Summer 2013 Handout 7

Build your own version of . . .

If you wrote **who** instead of **/bin/who** in the pipeline in the following shellscript, it would go into an infinite loop and crash the machine without outputting anything.

```
1$ which who
/bin/who
```

```
#!/bin/ksh
#This shellscript is ~/bin/who, my own version of who.
#Pass the arguments along, unchanged, to the real who.

echo This is my own version of who. 1>&2     #Handout 5, p. 5 for 1>&2
/bin/who $* | sort     #Handout 4, pp. 22, 24; Handout 5, p. 18 for $*
exit 0
```

```
2$ chmod a+rx who                    after you've gotten out of the editor; Handout 3, p. 18
3$ ls -l who                         Make sure the chmod worked.

4$ which who
/home1/a/abc1234/bin/who             Now which gives us different output.

5$ who
This is my own version of who.       and the following output is in alphabetical order
```

```
#!/bin/ksh
#This shellscript is ~/bin/vim, my own version of vim
#(Handout 3, p. 6).

/bin/vim -g -geometry=80x36 $*
```

```
#!/bin/ksh
#This shellscript is ~/bin/vi, my own version of vi.
#For -f see p. 140; Handout 4, p. 21; Handout 5, p. 20; ksh(1) p. 19.

if [[ -f "$1" ]]
then
    chmod u+w $1     #Turn on the leftmost w; Handout 3, p. 18.
fi

/bin/vi $1

if [[ -f "$1" ]]     #double quotes in Handout 5, p. 14
then
    chmod a-w $1     #Turn off all three w's.
fi

exit 0
```

The shell would normally change **-P*** into the names of all the things starting with **-P** in the current directory, but this is not done within **[[**double square brackets**]]**. See **ksh**(1) p. 19. The pattern must be to the right of the **==** or **!=**.

```
#!/bin/ksh
#This shellscript is ~/bin/lpr, my own version of lpr.
#Pass all the arguments along, unchanged, to the real lpr.
#If an argument starts with -P, pass it to lpq.
#Otherwise, give lpq no arguments.
#"in $*" is optional: pp. 144-145; Handout 4, p. 24; ksh(1) p. 2.

/usr/ucb/lpr $*

for argument in $*
do
    if [[ $argument == -P* ]]
    then
        printer=$argument
        break        #abandon the "for" loop: p. 160; ksh(1) p. 39
    fi
done

lpq $printer
exit 0
```

If we feed only one input file to **grep**, it will show us each line that it finds. If we feed it two or more input files, it will also show us the name of the file where it found each line. We therefore feed it the permanently empty file **/dev/null** as well as the file we want to search.

```
#!/bin/ksh
#This shellscript is ~/bin/gcc, my own version of gcc (Handout 3, p. 24).
#Complain if any .c file contains if (a = b) with a single equal sign.
#See Handout 6, p. 17.

cleanbill=1             #of health

for argument
do
    if [[ $argument == *.c ]]
    then
        if grep -n 'if *(.*[^=!<>]=[^=].*)' $argument /dev/null 1>&2
        then
            cleanbill=0
        fi
    fi
done

if [[ $cleanbill -eq 0 ]]
then
    exit 1              #Do not attempt to compile.
fi

#This shellscript's exit status will be that of the real gcc.
/opt/gcc/bin/gcc $*
```

Build your own version of the Unix rm command: p. 130, ex. 4−14

```
1$ cd
2$ pwd

3$ mkdir .trash
4$ ls -l | more                    Why doesn't ls -l list the .trash subdirectory?
5$ ls -la | more                   "all", even the ones whose names start with dot
```

```
#!/bin/ksh
#This shellscript is ~/bin/rm, my own version of rm.
#Instead of destroying the file, move it to my ~/.trash directory
#in case I change my mind later.

mv $1 ~/.trash
```

```
6$ date > junk                     Don't experiment with a valuable file.
7$ rm junk
8$ ls -l                           junk is gone.

9$ cd ~/.trash
10$ pwd
/home1/a/abc1234/.trash

11$ ls -l | more                   Now you see junk.
12$ /bin/rm junk                   Run the real rm.
13$ ls -l | more
```

(1) What goes wrong when you try

```
14$ rm junk
15$ rm garbage
16$ rm junk
```

The process ID number **$$** will be different each time you run this shellscript. See p. 146; Handout 3, p. 23; **ksh**(1) p. 12.

```
#!/bin/ksh
#This shellscript is ~/bin/rm, my own version of rm.
#Rename the file as you move it to the ~/.trash directory.

mv $1 ~/.trash/$1$$
```

```
17$ date
Tue May 28 15:22:52 EDT 2013

18$ date | tr ' ' -
Tue-May-28-15:22:52-EDT-2013
```

```
#!/bin/ksh
#This shellscript is ~/bin/rm, my own version of rm.
#Rename the file as you move it to the ~/.trash directory.

mv $1 ~/.trash/$1$$‘date | tr ' ' -‘
```

(2) When you say

```
19$ rm ~/junk
```

rm actually sees this:

```
20$ rm /home1/a/abc1234/junk
```

each of the two **$1**'s in the first shellscript in ¶ (1) will balloon into **/home1/a/abc1234/junk**. The shellscript will therefore attempt to execute the erroneous command

```
21$ mv /home1/a/abc1234/junk /home1/a/abc1234/.trash//home1/a/abc1234/junk123
```

Oddly enough, the double slash is not a problem. The command is wrong because it mentions a subdirectory of **/home1/a/abc1234/.trash** that does not exist.

It's okay to use the entire **$1** as the first argument of the **mv**, but we want to use only the rightmost segment of **$1** in the second argument of **mv**. ***/** looks for any string ending with a slash. And **##** tells the shell to use the preceding variable (in this case, **$1**) with the following string (in this case, the longest one ending with a **/**) removed from the front of the variable. For example,

```
22$ echo $HOME
/home1/a/abc1234

23$ echo ${HOME##*/}
abc1234
```

See Handout 4, pp. 3−4, 24; **ksh**(1) pp. 10−12.

```
#!/bin/ksh
#This shellscript is ~/bin/rm, my own version of rm.
#Rename the file as it is moved to the ~/.trash directory.
#Use only the basename from $1.

mv $1 ~/.trash/${1##*/}$$
```

```
24$ mv /home1/a/abc1234/junk /home1/a/abc1234/.trash/junk123
```

(3) What goes wrong when you try

```
25$ rm junk garbage
```

You can move and rename a file with one **mv** command: we just did it. You can also move several files with one **mv** command: see Handout 2, p. 4, ¶¶ (3) and (4). But to move and rename several files, you need a separate **mv** command for each file:

```
#!/bin/ksh
#This shellscript is ~/bin/rm, my own version of rm.
#Rename three files as they are moved to the ~/trash directory.

mv $1 ~/.trash/${1##*/}$$
mv $2 ~/.trash/${2##*/}$$
mv $3 ~/.trash/${3##*/}$$

#Are three enough?  Are three too many?
```

Write one **mv** in a loop.

```
#!/bin/ksh
#This shellscript is ~/bin/rm, my own version of rm.
#Allow more than one filename.

for filename
do
    mv $filename ~/.trash/${filename##*/}$$
done
```

(4) What goes wrong when you try

```
26$ rm -i junk garbage                    "interactive" option
```

We also make our **rm** accept the **-f** option, since our **.profile** passes this option to **rm** (Handout 2, p. 13, line 37).

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

```
#!/bin/ksh
#This shellscript is ~/bin/rm, my own version of rm.
#First argument can be an optional -i or -f.

interactive=0
while [[ "$1" == -i || "$1" == -f ]]   #two =s for string compare
do
    if [[ "$1" == -i ]]
    then
        interactive=1                  #one = to assign
    fi
    shift          #move args left, subtract 1 from $#, p. 155
done

for filename
do
    if [[ $interactive -eq 1 ]]   #-eq to compare two numbers
    then
        echo $0: remove $filename'? \c'   #Hand 5, pp. 14, 21  for \c
        read answer
        if [[ "$answer" != y* ]]          #Like -P* in Handout 7, p. 2.
        then
            continue          #go back to top of "for": ksh(1) p. 39
        fi
    fi
    mv $filename ~/.trash/${filename##*/}$$
done

if [[ -t 2 && `ls ~/.trash | wc -l` -ge 20 ]]
then
    echo $0: time to empty or compress the .trash 1>&2
fi

exit 0
```

```
27$ rm -i junk garbage
rm: remove junk? yes
rm: remove garbage? no
rm: time to empty or compress the .trash
28$
```

(5) What goes wrong when you try to remove a file whose name starts with a dash?

```
29$ rm -myfile
```

The **--** means that none of the following arguments are options (Handout 6, pp. 10−11, line 19). The following arguments must therefore all be filenames or directory names.

```
#Change the mv in the above shellscript to
mv -- $filename ~/.trash/${filename##*/}$$
```

Run a command once at a future time: p. 35

Give the **at** command in the directory that should be the program's current directory when it runs in the future. Your account must be ''unlocked''; see **at**(1) and **shadow**(4).

```
#!/bin/ksh
#This shellscript is ~/bin/wee.

date > ~/wee.out
```

```
1$ cd
2$ pwd

3$ at -f bin/wee 3:15am
commands will be executed using /bin/ksh
job 1369811700.a at Wed May 29 03:15:00 2013

4$ at -l                                minus lowercase L to list your at jobs.
1369811700.a Wed May 29 03:15:00 2013
```

You can even supply a day, month, and year:

```
5$ at -f wee 3:15am May 28, 2013
```

To discover the abbreviation for the month name,

```
6$ locale -ck LC_TIME | awk -F= '$1 == "mon" || $1 == "abmon"'
mon="January";"February";"March";"April";"May";"June";"July";"August";"September";"October";"Novem
abmon="Jan";"Feb";"Mar";"Apr";"May";"Jun";"Jul";"Aug";"Sep";"Oct";"Nov";"Dec"
```

Unix was invented at midnight on January 1, 1970:

```
7$ bc                   "binary calculator": Handout 2, pp. 20−21; Handout 4, pp. 25−26
scale = 5               request answers with five digits to right of decimal point
1369811700 / (60 * 60 * 24 * 365.25)
43.40671
control-d
8$

9$ cd /var/spool/cron/atjobs        man at told me which directory to go to
10$ pwd
/var/spool/cron/atjobs
```

For the strange uppercase **S** permission, see Handout 6, p. 14, Homework 6.7.

```
11$ ls -l | more                    see a list of everyone's at jobs
-r-Sr--r--   1 mm64     users       4377 May 28 15:22 1369811700.a

12$ grep '^cd ' 1369811700.a | head -1
cd /home1/a/abc1234

13$ rm 1369811700.a
rm: 1369811700.a: Permission denied
```

To see why **rm** refused to remove your file **1369811700.a**,

```
14$ ls -ld | more
drwxr-xr-x   2 root     sys            4 May 28 15:22 .

15$ at -l                           minus lowercase L for "list"
16$ at -r 1369811700.a              "remove"
17$ at -l                           Make sure the file was removed.
```

Run a shellscript with command line arguments

The program that you run with **at** cannot have command line arguments or I/O redirection. That's one of the reasons why the following command fails.

```
1$ at -f mail 3:15am abc1234@hostname.com < ~/letter          wrong
```

The workaround is to create a shellscript containing the arguments and redirection. Then run the shellscript with **at**:

```
#!/bin/ksh
#This shellscript is named latemail.

mail abc1234@hostname.com < ~/letter
```

```
2$ at -f latemail 3:15am
```

Run a command at regular intervals forever: p. 129

Why do you now have to say **/bin/rm** instead of **rm**?

```
#!/bin/ksh
#This shellscript is named cleanup.
#It removes all the files in the ~/.trash directory.

at -f cleanup 3:15am tomorrow
/bin/rm ~/.trash/*
```

The first line of the shellscript could also be one of

```
at -f cleanup now + 2 days
at -f cleanup now + 48 hours          sounds more operational
at -f cleanup now + 15 minutes
at -f cleanup midnight next week
```

In your **.profile** you could say

```
if [[ 'at -l | wc -l' -le 0 ]]
then
    echo Your at job is no longer scheduled. 1>&2
fi
```

More selective versions of the cleanup shellscript

The above shellscript has a bug: it tries to remove subdirectories as well as files. To fix it, pipe the output of **ls -lat** into **grep '^-'** to list only the files, not the subdirectories. See the **rm** examples in Handout 5, p. 16.

```
#!/bin/ksh
#Remove the five oldest files in the ~/.trash directory.

at -f cleanup 3:15am tomorrow
/bin/rm 'ls -tr ~/.trash | head -5'
```

```
#!/bin/ksh
#Remove the five largest files in the ~/.trash directory.  This is
#harder than the previous example because ls has no "size order"
#option.  sort +4nr ignores the first 4 fields on each line (p. 106).

at -f cleanup 3:15am tomorrow

/bin/rm `ls -l ~/.trash | tail +2 |
    sort +4nr | awk 'NR <= 5 {print $NF}'`
```

```
#!/bin/ksh
#Remove the files in the ~/.trash directory that belong to people
#other than abc1234.

at -f cleanup 3:15am tomorrow

/bin/rm `ls -l ~/.trash | tail +2 | awk '$3 != "abc1234" {print $NF}'`
```

Run a command at regular intervals for a finite number of times

To print 10 copies of a file at 15-minute intervals, create a file named **`~/count`** and write the number 10 in it:

`1$ echo 10 > ~/count` *faster than* **`vi ~/count`**

Then run the following shellscript.

```
#!/bin/ksh
#This shellscript is ~/bin/multi.  Print multiple copies of myfile
#at 15 minute intervals.  The number of copies is read from the file
#~/count.

if [[ ! -f ~/count ]]         #p. 140 for -f, 143 for !
then
    echo $0: must first create ~/count 1>&2
    exit 1
fi

n=`cat ~/count`               #Handout 5, pp. 19-22 for =``

if [[ $n -gt 0 ]]
then
    at -f ~/bin/multi now + 15 minutes
    lpr ~/myfile
    let n=n-1                 #Handout 4, pp. 24 for let
    rm ~/count                #noclobber: Handout 2, p. 13, ll. 57-58
    echo $n > ~/count
else
    rm ~/count
fi

exit 0
```

A gateway that counts how many times it's been executed

Create a **rw-rw-rw-** file named **~/public_html/cgi-bin/count**, containing the number **0**.

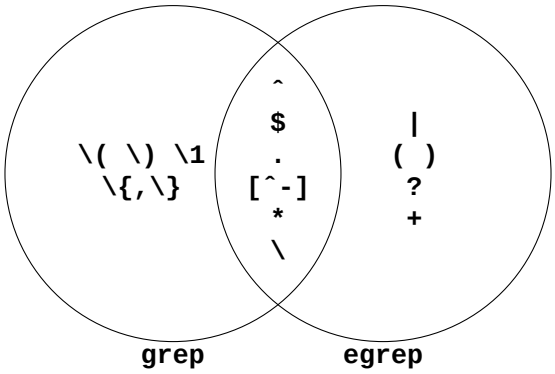
```
#!/bin/ksh
#This file is /home1/a/abc1234/public_html/cgi-bin/home_page

echo Content-type: text/html
echo
echo '<HTML>'
echo '<HEAD>'
echo '<TITLE>John Doe</TITLE>'
echo '</HEAD>'
echo '<BODY>'
echo "<H1>John Doe's Home Page</H1>"  #doubles around single
echo Welcome to my home page.
echo You are visitor number

n=`cat /home1/a/abc1234/public_html/cgi-bin/count`
let n=n+1
rm /home1/a/abc1234/public_html/cgi-bin/count
echo $n > /home1/a/abc1234/public_html/cgi-bin/count

echo $n.
echo '</BODY>'
echo '</HTML>'
exit 0
```

Your **index.html** file could then be

```
<HTML>
<HEAD>
<TITLE>John Doe</TITLE>
</HEAD>
<BODY>
Click
<A HREF = "/cgi-bin/cgiwrap/abc1234/home_page">here</A>
to enter.
</BODY>
</HTML>
```

grep vs. egrep

/usr/xpg4/bin/grep has a **-E** option that makes it smart enough to do everything that **egrep** can do. Perl can do everything that **grep** and **egrep** can do, and does not require (or permit) the back-slashes in **\(\) \{ \}**.




```
1$ egrep '^(..)*$'                    lines that consist of an even number of characters
2$ egrep '^..*$'                      lines that consist of one or more characters
3$ egrep '^.+$'                       lines that consist of one or more characters
```

A **|** in the argument of **egrep** means "or"; see p. 104.

```
4$ egrep 'prochoice|prolife'          No blanks around the |.  Why do we need quotes?
5$ egrep 'pro(choice|life)'           a*b+a*c = a*(b+c)  Why do we need parentheses?
6$ egrep '^(anti|pro)(choice|life|abortion)'      six combinations, no space

7$ egrep -i 'q([^u]|$)'               needed 2 grep's in Handout 6, p. 13, first lines 7-8
8$ egrep '(^|[^0-9])100([^0-9]|$)'    needed 4 grep's in Handout 6, p. 13, last lines 1-5

9$ egrep 'how you have to look for \(parentheses\) or \| bars in egrep'
10$ egrep 'A|B|C|D|F'                 Don't use | if each alternative is exactly one character.
11$ grep '[ABCDF]'                    shorter way to do the above; no longer need egrep
12$ grep '[A-DF]'                     even shorter way to do the above

13$ egrep 'colou?r'                   optional u: Handout 1, p. 8, line 5
14$ egrep 'Homoi?ousian' decline_and_fall_21      optional i
15$ egrep 'dialog(ue)?s'              optional ue
16$ egrep 'Oh No+!'                   Mister Bill: one or more consecutive o's
17$ egrep 'Tra( la)+!'
18$ egrep ':\+\)'                     look for the smiley face :+)
```

RFC (Request for Comments) 1034 domain name: one or more dot-separated labels, each starting with a letter. If the label contains additional characters, the last character must be a letter or digit. The characters in the middle of the label could be letters, digits, or hyphens. A label must be less than 64 characters long.

```
19$ perl -ne 'print if /^[a-z]([a-z0-9-]{0,61}[a-z0-9])?$/i;'

20$ perl -ne 'print if
/^[a-z]([a-z0-9-]{0,61}[a-z0-9])?(\.[a-z]([a-z0-9-]{0,61}[a-z0-9])?)*$/i;'

21$ perl -ne '$w = "[a-z]([a-z0-9-]{0,61}[a-z0-9])?"; print if /^$w(\.$w)*$/i;'
```

RFC 5322, §3.4.1 says that an email address has a **@** in the middle. Let's assume that we can have at most 64 characters before the **@**. The first and last characters before the **@** cannot be periods. Other than that, the characters can be letters, digits, or any one of **#$%'*+/=?^_'{|}.-**. You can't have consecutive periods. To put a **$** or **@** into the middle of a double-quoted Perl string or into a diagonal-slashed regular expression, you have to write a backslash in front of them. To put a single quote into a single-quoted argument in the Korn shell, you have to write a backslash in front of it.


```
22$ perl -ne '
    $c = "a-z0-9#\$%\'*+/=?^_`{|}-";
    $w = "[a-z]([a-z0-9-]{0,61}[a-z0-9])?";
    print if /^[$c][.$c]{0,62}[$c]\@$w(\.$w)*$/i && !/\.\./;
'
```

"Or more"

	means	*can be used in*
*****	*zero or more consecutive copies*	**grep** *and* **egrep**
?	*zero or one copy*	**egrep** *but not* **grep**
+	*one or more consecutive copies*	**egrep** *but not* **grep**

For repetition counts with **\{\}** inside the argument of **grep**, see **regexp**(5) pp. 2−3, ¶ 2.3.

Tagged regular expressions: \(\) \1

See **regexp**(5) p. 3, ¶¶ 2.5 and 2.6. The text only hints about tagged regular expressions on pp. 105, 326−327.

Output the lines that contain a double character:

```
1$ grep '..' /usr/dict/words                    Why doesn't this work?
2$ grep -i 'aa' /usr/dict/words                 brute force
3$ grep -i 'bb' /usr/dict/words
4$ grep -i 'cc' /usr/dict/words
```

As in Handout 6, p. 9, the backslashes turn *on* the special meaning that the characters **(** and **)** have to **grep**. The parentheses were added as an after thought.

```
5$ grep -i '\(.\)\1' /usr/dict/words            a better way
accept
too
```

It's illegal to use **\1** unless you have a **\(\)** somewhere to its left—otherwise, the computer wouldn't know what **\1** is asking for another copy of. And it's unnecessary to use **\(\)** unless you have a **\1** somewhere to its right—there's no reason to name part of the regular expression unless you're planning to mention the name farther on.

Output the lines that begin with a double character:

```
6$ grep -i '^\(.\)\1' /usr/dict/words
eel
ooze
```

Output the lines that begin and end with the same character:

```
7$ grep -i '^\(.\).*\1$' /usr/dict/words
algebra
Celtic
```

Output the lines that contain a triple character:

```
8$ grep -i '\(.\)\1\1' /usr/dict/words
IEEE            Institute of Electrical and Electronics Engineers
viii            Roman numeral 8
```

Output the lines that consist of two or more identical characters:


```
grep '^\(.\)\1\1*$' /usr/dict/words
AAA
ii
iii
```

Output the lines that contain only a double, but not a triple, character:

```
9$ grep -i '\(.\)\1' /usr/dict/words | grep -iv '\(.\)\1\1'
vii
```

▼ Homework 7.1: illegal, immoral, fattening —Alexander Woollcott (1887−1943)

Count the lines in the files **/usr/dict/words** and **/usr/dict/websters** that start with a lowercase ''i'', followed by a double letter. (The two copies of the double letter must be the same case.) Count only the lines that are at least six characters long. You get no credit if you count ''inner'' or if you fail to count ''immune''. On May 28, 2013, 91 of the 25,146 lines in the file **/usr/dict/words**, and 744 of the 234,936 lines in the file **/usr/dict/websters**, matched this pattern.

Use exactly one **grep**. What do most of these words have in common?

▲

Give names to two or more sections

```
1$ grep -i '\(.\)\1\(.\)\2' /usr/dict/words
raccoon
Tallahassee

2$ grep -i '^\(.\)\1\(.\)\2' /usr/dict/websters
eellike

3$ grep -i '^\(.\)\1.*\(.\)\2$' /usr/dict/words
eelgrass

4$ grep -i '\(.\)\1.*\(.\)\2.*\(.\)\3' /usr/dict/words
committee
Mississippi
Tennessee
```

▼ Homework 7.2: repeated strings of characters

Find all the lines in **/usr/dict/words** and **/usr/dict/websters** that contain two consecutive copies of the same group of four characters. Use this method:

```
1$ grep -i '\(...\)\1' /usr/dict/words
alfalfa
clinging
instantaneous
murmur
```

▲

▼ Homework 7.3: lines composed of three identical parts

Find all the lines in **/usr/dict/words** and **/usr/dict/websters** that are composed of three identical parts. Each part must consist of one or more characters. Ignore the difference between upper and lowercase. 8 of the 17 lines in the file **$S45/thirds** are of this form. Don't hand it in until you agree with this result.

Use this method:

```
1$ grep -i '^\(..*\)\1$' /usr/dict/words
beriberi
ii                Roman numeral 2
murmur
tutu
```

▲

Search for palindromes

Sit on a potato pan, Otis. A man, a plan, a canal: Panama! Able was I ere I saw Elba. Do geese see God? A Toyota. N*ίψον ανομήματα μη μόναν όψιν.* (Wash your sins, not only your face.)

Output the three-character palindromes:

```
1$ grep -i '^\(.\).\1$' /usr/dict/words
eye
gag
```

Output the four-character palindromes:

```
2$ grep -i '^\(.\)\(.\)\2\1$' /usr/dict/words
peep
toot

3$ grep -i '^\(..\)\1$' /usr/dict/words                    not palindromes
Mimi
papa
```

Output the five-character palindromes:

```
4$ grep -i '^\(.\)\(.\).\2\1$' /usr/dict/words
Ababa          Addis
madam          Madam, I'm Adam.
```

▼ **Homework 7.4: six- and seven-character palindromes: hallah, reviver**

Write one **grep** command to output the six- and seven- character palindromes in **/usr/dict/words** and **/usr/dict/websters**. There are 9 six-character and 8 seven-character palindromes.

▲

Wildcards in C scanf, fscanf, and sscanf

The **scanf** in line 3 inputs all the characters up to but not including the next blank, tab, or newline. The **scanf** in line 4 inputs all the characters up to but not including the next character that is not a letter. In other words, it keeps inputting characters as long as they are letters. The **scanf** in line 5 inputs all the characters up to but not including the next colon. See **scanf**(3c), and the K&R C book, Second Edition, p. 226.

```
1     char string[100];
2
3     scanf("%s", string);
4     scanf("%[A-Za-z]", string);
5     scanf("%[^:]", string);
```


Regular expressions in C

To write a C program that searches like **grep** for a regular expression, see the functions in **regcomp**(3c) and **regexp**(5).

—On the Web at
http://i5.nyu.edu/~mm64/INFO1-CE9545/src/regexp.c

```
 1 /* Output every line of standard input that matches the regular expression. */
 2
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5 #include <sys/types.h>
 6 #include <regex.h>
 7
 8 int main(int argc, char **argv)
 9 {
10     regex_t expression;
11     const int error = regcomp(&expression, "^moe[0-9]", 0);
12     char line[1024];
13
14     if (error != 0) {
15         const size_t length = regerror(error, &expression, NULL, 0);
16         if (length == 0) {
17             fprintf(stderr, "%s: regerror not implemented\n", argv[0]);
18         } else {
19             char *const p = malloc(length);
20             if (p == NULL) {
21                 fprintf(stderr, "%s: can't malloc space for error message\n",
22                     argv[0]);
23             } else {
24                 regerror(error, &expression, p, length);
25                 fprintf(stderr, "%s: %s\n", argv[0], p);
26                 free(p);
27             }
28         }
29         return EXIT_FAILURE;
30     }
31
32     while (fgets(line, sizeof line, stdin) != NULL) {
33         if (regexec(&expression, line, 0, NULL, 0) == 0) {
34             fputs(line, stdout);
35         }
36     }
37
38     regfree(&expression);
39     return EXIT_SUCCESS;
40 }
```

What a regular expression can't do

There are some things that can't be described with a regular expression. For example, you can't **grep** for the syntactically correct lines in a file of arithmetic expressions, even assuming that each number is exactly one digit, and that the only operators are plus, minus, times, divide, and parentheses.

1+2*3	*correct*
1+2*	*incorrect*
(1+2)*3	*correct*
(1+2(*3	*incorrect*

In fact, you can't even **grep** for palindromes of any length, only for palindromes of a specific length. For these and other searching tasks, use **awk**, Perl, or **yacc**. See pp. 233−287.

Archive several files into one big .tar file

tar is a utility for writing a group of files onto a tape, creating a *tape archive.* But the output of **tar**, like that of any Unix program, can be directed to a file instead of to a hardware device. Give the file a name ending with **.tar**.

The following example could have used a device name such as **/dev/rmt/0** (raw magnetic tape) instead of the filename **date.tar**. In that case you'd also need the **mt rewind** command.

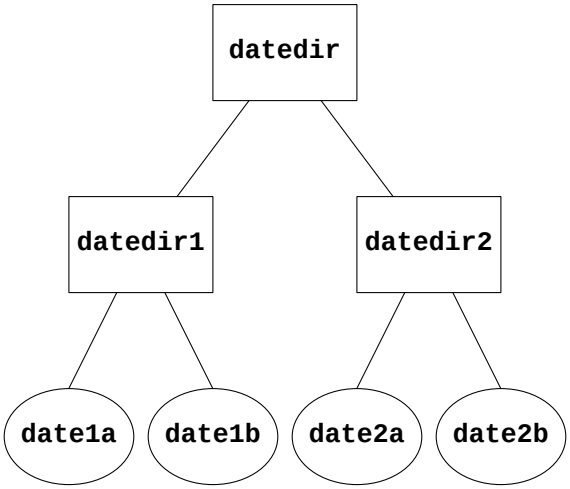
```
1$ cd
2$ echo hello > README
3$ date > date1
4$ date > date2
5$ date > date3

6$ ls -l README date1 date2 date3
7$ ls -l README date[1-3]           or use wildcard in p. 28; ksh(1) p. 18
-rw-------   1 mm64     users          6 May 28 15:22 README
-rw-------   1 mm64     users         29 May 28 15:22 date1
-rw-------   1 mm64     users         29 May 28 15:22 date2
-rw-------   1 mm64     users         29 May 28 15:22 date3

8$ tar cvf date.tar README date[1-3]                    create date.tar
a README 1K
a date1 1K
a date2 1K
a date3 1K

9$ ls -l date.tar
-rw-------   1 mm64     users       5120 May 28 15:22 date.tar

10$ tar tvf date.tar | more          Output a table of contents of the .tar file.
tar: blocksize = 10
-rw------- 50766/15       6 May 28 15:22 2013 README
-rw------- 50766/15      29 May 28 15:22 2013 date1
-rw------- 50766/15      29 May 28 15:22 2013 date2
-rw------- 50766/15      29 May 28 15:22 2013 date3

11$ awk -F: '$1 == "mm64"' /etc/passwd
mm64:x:50766:15:Mark Meretzky:/home1/m/mm64:/bin/ksh
```

The **x** flag extracts a copy of one or more of the little files archived in the **.tar** file. To extract everything archived in the **.tar** file, simply give no arguments after the name of the **.tar** file. The files you recreate will be owned by their original owners if you are the superuser. Extraction does not change the contents of the **tar** file.

```
12$ rm README date[1-3]
```

```
13$ tar xvf date.tar README          recreate README
tar: blocksize = 10
x README, 6 bytes, 1 tape blocks

14$ ls -l README
-rw-------   1 mm64     users          6 May 28 15:22 README
```

Archive an entire directory into one big .tar file



If you give one or more directory names instead of one or more filenames to **`tar cvf`** after the name of the **`.tar`** file, **`tar`** will archive the directories and all of their descendants, including all of the files they contain.

```
1$ cd
2$ pwd
/home1/a/abc1234

3$ mkdir datedir

4$ mkdir datedir/datedir1
5$ date > datedir/datedir1/date1a
6$ date > datedir/datedir1/date1b

7$ mkdir datedir/datedir2
8$ date > datedir/datedir2/date2a
9$ date > datedir/datedir2/date2b

10$ tar cvf date.tar datedir
a datedir/ 0K
a datedir/datedir1/ 0K
a datedir/datedir1/date1a 1K
a datedir/datedir1/date1b 1K
a datedir/datedir2/ 0K
a datedir/datedir2/date2a 1K
a datedir/datedir2/date2b 1K
```



```
11$ tar tvf date.tar | more
tar: blocksize = 13
drwx------ 50766/15       0 May 28 15:22 2013 datedir/
drwx------ 50766/15       0 May 28 15:22 2013 datedir/datedir1/
-rw------- 50766/15      29 May 28 15:22 2013 datedir/datedir1/date1a
-rw------- 50766/15      29 May 28 15:22 2013 datedir/datedir1/date1b
drwx------ 50766/15       0 May 28 15:22 2013 datedir/datedir2/
-rw------- 50766/15      29 May 28 15:22 2013 datedir/datedir2/date2a
-rw------- 50766/15      29 May 28 15:22 2013 datedir/datedir2/date2b

12$ rm    datedir/datedir[12]/*
13$ rmdir datedir/datedir[12]
14$ rmdir datedir
```

The following **tar xvf** command will re-create the directory **datedir**, its subdirectory **datedir2**, and the file **date2a**:

```
15$ tar xvf date.tar datedir/datedir2/date2a
tar: blocksize = 13
x datedir/datedir2/date2a, 29 bytes, 1 tape blocks

16$ ls -l | more
drwx------   3 mm64     users        182 May 28 15:22 datedir

17$ ls -l datedir | more
drwx------   2 mm64     users        180 May 28 15:22 datedir/datedir2

18$ ls -l datedir/datedir2/date2a
-rw-------   1 mm64     users         29 May 28 15:22 datedir/datedir2/date2a
```

Include the full pathname in the .tar file

If you specify the full pathnames of the files and directories to be archived, then their full pathnames will be stored in the **.tar** file:

```
1$ rm date.tar
2$ tar cvf date.tar ~/datedir
a /home1/a/abc1234/datedir/ 0K
a /home1/a/abc1234/datedir/datedir2/ 0K
a /home1/a/abc1234/datedir/datedir2/date2a 1K
a /home1/a/abc1234/datedir/datedir2/date2b 1K
a /home1/a/abc1234/datedir/datedir1/ 0K
a /home1/a/abc1234/datedir/datedir1/date1a 1K
a /home1/a/abc1234/datedir/datedir1/date1b 1K
```

```
3$ tar tvf date.tar
tar: blocksize = 13
tar: Removing leading '/' from '/home1/a/abc1234/datedir/'
tar: Removing leading '/' from '/home1/a/abc1234/datedir/datedir2/'
tar: Removing leading '/' from '/home1/a/abc1234/datedir/datedir2/date2a'
tar: Removing leading '/' from '/home1/a/abc1234/datedir/datedir2/date2b'
tar: Removing leading '/' from '/home1/a/abc1234/datedir/datedir1/'
tar: Removing leading '/' from '/home1/a/abc1234/datedir/datedir1/date1a'
tar: Removing leading '/' from '/home1/a/abc1234/datedir/datedir1/date1b'
drwx------ 50766/15       0 May 28 15:22 2013 tmp/23949.dir/datedir/
drwx------ 50766/15       0 May 28 15:22 2013 tmp/23949.dir/datedir/datedir2/
-rw------- 50766/15      29 May 28 15:22 2013 tmp/23949.dir/datedir/datedir2/date2a
-rw------- 50766/15      29 May 28 15:22 2013 tmp/23949.dir/datedir/datedir2/date2b
drwx------ 50766/15       0 May 28 15:22 2013 tmp/23949.dir/datedir/datedir1/
-rw------- 50766/15      29 May 28 15:22 2013 tmp/23949.dir/datedir/datedir1/date1a
-rw------- 50766/15      29 May 28 15:22 2013 tmp/23949.dir/datedir/datedir1/date1b
```

▼ Homework 7.5: create a tar file

Verify that all of the above works. If you're allowed to use a tape drive, **`tar`** some files to tape instead of to a **`.tar`** file.

▲

▼ Homework 7.6: copy all your files from one machine to another

```
1$ cd
2$ pwd

3$ tar cvf all.tar .
4$ tar tvf all.tar | more
5$ tar tvf all.tar | lpr
```

Use **`sftp`** to copy the above **`binary`** file named **`all.tar`** to your home directory on another machine. Then on the other machine,

```
$ cd
$ pwd

$ ls -l all.tar
$ tar tvf all.tar | more
$ tar tvf all.tar | lpr
$ tar xvpf all.tar        p 'cause hosts have different umask (Hand 2, p. 14, ll. 70−73)
```

▲

Compress and uncompress

```
1$ cd
2$ cp /etc/passwd .                    Copy /etc/passwd to your current directory.
3$ ls -l passwd
-rw-------   1 abc1234  users     433820 May 28 15:22 passwd

4$ compress passwd                          remove passwd and create passwd.Z
5$ ls -l passwd.Z
-rw-------   1 abc1234  users     138164 May 28 15:22 passwd.Z
```


Do not **cat** to the screen or **lpr** a compressed file (in this case, a **.Z** file). Instead,

```
6$ zcat passwd.Z | head -3
root:x:0:0:root@i5:/root:/usr/bin/bash
daemon:x:1:1::/:
bin:x:2:2::/usr/bin:

7$ uncompress passwd.Z                    remove passwd.Z and create passwd
8$ ls -l passwd
-rw-------   1 abc1234  users     433820 May 28 15:22 passwd
```

The compression and decompression is not ‘‘lossy’’:

```
9$ cmp passwd /etc/passwd                 No output if identical: Handout 2, p. 12.
10$ rm passwd
```

Other compression programs

suffix	*compress*	*decompress*	*website*
.Z	**compress**	**uncompress**	**http://en.wikipedia.org/wiki/Compress**
.z	**pack**	**unpack**	
.zip	**zip**	**unzip**	
.gz	**gzip**	**gunzip**	**http://www.gnu.org/software/gzip/gzip.html**
.bz2	**bzip2**	**bunzip2**	**http://www.bzip.org/**
.rar	**rar**	**unrar**	**http://www.rarlabs.com/**

Programs written by **www.gnu.org** often start with **g**. To list the pairs of utilities for compression and decompression on i5.nyu.edu,

```
1$ man -k compress | more                 "keyword"; go on a fishing trip
```

▼ Homework 7.7: compress and uncompress a file

Compress and uncompress a file (ASCII or binary). Use **bc** to find the percent by which it was compressed. Which pair of utilities yields the most compression? Do text files compress further than image files?

```
1$ bc                          Handout 2, pp. 20−21; Handout 4, pp. 24−25; Handout 7, p. 7
scale = 2                      Output answers to two decimal places.
100 * 138164 / 433820
31.84
control-d
2$
```

▲

A web page that contains a form

A *form* is a part of a World Wide Web page that contains buttons, checkboxes, places to fill-in words, etc. Here are the URL's of pages containing a form:

http://i5.nyu.edu/~mm64/cal.html	*the example below*
http://i5.nyu.edu/~mm64/man/	*Unix* **man** *page*
http://i5.nyu.edu/~mm64/x52.9232/#gateway	*C program to output a flag*
http://i5.nyu.edu/~mm64/INFO1-CE9264/#gateway	*C++ program to output a flag*
http://zip4.usps.com/zip4/	*nine-digit zip codes*

The following file is **~/public_html/cal.html**. Its URL is therefore **http://i5.nyu.edu/~abc1234/cal.html**. It should have the same nine permission bits (**rw-r--r--**) as your web home page **~/public_html/index.html**.

The **INPUT** widgets must be between the **FORM** tags in lines 8 and 50. To show the user where the form begins and ends, surround it with horizontal rules in lines 7 and 51. When you press the **SUBMIT** button in line 46, the gateway in line 8 will be run.

```
 1 <HTML>
 2 <HEAD>
 3 <TITLE>Calendar form</TITLE>
 4 </HEAD>
 5 <BODY>
 6
 7 <HR>
 8 <FORM METHOD = POST ACTION = "http://i5.nyu.edu/cgi-bin/cgiwrap/mm64/cal">
 9 <H2>Calendar form</H2>
10
11 <P>
12 This form runs the gateway
13 <CODE>/home1/m/mm64/public_html/cgi-bin/cal</CODE>
14 on the host
15 <CODE>i5.nyu.edu</CODE>.
16 </P>
17
18 <P>
19 Which year do you want to see?
20 <INPUT TYPE = TEXT NAME = "year" SIZE = 5>
21 </P>
22
23 <P>
24 Which month do you want to see?
25 <BR><INPUT TYPE = RADIO NAME = "month" VALUE = 1>January
26 <BR><INPUT TYPE = RADIO NAME = "month" VALUE = 2>February
27 <BR><INPUT TYPE = RADIO NAME = "month" VALUE = 3>March
28 <BR><INPUT TYPE = RADIO NAME = "month" VALUE = 4>April
29 <BR><INPUT TYPE = RADIO NAME = "month" VALUE = 5>May
30 <BR><INPUT TYPE = RADIO NAME = "month" VALUE = 6>June
31 <BR><INPUT TYPE = RADIO NAME = "month" VALUE = 7>July
32 <BR><INPUT TYPE = RADIO NAME = "month" VALUE = 8>August
33 <BR><INPUT TYPE = RADIO NAME = "month" VALUE = 9>September
34 <BR><INPUT TYPE = RADIO NAME = "month" VALUE = 10>October
35 <BR><INPUT TYPE = RADIO NAME = "month" VALUE = 11 CHECKED>November
36 <BR><INPUT TYPE = RADIO NAME = "month" VALUE = 12>December
37 <BR><INPUT TYPE = RADIO NAME = "month" VALUE = "">All twelve months
38 </P>
39
40 <P>
41 <INPUT TYPE = CHECKBOX NAME = "date">Check here
42 to see the current date and time too.
43 </P>
44
45 <P>
46 <INPUT TYPE = SUBMIT VALUE = "See the calendar.">
47 <BR>
48 <INPUT TYPE = RESET VALUE = "Start again.">
```

```
49 </P>
50 </FORM>
51 <HR>
52
53 </BODY>
54 </HTML>
```

The above line 8 can have a relative URL as in Handout 3, p. 30, ¶ (5):

```
55 <FORM METHOD = POST ACTION = "/cgi-bin/cgiwrap/mm64/cal">
```

To make a pop-up menu instead of radio buttons, change the above lines 25−37 to

```
56 <SELECT NAME = "month">
57 <OPTION VALUE = 1>January</OPTION>
58 <OPTION VALUE = 2>February</OPTION>
59 <OPTION VALUE = 3>March</OPTION>
60 <OPTION VALUE = 4>April</OPTION>
61 <OPTION VALUE = 5>May</OPTION>
62 <OPTION VALUE = 6>June</OPTION>
63 <OPTION VALUE = 7>July</OPTION>
64 <OPTION VALUE = 8>August</OPTION>
65 <OPTION VALUE = 9>September</OPTION>
66 <OPTION VALUE = 10>October</OPTION>
67 <OPTION VALUE = 11 SELECTED>November</OPTION>
68 <OPTION VALUE = 12>December</OPTION>
69 <OPTION VALUE = "">All 12 months</OPTION>
70 </SELECT>
```

We'll start with a simple gateway that merely displays the data it received from the above form. As in Handout 5, p. 27, the permissions of the gateway should be **`rwxr-xr-x`**.

The environment variable **`$CONTENT_LENGTH`** tells the gateway the number of bytes it should read the standard input. These bytes constitute one long line, without an end-of-file or even a newline character (**`\n`**) at the end:

```
year=2013&month=5&date=on
```

The command **`head -25`** inputs 25 lines from the standard input and outputs them to the standard output. The command **`/usr/bin/ghead -c25`** (g for GNU) inputs 25 characters from the standard input and outputs them to the standard output; see Handout 4, p. 30, line 13. The **`/usr/bin/ghead`** command in the following gateway therefore inputs **`$CONTENT_LENGTH`** bytes from the gateway's standard input and outputs them to the standard output, which is displayed in the browser.

```
1$ man head                    documentation about /bin/head
2$ man ghead                   documentation about /urs/bin/head
```

```
#!/bin/ksh
#This gateway is ~/public_html/cgi-bin/cal.

echo Content-type: text/html
echo
echo '<HTML>'
echo '<HEAD>'
echo '<TITLE>Calendar</TITLE>'
echo '</HEAD>'
echo '<BODY>'
echo '<H1>Calendar</H1>'
echo The first $CONTENT_LENGTH bytes of standard input are
echo '<PRE>'
/usr/bin/ghead -c$CONTENT_LENGTH
echo '</PRE>'
echo '</BODY>'
echo '</HTML>'
exit 0
```

Here is the output of the above gateway if the user checks the **CHECKBOX**. If the user doesn't check it, the **&date=on** would not be there.

```
Content-type: text/html

<HTML>
<HEAD>
<TITLE>Calendar</TITLE>
</HEAD>
<BODY>
<H1>Calendar</H1>
The first 25 bytes of standard input are
<PRE>
year=2013&month=5&date=on</PRE>
</BODY>
</HTML>
```

It appears in your browser like this:

Calendar

The first 25 bytes of standard input are
year=2013&month=5&date=on

Pipe the standard output of **head** into **tr**, which chops the long line of bytes into several shorter lines. See the **tr** in Handout 3, p. 15.

```
#!/bin/ksh
#This gateway is ~/public_html/cgi-bin/cal.

echo Content-type: text/html
echo
echo '<HTML>'
echo '<HEAD>'
echo '<TITLE>Calendar</TITLE>'
echo '</HEAD>'
echo '<BODY>'
echo '<H1>Calendar</H1>'
echo The first $CONTENT_LENGTH bytes of standard input are
echo '<PRE>'
#Change ampersands to newlines.
/usr/bin/ghead -c$CONTENT_LENGTH | tr '&' '\012'
echo                                  #Output a final newline.
echo '</PRE>'
echo '</BODY>'
echo '</HTML>'
exit 0
```

Here is the output of the above gateway:

```
Content-type: text/html

<HTML>
<HEAD>
<TITLE>Calendar</TITLE>
</HEAD>
<BODY>
<H1>Calendar</H1>
The first 25 bytes of standard input are
<PRE>
year=2013
month=5
date=on
</PRE>
</BODY>
</HTML>
```

It appears in your browser like this:

Calendar

```
The first 25 bytes of standard input are
year=2013
month=5
date=on
```

Now that we've verified that the gateway is receiving data from the form, let's make the gateway chop the data up and store it in shell variables. Could you use the parentheses from the double extra credit part of Homework 3.9 (Handout 3, p. 26) to create the file **`/tmp/$$`** more simply?


```
#!/bin/ksh
#This gateway is ~/public_html/cgi-bin/cal.
#It displays a calendar and is executed from the form in the page
#~/public_html/cal.html
#The URL of that page is
#http://i5.nyu.edu/~mm64/cal.html

echo Content-type: text/html
echo
echo '<HTML>'
echo '<HEAD>'
echo '<TITLE>Calendar</TITLE>'
echo '</HEAD>'
echo '<BODY>'
echo '<H1>Calendar</H1>'
echo '<PRE>'

/usr/bin/ghead -c$CONTENT_LENGTH | tr '&' '\012' > /tmp/$$
echo >> /tmp/$$   #append a newline to the end of the /tmp/$$ file

 year=`awk -F= '$1 == "year"  {print $2}' /tmp/$$`
month=`awk -F= '$1 == "month" {print $2}' /tmp/$$`
 date=`awk -F= '$1 == "date"  {print $2}' /tmp/$$`
rm /tmp/$$

#Quotes make sure that cal will receive a second argument even if
#$year is the null string.  In the absence of a second argument,
#cal would think that the first (and only) argument was the year.

cal $month "$year"
echo '</PRE>'

#Need double quotes because $date would be the null string if checkbox
#not checked.  See double quotes in Handout 5, pp. 14, 19.
if [[ "$date" == on ]]
then
    echo '<P>'
    date
    echo '</P>'
fi

echo '</BODY>'
echo '</HTML>'
exit 0
```

The **/tmp/$$** file contains three lines:

```
year=2013
month=5
date=on
```

Here is the output of the above gateway:


```
Content-type: text/html

<HTML>
<HEAD>
<TITLE>Calendar</TITLE>
</HEAD>
<BODY>
<H1>Calendar</H1>
<PRE>
   May 2013
 S  M Tu  W Th  F  S
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

</PRE>
<P>
Tue May 28 15:22:59 EDT 2013
</P>
</BODY>
</HTML>
```

It is rendered in your browser like this:

Calendar

```
   May 2013
 S  M Tu  W Th  F  S
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31


Tue May 28 15:22:59 EDT 2013
```

▼ Homework 7.8: write a form that gives input to a gateway

Write a form that gives input to a gateway that doesn't always produce the same output each time you run it. It can't be the gateway and form shown above.

For example, let the user type in a loginname and then tell them if that person is logged in right now. Or make a guestbook:

```
<TEXTAREA NAME = "message" ROWS = 24 COLS = 80>
Write your message here.
</TEXTAREA>
```

Or let the user type in a date, month, and year, and then show the phase of the moon for the midnight at the start of that date. See Handout 2, p. 14, line 88.

```
1$ ~mm64/bin/moon 28 5 2013                    day, month, year
```

Hand in a printout of the **`.html`** file that contains the form, your gateway shellscript, and (if possible) snapshots of the form and the output of the gateway.

▲
□

