
up
    abc xyz def
s/xyz/\\\&/p
    abc \& def
```

What about sequences like `\\B`? Well, `\B` is not a character at all, but a *special character* (sorry for the terminology) since it is **immediately**, at the lowest level of input, replaced by the current buffer name. Since `\\` is *not* a special character, and has non-literal meaning only when found between regular expression delimiters, the substitute command itself never sees the second backslash. All interpretation of special characters is done *before* the substitute command sees them. If the current buffer is buffer `a`, then

```
s/\\B/x/
```

does exactly the same thing as

```
s/\a/x/
```

Also, because **Qed** next converts `\\` to `\` in regular expressions,

```
s2/"/\\n"/p
```

is the same as

```
s2/"/\n"/p
```

since `\n` is not a special character.

Qed saves the last used regular expression and replacement text used in an `s` or `j` command, so that they can be called back using `\p` (for *pattern*) and `\r` (for *righthand side*). `\p` is handy when you want to change the saved pattern. If, for example, you start searching for `proc()` and want the declaration, but find there are very many usages of `proc()`, it is simple to find an occurrence of `proc()` at the beginning of a line:

```
/proc()/
        x=proc();
//
        x=proc()2;
*/^\p/
    proc(){
```

`\p` is of somewhat limited usefulness, as the null regular expression `//` is essentially the same as `/\p/`; but `\r` provides a new convenience. Browsing through text doing repetitive substitution is simplified considerably by using `\r` :

```
s/apples/mangos and pears/p
    I ain't got no mangos and pears
//
    your mother's apples smelled like they were
s//\r/p
    your mother's mangos and pears smelled like they were
```

There is a danger with `\p` and `\r` : if they contain delayed special characters, each usage of `\p` or `\r` removes one delay. If the current file name is `wylbur.ms`, it may be difficult to deal with `troff` font changes:

```
p
    editors such as Wylbur are so
s/Wylbur/\cfBWylbur\cfP/p
    editors such as \fBWylbur\fP are so
//
    Wylbur is also no good for
s//\r/p
    wylbur.msBWylburwylbur.msP is also no good for
" Oops
```

This is the sort of trouble which the `\'` special character can circumvent. `\'r` means the usual `\r`, but with special characters inside *uninterpreted*. If we had used it above, things would have worked properly:

```
p
    editors such as Wylbur are so
s/Wylbur/\cfBWylbur\cfP/p
    editors such as \fBWylbur\fP are so
//
    Wylbur is also no good for
s//\'r/p
    \fBWylbur\fP is also no good for
" Much better
```

`\er` is also handy for fixing a certain class of mistakes:

```
p
    textp=get(a->text.fdes);
s/text/tbuf/p
    tbufp=get(a->text.fdes);
" Oops again
us2//\'r/p
    textp=get(a->tbuf.fdes);
" Got it!
```

The **Qed** idiom `us2//\'r/p` undoes (`u`) what you just did wrong, then substitutes (`s`) again (`//`) on the *second* (`2`) match in the line.

Now, as an exercise, use **Qed** for a while until you feel comfortable with the use of backslashes. If you find them confusing, work with **Qed**, doing fancy things if you feel up to it, until the confusion disappears. What follows will be much stranger …

5. Special Characters (3)

Now that we've established the ground rules, we can begin to use some of the fancier stuff in **Qed**.

The special character `\l` (for *line*) returns a line of text from standard input, usually the user at the terminal. In other words, if, say, input is coming from a buffer, then the input will be temporarily redirected to come from the terminal. The terminating newline is stripped away. Since it is interpreted immediately (being a special character), `\l` is rarely of value *except* when delayed. Nevertheless, let's look at how it behaves in immediate mode:

```
""\lMessage\N
    Message
""\lMessage

    Message
""\lMessage
s of words
    Messages of words
```

The extra newline, whether provided by the `\N`, or by a typed second carriage return, is necessary because the `\l` strips its terminating newline away, but the comment (`""`) command is looking for a newline itself in order to terminate.

[Some questions to consider: If `\bx` is used instead of `\l`, the second newline is not required. Why? In the last example above, which characters are returned by `\l`? What is the origin of the others, if any? What would the above examples do if the comments were terminated with a double quote?]

Well, `\l` is clearly of little use if not delayed, but it *is* important to understand how it behaves.

An early version of U. of T. **Qed** had only lower case buffer names, and when the names `{` through `~` were added it was necessary to go through the manual changing some of the instances of `` `z' `` into `` `~' ``, but not all of them. The following single line made the job very simple:

```
g/`z'/ p ""replacement:" s//`\cl'/p
```

Each line (`g`) with a `` `z' `` (`` /`z'/ ``) is printed (`p`), the user is prompted for the replacement (`""replacement:"`), and the response (`\l`) — either a `z` or a `~` in our case — is substituted (`` s//`\cl'/ ``). The single delay ensures that `g` places a literal `\l` in the substitution string, which is then interpreted when each call to the substitute command builds its replacement (*right-hand side*) text. This sort of operation can also be performed using an `x` command driven by a global, but **Qed** can be programmed to do most of the work.

Here's another example:

```
bz ,d
ap ""Comment:\cl" s|$|  /* \cl */|p
    ""Comment:\l" s|$|  /* \l */|p
```

On any line which invokes this buffer with a `\bz`, the first `\l` in the comment will “eat” any input remaining on the line after the `\bz` (and inserts it into the comment command (`""`) where the first `\l` appears, namely after the `:`. Perhaps not very useful!).

```
ba a    c.code;
\bz
    Comment:stylish                ①
        c.code; /* stylish */

a   more.code;
\bz foobar
    Comment:foobarstylish          ②
        more.code;  /* stylish */
```

① Prompt is `Comment:`, user types `stylish`

② Prompt is `Comment:foobar` (*sic!*), user types `stylish`

(Also, of course, if what the user typed at the prompt contains the character `|`, problems will occur.)

Now, if we intend to be able to type our C comment text on the same line as the invocation of the buffer, as if passing it as an argument, we want neither the `Comment:` message nor the extra `\l` which clears the input line. Assuming buffer `z` still contains our commenting program, let's edit out the `Comment:\l` prompt, and try it again:

```
bz s/".*" //p
    s|$|        /* \l */|p
ba a    yetmore.code;
\bzstylish
        yetmore.code;         /* stylish */
```

This latter form is likely more useful, as it can be called from a global (the previous version could, but required the user to type extra newlines). For example, to comment all occurrences of a variable named `var`:

```
g/\{var\}/ p \cbz
```

Each line is printed, and the user's response is appended as a C comment. No extra `\l` is needed at the end to clear the input line as the `g` already reads the line up to and including the terminal newline. Thus the single `\l` in the `s` command in buffer `z` returns the next line typed in. Note that the `\bz` command is delayed so that it is interpreted for each selected line.

We could alternatively have set up our `z` buffer so that the `\l` itself was delayed, using `\cl` instead. The buffer could then be invoked (in a global command) as `\bz`, without the delay.

In effect, then, the `c` in the original buffer call `\cbz` above acts to delay the `\l`. If the buffer had only literal text, no delay would be necessary. Our choice of where to put the delay was made by wanting the buffer to be invocable directly from the keyboard.

Just for the record, note that we can achieve the effect of `\cb` above by typing `\'b`, although the manner in which it works is quite different.

Although these examples are somewhat low-key, they do begin to show how the parts of **Qed** fit together. Later, we will see how the `\l` can be used to control execution of commands.

6. Registers

Qed has 56 registers, with the same names as the buffers: `a` to `z`, `A` to `Z`, `{`, `|`, `}`, and `~`. Buffers and registers are otherwise unrelated. The registers are used to store simple text and short command sequences. In fact, most of the command buffers we have created so far would be better suited to storage in registers; buffers are generally used for storage of file text proper and multiline command sequences. The two main advantages of using registers to store text are: they can be set and manipulated without leaving the current buffer, and they do not appear in the output from `n` commands, which is significant because a user may typically have twenty or more defined registers.

Registers are manipulated with the `z` (for *zdring*!) command. The character after the `z` is the name of the register being operated on, and the next character is an *operation code*. The most straightforward operations are assignment (`:`) and printing (`p`):

```
za:procrastination
zap
    procrastination
```

The string being assigned to the register is terminated by a newline. If a newline is to be embedded in the register, `\N` provides the cleanest mechanism:

```
za:line1\cNline2
zap
    line1\Nline2
a
\za
.
-,p
    line1
    line2
```

As a cconvenience, **Qed** also allows multi-line assignment to a register by escaping a literal newline with `\` :

```
za:line1\
line2
zap
    line1
    line2
a
\za
.
-,p
    line1
    line2
```

Registers are invoked in the obvious way: `\za` inserts the contents of register `a` into the input stream. Note in the above example that the append (`a`) could not be done on one line, as the embedded newline in the register would cause the first line (`line1`) of the register to be appended, and the second (`line2`) to be interpreted as (ill-formed) command input:

```
zap
    line1\Nline2
a \za
    line1
    ?za10 ?x
```

This is another example of embedded newlines causing trouble: be careful!

There are many *operation characters* for registers; they are listed in full in the `qed(1)` manual section. For example, we can perform a substitute operation on the contents of the register with `zas/xxx/yyy/` (the same syntax as for the substitute command `s`); increment and decrement the Unicode codepoint values of the characters in the register with `za+N` and `za-N`, where `N` is an integer; and do subzdring (!) operations with the *take* and *drop* functions `za)N` and `za(N`. One particularly handy form is

register set to match

```
za/regular expression/
```

which saves in register `a` the string in the current line which matches the regular expression. There are several other register operations we will introduce when required.

These operations are quite straightforward; we will see them all used when we start to program **Qed**.

Registers can also be manipulated numerically. This is indicated by the `#` (for *number*) operation: `za#...` . The text following the `#` is then interpreted specially. Decimal numbers (integers) stand for themselves, and numeric assignment to the register is `:`. So:

```
za#:42
zap
    42
za#:-42
zap
    -42
```

In this numeric-register context, the letters `a`, `r`, `n`, `N`, `p` and `P` have special meanings. For example `p` means *print the current value of the register*. We will look at `a` and `r` shortly. The rest are described in the `qed(1)` manual. We can chain numeric operations together (without spaces):

```
za#:42p
    42
za#:-42p:99p
    -42
    99
zap
    99
```

In the last example, -42 is assigned (`:-42`) to `a`, then the current value of `a` is printed (`p`), then 99 is assigned (`:99`) to `a`, and the current value of `a` is again printed. Register `a` contains 99 at the end of the numerical context.

In this numeric context, the value of the register can be updated by one of the arithmetic operations `+`, `-`, `*`, `/`, and `%`, which have their familiar C language meanings:

```
za#:1
za#+1p
    2
za#*2p
    4
za#*2+1p
    9
```

Numeric-register context ends at the first character which is invalid in a numeric-register context, or the first newline, whichever comes first. When numeric-register context ends, the entire `z…` register command ends, and **Qed** will start processing any remaining characters on the line as regular **Qed** commands. This can have some bizarre side-effects:

```
ba Z
a foo.bar;
za#:99p
    99
za#:99 p
    foo.bar;
```

In `za#:99 p`, the space (after the `99`) is *not* a valid character in numeric-register context, so it terminates the `za#:99` command. The remainder of the line (` p`) is then interpreted as a regular **Qed** command (namely, *print the current line*), which it duly does.

It is an error (`?#`) to try to perform numeric operations on a register which does not contain a (possibly negative) decimal integer:

```
za:44moose
za#*2
    ?#
```

The main difference between `zap` and `za#p` is semantic. In `zap`, the contents of register `a` are interpreted as a string, and the (string-)register operation `p` prints the string held in `a`. In `za#p`, the contents of register `a` are interpreted in numeric context, and the numeric-register operation `p` prints the (numeric) value in `a`. In practice, the outcome of `zap` and `za#p` is the same if `a` contains a decimal integer.

Perhaps the most important use of numeric-register operations is in addressing. The numeric-operation character `a` causes the register to receive the line number of the address of the `z` command:

```
$za#a
```

assigns to register `a` the line number of the last line (`$`), *i.e.* the number of lines in the current buffer; and

```
/xxxx/za#a
```

saves in `za` the address of the first forward occurrence of `xxxx`.

The `r` operation character (for *range*) stores the first given address in the named register, and the second address in the register whose name is lexically one greater:

```
1,$ za#r
,za#r
```

Both put `1` in register `a` and the value of `$` in register `b`. Neither `a` nor `r` changes the value of *dot*.

These operations are usually used to pass addresses to an execution buffer: if the first line of a buffer is

```
za#r
```

then if the buffer is invoked as, say,

```
-5,.\bz
```

then registers `a` and `b` contain the addresses of the first and last lines of the range to be operated on by the buffer.

Numerical operations are also frequently useful in text editing, such as when generating defined constants for a table:

```
a
read
write
open
close
creat
.
" capitalize
?read?,.s/.*/^/p
    CREAT
za#:0
?READ?,.g/^/s/.*/#define &  \cza/p za#+1
    #define READ    0
    #define WRITE   1
    #define OPEN    2
    #define CLOSE   3
    #define CREAT   4
```

The `^` (caret) character in the right hand side of a substitute behaves like `&`, but flips the case of ASCII alphabetics in the matched string.

7. Control Structures

the global command, `g`, which is remarkably powerful and versatile, as the previous example demonstrates. The ability to place several commands on a line, and the simplicity of `\N`, make globals even easier to use in **Qed** than in **Ed**.

Along with the concept of line-by-line execution goes that of buffer-by-buffer execution, which is provided in **Qed** by the *globuf* commands `G` and `V`. They are quite simple to use: their format is identical to the regular globals `g` and `v`, but the regular expression is used to match the output which would be produced by an `f` command in each buffer. Only buffers which contain text or have a remembered file name are tested for a match. If a buffer matches the regular expression, the command list is executed in that buffer. For example,

```
G/.'.*	./w
```

writes out (`w`) all buffers which have been modified since last written. That is, all buffers which the `f` command reports with a prime (`'`) after the buffer name. The white space in the above example is a tab, which is the actual delimiter used by the `f` (and `n`) commands between the number of lines in the buffer and the file name. Here's a fancier example:

```
G/./ g/thing/ ""\cB \cf: " p
```

It takes all non-null buffers (`G/./`), and for each one it looks for occurences of `thing` (`g/thing/`). Whenever it finds one, it prints out (`""…"`) the buffer (`\B`), the file name (`\f`), and line (`p`). It's a bit like a super-grep, or *Gregrep*.

Qed has a *loop* control structure, the `h` command (for *h-until*!). `h`, like `g`, takes a line of commands and executes it repeatedly. It has four forms:

loop

- `hN` executes the line N times
- `ht` executes the line until the *truth flag* is true
- `hf` executes the line until the *truth flag* is false
- `ha` (for *always*) executes the line forever, or until an error

Although the loop is an *until*,

no-op

```
h0 p
```

is guaranteed to execute zero times. An infinite loop can be halted by sending `^C` from the terminal.

The *truth flag* is set by substitutions, and comparison operations in registers. When a register is compared to some value, the truth flag is set according to the success of the comparison. When a substitution is made, the truth flag is set if a substitution was performed. As a simple example, say you have prepared a letter to be sent to someone, using **Qed**, only to find that the erase character is a backspace, not `#` as you had been using. To fix the problem,

```
g/^/ hf s/.#//
```

Note that *huntil*-s can be run inside globals, and, in fact, can be nested arbitrarily deep. Globals can also be run from *huntils*; the only restriction is that globals cannot be called from globals, as **Qed** can only mark a line for a global once. Similarly, *globufs* cannot be called from *globufs*.

Exercise

Change the example where `z`-s were replaced by `~`-s so that it works properly when there is more than one `z` on a line.

As in globals, *huntils* stop the scan of the command sequence at the first newline. To build an alphabet in register `A`:

```
za:a
zA:
h26 zAs/$/\cza/ za+1
zAp
    abcdefghijklmnopqrstuvwxyz
```

Note that **Qed** code is not always easy to read! If you happen to know that the character which precedes `a` in Unicode is back-quote (`` ` ``), you could also build the alphabet with:

```
za:`
zA:
h26 za+1 zAs/$/\cza/
zAp
    abcdefghijklmnopqrstuvwxyz
```

And now, register `a` could also be used in *auto-increment mode* to simplify things further:

```
za:`
zA:
h26 zAs/$/\cz+a/
zAp
    abcdefghijklmnopqrstuvwxyz
```

The `+` between the special character `\z` and the `a` in the register call causes the values of the Unicode codepoints of the character(s) in `a` to be incremented before being placed in the input stream:

```
za:moose
""\z+a
    npptf
```

Auto-decrements are also possible (`\z-a`) as are numerical increments and decrements (`\z#+a` and `\z#-a`). Only auto-increments of +1 and auto-decrements of -1 are possible.

As a less frivolous example (one that was used in writing an earlier version of this tutorial), a *huntil* makes it simple to convert, say, the *troff* command `.ul 5` to five `.ul`-s, one after each affected line:

```
g/^\.ul / zn/[0-9]/ zn#-1 s/ [0-9]+// h\czn +a .ul
```

It looks horrible, but it works, and can save much trouble if there are (as in the tutorial) twenty or more places where the fix needs to be made. (The `+` character in regular expressions is like `*`, but guarantees at least one match.)

Of course, until familiarity with **Qed** is developed, the mental effort required to write a line like this and have it work is probably considerably greater than the physical effort required to type in the changes individually. Even for beginning users, though, saving the complicated patterns, and commands such as `ap .ul` in registers would make the job much more pleasant.

Again, care must be taken when invoking registers or buffers in *huntil*-s:

```
h20 \bz
" or "
h20 \cbz
```

will likely not do what is expected if buffer `z` contains more than one line.

The other major new control structure in **Qed** is the `y` command (for *yump;* think of *jump* pronounced with a Swedish accent). The syntax is:

jump

```
y[t|f][N|o|'label|`label]
```

which translates as follows: If the `t` or `f` is present, jump only if it matches the *truth flag;* otherwise jump unconditionally. The `N`, if present, is a number, and is interpreted as a line number in the current executing buffer, which becomes the next line read for commands. An `o` (for *out*) causes the current input source, such as a global command string or buffer, to be terminated. If the input source is a buffer, the effect is to return from the buffer; if a global, the execution of the global (or *huntil*) is stopped. For example,

```
za#:1
h50 za#+1 za#>20 yto
zap
    21
```

executes 21 times, leaving register `a` set to 21.

The forms

conditional jump forward to label

```
y[t|f]'label
```

and

conditional jump backward to label

```
y[t|f]`label
```

are similar to `y[t|f]N`, but the line to which control is transferred is the first line found which begins with the comment `"label`, searching forward in the buffer in the case of `y[t|f]'label`; or backward in the buffer in the case of `y[t|f]`label` (where the operation character is a *back-tick*). Initial blanks and tabs on the line before the comment character `"` are ignored, and the scan of the label stops at the first blank, tab, newline or double quote. If the first character after the double quote is a space, tab, newline or double quote, the label is null and can never be matched. If no matching label is found in the executing buffer, execution resumes at the first character after the label in the *yump* command. Note that the label must be matched exactly; it is not interpreted as a regular expression.

There are a few non-trivial small examples which illustrate the use of *yumps*, but they will be used later on in the tutorial. For the moment, a remark on style: clearly, with only a *goto*, flow of control in **Qed** can become messy if care is not taken. It is recommended that *yump*-s only be used in easily identifiable forms such as *if ... then ... else ...*:

if <condition> then ... else ...

```
<condition> yf'else
    ...
y'fi
"else
    ...
"fi
```

and, *do ... until ...*

do ... until <condition>

```
"do
    ...
    <condition> yf`do
"od
```

or, *while ... do ...*

while <condition> do ...

```
"{
    <condition> yf'}
    ...
    y`{
"}
```

One particularly useful form of labeled *yump*-s is a *switch* statement based on a line of input from the user. This mechanism makes command interpretation very simple. It is essentially a fancy switch statement:

switch

```
y'X\l
"default:
    ...
    yo
"Xcase1
    ...
    yo
"Xcase2
    ...
    yo
...
```

One other form of *yump* exists; it is intended primarily to skip the rest of a *global* or *huntil* command sequence, without stopping the execution completely. Its form is simply `yt`, or `yf`. When invoked, it jumps over the current input source up to and including the next newline. It can also be used as a shorthand in buffers, but such usage is discouraged.

8. Calling the Shell

Qed has three methods of calling the Shell aside from the `!` (*bang*) command: *crunch* (`<`), *zap* (`>`), and *pipe* (`|`). All of these commands cause the **UNIX** commandline they last invoked to be stored in register `U`. This commandline can be recalled by doubling the command character, as if the command `\'zU` was issued:

```
!echo fun
    fun
    !
zUp
    echo fun
!!
    fun
    !
```

You can even extend the saved commandline:

```
!! | wc -c
    4
    !
zUp
    echo fun | wc -c
```

And edit it, just like any other register:

```
zUs/fun/funny/
!!
    6
    !
zUp
    echo funny | wc -c
```

Crunch (`<`) takes the standard output from the Shell command and reads it into the current buffer, as if the output from the Shell command had been redirected to a temporary file, which is then read in with an `r` command. Like the `r` command, `<` takes an optional address which specifies the line (defaulting to `$`) at which the text is to be read in. (As above, the **UNIX** command last executed can be reinvoked, as a *crunch*, with `<<`.)

```
< ls
    !
```

appends a list of the files in the current directory to the end of the buffer.

One very common usage of the crunch command is to create a to-do list, by a command such as

```
< grep "FIX ME!" *.c
    !
```

or using a buffer as a sort of checklist by capturing diagnostic output from a compiler, say:

```
bz Z
< cc -c *.c | tee /dev/tty
    ... diagnostic messages ...
    !
```

saves the listing of the compile errors so you can let `cc` run through everything before fixing typing mistakes, etc.

Zap (`>`) is to *crunch* as `w` is to `r`: it writes out the contents of the addressed lines (defaulting to the entire current buffer) as standard input to the Shell command. It is can be used to send e-mail. The e-mail can be prepared in a buffer, edited as desired, and then sent easily by

```
> mail joe
    !
```

or even

```
> nroff | mail joe
    !
```

Zap and *crunch* work nicely together. We can perform an interactive file-delete function, (like the ancient `dsw` command) using *crunch* to read the files in, modifying the list as appropriate, and sending it out to (another ancient command) `args`:

```
< ls
    !
... editing commands ...
> args rm
    !
```

(`args` was a command that took each line on its standard input and maked it an argument to the command, which was then `exec`-ed in the normal manner.)

The following commands can initiate the construction of a dependency-list file for `make`:

```
<grep "#include" *.c
    !
,s/:#include[   ]"/     //
,s/"$//p
    utfio.c qed.h
```

At U. of T., the Shell takes a `-e` option which tells it to echo on the diagnostic output the commands it is executing, which works nicely with *zap:*

```
a
command1
command2
command3
.
> sh -e
    % command1
    % command2
    % command3
    !
```

In short, the *crunch* and *zap* commands are used very frequently.

The *pipe* command (`|`) is very similar to *crunch*. Whereas *crunch* takes a single address, and *inserts* the standard output of the commandline at that address, *pipe* takes an address range, and it *replaces* the addressed range with the standard output of its commandline. The default address range for *pipe* is the entire buffer (`1,$`), so the command `< ls` will append a directory listing to the current buffer, but the command `| ls` will *replace* the contents of the buffer with the directory listing.

9. Programming (1)

Now that we've seen all the primitives, we can begin using buffers and registers to build more sophisticated commands. The first step is to assemble a few useful command sequences in registers. Harking back to our function-declaration-finding buffer in Section 3, define register `f` (for *function*):

```
zf:?^[a-zA-Z_].*(?
zfp
    ?^[a-zA-Z_].*(?
```

As a *global* search, the regular expression `/^[a-zA-Z_].*(/` found all function declarations, provided, of course, that the usual paragraphing style is used.

Exercise

Write another definition to perform this function which uses the *beginning of identifier* (`\{`) metacharacter.

Now, with the regular expression enclosed in `?…?`, register `f` finds the first *previous* function declaration. This seems like an odd concept at first, but works well. For example, to see which function's source is being browsed:

```
\zf
    function(x)
```

Or, to find the declaration of a local variable:

```
p
        variable=0;
\zf/variable/
        int variable;
```

(This works by searching back to the closest previous function definition using the saved command (`\zf`) and then searching forward for the first occurence of the name `variable` with `/variable/`.)

To print out to the line printer the listing of the function currently being browsed:

```
\zf, /^}/ > lpr
```

There are fancier things, too. If we want to know which subroutines call `proc()`, we can again use `\zf`:

```
g/proc()/\zf
    func1(x)
    func2(y)
    func3()
```

After using macros like `\zf` for a while, they become familiar to the point that they become idiomatic, a part of the **Qed** language. To help the user develop a personal working environment, **Qed** provides a simple mechanism for initializing. Typing (to the Shell)

```
$ qed -x qfile file1 file2
```

causes **Qed** to load the file named `qfile` into buffer `~` (*tilde*) **and execute it,** before reading in the files to be edited (`file1`, `file2`), and beginning the normal editing session. Typically, the startup file is used to initialize options and registers; it might contain something like:

qfile listing

```
""Qed
zc:s|$| /* \cl */|p
zf:?^[a-zA-Z_].*(?
b~Z    " destroy buffer after execution
```

which prints a message (`Qed`); defines a couple of handy registers (`zc` and `zf`); and obliterates itself (`b~Z`). If no `-x` option is given when **Qed** is invoked, **Qed** will try to load the file named by the (Shell) environment variable `QEDFILE` into register `~`, and run that instead. If the variable `QEDFILE` is undefined then no startup file is loaded.

Browsing through the startup files of a few experienced **Qed** hacks, a few interesting things come to light. One simple but rather pretty option is

```
ob""\x1a"+p
```

ASCII `0x1A` is a reverse line-feed on most of the U. of T. terminals; The *browse option* (`ob`) defines a special register which is executed, if defined, when a simple newline is typed at the terminal, rather than doing the default `+p` (*print next line*). Printing a reverse line-feed before the `+p` means that no empty lines appear on the screen when browsing through text.

It is sometimes useful to set the browse register to something like `+b` for easy paging through text, or to `P` or `L`, which cause the line to be displayed in the format of `p` or `l`, but with line numbers at the beginning of the line:

```
22i Line	22
p
    Line    22
l
    Line\t22
P
    22  Line    22
L
    22  Line\t22
```

These other display formats are also sometimes handy in global searches:

```
g/proc()/ \zf P
    104 func1(x)
    118 func2(y)
    221 func3()
```

Another nice register to have tucked away (as it is above) is the C-commenting command we saw earlier:

```
zc:s@$@ /* \cl */@ p
```

We can call it up when desired:

```
p
    bizarre();
\zc(A Kludge)
    bizarre();  /* (A Kludge) */
g/xxxxx/ p \czc
    yyy xxxxx yyy
needles
    yyy xxxxx yyy   /* needles */
    zzz xxxxx zzz
haystacks
    zzz xxxxx zzz   /* haystacks */
...
```

The following register definition allows the user to find a buffer by its file name:

```
zb:G/\cl/ f\cN
\zbfile
    g'.34   file.c
```

We don't even need to type the suffix `.c`!

Here is a rather complicated, but conceptually simple, register, `\zs` (for *search*), which globally searches for a pattern in all the buffers from `a` through `z`, and leaves dot at the last occurrence found. For readability, from here on, we will usually list the contents of a register with real newlines in place of the `\N`-s, or the escaped newlines, and without the delays to special characters, that would be necessary when actually assigning the program to a register. Compare what you would have to type at the keyboard to assign this program to register `s`, with the listing that follows!

writing to zs at the terminal (with escaped newlines)

```
zs:zB:\cB\
zP:\cl\
zI:`\
h26 zI+1 b\cczI $zD#a=0 yt g/\czP/ ""\ccB:" P zB:\ccB\
b\czB
```

Or worse (!):

writing to zs at the terminal (with delayed newlines)

```
zs:zB:\cB\cN zP:\cl\cN zI:`\cN h26 zI+1 b\cczI $zD#a=0 yt g/\czP/ ""\ccB:" P
zB:\ccB\cN b\czB
```

Versus a comparatively sane listing:

zs program listing

```
zB:\B
zP:\l
zI:`
h26 zI+1 b\czI $zD#a=0 yt g/\zP/ ""\cB:" P zB:\cB
b\zB
```

But what does all this mean !? One step at a time:

zs program walk-through

- `zB:\B` sets register `B` to the current buffer name;
- `zP:\l` sets register `P` to the rest of the commandline (*i.e.* the user-supplied pattern; if there were special characters in the pattern, we would probably have to delay them once more than usual to achieve the desired result.)
- `` zI:` `` sets register `I`, a counter, to the character immediately prior to `a` in Unicode.

The next line does all the work, and reads something like:

- `h26` *for* 26 iterations *do*
  - `zI+1` increment I (iterates over the alphabet)
  - `b\czI` switch to (new) buffer \zI
  - `$zD#a` set D to the address of the last line of the buffer
  - `=0` test for D = 0 ?
  - `yt` *if* (D=0) is true *then* skip the rest of the line, resuming the next iteration
  - `g/\zP/` *else for* every line matching the given pattern \zP *do*
    - `""\cB:"` print the current buffer name and a colon
    - `P` print the matched line, with line number
    - `zB:\cB` set register B to the current buffer name

Finally, the last command

- `b\zB` switch to buffer name in register `B`.

After execution, `zB` contains the last buffer name in which a match was found, and **Qed** automatically keeps track of the line number on which the match was found. The last line of `zs` therefore changes back to buffer `\zB`, which leaves *dot* at the last line printed, similar to the behaviour of `g/xxx/p`.

Got that?

Make sure you understand how the `s` register operates, as it utilizes many of the standard **Qed** programming techniques, such as nesting a *global* inside a *huntil*.

Well, that was instructive, but rather revolting. If you understood how the search register works, you're doing very well, but it's not a good example of how to program **Qed**, just a pedagogical one. Here's how to *really* do it:

global search program listing

```
G/^[a-z]/ g/\l/ ""\cB:" P
```

Exercise

Modify this version so that it remembers the last buffer in which a match was found.

You'll find as you gain experience that *huntil*-s are rarely used, but they do have their moments.

Using the register is quite easy; just type `\zs` followed by the pattern being searched for:

```
\zs^func()
    a:86    func()
    b:102   func() {
f
    b .209  junk.c
```

Exercise

Set up your startup buffer to include the original definition of `zs` using delayed `\N`-s where necessary. Is a delayed newline necessary at the end of the register? Why, or why not? (**Hint:** Where does the newline at the end of the invocation line end up?) Define a second register like `s`, but which executes a definable register, say `e`, for *execute*, rather than just printing the line. You can use our one-liner version (above) that we stored in register `x` here. What useful things might be put in register `e`?

Registers can also be used to call the Shell. Register `d`, defined below, calls `pwd` to get the current directory, saving the result in register `e`, so that the user can quickly return after changing working directory.

```
zd:ovr zB:\cB\cN bX <pwd \cN ze. d b\czB ovs zep zB:
```

This definition of register `d` is also exactly as it would appear in a startup file. The `ze.` command puts a copy of the current line in register `e`. The delayed newlines are necessary; unpacked, the string looks like

zd program listing

```
ovr zB:\B
bX <pwd
ze. d b\zB ovs zep zB:
```

Briefly: turn verbose mode off (`ovr`); save the current buffer name into register `B` (`zB:\B`); change to buffer `X` (`bX`) and get the directory (`<pwd`); save it in register `e` (`ze.`); delete the line from the buffer (`d`); change back to the original buffer (`b\zB`); turn verbose mode back on (`ovs`); print the saved directory (`zep`); and clear register `B` (`zB:`).

10. Programming (2)

So far, the emphasis has been on using registers as programming elements, primarily because the size and complexity of the problems being handled has been small enough that registers are really the way to deal with them. Ultimately, though, more complicated problems arise and it becomes necessary to store command sequences in buffers. In light of that, one more register definition, `r` for *run*, will make using program buffers somewhat simpler. Called as

```
\zrbuffer
```

it reads file `buffer.q`, prepended by the search path stored in register `q`, into a scratch buffer, executes it, and then clears the scratch buffer. Typically, register `q` would be set by the startup buffer to contain something like `/home/rob/q/` so

```
\zrcommand
```

runs the buffer in the file `/home/rob/q/command.q`. Register `r` is long, but linear, having no loops. As an unpacked listing, with newlines, it looks like:

zr program listing

```
zL#r
z{:\l
z|:\B
ovr b{
e \zq\z{.q
ovs b\z| \b{
b{ Z
b\z|
```

This looks considerably more bizarre than our earlier definitions, because it follows some conventions that have proven useful. The registers and buffers with funny names (`{`, `|`, `}`, and `~`) are (unofficially) reserved as scratch areas: anything you put in one is not guaranteed to stay there if you call in an external buffer. `zr` uses registers `{` and `|` to hold the program name typed by the user and the buffer the register was called from, and buffer `{` to hold the program. **Qed** itself uses register `~` to hold the initialization code to bootstrap the startup buffer, but clears it before going to the terminal for input. (A side effect of this is that your startup buffer can zap `z~` to alter the bootstrap procedure.)

Other conventions are that upper case registers and buffers are reserved for use by program buffers, such as the ones we will be developing in this section; and lower case letters are reserved for the user.

`zr` stores in registers `L` and `M` the addressed lines for the buffer being called (via the *range* operator: `zL#r`). Following these conventions means that a user can call a copy of someone else's program buffer, for example, without worrying about which registers and buffers it uses.

The `ovr` and `ovs` calls in register `r` set the *verbose flag* off and on when appropriate, to suppress the occasional character counts on i/o. The register as defined here actually works, but what we really want is something a bit spiffier, so, let's have `\zr` load a particular buffer file, which we will describe immediately afterwards:

```
zr:zL#r z}:\cB\cN ovr b~e \czqrun\cN \cb~\cN b\cz}
```

This loads buffer `~` with the file (say) `/home/rob/q/run.q`. The buffer is then executed, and the user is returned to the original buffer. The `run.q` buffer file contains:

run.q program listing

```
" Run a qed buffer `off line'
z{:\l
z{C
" the next line puts a space at the end of the register
z{$
" the next line looks for a space in the argument string
z|'{  z{[
z~#c  z{)\z~  z|(\z~  z|C
" z{: command   z|: argument string   z}: return buffer set by zr
b{ e \zq\z{.q
ovs
b\z} \b{
" Note! ok to ZERO buffer ~ (this buffer); the line will finish executing
b{Z b~Z
```

The `zXC` command is kludgey but handy: it collapses multiple blanks and tabs in the register to single blanks, and deletes leading blanks. The first few lines of the buffer put the command and its arguments (if present) into registers `{` and `|`. The new register commands introduced here are:

register commands

- **`zx[string`** stores in the *count* the starting index of `string` in register `x`
- **`zy#c`** saves the *count* in register `y`
- **`zx)N`** *takes* the first `N` characters from the register, and discards the remainder
- **`zx(N`** *drops* the first `N` characters from the register, and preserves the remainder

Although it's a little unfair to show these commands to you this late in the game, they didn't really need showing earlier on, and they are quite simple to master. The *count* also hasn't shown up before, so we'd best explain it now. It is a special place, something like the *truth*, which gets set to the number of characters transferred during i/o operations; the number of substitions made during an `s` command; and to other such numbers, as above. Although rarely used, it, too, has its moments.

The requested buffer is then loaded into buffer `}` and executed. Finally, the loaded buffer and buffer `~` are zeroed, and `run.q` returns.

Although its coding is not particularly pretty, the power register `r` gives us is dramatic. It is really part of the **Qed** language, since it allows the user to store many command buffers in the file system, but get at them easily and in a mnemonic fashion. It itself employs two conventions which are therefore ubiquitous: registers `L` and `M` hold the lines being addressed by a buffer call, and buffers `{` and `~` are off-limits to command buffers. The latter point, of course, shows the weakness of a language in which all the variables are global, but let's ignore that theoretical issue for the moment: **Qed** has many other weaknesses which are far more important!

`zr` is only useful if we have some buffers to drive with it. For starters, we can take our `search` register and put it in a buffer (say `/home/rob/q/grep.q`):

grep.q file listing

```
" Grep for z| (possibly set by caller) in all buffers
z|=
yf'fi
    ""pattern:" z|:\l
"fi
G/^[a-zA-Z]/ g/\z|/ ""\cB:" P
```

A few noteworthy points occur: firstly, we can prompt the user for missing arguments. If the user types

```
\zr grep expr
```

(notice the blanks, which are deleted by the `run` buffer) we can search for `expr` directly; but if no expression is specified, we just ask for it.

Secondly, putting the code into a buffer means everything can be delayed one less time, which makes it more readable, and the initialization and cleanup code is shared by all command buffers, providing a clean and uniform interface. Also, after execution, register `r` returns the user to the buffer they started in, rather than leaving them in some random place. For this example, it may or may not matter, but in some cases it is advantageous to return 'home'.

Here is a new example. It right justifies the addressed lines, something of mild utility, but too special purpose to keep around as a real program. It only takes a couple of minutes, though, to write a **Qed** buffer to do it, which can then be saved away:

rjust.q program listing

```
" Right justify addressed lines (default to (1,$))
zL#=\zM yf'fi
    1,$zL#r
"fi
" The white space below is a space and a tab
\zL,\zMs/^[ 	]*//
\zL,\zMs/[ 	]*$//
zW:0
\zL,\zM g/^/ zC#l#<\czW yt zW:\czC
zW#>35 yf zW:35
zD:                    |
zD)\zW
\zL,\zM s/^/\zD/
" Turn spaces into periods
zD+14
\zL,\zM s/^ ,\(\zD\)$/\1/
zD-14
\zL,\zM s/^\zD//
zL:\N zM:\N zC:\N zD:\N zW:
```

(Another new command (sorry!): `zC#l` sets register `C` to the length of the current line.)

This buffer illustrates how command buffers use the `(zL,zM)` address pair. Clearing the registers afterwards is a good practice for program buffers to follow.

Exercise

Why is there no `\N` on the end of the last line?

To invoke this program on a suitable buffer full of, say, words one to a line, we save it away in `/home/rob/q/right.q` and type:

```
ba  " where the data is
,p
    excle
    ficatings
    criminter
    con
    explasence
    des
    ofh
    fultesibe
    shispensitment
    dedgearing
    expers
" yes, they're random words
\zrright
,p
             excle
         ficatings
         criminter
               con
        explasence
               des
               ofh
         fultesibe
    shispensitment
        dedgearing
            expers
```

As the *Ronco* man would say, “Isn’t that amazing!”

Can we do anything useful with all this power? Well, we can write a buffer `un` (for *run* or *unix*) which pipes the addressed lines out to a shell command line, and replaces them in the buffer with the output of the command. This functionality is now largely provided by the *pipe* command (`|`), but creating an implementation of *pipe* in pure **Qed**, is itself instructive:

un.q program listing

```
" un.q -- replace addressed lines of current buffer by result
"      of passing them through pipeline
"      Looks in z| for pipeline; if empty, prompts & reads from terminal
"      Called as addr1, addr2 \zrun; defaults to (1,$).
z|=
yf'fi
    ""<> "
    z|:\l
"fi
zL#=\zM yf 1,$zL#r
ovr
\zL,\zM > \z| > /tmp/qed
zT#t    " zT gets return status from truth
\zMr /tmp/qed
!rm /tmp/qed
ovs
zT#=0 yf'else
    ""Invalid status return - lines not deleted
    y'fi
"else
    \zL,\zMd
"fi
zL:\NzM:\NzT:
""!\N
```

The prompt is reminiscent of *crunch-zap*. The `yf'else` tests the return status of the command, and decides not to delete the original lines if the status was bad (*i.e.* non-zero). Using the `\zrun` (*run buffer* `un.q`) combination, we can process the data in a buffer through any arbitrary pipeline, such as

```
,p
    excle
    ficatings
    criminter
    con
    explasence
    des
    ofh
    fultesibe
    shispensitment
    dedgearing
    expers
\zrun sort
    !
,p
    con
    criminter
    dedgearing
    des
    excle
    expers
    explasence
    ficatings
    fultesibe
    ofh
    shispensitment
```

To send out only a portion of the buffer to the pipeline, the usual convention is used:

```
.,/ful/ \zrun sort
```

11. Final Comments

Qed is a large system, but its concepts are, for the most part, simple extensions from those of **Ed**. Although it provides no new functionality in **UNIX**, it can greatly simplify many text-manipulation tasks, ranging from day-to-day editing problems to production-level text processing. By striking a harmonious balance between **Qed** and **UNIX**'s other tools, the intelligent user will find **Qed** powerful, flexible, easy to master, and fun!

12. BUGS

The Tutorial really does assume fluency in **Ed**. An updated Tutorial for a modern audience should certainly begin with an introduction to the line-oriented editing paradigm, and the basic **Ed**-like functionality in **Qed**.

Some of the original examples are pretty anachronistic, and would have seemed less exotic to someone sitting at a U. of T. terminal back in the early '80s. An updated Tutorial should choose examples which would seem familiar to today's audience. Perhaps some programs for doing common `git` tasks.

The section on Registers needed a major overhaul, as the mini-languages used in register and numeric-register operations had changed significantly.

There were surprisingly few typos in the original, quite a feat considering that many of the examples had **Qed** code interspersed with *troff* code!

13. HISTORY

Originally written by Robert Pike.

This version was edited and updated by Sean Jensen.

14. RESOURCES

Rob Pike's original U. of T. Qed tarball: https://github.com/arnoldrobbins/qed-archive/unix-1992

Sean Jensen's port of Qed, including this tutorial: https://github.com/phonologus/QED