Summer 2013 Handout 8

▼ Homework 8.1: Concurrent Versions System (CVS)

```
http://www.nongnu.org/cvs/
```

If we put a file under the protection of CVS, the people who want to edit the file can be forced to do so one at a time. CVS sits on top of the Revision Control System RCS. An older program similar to RCS was the Source Code Control System SCCS.

Do not attempt this homework before I discuss it in class, or if you were absent when I discussed it in class, or if you have not copied my **`.profile`** file into your home directory, or if you do not currently have a file in the **`~mm64/public_html/INFO1-CE9545/bio`** directory (make sure that I haven't moved your file to the **`~mm64/public_html/INFO1-CE9545/bb`** directory), or if there is other due homework which you have not handed in, or if your **`vi`** says **`[Using open mode]`** when you launch it (Handout 3, p. 1), or if **`echo $S45`** shows you nothing (Handout 2, p. 13, lines 29−30). Do not write a lawyer joke.

Do not edit any file in the **`~mm64/45/cvs`** directory or its subdirectories. Do not move or copy any file into, or remove any file from, the **`~mm64/45/cvs`** directory or its subdirectories.

(1) The **`jokes`** file is stored in the directory **`~mm64/45/cvs/CVSROOT`** (the "repository") and its subdirectories. Before giving any CVS commands, you must put the name of this directory's parent into the environment variable **`$CVSROOT`**. If you use CVS frequently, you can do this in your **`.profile`** file. For environment variables, see Handout 3, p. 16; see Handout 4, p. 4.

```
1$ export CVSROOT=~mm64/45/cvs

2$ echo $CVSROOT
/home1/m/mm64/45/cvs
```

To verify that **`$CVSROOT`** is an environment variable, not a local variable, see if it is output by the **`env`** program (Handout 1, p. 3, line 61).

```
3$ env | awk -F= '$1 == "CVSROOT"'
CVSROOT=/home1/m/mm64/45/cvs
```

(2) To check out the **`jokes`** file, start in your home directory.

```
4$ cd
5$ pwd
```

The **`cvs checkout`** command should create a subdirectory named **`jokesdir`** (if you do not already have one) in the current directory, and should create a file named **`jokes`** in the **`jokesdir`** directory. Let's anticipate four things that could prevent this.

(a) If the current directory already contains something named **`jokesdir`** that is not a subdirectory, you must first rename it, remove it, or move it elsewhere.

(b) If the current directory contains no subdirectory named **`jokesdir`**, make sure you have **`w`** permission for the current directory.

(c) If the current directory already has a subdirectory named **`jokesdir`**, make sure that the subdirectory is **`chmod`**'ed to at least **`rwx------`**.

(d) If the current directory already has a subdirectory named **`jokesdir`**, and the subdirectory contains anything named **`CVS`** or **`jokes`**, you must first rename them, remove them, or move them elsewhere.

```
6$ ls -ld jokesdir                          The jokesdir directory does not yet exist.
jokesdir: No such file or directory
```

The **`cvs checkout`** command should create the **`jokesdir`** directory and the **`jokesdir/jokes`** file. The **`cvs admin -l`** command (minus lowercase L) should lock the **`jokesdir/jokes`** file so that no one else can edit it and commit their changes.* Execute the two **`cvs`** commands together by joining them with a semicolon.

```
7$ cvs checkout jokesdir; cvs admin -l jokesdir/jokes

sh[1]: cvs: not found [No such file or directory]
sh[1]: cvs: not found [No such file or directory]
```

Let's verify that we created the **`jokesdir`** directory and **`jokesdir/jokes`** file:

```
8$ ls -ld jokesdir

9$ ls -l jokesdir
```

Let's verify that the **`jokesdir/jokes`** file is locked. The times are in UT:

```
10$ cvs log jokesdir/jokes | more
```

(3) Add your joke to the bottom of the **`jokesdir/jokes`** file.

```
11$ vi jokesdir/jokes
```

(4) Unlock the **`jokesdir/jokes`** file and commit the changes.

```
12$ cvs admin -u jokesdir/jokes; \
cvs commit -m 'I added the joke about the $18 bills.' jokesdir/jokes

sh[1]: cvs: not found [No such file or directory]
sh[1]: cvs: not found [No such file or directory]
```

Let's verify that the **`jokesdir/jokes`** file is now unlocked, and that a new revision has been created.

```
13$ cvs log jokesdir/jokes | more
```

Post the **`jokes`** file on the web and print it from there. Circle your joke and hand it in.

```
14$ cp jokes ~/public_html
15$ cd ~/public_html
16$ chmod 444 jokes                   r--r--r-- to make jokes visible on the web
17$ ls -l jokes
```

(5) Remove your copy of **`jokesdir`** and its contents with **`cvs release -d`** instead of **`rmdir`**. It will warn you if you have accidentally changed your copy of the **`jokes`** file after the **`commit`**. Go to the parent directory of **`jokesdir`**.

* To get these *reserved checkouts,* p. 9 of **`cvs`**(1) says to run the Perl program **`/opt/sfw/share/cvs/contrib/rcslock`**. But this program tries to run the **`rlog`** program of the Revision Control System RCS, which we no longer have on i5.nyu.edu.

```
18$ cd
19$ pwd

20$ ls -ld jokesdir                    Make sure you're in the parent directory of jokesdir.
```

We have no altered files because we have not modified any files since the **commit**.

```
21$ cvs release -d jokesdir            -d for "delete"

22$ ls -ld jokesdir                    The jokesdir directory should be gone.
jokesdir: No such file or directory
```

You can give the **cvs history** command even after you have released the directory. You can give the **cvs log** command only before you have released the directory.

```
23$ cvs history -m jokesdir
```

How I created the jokes file and put it under the protection of CVS

Don't do this. I already did it.

```
1$ export CVSROOT=~mm64/45/cvs         Create the environment variable.
2$ echo $CVSROOT
3$ cvs init             Create the directory ~mm64/45/cvs and its subdirectory CVSROOT.
4$ ls -ld ~mm64/45/cvs
5$ ls -ld ~mm64/45/cvs/CVSROOT

6$ cd
7$ pwd

8$ mkdir jokesdir
9$ chmod 755 jokesdir
10$ ls -ld jokesdir

11$ cd jokesdir
12$ vi jokes                           Create the jokes file.
13$ chmod 644 jokes
14$ ls -l jokes
```

Copy the file(s) in the current directory (and its descendants, if any) into the directory **$CVSROOT/jokesdir**.

```
15$ cvs import -m 'INFO1-CE9545/Y12.1005 Summer 2013 jokes file' jokesdir vendor release
```

Substitute commands in vi and ed **Substitute commands in sed**

In **vi**, you must not be in insert mode when you give the commands in column 1. Press **ESC** to make sure that you are not in insert mode. Each of the **vi** commands in column 1 begins with a **:** and ends with a **RETURN**.

For the presence or absence of the trailing **g**, see p. 323. For the **g/bystander/** and **v/bystander/** prefixes, see p. 325. **s** will accept all the same prefixes that were used in front of the **w** and **p** in Handout 6, p. 4; the **y** in Handout 4, p. 14; and the **s** in Handout 3, p. 4. Perl has the same **s///** commands, except that the prefix is written with the keyword **if**. **s**. See pp. 18−19 of **perlop**(1); you'll have to give the argument **-M/usr/perl5/man** to **man**.

```
:s/old/new/                            (sed has no "current line".)
:s/old/new/g
```

`:10s/old/new/`	**`10s/old/new/`**	
`:10s/old/new/g`	**`10s/old/new/g`**	
`:10,20s/old/new/`	**`10,20s/old/new/`**	
`:10,20s/old/new/g`	**`10,20s/old/new/g`**	
`:1,$s/old/new/`	**`s/old/new/`**	
`:1,$s/old/new/g`	**`s/old/new/g`**	
`:g/bystander/s/old/new/`	**`/bystander/s/old/new/`**	
`:g/bystander/s/old/new/g`	**`/bystander/s/old/new/g`**	
`:v/bystander/s/old/new/`	**`/bystander/!s/old/new/`**	*p. 110 for* **`!`**
`:v/bystander/s/old/new/g`	**`/bystander/!s/old/new/g`**	
`:10,20g/bystander/s/old/new/`	*(beyond the scope of this course)*	
`:10,20g/bystander/s/old/new/g`	*(beyond the scope of this course)*	
`:10,20v/bystander/s/old/new/`	*(beyond the scope of this course)*	
`:10,20v/bystander/s/old/new/g`	*(beyond the scope of this course)*	
`:g/bystander/+1s/old/new/`	*(***`sed`*** has no relative addressing.)*	
`:g/bystander/+1s/old/new/g`		

Substitute commands to remove entirely

I didn’t bother to show the prefix (e.g., **`:1,$`**) in front of the following **`s`** commands

`s/old//`	*Remove the first* **`old`** *from the line. No space between slashes.*
`s/old//g`	*Remove every* **`old`** *from the line.*

Substitute commands with regular expressions

You can use any regular expression in place of **`old`** or **`bystander`**; for example **`^old`**, **`old$`**, **`[Oo]ld`**, etc. See pp. 323−325 for **`s`** commands in **`ed`**, which are the same as in **`vi`**.

`s/old/new/g`	
`s/^old/new/`	
`s/[Oo][Ll][Dd]/Old/g`	*instead of 7 separate commands*
`s/^/new/`	*Add* **`new`** *to the start of the line.*
`s/$/new/`	*Add* **`new`** *to the end of the line.*
`s/max/MAX/g`	
`s/\<max\>/MAX/g`	*Handout 6, p. 14, line 12*

To write the Christmas message (Handout 5, p. 11, Homework 5.4), I created a garbage file with **`vi`** by slapping the keyboard with the heels of my palms. Then

`:1,$s/[^a-z ]//g`	*Remove every character except lowercase letters and blanks.*
`:1,$s/l//g`	*Remove every lowercase L.*
`:1,$s/[^a-km-z ]//g`	*Do both of the above with a single command.*

Substitute commands with a bystander

To make a change on every line that starts with a `#`,

```
g/^#/s/old/new/g
```

To make a change on every line that doesn't start with a `#`,

```
v/^#/s/old/new/g
```

Substitute commands with tagged regular expressions

`&` is a special character in the replacement, standing for a copy of the character(s) that were matched by the pattern (pp. 323−4). For example, the `&` in example 2 stands for the word `trash`.

```
1    :1,$s/trash/.trash/g                 The dot is not a wildcard here, so it needs no backslash.
2    :1,$s/trash/.&/g
3    :1,$s/\(trash\)/.\1/g                another way to do the same thing
4    :1,$s/-\([0-9]\{1,\}\)/(\1)/g        parenthesize every negative number; H 6:17 for \{1,\}
5    perl -pe 's/-(\d+)/($1)/g;'          easier in Perl; H 7:11 for +
```

The `trash` in the patterns in the above lines 1−3 should be `\<trash\>`. See Handout 6, p. 14, line 12.

Only the pattern, not the replacement, is a regular expression.

The two parts of a substitute command are called the

s/*pattern***/***replacement***/g**

Compare the `awk` nomenclature in Handout 4, p. 16, line 2.

```
grep '\$'                search for a dollar sign
```

The `$` and other regular expression characters are special in the pattern but not in the replacement.

```
1    :1,$s/\$/dollar/g        Requires a backslash.
2    :1,$s/dollar/$/g         ★ Requires no backslash.  Easier to say u for "undo": Handout 3, p. 3
```

The `&` is special in the replacement but not in the pattern.

```
3    :1,$s/ampersand/\&/g     Requires a backslash.
4    :1,$s/&/ampersand/g      Requires no backslash.
```

The `/` is special in the pattern and in the replacement.

```
5    :1,$s/half/1\/2/g        Requires a backslash.
6    :1,$s/1\/2/half/g        Requires a backslash.
```

This `/` is special neither in the pattern nor in the replacement.

```
7    :1,$s:half:1/2:g         Use three colons instead of three slashes.
8    :1,$s:1/2:half:g
```

The vi example that made me a true believer

```
1$ grep ',,,,'                lines with four (or more) consecutive commas
2$ grep ',.*,.*,.*,'          lines with four (or more) commas, not necessarily consecutive

3$ grep -i 'i.*i.*i.*i' /usr/dict/words | more
Mississippi
primitivism
```

```
/* part of a C program */

typedef struct {
    char   field1;
    int    field2;
    double field3;
    char  *field4;
} stooge_t;

stooge_t stooge[] = {
      1.11, 1,    'a',  "moe",
      2.22, 2,    'b',  "larry",
      3.33, 3,    'c',  "curly",
      4.44, 4,    'd',  "shemp",
      5.55, 5,    'e',  "Buster Keaton",
};
```

I typed the columns in the wrong order. The following **s** command in **vi** fixes this by taking advantage of the fact that each line has exactly four commas.

```
:11,2000s/^\(.*,\)\(.*,\)\(.*,\)\(.*,\)$/\3\2\1\4/
```

```
      'a',  1,    1.11, "moe",
      'b',  2,    2.22, "larry",
      'c',  3,    3.33, "curly",
      'd',  4,    4.44, "shemp",
      'e',  5,    5.55, "Buster Keaton",
```

Suppose one of the strings contained commas as well as blanks:

```
     5.0, 5,   'e', "moe, larry, curly, shemp",
```

Once again, we will require the last character of part 4 to be a comma. But this time, it can't be any old comma: it has to be a comma that comes right after a double quote. And not just any old double quote: it has to be a double quote that is not the first one on the line.

```
:11,2000s/^\(.*,\)\(.*,\)\(.*,\)\(.*".*",\)$/\3\2\1\4/
```

Vi calisthenics I

The following commands can also be used in **sed**, **ed**, and Perl. Remove the leading **:** in **ed**, and remove the leading **:1,$** in **sed** and Perl.

```
3
01000
00012345
123456789
```
1

:1,$s/^0*// *Remove leading zeros.*

```
3
1000
12345
123456789
```
2

```
    :1,$s/^/000000000/
```
Right-justify the numbers with leading zeros (two steps).

```
0000000003
0000000001000
00000000012345
000000000123456789
```
3

```
    :1,$s/^.*\(.........\)$/\1/
    :1,$s/^.*\(.\{9\}\)$/\1/
```
Remove all but last 9 characters.
Another way to do the same thing: Handout 6, p. 9, lines 10−11.

```
000000003
000001000
000012345
123456789
```
4

```
    :1,$s/.../&,/g
```
Insert commas every 3 digits, non-overlapping: pp. 323−4 for **&**.

```
000,000,003,
000,001,000,
000,012,345,
123,456,789,
```
5

```
    :1,$s/,$//
```
Remove trailing comma.

```
000,000,003
000,001,000
000,012,345
123,456,789
```
6

```
    :1,$s/^[0,]*//
```
Remove leading zeros and commas.

```
3
1,000
12,345
123,456,789
```
7

```
    :1,$s/.*/$&.00/
```
Add a leading dollar sign and trailing **.00**.

```
$3.00
$1,000.00
$12,345.00
$123,456,789.00
```
8

```
    :1,$s/^/***************/
```
Add 15 leading asterisks.

```
***************$3.00
***************$1,000.00
***************$12,345.00
***************$123,456,789.00
```
9

`:1,$s/^.*\(...............\)$/\1/`	*Remove all but last 15 characters; don’t need* **`$`**.
`:1,$s/^.*\(.\{15\}\)$/\1/`	*Another way to do the same thing: Handout 6, p. 9, lines 10−11.*

```
**********$3.00
******$1,000.00
*****$12,345.00
$123,456,789.00
```
10

`:1,$s/\$\./$0./`	*If there’s no digit to the left of the decimal point, add one.*
`:1,$s/\([0-9]\)\([.,]\)/\2\1/g`	*Deflation: move each decimal point and comma to the left.*
`:1,$s/\$,/*$/`	*Prevent* **`$1,000.00`** *from becoming* **`$,100.000`**
`:1,$s/0$//`	*Remove trailing zero, if there is one.*

```
**********$.30
*******$100.00
*****$1,234.50
$12,345,678.90
```
11

`:1,$s/\.$/.0/`	*If there’s no digit to the right of the decimal point, add one.*
`:1,$s/\([.,]\)\([0-9]\)/\2\1/g`	*Inflation: move each decimal point and comma to the right.*
`:1,$s/\*\(\$[0-9]\)\([0-9][0-9][0-9]\)/\1,\2/`	*Prevent* **`$100.000`** *from becoming* **`$1000.00`**
`:1,$s/$/0/`	*Add a trailing zero.*

```
**********$3.00
******$1,000.00
*****$12,345.00
$123,456,789.00
```
12

Vi calisthenics II

```
Yevgeniy Berezovskiy
Andrew  Wong
```
1

`:1,$s/ \{2,\}/ /g`	*Reduce groups of 2 or more consecutive blanks to a single blank.*
`:1,$s/[ ]\{1,\}/ /g`	*Reduce groups of 1 or more consecutive blanks and/or tabs to a single blank. (The wildcard contains one blank and one tab.)*

```
Yevgeniy Berezovskiy
Andrew Wong
```
2

```
:1,$s/^\(.*\) \(.*\)$/\2, \1/
```

```
Berezovskiy, Yevgeniy
Wong, Andrew
```
3

```
:1,$s/\( .\).*$/\1./
```

```
Berezovskiy, Y.
Wong, A.
```
4

```
    :1,$s/^\(.\).*, \(.\)/\2\1 &/
```

```
YB Berezovskiy, Y.
AW Wong, A.
```
5

```
    :1,$s/^...//
```
or **`u`** *for "undo": Handout 3, p. 3*

```
Berezovskiy, Y.
Wong, A.
```
6

```
    :1,$s/,/------,/
```
Pad the last name to six characters with dashes (two steps).

```
Berezovskiy------, Y.
Wong------, A.
```
7

```
    :1,$s/^\(......\).*,/\1,/
```
Truncate the last name to at most 6 characters: Procrustean bed.

```
Berezo, Y.
Wong--, A.
```
8

```
    :1,$s/./& /g
    :1,$s/ $//
```
Horizontal double space.
Remove trailing blank.

```
B e r e z o ,   Y .
W o n g - - ,   A .
```
9

```
    :1,$s/\(.\) /\1/g
```
Remove the double space.

```
Berezo, Y.
Wong--, A.
```
10

The first command puts the tags around everything up to but not including the first comma. The second command puts the tags around everything up to but not including the last comma. If the line has exactly one comma, the commands do the same thing. See the three colons in Handout 8, p. 7, lines 7−8.

```
    :1,$s:^[^,]*:<EM>&</EM>:
    :1,$s:[^,]*$:<EM>&</EM>:
```
Surround last name w/ pair of tags: Handout 3, p. 9, l. 50.

```
<EM>Berezo</EM>, Y.
<EM>Wong--</EM>, A.
```
11

A problem case

`vi` Calisthenics II assumes that each line contains two names. What if some lines contain three?

```
Yevgeniy Berezovskiy
```

```
1 grep '^[^ ]* [^ ]*$'
2 grep '^[^ ]* [^ ]* [^ ]*$'

3 :g/^[^ ]* [^ ]*$/s/old/new/g
```
1 *Output the lines that have exactly one space.*
2 *Output the lines that have exactly two spaces.*
3 *Change* **`old`** *to* **`new`** *only on lines that have exactly one space.*

```
 4 :g/^[^ ]* [^ ]* [^ ]*$/s/old/new/g   Change old to new only on lines that have exactly two spaces.
```

Chop off a fixed number of characters

The following commands have no effect on a line with less than three characters.

```
 1 s/^...//             Remove the first three characters; ^ is optional.
 2 s/...$//             Remove the last three characters; $ is required.

 3 s/^\(...\).*$/\1/    Remove all but the first three characters.
 4 s/^.*\(...\)$/\1/    Remove all but the last three characters.

 5 s/^\(..\).../\1/     Remove the third, fourth, and fifth characters; no effect on line with less than 5 chars.
 6 s/...\(..\)$/\1/     Remove the fifth from last, fourth from last, and third from last characters
```

Chop off a variable number of characters

The following commands have no effect on a line that has no colons.

```
 1 s/://                Remove the first colon.
 2 s/\(.*\):/\1/        Remove the last colon.
 3 s/:\([^:]*\)$/\1/    Another way to remove the last colon.

 4 s/:.*$//             Remove everything after and including the first colon.
 5 s/:.*$/:/            Remove everything after but excluding the first colon.

 6 s/^.*://             Remove everything up to and including the last colon.
 7 s/^.*:/:/            Remove everything up to but excluding the last colon.

 8 s/^[^:]*://          Remove everything up to and including the first colon.
 9 s/^[^:]*:/:/         Remove everything up to but excluding the first colon.

10 s/:[^:]*$//          Remove everything after and including the last colon.
11 s/:[^:]*$/:/         Remove everything after but excluding the last colon.

12 s/:[^:]*:[^:]*$//    Remove everything after and including the next-to-last colon.
13 s/^\([^:]*:\)[^:]*/\1/ Remove password field of /etc/passwd; leave 6 colons intact.
```

Often **vi** requires fewer carets and dollar signs than **grep**:

```
14 s/^.//               Remove the first character on the line.
15 s/.//                another way to do the same thing—the caret is optional
```

Some of the carets and dollar signs shown above are therefore not required, but it’s simpler to always put them in.

Definitive Statement on 'Single', "Double", and `Back` Quotes

(1) Back quotes are the simplest. They can enclose only a command whose output is to become part of a longer command.

```
1$ echo `hello`                bad: the shell will think hello is the name of a program
ksh[1]: hello: not found [No such file or directory]
```

For example, to mail a letter to everyone in the class who is logged in,

```
2$ mail `classmates` < ~/letter
```

More examples are in Handout 5, pp. 15−23.

(2) Single or double quotes make the shell treat two or more words as a single word. In the following example, the quotes prevent **grep** from thinking that you were searching for the word **To** in two files named **be** and **$S45/Shakespeare.complete**:

```
3$ grep word infile1 infile2 infile3
4$ cd $S45
5$ pwd
6$ grep 'To be' Shakespeare.complete

7$ grep To be Shakespeare.complete | head -3
grep: can't open be
Shakespeare.complete:    (Townsman:)
Shakespeare.complete:    To marry Princess Margaret for your grace,
Shakespeare.complete:    So, in the famous ancient city, Tours,

8$ awk '{print $1}' infile          {print $1} must be a single argument
9$ awk {print $1} infile
awk: syntax error near line 1
awk: illegal statement near line 1

10$ x=hello                         Handout 4, p. 3
11$ cf=chow fon                     Puts only the word chow into $cf.
12$ cf='chow fon'
```

```
for things in flowers 'young girls' husbands soldiers        Handout 4, p. 8
```

To print a file named **starmaker** with a title, page number, and date on top of each page,

```
13$ lpr starmaker              1937 science fiction novel by Olaf Stapledon (1886−1950)
14$ pr -l51 starmaker | lpr              add page headings: minus lowercase L 51
15$ pr -h Starmaker -l51 starmaker | lpr
16$ pr -h 'Star Maker' -l51 starmaker | lpr
```

For the **-i** option, see Handout 4, pp. 1−2; Handout 6, p. 9.

```
17$ pr -i' '1 -l1 -m -t -w80 $S45/encrypted decrypted | head -38
```

(3) Use single or double quotes when a string contains leading or trailing blanks or tabs:

```
18$ prog1 | tr : ' ' | prog2             Change every colon to a blank.
```

(4) Certain characters have special meanings for the shell, including

```
| < > ' " ` ( ) { } & ; $ * ? [ ] # \ ~
```

and **!** in the C shell (**csh**(1) p. 3). For example,

```
19$ expr 2 + 3
20$ expr 2 '*' 3
21$ expr 2 \* 3           A backslash can turn off the special meaning of only one character.
22$ awk '{print $3}'  A pair of quotes can turn off the special meanings of many characters.

23$ egrep 'prochoice|prolife' infile        What would the error be without quotes?
24$ echo '>>>>>>>>>>>>>>>>>>>>>>>>>'
25$ echo 'Was it good or bad? \c'           Output a blank after the question mark.
```

The arguments of **grep**, **egrep**, **sed**, **awk**, and Perl are so rich in these special characters that they should always be quoted for safety:

```
26$ grep ’ism$’
27$ sed ’s/:.*//’
28$ awk ’{print $2}’
29$ perl -ane ’print "$F[0]\n";’
```

means **awk ’{print $1}’**

Single quotes can be used to surround a double, and vice versa.

```
30$ echo ’He said "Make my day".’
31$ echo "Monday’s child is fair of face"
32$ echo Monday\’s child is fair of face
33$ echo There are three kinds of quotes: "’" ’"’ ’‘’
```

```
#!/bin/ksh
#Output the number of single quotes in the input.
#See Handout 5, p. 11 for fold and tr -c.
#Sample use: quotecount < infile

fold -b -1 | grep "’" | wc -l
tr -cd "’" | wc -c                  #simpler way to do the same thing

exit 0
```

(5) If the value of a shell variable could be the null string, surround it with double quotes. When the variable becomes null, the quotes will hold the variable’s place and avoid a syntax error. Examples were in Handout 5, pp. 14, 19; Handout 7, pp. 6, 24.

```
read verdict
if [[ "$verdict" == bad ]]
```

(6) So what’s the difference between single and double quotes? The shell normally expands all three kinds of abbreviations:

$*variables*	*wildcards*	**‘***back quotes***‘**
echo $x	**rm *.c**	**mail ‘classmates‘ < ~/letter**
echo $PATH	**rm file?.html**	**if [[‘who | wc -l‘ -gt 20]]**
if [[$1 == Mon]]	**rm file[1-6].html**	**for filename in ‘ls -t‘**

Within single quotes, however, the shell expands no abbreviations, and within double quotes the shell expands only variables and back quotes.

```
34$ rm *                         Remove all the files in the current directory.
35$ rm ’*’                       Remove only the file whose name is *.

36$ echo $HOME
/home1/a/abc1234

37$ echo ’$HOME’
$HOME
```

This is why we use double quotes instead of single quotes around a shell variable and/or back quotes:

```
if [[ "$verdict" == bad ]]
if [[ "‘echo $verdict | tr ’[A-Z]’ ’[a-z]’‘" == bad ]]
```

Gerrymandering:
pp. 112−113, 126

```
1$ grep '^.[0-9]'                  Sunday: lines whose second character is a digit (Homework 6.6)
2$ grep '^....[0-9]'               Monday: lines whose fifth character is a digit
3$ grep '^.......[0-9]'            Tuesday: lines whose eighth character is a digit

4$ grep '^.\{1\}[0-9]'             See Handout 6, p. 9, lines 10−11 for \{\}
5$ grep '^.\{4\}[0-9]'
6$ grep '^.\{7\}[0-9]'

7$ n=1
8$ n=4
9$ n=7

10$ grep "^.\{$n\}[0-9]"           $n won't work within single quotes
11$ grep '^.\{'$n'\}[0-9]'         Gerrymander (Polish corridor): pp. 112−113, 126

12$ grep '^[^aeiou]*a[^aeiou]*e[^aeiou]*i[^aeiou]*o[^aeiou]*u[^aeiou]*$'
13$ c='[^aeiou]*'                                     Handout 6, p. 17, Homework 6.8
14$ grep '^'$c'a'$c'e'$c'i'$c'o'$c'u'$c'$'
15$ grep "^${c}a${c}e${c}i${c}o${c}u${c}\$"           {curly braces} in Handout 4, pp. 9−10
```

http://i5.nyu.edu/~mm64/INFO1-CE9545/src/needksh

```
#!/bin/ksh
#Output the loginname of everyone in the class whose
#login shell is not /bin/ksh.
#set -x

for loginname in `~mm64/bin/roster 45`
do
    awk -F: '$1 == "'$loginname'" && $NF != "/bin/ksh" {print $1}' /etc/passwd
done
```

If we uncomment the **set -x** (Handout 5, pp. 8−9), the standard error output would show that the **awk** is unaware that a shell variable was used.

```
16$ needksh
+ /home1/m/mm64/bin/roster 45
+ awk -F: '$1 == "yb610" && $NF != "/bin/ksh" {print $1}' /etc/passwd
+ awk -F: '$1 == "ic297" && $NF != "/bin/ksh" {print $1}' /etc/passwd
```
etc.

In Handout 7, p. 9, there is no need to hardwire the **abc1234** into the argument of **awk**.

```
17$ /bin/rm $(ls -l ~/.trash | tail +2 | awk '$3 != "'$(whoami)'" {print $NF}')
```

Quoting bibliography

See p. 75 for single quotes, 85 for double quotes, 86−88 for back quotes. In **ksh**(1), see p. 18 for single and double quotes, pp. 6−7 for back quotes.

Wildcards in the shell language

The shell notation differs from regular expressions. See pp. 26−29; **ksh**(1) pp. 17−18.

shell language filename abbreviation	*regular expression*
`*`	**`.*`**
`?`	**`.`**
`.`	**`\.`**
`'*'` *or* **`\*`**	**`\*`**
`[abc]`	**`[abc]`**
`[a-z]`	**`[a-z]`**
`[!a-z]` *in* **ksh** *and* **bash**	**`[^a-z]`**

□