Fall 2006 Handout 10

join

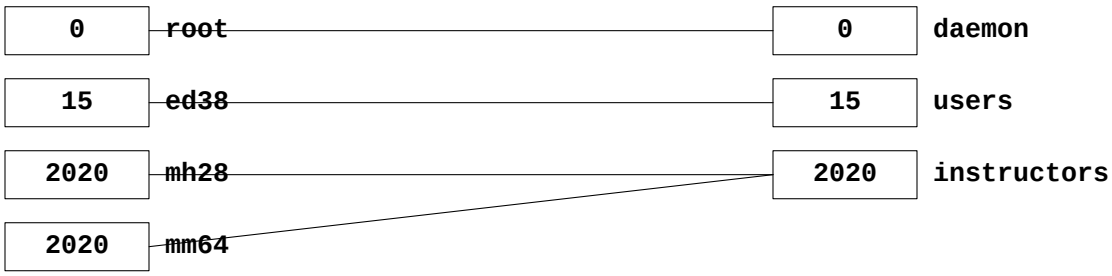
The join field in each file must be sorted alphabetically, even if it's numeric.

```
1$ awk -F: '{print $4, $1}' /etc/passwd | sort > ~/people1
0 root
15 mm64

2$ awk -F: '{print $3, $1}' /etc/group | sort > ~/group1
12 daemon
15 users
```

To output each person alongside the name of the group to which they belong,

```
3$ join ~/people1 ~/group1
15 mm64 users
```

The above example outputs four lines because there are four pairs of identical keys:



Generate a Cartesian product: Hamlet II, ii, 387−390
(lines 138703−138709 in `$S45/Shakespeare.complete`)

To form the Cartesian product of two files,

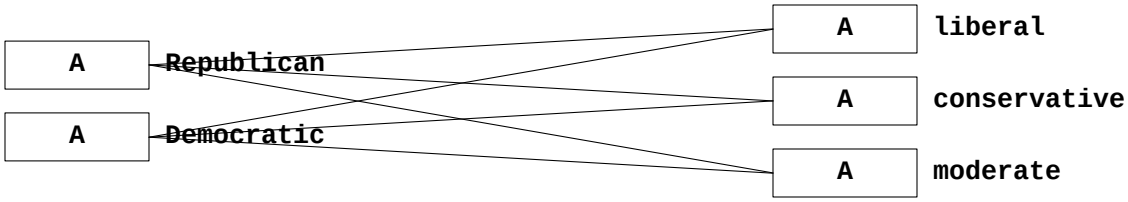
```
Republican
Democratic
```

This is **`party`**.

```
liberal
conservative
moderate
```

This is **`leaning`**.

```
1$ sed 's/^A /' party > ~/party_temp          or perl -pe 's/^A /'
2$ sed 's/^A /' leaning > ~/leaning_temp
3$ join ~/party_temp ~/leaning_temp
```




```
A Republican liberal
A Republican conservative
A Republican moderate
A Democratic liberal
A Democratic conservative
A Democratic moderate
```

Then remove the leading **A** and blank:

```
sed 's/^A //'
perl -pe 's/^A //'
```

The -j1 and -j2 options of join

By default, the join field is the first field on each line of input. Use the **-j1** option to use a different field as the join field in the first input file; **-j2** in the second input file. For example, to use the second field in each input file as the join field,

```
1$ awk -F: '{print $1, $4}' /etc/passwd | sort +1 > ~/people2
root 0
mm64 15

2$ awk -F: '{print $1, $3}' /etc/group | sort +1 > ~/group2
daemon 12
users 15

3$ join -j1 2 -j2 2 ~/people2 ~/group2
15 mm64 users
```

Never give the options **-j1 1** or **-j2 1**: they're the default.

The -o option of join

The default output is the join field, then the rest of the line from the first input file, then the rest of the line from the second input file. Use the lowercase **-o** option to compose a different output line consisting of fields from either input file. For example, to make the above **join** command output only the person's loginname (the first word of the first input file, **1.1**) followed by the group name (the first word of the second input file, **2.1**), change it to

```
1$ join -j1 2 -j2 2 -o 1.1 2.1 ~/people2 ~/group2
mm64 users

2$ join -j1 2 -j2 2 -o 2.1 2.2 1.1 ~/people2 ~/group2
users 15 mm64
```

The **-t** option of **join** is just like the **-F** option of **awk** and the **-t** option of **sort**. See **man -t join**.

▼ Homework 10.1: a join application

Columns two and three of the output of **ps -lax** are the **PID** and parent's **PID** of every process.

```
1$ ps -lax
   F UID   PID  PPID CP PRI NI  SZ  RSS WCHAN    STAT TT  TIME COMMAND
8201 420 11395 11394  0  15  0  64  352 pause    I    p4  0:04 -csh (csh)
8021 420 15584 11395  4   1  0 168  480 socket   S    p4  0:00 vi handout10.ms
   1 420 15604 15603 29  32  0 152  408          R    p4  0:00 ps -lx
```

The first and last columns of the output of **ps -acux** are the owner's loginname and name of every process. Why should **awk print $NF** instead of **$11**?

```
2$ ps -acux
USER       PID %CPU %MEM   SZ  RSS TT STAT START  TIME COMMAND
meretzky 15611 15.4  0.7  152  384 p4 R    14:20   0:00 ps
meretzky 15584  0.0  0.9  168  496 p4 S    14:17   0:00 vi
meretzky 11395  0.0  0.6   64  344 p4 I    11:13   0:04 csh
```

Write a shellscript named **myps** that will output one line for each process, with the four columns described above. Output no header line.

```
3$ myps
meretzky    11395    11394    csh
meretzky    15584    11395    vi
```

myps will call **ps** twice, with the two different arguments. Feed selected columns of the output of the two **ps**'s into **join**, which will use the **PID** numbers as the join field. You get no credit unless you sort the join field alphabetically. You get no credit if you give the **-j1 1** or **-j2 1** options to **join**. **join** will output only the processes whose **PID** numbers appear in the output of both **ps**'s; it will remove the others automatically.

You get no credit for any shellscript unless you put the temporary files you create into your home directory:

```
prog1 | prog2 | prog3 > $HOME/program
prog4 | prog5 | prog6 > $HOME/errors
```

You get no credit if you input a file into a program like this:

```
cat file | prog
```

Do it this way instead:

```
prog file
```

sort the output of **join** in order of ascending **PID** numbers. Line the columns up nicely by filtering them through the **printf** statement of **awk** or **perl**.

Hand in only the first 50 lines of output. You get no credit if the same PID number appears on two different lines.

▼ Homework 10.2: log off the idlers

Use **join** to eliminate the **for** loop in the **awk** example that killed programs running on idle terminals.

▲

▼ Homework 10.3: a join application

The **df** command (for ''disk free'') outputs a table showing how full each subtree is. Each subtree is listed by the name of its top (i.e., root) directory.

```
1$ df | more
/                  (/                  ): 5239602 blocks   477535 files
```

Write a shellscript named **full** that will output the login name of everyone whose home directory is in a subtree that's more than 80% full. This could be used to **mail** them a letter telling them to remove unnecessary files.

Create two files and feed them into **join**. The first file will list the directories that are more than 80% full, sorted in alphabetical order of the directory name. Don't list the root directory **/**.

The second file will list the loginname of each person in **/etc/passwd**, preceded by the first component of the full pathname of their home directory. Sort it in alphabetical order of the directory name:

```
/home1 aa947
/home1 aas269
/home1 aav232
/home1 ab215
```

Then **join** the two files.

▲

Outline for the advanced perl course

1. Introduction
   - A. the Unix philosophy of simple tools connected by pipes; i/o redirection
   - B. **perl** as a successor of **grep**, **sed**, **awk**, **sh**, and C
   - C. advantages of **perl** over a shellscript of connected tools:
     1. better notation (e.g., **perl** contains the union of **grep** and **egrep**)
     2. not limited to one-way sequential communication (i.e., pipes) between parts of the program. In **perl**, data can be loaded into a data structure (i.e., a list) and processed non-sequentially (e.g., in contrast to **sed**)
   - D. tasks for which you would otherwise have to resort to C:
     1. process binary data in addition to ASCII data
     2. format individual bits and bytes in memory with **pack** and **unpack**
     3. call the Unix system calls (e.g., **stat**, **fork**, **socket**, etc.)

2. Running perl: **print "hello\n";**
   - A. interactively, in response to the C shell prompt (**-e** option)
   - B. in a Bourne shellscript
   - C. as a perlscript starting with **#!**

3. Programming with scalar variables
   - A. scalar variables, assignment, initialization, undefined values
   - B. string operators: **. x**
   - C. output with **print**, input with **<>**, **$/**, and **read**.
   - D. **'**single quotes**'**, **"**double quotes**"**, and **`**back quotes**`**
   - E. control structure: loops , **if**, **{**curly braces**}**

4. I/O with files and pipes

A. filehandles, **open** with **|**, **<**, **>**, **>>**

B. error messages with **die "$0: $!";**, **||** as a shorthand for **if not**

C. **while (<>)** and the special variable **$_**

D. loop through a list of filenames with **<*.c>**; file test operators; **unlink**, **rename**, **chmod**, **chown**; **system**.

5. Lists

A. store a list in an array: **@a = (1, 2, 3);**, subscripts, **$#a**, **@ARGV**

B. scalar context vs. array context

C. loop through a list with **foreach**

D. loop through a list with **grep**

E. **sort** a list; **sort** subroutines

F. associative arrays, **%ENV**

6. Implicit input-driven looping

A. emulate **grep** and **awk** with the **-n** option; **$.**

B. emulate **awk** with the **-a** option (autosplit), **@F**

C. emulate **sed** with the **-p** option

7. Regular expressions (a large topic)

A. a complete exposition of regular expressions, assuming no previous knowledge

B. tagged regular expressions (parentheses and **\1**, **\2**, **\3**), important in data transformation

8. Data transformation (a large topic)

A. **s/pattern/replacement/** with regular expressions; **$`**, **$&**, **$'**, **$1**, **$2**, **$3**

B. **s/pattern/replacement/** with interpolated variables: a combination of **sed** and **awk**

9. Calling the Unix system calls

A. **pack** and **unpack** binary data

B. **require** (just like **#include** in C)

C. To illustrate that the system calls are equally accessible in **perl** and **C**, students will be given the following examples in both languages.

1. directory and file access with **opendir**, **readdir**, **closedir**, and **stat**
2. create a child process with **fork**, **exec**, and **wait**
3. shared memory with **shmget**, **shmread**, **shmwrite**, **shmctl**
4. **SOCK_STREAM** sockets with **socket**, **connect**, **bind**, **listen**, **accept**

10. Miscellaneous

A. formats

B. subroutines, **local**, packages

C. convert **sed**, **awk**, **find**, and C header files to **perl**: **s2p**, **a2p**, **find2perl**, **h2ph**

D. the **perl** debugger

Introduction

Information about perl

(1) *Programming perl* by Larry Wall and Randal L. Schwartz. O'Reilley & Associates; ISBN 0-937175-64-1.

(2) *Leraning perl* by Randal L. Schwartz. O'Reilley & Associates; ISBN 0-56592-042-2.

(3) the Unix manual page: display on screen with **`man perl`**, print out with **`man -t perl`**.

(4) the **`comp.lang.perl`** newsgroup; use **`nn`**

(5) Don't bother the author:

```
1$ finger lwall@jpl-devvax.jpl.nasa.gov                    lowercase LWALL
[jpl-devvax.jpl.nasa.gov]

Login name: lwall                In real life: Larry Wall
Directory: /u/sfoc/lwall             Shell: /usr/etc/exp
Last login Sun Apr 11, 1993 on ttyp1 from vaccine.netlabs.
Mail last read Tue May 24 21:58:59 1994
No Plan.
```

A Unix pipeline

Output the login name of everyone who is logged into two or more terminals:

```
1$ who
operator console Apr 30 14:14
bidof    ttyp0   May 25 08:10       (128.122.128.214)
nabutvsk ttyp1   May 25 10:28       (128.122.128.214)
laik     ttyp2   May 25 12:01       (128.122.150.94)
bidof    ttyp3   May 25 10:12       (128.122.128.61:0)

2$ who | awk '{print $1}'
operator
bidof
nabutvsk
laik
bidof

3$ who | awk '{print $1}' | sort
bidof
bidof
laik
nabutvsk
operator

4$ who | awk '{print $1}' | sort | uniq -d
bidof
```

An interpreter script (shellscript, perlscript, awkscript, etc.)

(1) Put the script in the **`bin`** subdirectory of your home directory.

(2) Scripts may be written in any one of several languages; the **`#!`** on the first line shows which one.

(3) Type **`chmod`** to turn on its **`x`** bits.

(4) Type **rehash** after creating any new program (including a shellscript or perlscript).

```
#!/bin/sh
#Output the login name of everyone who is logged into two
#or more terminals.

who | awk '{print $1}' | sort | uniq -d
```

```
1$ ls -l twoormore
-rw-r--r--  1 meretzky      165 May 25 12:09 twoormore

2$ chmod 755 twoormore
3$ ls -l twoormore
-rwxr-xr-x  1 meretzky      165 May 25 12:09 twoormore

4$ rehash

5$ twoormore                          Run the program.
bidof
```

A perlscript to produce the same output

```
 1 #!/bin/perl
 2 #Output the login name of everyone who is logged into two
 3 #or more terminals.
 4
 5 foreach (‘who‘) {
 6     ($loginname) = split;
 7     $count{$loginname} = $count{$loginname} + 1;
 8     if ($count{$loginname} == 2) {
 9         print "$loginname\n";
10     }
11 }
```

The following task is just as easy as a perlscript, but would be harder as a shellscript. In fact, you would probably have to resort to **awk**.

```
 1 #!/bin/perl
 2 #Everyone is allowed to use one terminal.  If someone is using
 3 #more than one terminal, output the names of the second, third,
 4 #fourth, etc. terminals that they’re using.
 5
 6 foreach (‘who‘) {
 7     ($loginname, $terminal) = split;
 8     $count{$loginname} = $count{$loginname} + 1;
 9     if ($count{$loginname} >= 2) {
10         print "$terminal\n";
11     }
12 }
```

```
1$ perlscript
ttyp3
```

A perlscript that creates a list

```
1$ ps -acux
USER       PID %CPU %MEM   SZ  RSS TT STAT   TIME COMMAND
meretzky 26721 41.7  0.6 1164  960 q6 R      0:00 ps
root     16226 23.5  0.4  968  660 ?  R    1194:05 sendmail
rmchndrn 26671 17.0  0.9 2356 1524 r0 R      0:19 lwraster
root     26715  1.2  0.2  416  296 s6 S <    0:00 login
```

```
 1 #!/bin/perl
 2 #If someone is using more than one terminal, kill all their processes
 3 #except for those running on their first terminal.
 4
 5 #Make a list of the doomed terminals.
 6 foreach (‘who‘) {
 7     ($loginname, $terminal) = split;
 8     $count{$loginname} = $count{$loginname} + 1;
 9     if ($count{$loginname} >= 2) {
10         push (@doomed, $terminal);     #Put the $terminal on the @doomed list.
11     }
12 }
13
14 #Kill each process running on a dooomed terminal.
15 foreach (‘ps -acux‘) {
16     ($loginname, $pid, $cpu, $mem, $sz, $rss, $tt) = split;
17     foreach $terminal (@doomed) {
18         if ($tt eq substr($terminal, -2)) {
19             print "killing $pid on terminal $terminal\n";
20             kill 9, $pid;
21         }
22     }
23 }
```

Print the numbers from 1 to 10

The shell language cannot perform arithmetic, compare two numbers, or even print a number. To perform these tasks, a program written in the shell language (i.e., a shellscript) must run other programs, which are underlined in the following example. The shell language is a language foir running other programs; it can do nothing else.

```
#!/bin/sh

i=1
while [ $i -le 10 ]
do
    echo $i
    i=‘expr $i + 1‘
done
```

```
1 #!/bin/perl
2
3 $i = 1;
4 while ($i <= 10) {
5     print "$i\n";
```

```
6     $i = $i + 1;
7 }
```


Three ways to run perl: pp. 4−5

(1) In response to the C shell prompt

Put ’single quotes’ around any command line argument that contains unusual characters such as \.

```
1$ perl -e ’print "hello\n";’                    Final semicolon is optional.
hello
```

(2) As one command in a shellscript

```
1 #!/bin/sh
2 #This file is named shelly.
3
4 date
5 perl -e ’print "hello\n";’
6 whoami
```

```
1$ shelly
Wed Nov 22 13:31:00 EST 2006
hello
mm64
```

(3) As a perlscript

You will often run a shellscript that contains one **perl** command and nothing else:

```
1 #!/bin/sh
2
3 perl -e ’print "hello\n";’
```

In this case, it is faster and simpler to write a perlscript instead of a shellscript:

```
1 #!/bin/perl
2 #This file is named perlscript.
3
4 print "hello\n";
```

```
1$ perlscript
```

A perlscript can be more than one statement:

```
 1 #!/bin/perl
 2
 3 print "* * * * * * ====================\n";
 4 print " * * * * * *  ====================\n";
 5 print "* * * * * * ====================\n";
 6 print " * * * * * *  ====================\n";
 7 print "* * * * * * ====================\n";
 8 print "==================================\n";
 9 print "==================================\n";
10 print "=========E pluribus unum=========\n";
11 print "==================================\n";
12 print "==================================\n";
```

Scalar variables

You don't have to declare variables: they pop into existence as soon as you mention them. See pp. 79−93 for the operators in an expression.

```
1 #!/bin/perl
2
3 $i = 1;
4 $j = 2;
5 $k = $i + $j;
6
7 print $k, "\n";
8 print $k, "\n";
9 print $i + $j, "\n";
```

```
2$ perlscript
3
3
```

▼ **Homework 10.4: omit the `, "\n"`**

What happens if you omit the two occurrences of `, "\n"` from the above perlscript?

▲

Division and remainder

```
1 #!/bin/perl
2
3 print 38 / 5, "\n";
4 print int(38/5), "\n\n";
5
6 print 38 % 5, "\n";
7 print 39 % 5, "\n";
8 print 40 % 5, "\n";
```

```
1$ perlscript
7.5999999999999996
7

3                       because 38 is 3 more than a multiple of 5
4                       because 39 is 4 more than a multiple of 5
0                       because 40 is a multiple of 5
```

String variables

Any variable can hold a number or a string:

```
#!/bin/perl

$segment = "+=";
$short = $segment . $segment;
$long = $segment x 30;

print $short, "\n";
print $long, "\n";
```

```
1$ perlscript
+=+=
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=
```

Variables are initialized to 0 or the null string

If you use the value of a variable without first assigning a value to it, the variable is automatically given a default initial value. The context in which the variable is used determines whether this default initial value will be 0 or the null string.

```
#!/bin/perl

$n = 10;
$s = "hello";

print $n + $u, "\n";
print $s . $v, "\n";
```

```
1$ perlscript
10
hello
```

In the above example, the variables **$u** and **$v** are undefined before the **print**'s and defined after the **print**'s. To test if a variable is defined,

```
if (defined $u) {
if (! defined $u) {
```

p. 82 for **!**

Input and output

You can remove the **STDOUT** and **STDIN** (but not the **< >**): they're the default. You can change **STDOUT** to **STDERR**.

The assignment statement stores the entire line of input, including the terminating **\n**, into the variable **$year**. This line may consist of any string, not just a number.

```
1 #!/bin/perl
2
3 print STDOUT "What year is this? ";
4 $year = <STDIN>;
5 print STDOUT "In ", 1997 - $year, " years,\n";
6 print STDOUT "Hong Kong goes back to Chinese rule.\n";
```

A quoted string may occupy more than one line (p. 69):

```
1 #!/bin/perl
2
3 print "What year is this? ";
4 $year = <>;
5 print "In ", 1997 - $year, " years,
6 Hong Kong goes back to Chinese rule.\n";
```

▼ **Homework 10.5: input and output a variable**

Write a perlscript to conduct a dialog such as

```
How old are you? 38
That's 266 dog years!
```

▲

Quotation marks

Singles and doubles

Like C but unlike the shell, **perl** requires quotes around every variable. **perl** has the same three kinds of quotes that the shell has: **'**single**'**, **"**double**"**, and **‘**back quotes**‘**.

You can use variables and **\n** within single quotes but not within double quotes:

```
 1 #!/bin/perl
 2
 3 $i = 100;
 4
 5 #Three ways to do the same thing:
 6 print $i, "\n";
 7 print $i . "\n";
 8 print "$i\n";
 9
10 print "You have just won $i dollars.\n";
11 print ’You have just won $i dollars.\n’;
```

```
1$ perlscript
100
100
You have just won $i dollars.\n
```

These rules also apply in the shell language:

```
2$ echo "$HOME"
/home1/m/mm64

3$ echo ’$HOME’
$HOME
```

Back quotes

```
 1 #!/bin/perl
 2
 3 $a = ’date’;
 4 print "$a\n";
 5
 6 $a = "date";
 7 print "$a\n";
 8
 9 $a = ‘date‘;
10 print "$a";             #$a contains a \n
11
12 print ‘date‘;
13 print ‘who | wc -l‘;
```

```
1$ perlscript
date
date
Wed Nov 22 13:31:00 EST 2006
Wed Nov 22 13:31:00 EST 2006
9
```

▼ **Homework 10.6: print the total size in bytes of the files in the current directory**

Assume that the current directory contains only files, so you don't have to worry about subdirectories. Loop through all the lines output by **`ls -l`** with a **`foreach`** loop. **`split`** the first four fields of each line into variables named **`$perms`**, **`$links`**, **`$owner`**, and **`$size`**. Sum up the **`$size`** of each file in a variable named **`$sum`**. After the loop is over, **`print`** the **`sum`**.

You can initialize **`$sum = 0;`** if it makes you feel more secure, but it isn't necessary.

✎ Extra credit. Also output the average size in bytes of all the files. Create a variable named **`$count`** and increment it every time we loop through another line. Assume that the current directory contains at least one file, so you don't have to worry about division by zero.

```
1$ ls -l
-r--r--r--  1 meretzky     5045 May  4 12:31 p.associative.ms
-rw-r--r--  1 meretzky      823 May  5 13:13 p.input.ms
drw-r--r--  1 meretzky      512 May  5 12:28 mysubdirectory
```

▲

Input chunks of various size

To input an entire line,

```
$a = <STDIN>;
```

To input everything up to and including the next colon,

```
$/ = ':';               #The default value of $/ is "\n", p. 113.
$a = <STDIN>;
```

To input everything up to the end of the input file,

```
undef $/;               #Remove the value of $/.
$a = <STDIN>;
```

To input a single byte,

```
$n = read(STDIN, $a, 1);
```

To input exactly four bytes,

```
$n = read(STDIN, $a, 4);
```


"while" and "for" loops

See p. 88 for the six relational operators **==**, **<=**, etc.

Always enclose the body of the loop in **{**curly braces**}**. Unlike C, **perl** requires the curly braces even if the body contains only one statement.

```
1 #!/bin/perl
2 #Print the numbers from 1 to 10, one per line.
3
4 $i = 1;
5
6 while ($i <= 10) {
7     print "$i\n";
8     $i = $i + 1;
9 }
```

As in C, a **for** loop is another notation for a **while** loop. It lets you write the three vital statistics all on one line:

```
1 #!/bin/perl
2 #Print the numbers from 1 to 10, one per line.
3
4 for ($i = 1; $i <= 10; $i = $i + 1) {
5     print "$i\n";
6 }
```

▼ **Homework 10.7: write a program with nested `for` loops**

Write a program with two **for** loops and two **print**'s whose output is

```
1$ perlscript
Lucy in the sky with diamonds
Lucy in the sky with diamonds
Lucy in the sky with diamonds
Aaaaaaaaaaaaaaaaaaaaaaaaahhh

Lucy in the sky with diamonds
Lucy in the sky with diamonds
Lucy in the sky with diamonds
Aaaaaaaaaaaaaaaaaaaaaaaaahhh
```

▲

Geometric progression, right justification

```
1 #!/bin/perl
2 #Output the powers of 2 from 1 to 64 inclusive, one per line.
3
4 for ($i = 1; $i <= 64; $i = $i * 2) {
5     printf STDOUT "%2d\n", $i;
6 }
```

```
1$ perlscript
 1
 2
 4
 8
16
32
64
```

printf conversion characters

```
1 #!/bin/perl
2
3 print "decimal   octal     hex\n";
4
5 for ($i = 0; $i < 32; $i = $i + 1) {
6     printf "%7d %7o %7X\n", $i, $i, $i;
7 }
```

```
1$ perlscript
decimal   octal     hex
      0       0       0
      1       1       1
      2       2       2
      3       3       3
      4       4       4
      5       5       5
      6       6       6
      7       7       7
      8      10       8       etc.
```

```
1 #!/bin/perl
2
3 print "hex ascii\n";
4
5 for ($i = ord(' '); $i < ord('~'); $i = $i + 1) {
6     printf "%3d %c\n", $i, $i, $i;
7 }
```

```
2$ perlscript
hex ascii
 32
 33 !
 34 "
 35 #
 36 $
 37 %
 38 &
 39 '
 40 (        etc.
```

▼ **Homework 10.8: A Hundred Bottles of Beer on the Wall**

Write a perlscript to output the words of this song. Between verses, output an empty line and

```
sleep 1;
```

▲

▼ **Homework 10.9: the New York State Thruway**

Write a perlscript to output eight signs at intervals of 10 miles, heading north. Right-justify the numbers. Output an empty line after each sign.

```
+--------------+
| Albany   108 |
| Montreal 328 |
| Buffalo  388 |
+--------------+

+--------------+
| Albany    98 |
| Montreal 318 |
| Buffalo  378 |
+--------------+

+--------------+
| Albany    88 |
| Montreal 308 |
| Buffalo  368 |
+--------------+
```

etc.

▲

An infinite loop: p. 96

```
1 #!/bin/perl
2
3 for (;;) {
4     print "It was a dark and stormy night.\n";
5     print "Some Indians were sitting around a campfile.\n";
6     print "Then their chief rose and said:\n\n";
7 }
```

Pipe the output of the above perlscript into **more**:

```
1$ perlscript | more
```

A loop that always iterates at least once: p. 94

```
 1 #!/bin/perl
 2
 3 srand;
 4
 5 print "Welcome to Russian Roulette.\n";
 6 print "Any number of players can take turns.\n";
 7 print "To pull the trigger when you see the ->, press RETURN.\n\n";
 8
 9 do {
10     print '-> ';
```



```
11      $dummy = <>;
12  } while rand(6) >= 1;         #one in six chance of death
13
14  print "BANG!\n";
```

```
 1  #!/bin/perl
 2
 3  srand;
 4
 5  print "Welcome to Russian Roulette.
 6  Any number of players can take turns.
 7  To pull the trigger when you see the ->, press RETURN.\n\n";
 8
 9  do {
10      print '-> ';
11      $dummy = <>;
12  } until rand(6) < 1;      #one in six chance of death
13
14  print "BANG!\n";
```

"if" statements

Like C, **perl** requires **(**parentheses**)** around the logical expression. Unlike C, **perl** requires **{**curly braces**}** around the body of an **if** even if the body consists of only a single statement.

See p. 88 for the relational operators **==**, **eq**, etc.; p. 89 for the logical operators **&&**, **||**, etc.; p. 103 for regular expressions within **/**slashes**/**; p. 82 for the pattern binding operator **=~**; and p. 84 for the file test operators **-r**, **-w**, etc.

```
 1 #!/bin/perl
 2
 3 $a = 1;
 4 $b = 2;
 5
 6 if ($a == $b) {
 7     print "They're equal.\n";
 8 }
 9
10 if ($a >= 20) {
11     print "It's greater than 20.\n";
12 }
13
14 if (`who | wc -l` >= 20) {
15     print "At least 20 people are logged in right now.\n";
16 }
17
18 $a = 'yes';
19 $b = 'y';
20
21 if ($a eq $b) {
22     print "They're equal.\n";
23 }
24
25 if (`whoami` ne 'abc1234') {
26     print "Sorry, only abc1234 has permission to run this perlscript.\n";
27     exit 1;                   #Non-zero exit status indicates failure in Unix.
28 }
```

```
 1 #!/bin/perl
 2
 3 $a = <STDIN>;           #You can omit the STDIN.
 4
 5 if ($a =~ /^y/) {       #The ^y between the slashes is a regular expression.
 6     print "The value of the variable starts with a y,\n";
 7     print "and I think anything that starts with a y means yes.\n";
 8 }
 9
10 if (`date` =~ /^Fri/) {
11     print "Today is Friday.\n";
12 }
13
14 if (`who | wc -l` >= 20 && `date` =~ /^Fri/) {
15     print "Aw, give up: At least 20 people are logged in, and it's Friday.\n";
16     exit 1;
17 }
```



```
18
19 if (-r bigfile) {
20     print "I have permission to read the file bigfile.\n");
21 }
```

▼ **Homework 10.10: output the names of the filesystems that are full**

Loop through all the lines output by **`df`** with a **`foreach`** loop. **`split`** the first five fields of each line into variables named **`$filesystem`**, **`$total`**, **`$used`**, **`$free`**, **`$percent`**. If **`$percent`** is greater than or equal to 90, **`print`** the **`$filesystem`** and the **`$percent`**. (You can ignore the **`%`** at the end of the value of **`$percent`**.)

```
1$ df
Filesystem              Total    kbytes   kbytes   %
node                    kbytes   used     free     used  Mounted on
/dev/rz0a                 15823   13452      789    94%   /
/dev/rz0h               1233086 1084272   25506    98%   /usr
/dev/rz1h               1233086  706012  403766    64%   /home1
/dev/rz2h               1233086  987754  122024    89%   /home2
/dev/ra35c              1914797 1545621  177697    90%   /home3

2$ perlscript
/dev/rz0a 94%
/dev/rz0h 98%
/dev/ra35c 90%
```

✎ Extra credit. Use the statement

```
chop $percent;
```

to remove the last character from the variable **`$percent`**. Use **`printf "%-20s %5d\n",`** to left-justify the **`$filesystem`** and right-justify the **`$percent`**.

▲

▼ **Homework 10.11: total size in bytes of the files in the current directory**

An earlier exercise looped through every line output by **`ls -l`** to sum up the size of everything in the current directory. Change it so that it sums up only the files, not the subdirectories. Add **`$size`** to **`$sum`** only if **`$perms`** begins with a dash.

Also **`print`** the average size of all the files. Create a variable named **`$count`** that is incremented only when we encounter a file, not a subdirectory. After the **`foreach`** loop is over, write an **`if`** to prevent us from dividing by zero if **`$count`** is zero.

▲

if-then-else

You will often need a pair of consecutive **`if`**'s of which exactly one will be true: never both and never neither.

```
1 #!/bin/perl
2
3 $a = 1;
4 $b = 2;
5
6 if ($a == $b) {
7     print 'equal';
8 }
9
```

```
10 if ($a != $b) {
11     print 'not equal';
12 }
```

```
 1 #!/bin/perl
 2
 3 $a = 1;
 4 $b = 2;
 5
 6 if ($a == $b) {
 7     print 'equal';
 8 } else {
 9     print 'not equal';
10 }
```

```
 1 #!/bin/perl
 2 #A three-way if: three possibilities
 3
 4 $a = 1;
 5 $b = 2;
 6
 7 if ($a == $b) {
 8     print 'equal';
 9 } elsif ($a < $b) {
10     print 'less than";
11 } else {
12     print "greater than";
13 }
```

```
 1 #!/bin/perl
 2 #A four-way if: four possibilities
 3
 4 $a = 1;
 5 $b = 2;
 6
 7 if ($a == $b) {
 8     print "equal";
 9 } elsif ($a < $b) {
10     print "less than";
11 } elsif ($a > $b + 100) {
12     print "much greater than";
13 } else {
14     print "greater than, but not all that much greater than";
15 }
```

▼ Homework 10.12: what kind of election year is this?

Write a perlscript that asks the user to type in the current year. Then output exactly one of the following three statements. Use the **%** operator and **`elsif`**.

```
    This is a presidential election year.          (multiple of 4)
    This is a congressional election year.         (2 more than a multiple of 4)
    This is a local election year.                 (none of the above)
```

▲

▼ **Homework 10.13: simplify the following if statements**

Make the following statements more elegant. Do not change their output.

```
1 #Example 1.  This is how they programmed before they invented else.
2 $c = 10;
3 if ($a == $b) {
4     $c = 20;
5 }
```

```
1 #Example 1.  Use process of elimination.
2 if ($a == $b) {
3     print 'equal';
4 } elsif ($a != $b) {
5     print 'not equal';
6 }
```

```
1 #Example 2.  Use process of elimination.
2 if ($a == $b) {
3     print 'equal';
4 } elsif ($a < $b) {
5     print 'less than';
6 } elsif ($a > $b) {
7     print 'greater than';
8 }
```

```
1 #Example 3.  Use the distributive law: a * c + b * c == (a + b) * c
2 if ($a == $b) {
3     print "equal.\n";
4     exit 0;
5 } else {
6     print "not equal.\n";
7     exit 0;
8 }
```

```
1 #Example 4.  Use the distributive law: c * a + c * b == c * (a + b)
2 if ($a == $b) {
3     print 'They are ';
4     print "equal.\n";
5 } else {
6     print 'They are ';
7     print "not equal.\n";
8 }
```

```
1 #Example 5.  Never write a "null else" in perl.
```



```
2 if ($a == $b) {
3     print 'equal';
4 } else {
5 }
```

```
1 #Example 6.  Never write a "null then" in perl.
2 if ($a == $b) {
3 } else {
4     print 'not equal';
5 }
```

```
1 #Example 7.
2 if ($a == $b) {
3     exit 0;
4 } else {
5     print "Let's keep going.\n";
6 }
```

```
1 #Example 8.
2 if ($a == $b) {
3     print "Let's keep going.\n";
4 } else {
5     exit 0;
6 }
```

```
1 #Example 9.  Use && (p. 89).
2 if ($a == $b) {
3     if ($c == $d) {
4         print "Both pairs are equal.\n";
5     }
6 }
```

```
1 #Example 10.  Use || (p. 89).
2 if ($a == $b) {
3     print "At least one pair is equal.\n";
4 } elsif ($c == $d) {
5     print "At least one pair is equal.\n";
6 }
```

```
1 #Example 11.
2 if ($profit >= $loss) {
3     print "We're in the black.\n";
4 }
5
6 if ($profit == $loss) {
7     print "But only barely.\n";
8 }
```

```
 1 #Example 12.
 2 if ($a != $b) {
 3     if ($a < $b) {
 4         print 'less than';
 5     } else {
 6         print 'greater than';
 7     }
 8 } else {
 9     print 'equal';
10 }
```

▲

Swap the "then" and the "else"

Because the last example is so important, we show the solution. Change the first logical expression from **!=** to **==** to swap the ''then'' and the ''else'' :

```
1 if ($a == $b) {
2     print 'equal';
3 } else {
4     if ($a < $b) {
5         print 'less than';
6     } else {
7         print 'greater than';
8     }
9 }
```

Next, combine the consecutive words **else if** to **elsif**, and remove one pair of **{**curly braces**}** and one level of indentation:

```
1 if ($a == $b) {
2     print 'equal';
3 } elsif ($a < $b) {
4     print 'less than';
5 } else {
6     print 'greater than';
7 }
```

▼ **Homework 10.14: simplify the following if statement**

```
 1 if ($a != $b) {
 2     if ($a >= $b) {
 3         if ($a > $b + 100) {
 4             print 'much greater than';
 5         } else {
 6             print 'greater than, but not all that much greater than';
 7         }
 8     } else {
 9         printf 'less than';
10     }
11 } else {
12     print 'equal';
13 }
```

▲

"unless" means "if not"

Here are two ways to do the same thing. Was this a good idea?

```
 1 #!/bin/perl
 2
 3 $profit = 1;
 4 $loss = 2;
 5
 6 if ($profit < $loss) {
 7     print "We're in the red.\n";
 8 }
 9
10 unless ($profit >= $loss) {
11     print "We're in the red.\n";
12 }
```

Similarly, **`until`** means **`while not`**.

A shorthand for "if" or "unless": p. 94

All of the above **`if`** and **`unless`** constructions can contain one or more statements. The **`(`**parentheses**`)`** and **`{`**curly braces**`}`** are always required.

If your **`if`** or **`unless`** contains only a single statement, however, you can remove the **`(`**parentheses**`)`** and **`{`**curly braces**`}`**, and put the **`if`** or **`unless`** at the end of the statement:

```
 1 #!/bin/perl
 2
 3 $profit = 1;
 4 $loss = 2;
 5
 6 print "We're in the red.\n" if $profit < $loss;
 7 print "We're in the red.\n" unless $profit >= $loss;
```

```
1$ perlscript
We're in the red.
We're in the red.
```

A shorthand for "while" or "until": p. 94

Here are three ways to write the same infinite loop. You can use the third way only when the loop contains exactly one statement:

```
 1 #!/bin/perl
 2
 3 for (;;) {
 4     print "This is an infinite loop containing one statement.\n";
 5 }
 6
 7 while (1) {
 8     print "This is an infinite loop containing one statement.\n";
 9 }
10
11 print "This is an infinite loop containing one statement.\n" while 1;
```


Write as simply as possible

You don't have to use every feature of the language, at least not at the beginning. For example, if you know C, you can rewrite this loop on p. 246

```
rand($.) < 1 && ($it = $_) while <>;
```

as

```
while (<>) {
    if (rand($.) < 1) {
        $it = $_;
    }
}
```


File i/o

Standard output

The standard output of any Unix program can be directed to any file or hardware device (e.g., any terminal), or can be directed through a pipe into another program as input, or can be stored in a shell variable. The destination of the standard output is specified on the command line that runs the program, not in the program itself:

```
1 #!/bin/perl
2
3 print STDOUT "hello\n";
```

```
1$ perlscript                          to your terminal's sreen
2$ perlscript > outfile                destroy previous contents of outfile
3$ perlscript >> outfile               append to existing file
4$ perlscript > /another/directory/outfile

5$ perlscript > /dev/ttypa             someone else's terminal's screen
6$ perlscript > /dev/gizmo             another hardware device
7$ perlscript > /dev/null              black hole

8$ perlscript | anotherprog
9$ perlscript | lpr                    send the output directly to the printer
10$ anotherprog 'perlscript'           use output of perlscript as command line arg of anotherprog
```

```
#!/bin/sh
#This is a Bourne shellscript.
#Store the output of perlscript in the variable x.

x='perlscript'
echo $x                 #Make sure the assignment statement worked.
```

File output in perl

If your perlscript is to send all of its output to one destination (e.g., one output file or one terminal), the standard output shown above will suffice. But if output is to be sent to more than one destination (e.g., several output files, or an output file and a terminal), use the following machinery.

The first argument of the **open** function (p. 162) is called a *filehandle;* it tells subsequent **print**'s where to send their output to. A filehandle is written in all uppercase. The second argument of **open** can begin with either **>** or **>>**.

open returns a non-zero number for success. A Unix error message should always consist of the name of the program, a colon, a blank, and message itself. **perl** gives values automatically to certain special variables with non-alphanumeric names. The special variable **$0** is the name of the program (p. 114), and the special variable **$!** is the error message (p. 115).

```
1 #!/bin/perl
2
3 #Open an output file in the current directory.
4 if (open(OUTFILE1, '> outfile') == 0) {
5     print STDERR "$0: $!\n";
6     exit 1;
7 }
8
9 #Open an output file in some other directory.
```

```
10 if (open(OUTFILE2, '> /usr/outfile') == 0) {
11     print STDERR "$0: $!\n";
12     exit 1;
13 }
14
15 print OUTFILE1 "hello\n";
16 print OUTFILE2 "goodbye\n";
17
18 close OUTFILE1;
19 close OUTFILE2;
20 exit 0;
```

On my machine, I have no permission to create a file in the **/usr** directory, so line 11 outputs

```
perlscript: Permission denied
```

Two shorthands: p. 95

The **die** command (p. 137) **print**'s its string, the name of the perlscript, the line number, and a **\n** to the **STDERR** and **exit**'s with a non-zero exit status to indicate failure. We can change lines 5−6 above to

```
die "$0: $!";
```

In C and **perl**, the following **if**'s do not necessarily compare both pairs of numbers:

```
if ($a == $b && $c == $d) {
if ($a == $b || $c == $d) {
```

```
 1 #!/bin/perl
 2
 3 #Open an output file in the current directory.
 4 open(OUTFILE1, '> outfile') || die "$0: $!";
 5
 6 #Open an output file in some other directory.
 7 open(OUTFILE2, '> /usr/outfile') || die "$0: $!";
 8
 9 print OUTFILE1 "hello\n";
10 print OUTFILE2 "goodbye\n";
11
12 close OUTFILE1;
13 close OUTFILE2;
14 exit 0;
```

On my machine, the **die** in line 7 prints

```
perlscript: Permission denied at perlscript line 7.
```

▼ Homework 10.15: output to two terminals

You can **open** a terminal or any other hardware device in the same way that you **open** a file:

```
open(TERMINAL '> /dev/ttypa') || die "$0: $1";
```

The Unix command **who** lists the terminals that are currently in use; prepend **/dev/** to the name of each terminal.

Pick two terminals at which other people are logged in. Write a perlscript that **open**'s the two terminals and **print**'s a line to both of them. Then **close** the terminals.

▲

▼ Homework 10.16: output to every terminal

Use **`foreach`** and **`chop`** to loop through the names of all the terminals currently in use. During each loop, **`open`** a terminal, **`print`** a line to it, and **`close`** the terminal.

▲

Output to a pipe

```
 1 #!/bin/perl
 2
 3 #Pipe output directly to the printer.
 4 open(LPR, '| lpr') || die "$0: $!";
 5
 6 #Pipe to the printer, but alphabetize and number the lines along the way.
 7 open(SORTLPR, '| sort | cat -n | lpr') || die "$0: $!";
 8
 9 print LPR "hello\n";
10 print LPR "goodbye\n";
11
12 print SORTLPR "hello\n";
13 print SORTLPR "goodbye\n";
14
15 close LPR;
16 print "The exit status of the lpr was $?.\n";
17 close SORTLPR;
18 exit 0;
```

The above perlscript outputs two files to the printer:

```
hello
goodbye
```

```
1 goodbye
2 hello
```


Standard input

The standard input of any Unix program can be taken from any file or hardware device (e.g., any terminal), or can be the output of another program taken through a pipe. The source of the standard output is specified on the command line that runs the perlscript, not in the perlscript itself:

```
1 #!/bin/perl
2
3 $a = <STDIN>;
4 print "I have just input the following line:\n";
5 print $a;
```

```
1$ perlscript                          from your terminal's keyboard
2$ perlscript < infile                 from an input file
3$ perlscript < /another/directory/infile

4$ perlscript < /dev/ttypa             from someone else's terminal's keyboard
5$ perlscript < /dev/gizmo             another hardware device
6$ perlscript < /dev/null              zero bytes of input

7$ previousprog | perlscript
8$ echo $a | perlscript
```

```
#!/bin/sh
#This is a Bourne shellscript.
#Feed three lines of data into the perlscript's standard input.

perlscript <<\!
moe
larry
curly
!
```

File input in perl

If your perlscript is to take all of its input from one source (e.g., one input file or one terminal), the standard input shown above will suffice. But if input is to be taken from more than one source (e.g., several input files, or an input file and a terminal), use the following machinery. The **<** is optional in the second argument of **open**.

```
 1 #!/bin/perl
 2
 3 #Open an input file in the current directory.
 4 open(INFILE1, '< infile') || die "$0: $!";
 5
 6 #Open an input file in some other directory.
 7 open(INFILE2, '< /usr/infile') || die "$0: $!";
 8
 9 $a = <INFILE1>;
10 $b = <INFILE2>;
11
12 close INFILE1;
13 close INFILE2;
14 exit 0;
```

Input from a pipe

```
1$ cal
      May 1994
 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
```

It's traditional to deposit each line of input into the special variable **$_** (pp. 6−7, 112):

```
 1 #!/bin/perl
 2
 3 #Take input from cal.  It produces a calendar of the current month.
 4 open(CAL, 'cal |') || die "$0: $!";
 5
 6 $_ = <CAL>;
 7 $_ = <CAL>;
 8 $_ = <CAL>;
 9 $_ = <CAL>;
10
11 chop $_;
12 if ($_ eq ' 8  9 10 11 12 13 14') {
13     print "This month has a Friday the Thirteenth.\n";
14 }
15
16 close CAL;
17 print "The exit status of the cal was $?.\n";
18 exit 0;
```

Change lines 11−12 to any one of

```
if ($_ eq " 8  9 10 11 12 13 14\n") {
if ($_ =~ /^ 8/) {
if ($_ =~ / 14$/) {
if (/ 14$/) {
```

Loop through lines of input

```
 1 #!/bin/perl
 2
 3 open(CAL, 'cal |') || die "$0: $!";
 4
 5 $_ = <CAL>;
 6 while ($_ ne '') {
 7     chop $_;
 8     if (/ 14$/) {
 9         print "This month has a Friday the Thirteenth.\n";
10         last;                        #just like break in C
11     }
12     $_ = <CAL>;
13 }
14
15 close CAL;
16 exit 0;
```

Simplify the above example

= is an operator in C and **perl**, just as **+** and **-** are:

```
$a + 10
$_ = <CAL>
```

Therefore **$a + 10** and **$_ = <CAL>** are both expressions, and any expression can be used as part of a larger expression:

```
($a + 10) * $b          perform addition and multiplication in the same expression
($_ = <CAL>) ne ''      perform assignment and comparison in the same expression
```

In both cases, the parentheses are necessary to execute the operator with lower precedence before the operator with higher precedence. See the precedence chart on p. 79.

```
 1 #!/bin/perl
 2
 3 open(CAL, 'cal |') || die "$0: $!";
 4
 5 while (($_ = <CAL>) ne '') {
 6     chop $_;
 7     if (/ 14$/) {
 8         print "This month has a Friday the Thirteenth.\n";
 9         last;                                #just like break in C
10     }
11 }
12
13 close CAL;
14 exit 0;
```

C programmers use the same idiom to compress a **while** loop:

```
 1     int c;
 2
 3     c = getchar();
 4     while (c != EOF) {
 5         putchar(c);
 6         c = getchar();
 7     }
 8
 9     while ((c = getchar()) != EOF) {
10         putchar(c);
11     }
```

Simplify the above example: pp. 6, 8, 66

A string counts as ''true'' if it is not the null string. Therefore you can change

```
while (($_ = <CAL>) ne '') {
```

to

```
while ($_ = <CAL>) {
```

The special variable $_

If you don't specify which variable you want to **chop**, **split**, or **print**, **perl** will use **$_** by default. If you don't specify which variable you want to store the value of **<CAL>** into, **perl** will use **$_** by default. If you don't specify which variable you want to store each line into in a **foreach** loop, **perl**

will use **$_** by default. **$_** is reminiscent of **it** in Macintosh Hypertalk.

The **while** loop will keep going as long as the expression in the parentheses is neither 0 nor the null string **''**. Therefore the **ne ''** is redundant and may be omitted.

```
 1 #!/bin/perl
 2
 3 open(CAL, 'cal |') || die "$0: $!";
 4
 5 while (<CAL>) {
 6     chop;
 7     if (/ 14$/) {
 8         print "This month has a Friday the Thirteenth.\n";
 9         last;                                #just like break in C
10     }
11 }
12
13 close CAL;
14 exit 0;
```

If the input was coming from the standard input rather than a file that we **open**'ed ourselves, we could say (p. 7)

```
1 #!/bin/perl
2 #Copy the standard input to the standard output.
3
4 while (<>) {
5     #Each time we arrive here, $_ is the next line read from input.
6     print;                  #perl prints $_ by default.
7 }
```

Why don't we need a **\n** in the above **print**?

The special variable $.

$. gets a new value each time we perform input with a filehandle, i.e., each time we use the value of **<CAL>** or **<>**:

```
1 #!/bin/perl
2 #Copy the standard input to the standard output.
3 #Add a line number and a blank to the start of each line.
4
5 while (<>) {
6     print "$. $_";
7 }
```

```
1$ perlscript < /etc/passwd
1 root:*:0:1:root:/:/bin/csh
2 nobody:Nologin:-2:-2:nobody:/:/bin/date
3 operator:*:5:28:operator:/home1/acf/operations/operator:/bin/csh
4 ris:Nologin:11:11:ris:/usr/adm/ris:/bin/sh
5 daemon:*:1:1:daemon:/:
```

▼ Homework 10.17: different ways of numbering the lines of input

In place of the above **print**, what would the following ones do?

```
1     print $. - 1, " $_";
```



```
2     print $. + 999, " $_";
3     print 1001 - $., " $_";
4     print 10 * $., " $_";
5     print 1 + ($. - 1) % 10, " $_";
```

What goes wrong when you try

```
6     print "$. - 1 $_";
```

You can use a variable within double quotes, but you can't perform arithmetic there.

▲

▼ Homework 10.18: output only every other line

Write a perlscript that takes in lines of standard input and outputs a copy of only the odd numbered lines. Use **`$.`** and the **`%`** operator in an **`if`** inside the **`while`** loop. Write another perlscript that outputs only the even numbered lines.

▲

▼ Homework 10.19: how much disk space is held back?

```
1$ df
Filesystem              Total    kbytes   kbytes   %
node                    kbytes   used     free     used  Mounted on
/dev/rz0a                 15823   13452      789    94%   /
/dev/rz0h               1233086 1033172   76606    93%   /usr
/dev/rz1h               1233086  706050  403728    64%   /home1
/dev/rz2h               1233086  982109  127669    88%   /home2
/dev/ra35c              1914797 1539204  184114    89%   /home3
```

Is the **`Total`** column the sum of the **`used`** and **`free`** columns? Write a **`perlscript`** that takes lines of input from the **`df`** command, and outputs three columns: the name of the filesystem (column 1 of the input), the **`Total kbytes`** of the filesystem (column 2 of the input), and the **`Total kbytes`** minus the sum of the **`used`** and **`free`**. What determines the amount of disk space that is held back (i.e., counted as neither **`used`** nor **`free`**)?

Ignore the first two lines of input: do no computation until line 3. Use **`$.`** in an **`if`**.

```
2$ perlscript
Filesystem      total    heldback
/dev/rz0a       15823        1582
/dev/rz0h     1233086      123308
/dev/rz1h     1233086      123308
/dev/rz2h     1233086      123308
/dev/ra35c    1914797      191479
```

▲

Two different ways to loop through the lines output by a program

You now know two ways to do the same thing. The first is easier to write; the second uses less memory.

```
1 #!/bin/perl
2
3 foreach (‘prog‘) {
4     do something to $_;
5 }
6
7 open(PROG, ’prog |’) || die "$0: $!";
```

```
 8 while (<PROG>) {
 9     do something to $_;
10 }
11 close PROG;
```

Loop through a list of filenames: p. 78

Instead of looping through the lines of an input file, you can loop through a list of filenames:

```
 1 #!/bin/perl
 2
 3 while (<*>) {
 4     print "$_\n";
 5 }
 6
 7 while (<*.c>) {
 8     print "$_\n";
 9 }
10
11 while (</other/directory/*.c>) {
12     print "$_\n";
13 }
```

Things you can do to the files whose names you loop through: Quick Reference Guide, p. 8

The parentheses around the argument of **`unlink`** are required; without them, **`unlink`** would think that its argument was the entire expression **`$_ || die "$0: $!"`**. Apparently, the precedence of **`||`** is higher than that of the **`unlink`** operator. See pp. 11, 22, 86−87.

```
 1 unlink($_) || die "$0: $!";
 2
 3 rename($_, "new$_") || die "$0: $!";
 4 rename($_, '/new/directory/new$_') || die "$0: $!";
 5
 6 chmod(0755, $_) || die "$0: $!";
 7 chown(123, 456, $_) || die "$0: $!";
 8
 9 system('cp', $_, "new$_") || die "$0: could not copy $_";
10 system('cp', $_, "/new/directory/new$_") || die "$0: could not copy $_";
```

▼ Homework 10.20: clean up the current directory

To test if a file is owned by the person who is running the perlscript,

```
if (-o $filename) {
```

To test if a file is not owned by the person who is running the perlscript,

```
if (! -o $filename) {
```

To test if no one has fed a file as input into any program in the last two weeks,

```
if (-A $file > 14) {
```

Write a perlscript that will loop through all the files in the current directory and remove (i.e., **`unlink`**) every one that is owned by someone else, or hasn't been looked at in two weeks, or contains no bytes, or is not a text file, or whose size is greater than 100,000 bytes, or whose name is **`core`**, or some combination ofthe above. See p. 82 for **`!`**, p. 89 for **`&&`** and **`||`**, and p. 84 for the file test operators. Be careful not to remove your perlscript.

▲

Lists

An array can hold a list

Put a **$** in front of the name of a variable or individual element of an array. Like C, **perl** requires **[**square brackets**]** around the subscript. By default, subscripts start at zero.

```
 1 #!/bin/perl
 2
 3 $a[0] = 'moe';
 4 $a[1] = 'larry';
 5 $a[2] = 'curly';
 6 $a[3] = 10;
 7
 8 print "$a[0]\n";
 9 print "$a[1]\n";
10 print "$a[2]\n";
11 print "$a[3]\n";
12
13 print "$#a\n";       #the subscript of the last element in the array (p. 14)
```

```
1$ perlscript
moe
larry
curly
10
3
```

The special variable $[(p. 114)

By default, the subscripts of every array start at 0. To make them start at 1, say

```
$[ = 1;
```

at the start of your perlscript. Don't think about what would happen if you changed the value of **$[** after you deposited values into an array.

Multiple assignments

Put a **@** in front of the name of an entire array:

```
 1 #!/bin/perl
 2 #Another way to load the same list of values into an array.
 3
 4 @a = ('moe', 'larry', 'curly', 10);
 5
 6 print "$a[0]\n";
 7 print "$a[1]\n";
 8 print "$a[2]\n";
 9 print "$a[3]\n";
10
11 print "$#a\n";
```

To store all the lines of a program's output in an array,

```
@a = 'prog';
```

To copy all the elements of one array into another,


```
@b = @a;
```

To empty out an array (p. 19),

```
@a = ();                #Now $#a will be -1.
```

A list of values does not have to be stored in an array. It can also be stored in a list of individual variables:

```
1 #!/bin/perl
2
3 ($x, $y, $z, $w) = ('moe', 'larry', 'curly', 10);
4
5 print "$x\n";
6 print "$y\n";
7 print "$z\n";
8 print "$w\n";
```

```
1$ perlscript
moe
larry
curly
10
```

Use an array to avoid a list of "elsif"s

```
 1 #!/bin/perl
 2
 3 @animal = (
 4     'monkey',          #0
 5     'rooster',         #1
 6     'dog',             #2
 7     'pig',             #3
 8     'rat',             #4
 9     'ox',              #5
10     'tiger',           #6
11     'hare',            #7
12     'dragon',          #8
13     'snake',           #9
14     'horse',           #10
15     'sheep'            #11
16 );
17
18 print 'What year is this? ';
19 $year = <>;
20 print "This is the year of the $animal[$year % 12].\n";
```

```
1$ perlscript
What year is this? 1994
This is the year of the dog.
```

```
1 #!/bin/perl
2
3 print 'What year is this? ';
4 $year = <>;
5 $remainder = $year % 12;
```



```
 6
 7 if ($remainder == 0) {
 8     print "This is the year of the monkey.\n";
 9 } elsif ($remainder == 1) {
10     print "This is the year of the rooster.\n";
11 } elsif ($remainder == 2) {
12     print "This is the year of the dog.\n";
13 } elsif (
```

Scalar context vs. array context: pp. 18−20

The value of the expression **<STDIN>** can be either a scalar (a single number or string) or a list, depending on the context:

```
1 $a = <STDIN>;          Value of <STDIN> is one line of input.
2 @a = <STDIN>;          Value of <STDIN> is every line of input.
```

The value of the expression **@a** can be either a scalar or a list, depending on the context:

```
1 $count = @a;           Value of @a is the number of elements in @a.
2 @b = @a;               Value of @a is every element of @a.
3
4 if (@a > 0) {
5     print "The array contains at least one element.\n";
6 } else {
7     print "The array contains no elements.\n";
8 }
```

▼ **Homework 10.21: don't hardcode the size of the array**

How can you avoid hardcoding the **12** in the **print** in the Chinese year program?

▲

▼ **Homework 10.22: does "print" supply a scalar context or an array context?**

Create an array named **@a** containing several elements. What happens when you

```
1 print @a, "\n";
2 print "@a\n";
```

▲

Loop through a list

You can loop through a list of numbers with either **for** or **foreach**:

```
1 for ($i = 0; $i <= 4; $i = $i + 1) {
2 foreach $i (0, 1, 2, 3, 4) {
3 foreach $i (0 .. 4) {                    pp. 19, 89−91
4 foreach $prime (2, 3, 5, 7, 11, 13, 17, 19, 23) {
5 foreach $year (1914 .. 1918, 1939 .. 1945) {
6 foreach $i ($[ .. $#a) {
7 foreach $i (@a) {
8 foreach $i (‘prog‘) {
9 foreach (‘prog‘) {
```

but you must loop through a list of strings with **foreach**:

```
1 #!/bin/perl
2
```



```
3 foreach $things ('flowers', 'young girls', 'husbands', 'soldiers',
4     'graveyards') {
5
6     print "Where have all the $things gone?\n";
7 }
```

```
     1$ perlscript
     Where have all the flowers gone?
     Where have all the young girls gone?
     Where have all the husbands gone?
     Where have all the soldiers gone?
     Where have all the graveyards gone?
```

```
1 #!/bin/perl
2 #Produce the same output as above.
3
4 foreach ('flowers', 'young girls', 'husbands', 'soldiers', graveyards) {
5     print "Where have all the $_ gone?\n";
6 }
```

Loop through an array

```
1 #!/bin/perl
2 #Produce the same output as above.
3
4 @things = ('flowers', 'young girls', 'husbands', 'soldiers', graveyards);
5
6 foreach (@things) {
7     print "Where have all the $_ gone?\n";
8 }
```

▼ Homework 10.23: loop through the perlscript's command line arguments

Loop through the **@ARGV** array. Assume that each command line argument of the perlscript is the name of a terminal currently logged in. Within the loop, **open** each terminal, **print** an annoying message on its screen, and **close** the terminal.

Or build your own version of the Unix **rm** command by naming your perlscript **rm**. Assume that each command line argument of the **perlscript** is the name of a file. Within the loop, use the **rename** command to move each file to the **$HOME/.trash** directory. Add **$$** (p. 113) to the end of the name of each file as you move it.

```
1 if (@ARGV <= 0) {
2     die "$0: requires at least one command line argument\n";
3 }
```

▲

Several ways to loop through an array

```
1 #!/bin/perl
2
3 for ($i = $[; $i <= $#ARGV; $i = $i + 1) {
4     do something to $ARGV[$i];
5 }
6
7 $count = grep(do something to $_, @ARGV);       #Use grep in a scalar context.
```



```
 8 @a = grep(do something to $_, @ARGV);          #Use grep in an array context.
 9
10 while (@ARGV) {                                #Use @ARGV in a scalar context.
11     $argument = shift @ARGV;                   #Omit the @ARGV, p. 181
12     do something to $argument;
13 }
```

Nested loops

```
 1 #!/bin/perl
 2 #Output the Cartesian product.
 3
 4 foreach $leaning (’liberal’, ’moderate’, ’conservative’) {
 5     foreach $party (’Democrat’, ’Republican’) {
 6         print "$leaning $party\n";
 7     }
 8 }
```

```
1$ perlscript
liberal Democrat
liberal Republican
moderate Democrat
moderate Republican
conservative Democrat
conservative Republican
```

Could you print all the Democrats first, followed by all the Republicans? Could you print out the **move** instruction in your favorite assembly language with every combination of addressing modes?

▼ Homework 10.24: the 12 days of Christmas

Write a perlscript to output all the lyrics with an array such as

```
1 $[ = 1;              #Let the subscripts go from 1 to 12 instead of 0 to 11.
2
3 @gift = (
4     ’partridge in a pear tree’,
5     ’turtledoves’,
6     ’French hens’
7 );
```

▲

▼ Homework 10.25: clean up several directories

An earlier exercise looped through all the files in one directory and removed some of them. Beef it up to clean up several directories.

Use **chdir** to travel to from one directory to another:

```
1 #!/bin/perl
2
3 chdir ’/new/directory’ || die "$0: $!";
4
5 while (<*>) {
6     print "$_\n";
7 }
```

Put the above statements in a **foreach** loop that will loop through the directory names.

▲

Subroutines: pp. 50−53, 99−102

```
 1 #!/bin/perl
 2
 3 print "There's a man who lives a life of danger\n";
 4 print "To everyone he meets he stays a stranger\n";
 5 print "With every move he makes, another chance he takes\n";
 6 print "Odds are he won't live to see tomorrow.\n\n";
 7
 8 &chorus;
 9
10 print "Beware of pretty faces that you find\n";
11 print "A pretty face can hide an evil mind\n";
12 print "Be careful what you say, or you'll give yourself away\n";
13 print "Odds are he won't live to see tomorrow.\n\n";
14
15 &chorus;
16
17 print "Swinging on the Riviera one day\n";
18 print "Lying in a Bombay alley the next day\n";
19 print "Don't let the wrong words slip while kissing persuasive lips\n";
20 print "Odds are he won't live to see tomorrow.\n\n";
21
22 &chorus;
23
24 sub chorus
25 {
26     foreach (1 .. 2) {
27         print "Secret Agent Man, Secret Agent Man\n";
28         print "They've given you a number, and taken 'way your name.\n";
29     }
30
31     print "\n";
32 }
```

▼ **Homework 10.26: use an array**

Print the three verses with a single **`print`** statement instead of the 12 **`print`**'s used above. Do this with an array named **`@verse`** that will contain three elements. Each element will be a string containing four lines.

```
 1 @verse = (
 2
 3 "There's a man who lives a life of danger
 4 To everyone he meets he stays a stranger
 5 With every move he makes, another chance he takes
 6 Odds are he won't live to see tomorrow.",
 7
 8 "Beware of pretty faces that you find
 9 A pretty face can hide an evil mind
10 Be careful what you say, or you'll give yourself away
11 Odds are he won't live to see tomorrow.",
12
13 #etc.
```

▲

Arguments and return value

You rarely need the **`return`** statement, pp. 53, 99, 173.

```
 1 #!/bin/perl
 2
 3 $i = 10;
 4 $sum = &sum($i, 20);
 5 print "The sum is $sum.\n";
 6
 7 sub sum {
 8     $count = @_;
 9     print "I received $count command line arguments.\n";
10     print "The first argument is $_[0], the second is $_[1].\n";
11     $_[0] + $_[1];
12 }
```

```
     1$ perlscript
     I received 2 command line arguments.
     The first argument is 10, the second is 20.
     The sum is 30.
```

Call by reference

Call by reference allows a subroutine to change the value of a variable passed to it.

```
 1 #!/bin/perl
 2
 3 $i = 10;
 4 &f($i);
 5 print "The new value of \$i is $i.\n";
 6
 7 sub f {
 8     if (@_ <= 0) {
 9         die "$0: the function f requires at least one argument";
10     }
11     $_[0] = $_[0] + 1;
12 }
```

```
     1$ perlscript
     The new value of $i is 11.
```

Call by value

Call by value prohibits a subroutine from changing the value of a variable passed to it. Warning: without the word **`local`**, you would get call by reference.

```
 1 #!/bin/perl
 2
 3 $i = 10;
 4 $j = 20;
 5 &f($i, $j);
 6 print "The value of \$i is $i.\n";
 7
 8 sub f {
 9     die "$0: the function f requires at least two arguments" if @_ <= 0;
10     local($firstarg, $secondarg) = @_;
```



```
11      $firstarg = $firstarg + 1;
12  }
```


Associative arrays

What associative arrays are for: pp. 29−32

The first two examples are real C programs; the third is imaginary.

```
 1 /* Input a month number (1 to 12), output the number of days in the
 2 month (1 to 31). */
 3
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6
 7 int length[] = {
 8     0,      /* dummy, so that January will have subscript 1 */
 9     31,     /* January */
10     28,     /* February */
11     31,     /* March */
12     30,     /* April */
13     31,     /* May */
14     30,     /* June */
15     31,     /* July */
16     31,     /* August */
17     30,     /* September */
18     31,     /* October */
19     30,     /* November */
20     31      /* December */
21 };
22 const int n = 12;
23
24 main (int argc, char **argv)
25 {
26     int month;
27
28     scanf ("%d", &month);
29
30     if (1 <= month && month <= n) {
31         printf ("It has %d days.\n", length[month]);
32         exit (0);
33     } else {
34         fprintf (stderr, "%s: input must be in range 1 to %d inclusive\n",
35             argv[0], n);
36         exit (1);
37     }
38 }
```

```
 1 /* Input a month name, output the number of days in the month (1 to 31). */
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <string.h>
 5
 6 typedef struct {
 7     char *name;      /* name of the month */
 8     int length;      /* number of days */
 9 } month;
10
11 month a[] = {
```

```
12      {"January",      31},
13      {"February",     28},
14      {"March",        31},
15      {"April",        30},
16      {"May",          31},
17      {"June",         30},
18      {"July",         31},
19      {"August",       31},
20      {"September",    30},
21      {"October",      31},
22      {"November",     30},
23      {"December",     31}
24  };
25  const int n = 12;
26
27  main (int argc, char **argv)
28  {
29      char monthname[100];
30      month *p;
31
32      scanf ("%s", monthname);
33
34      for (p = a; p < a + n; ++p) {
35          if (strcmp(p->name, monthname) == 0) {
36              printf ("It has %d days.\n", p->length);
37              exit (0);
38          }
39      }
40
41      fprintf (stderr, "%s: input must be name of a month\n", argv[0]);
42      exit (1);
43  }
```

```
 1  /* Input a month name, output the number of days in the month (1 to 31). */
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <string.h>
 5
 6  int length[];
 7  const int n = 12;
 8
 9  main (int argc, char **argv)
10  {
11      char monthname[100];
12
13      length["January"] = 31;
14      length["February"] = 28;
15      length["March"] = 30;
16      length["April"] = 30;
17      length["May"] = 31;
18      length["June"] = 30;
19      length["July"] = 31;
20      length["August"] = 31;
21      length["September"] = 30;
```

```
22      length["October"] = 31;
23      length["November"] = 30;
24      length["December"] = 31;
25
26      scanf ("%s", monthname);
27      if (length[monthname] != 0) {
28          printf ("It has %d days.\n", length[monthname]);
29          exit (0);
30      } else {
31          fprintf (stderr, "%s: input must be name of a month\n", argv[0]);
32          exit (1);
33      }
34  }
```

```
 1  #!/bin/perl
 2  #Input a month name, output the number of days in the month (1 to 31).
 3
 4  $length{'January'} = 31;
 5  $length{'February'} = 28;
 6  $length{'March'} = 30;
 7  $length{'April'} = 30;
 8  $length{'May'} = 31;
 9  $length{'June'} = 30;
10  $length{'July'} = 31;
11  $length{'August'} = 31;
12  $length{'September'} = 30;
13  $length{'October'} = 31;
14  $length{'November'} = 30;
15  $length{'December'} = 31;
16
17  $monthname = <>;
18  chop $monthname
19
20  if (defined $length{$monthname}) {
21      print "It has $length{$monthname} days.\n";
22      exit 0;
23  } else {
24      die "$0: input must be name of a month\n";
25  }
```

```
 1  #!/bin/perl
 2  #Input a month name, output the number of days in the month (1 to 31).
 3
 4  %length = (
 5      'January',   31,
 6      'February',  28,
 7      'March',     30,
 8      'April',     30,
 9      'May',       31,
10      'June',      30,
11      'July',      31,
12      'August',    31,
13      'September', 30,
14      'October',   31,
```

```
15     'November',  30,
16     'December',  31
17 );
18
19 $monthname = <>;
20 chop $monthname;
21
22 if (defined $length{$monthname}) {
23     print "It has $length{$monthname} days.\n";
24     exit 0;
25 } else {
26     die "$0: input must be name of a month\n";
27 }
```

Another natural use for an associative array

```
1$ echo $HOME
/home1/m/mm64

2$ echo $LOGNAME
mm64
```

```
1 #!/bin/perl
2
3 print "$ENV{'HOME'}\n";       #the environment variable $HOME
4 print "$ENV{'LOGNAME'}\n";    #the environment variable $LOGNAME
5
6 print "$ARGV[0]\n";           #the first command line argument
7 print "$ARGV[1]\n";           #the second command line argument
8 print "$ARGV[2]\n";           #the third command line argument
```

How much memory is each person using?

```
1$ ps -acux
USER       PID %CPU %MEM   SZ  RSS TT STAT   TIME COMMAND
meretzky 28773  0.0  0.1  376  148 p8 I      0:02 csh
meretzky  9329  0.0  0.5 1048  844 p8 R      0:00 ps
meretzky  9328  0.0  0.1  348  104 p8 S      0:00 csh
abc1234   9327  0.0  0.0    0    0 p8 Z      0:00 <exiting>
meretzky  9180  0.0  0.1  344  176 p8 S      0:00 vi

2$ ps -acux | tail +2
meretzky 28773  0.0  0.1  376  148 p8 I      0:02 csh
meretzky  9329  0.0  0.5 1048  844 p8 R      0:00 ps
meretzky  9328  0.0  0.1  348  104 p8 S      0:00 csh
abc1234   9327  0.0  0.0    0    0 p8 Z      0:00 <exiting>
meretzky  9180  0.0  0.1  344  176 p8 S      0:00 vi
```

```
1 #!/bin/perl
2
3 $[ = 1;
4 open(PS, "ps -acux | tail +2 |") || die "$0: $!";
5
6 while (<PS>) {
```

```
 7     @F = split;
 8     $sum{$F[1]} = $sum{$F[1]} + $F[5];
 9 }
10
11 foreach $loginname (keys(%sum)) {
12     print "$loginname\t$sum{$loginname}\n";
13 }
```

```
nabutvsk          596
dube              1092
goodmanj          348
volchan           372
bogomolv          684
```

To loop through the array in alphabetical order, say

```
foreach $loginname (sort keys(%sum)) {
```

▼ Homework 10.27: how much time has each person used?

Write a perlscript to loop through the lines output by **`ps -acux`** and sum up the **`TIME`** instead of the **`SZ`**. Use **`split`** to deposit the **`TIME`** into **`$F[$#F - 1]`** (the next-to-last element of the array **`@F`**. Then **`split $F[$#F - 1]`** into two halves at the colon:

```
($minutes, $seconds) = split(':', $F[$#F - 1]);
```

Do the extra variables **`$number_of_fields`** and **`$time`** make the perlscript easier to understand or annoyingly verbose:

```
1 @F = split;
2 $number_of_fields = $#F;
3 $time = $F[$number_of_fields - 1];
4 ($minutes, $seconds) = split(':', $time);
```

▲

Implicit input-driven looping

Emulate grep and awk with the -n option: p. 355

A perlscript is often nothing more than a loop that processes each line of input:

```
1 #!/bin/perl
2 #Output the lines of input that are too long to print on a page.
3
4 while (<>) {
5     if (length($_) > 80) {
6         print $_;
7     }
8 }
```

```
1 #!/bin/perl
2 #Output the lines of input that are too long to print on a page.
3
4 while (<>) {
5     print if length > 80;
6 }
```

In this case, use the **-n** option instead of writing an explicit **while** loop:

```
1 #!/bin/perl -n
2 #Output the lines of input that are too long to print on a page.
3
4 print if length > 80;
```

```
1$ perl -ne 'print if length > 80;'          Final semicolon is optional.
2$ perl -ne 'print "$.\n" if length > 80'
3$ perl -ne 'print "$. $_" if length > 80'
```

When you combine two or more command line arguments such as **-p** and **-e** into the single argument **-pe**, the **e** must come last.

▼ **Homework 10.28: find all the anti's**

Fill in the single-quoted string so that

```
1$ perl -ne '???' /usr/dict/words
```

will output all the lines in the file **/usr/dict/words** that start with **anti**.

▲

Autosplit mode: p. 352

```
1 #!/bin/perl
2
3 while (<>) {
4     @F = split;
5     print "The first word on the line is $F[0]\n";
6 }
```

```
1 #!/bin/perl -n
2
3 @F = split;
```



```
4 print "The first word on the line is $F[0]\n";
```

```
1 #!/bin/perl -an
2
3 print "The first word on the line is $F[0]\n";
```

```
      1$ who | perl -ane 'print "$F[0]\n"'
```

Regular expressions: pp. 103−106

Print the lines of input that contain a given string

Always write a regular expression within /slashes/.

```
1 #!/bin/perl
2
3 while (<>) {
4     if ($_ =~ /mania/) {
5         print $_;
6     }
7 }
```

```
1 #!/bin/perl
2
3 while (<>) {
4     print $_ if $_ =~ /mania/;
5 }
```

```
1 #!/bin/perl
2
3 while (<>) {
4     print if /mania/;
5 }
```

```
1 #!/bin/perl -n
2
3 print if /mania/;
```

```
1$ perl -ne 'print if /mania/' /usr/dict/words | more
```

Try phobia, esque, etc.

Regular expressions containing ^

Search for your favorite prefixes: macro, mega, micro, octo, over, under, Italo, para, pre, pseudo, voodo, etc.

```
1 perl -ne 'print if /^anti/' /usr/dict/words | more
2 ls -l | tail +2 | perl -ne 'print if /^-/'        List only the files.
3 ls -l | tail +2 | perl -ne 'print if /^d/'        List only the directories.
4 ls -l | tail +2 | perl -ane 'print "$F[7]\n" if /^d/'
5 perl -ne 'print if /^#/' prog.c
6 perl -ne 'print if ! /^#/' prog.c
7 perl -ne 'print unless /^#/' prog.c
8 perl -ne 'print unless /^C/' prog.f              Output the Fortran program sans comments.
```

Regular expressions containing ^ and $

Search for your favorite suffixes: able, mania, maniac, ocracy, phobiac, oxide, tomy (as in ‘‘lobotomy’’), ist, ity, eqsue, fish, etc.

```
1 perl -ne 'print if /phobia$/' /usr/dict/words | more
```

```
 2 perl -ne 'print if /:/bin/csh$/' /etc/passwd | more                bad
 3 perl -ne 'print if mx:/bin/csh$x' /etc/passwd | more               p.125
 4 perl -ne 'print unless mx:/bin/csh$' /etc/passwd | more
 5 perl -ne 'print if /^$/' prog.c | wc -l
 6 perl -ne 'print unless /^$/' prog.c | lpr
 7 perl -ne 'print if /\^/'                                  search for a caret
 8 perl -ne 'print if /\$/'                                  search for a dollar sign
 9 perl -ne 'print if /\\/'                                  search for a backslash
```

▼ Homework 10.29: search for pathalogical lines

Search for lines whose first character is a caret. Search for lines whose last character is a dollar sign. Search for lines whose first two characters are a backslash and a caret.

▲

The . wildcard

```
 1 perl -ne 'print if /separate/' /usr/dict/words
 2 perl -ne 'print if /seperate/' /usr/dict/words
 3 perl -ne 'print if /sep.rate/' /usr/dict/words

 4 perl -ne 'print if /^c.k/' /etc/passwd
 5 perl -ne 'print if /\./'

 6 perl -ne 'print if /^...u.$/i' /usr/dict/words          case insensitive, p. 125
 7 perl -ne 'print if /^...u./i' /usr/dict/words
 8 perl -ne 'print if /...u.$/i' /usr/dict/words
 9 perl -ne 'print if /...u./i' /usr/dict/words
10 perl -ne 'print if /^b.g$/i' /usr/dict/words
11 perl -ne 'print if /^p.t$/i' /usr/dict/words
```

List the files that anyone can execute:

```
12 ls -l | perl -ne 'print if /^-.......x/' /usr/dict/words
13 ls -l | perl -ne 'print if /^-.{8}x/' /usr/dict/words          p. 103 for {}
```

▼ Homework 10.30: Capitalism, Communism, Nationalism

The Twentieth Century has been a battleground of conflicting isms, from absenteeism to Zionism. List the isms (i.e., words that end with ''ism'' in either upper or lowercase) in **`/usr/dict/words`** that are at least six characters long. Pipe the output of **`perl`** into

```
pr -4 -l22 -t                                    minus lowercase L twenty-two
```

to output the isms in four columns of 22 lines each.

▲

The [] wildcard

```
.                                   any character at all
[ABCDEFGHIJKLMNOPQRSTUVWXYZ]        any uppercase letter
[A-Z]                               any uppercase letter; no space on either side of dash, can't say [Z-A]
```

Suppose you had a file of names, one per line:

```
perl -ne 'print if /^[A-K]/' names > ak
perl -ne 'print if /^[L-Z]/' names > lz
```



```
perl -ne 'print if /^[A-K]/i' names > ak
perl -ne 'print if /^[A-Ka-k]/' names > ak
```

`[a-z]`	*any lowercase letter*
`[A-Za-z]`	*any letter; can't say* **`[A-z]`***—see* **`/usr/pub/ascii`**
`[a-zA-Z]`	*another way to do the same thing*
`[0123456789]`	*any digit*
`[0-9]`	*any digit*
`\d`	*any digit, p. 104*
`[0-7]`	*any octal digit*
`[0-9A-Fa-f]`	*any hexadecimal digit*
`[A-Za-z0-9_]`	*any character allowed in a C or C++ variable name*
`\w`	*any character allowed in a C or C++ variable name, p. 104*
`[ \t\n]`	*a blank, a tab, or a newline*
`\s`	*a blank, a tab, or a newline, p. 104*

```
perl -ne 'print if /[02468]$/'
```
assume each line ends with a number; search for even

```
perl -ne 'print if /^[0-9][0-9][0-9][0-9][0-9]$/'
```
United States zip code

```
perl -ne 'print if /^\d\d\d\d\d$/'
```
United States zip code, p. 104

```
perl -ne 'print if /^\d{5}$/'
```
United States zip code, p. 103

```
perl -ne 'print unless /^\d{5}$/'

perl -ne 'print if /^[A-Z][0-9][A-Z] [0-9][A-Z][0-9]$/'
```
Canadian zip code

```
perl -ne 'print if /^[A-Z]\d[A-Z] \d[A-Z]\d$/'
```
Canadian zip code

```
perl -ne 'print if /\[/'
```
search for a **`[`**

```
perl -ne 'print if /\]/'
```
search for a **`[`**

Would a line consisting of only the three characters **`ABC`** be output if you fed it into

```
perl -ne 'print if /^[ABC]$/'
perl -ne 'print if /[ABC]/'
```

▼ Homework 10.31: Bonfire of the Vanities

> Before losing consciousness, he was able to give his mother the first letter—R—and five possibilities for the second letter—E, F, B, R, P—of the license plate of the luxurious Mercedes-Benz that ran him down on Bruckner Boulevard and sped off.
>
> *—Chapter 12*

Write a command that will output the number of words in **`/usr/dict/words`** that match the above description and that are at most seven characters long. Output only one line, containing only one number. Search for both upper and lowercase: don't miss **`RPM`**.

Use **`length`** and **`&&`** inside the conditional expression of the **`perl if`**. Pipe the output of **`perl -ne`** into **`wc -l`**. Or dispense with **`wc -l`** and write a perlscript with an explicit **`while`** loop and a **`print`** statement after the **`while`** loop.

▼ **Homework 10.32: abc1234**

How many lines in the file **/etc/passwd** begin with login names of the form **abc1234**? Search for lines that start with three lowercase letters, four digits, and a colon. Output only one line, containing only one number. A colon is not a special character in a regular expression, so you need no \ in front of it.
▲

▼ **Homework 10.33: 737-3783**

Write a perlscript named **spelledby** to output all the words in **/usr/dict/words** that are spelled by your seven-digit phone number. Use someone else's number if yours contains a zero or one. Search for both upper and lowercase.

No luck? Try your first three and last four digits separately; or your first four and last three, etc. Remove the ^ and **$** only as a last resort.

2	**abc**	**6**	**mno**	
3	**def**	**7**	**prs**	*there's no* **q**
4	**ghi**	**8**	**tuv**	
5	**jkl**	**9**	**wxy**	*there's no* **z**

▲

▼ **Homework 10.34: administrative aid for the NYU School of Continuing Education**

Write a perlscript named **daycount** that will output how many of the specified week days there are in a given month. The first argument must consist of exactly one uppercase letter followed by exactly two lowercase letters. Spell Thursday **Thu**, to make it easy to use the first word output by **date** as the first argument of **daycount**. Output only one line, containing only one number. For example,

```
1$ daycount Tue 11 2006
4
```

This perlscript must take exactly three command line arguments: output an error message otherwise. Make sure that

```
$#ARGV == 2
@ARGV == 3
```

If the user typed the three command line arguments correctly, **daycount** must output only one line, containing only one number and nothing else.

To count how many Sundays there are, count the lines that have a digitin the first field:

```
2$ cal 11 2006
   November 2006
 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
```

```
1 #!/bin/perl
2 #Not a complete perlscript.
3
4 open (CAL, 'cal 11 2006 |'
5
6 while (<CAL>) {
7     if ($. >= 3) {
8         @F = split;
```



```
 9          if ($F[0] =~ /[0-9]/) {
10              $count = $count + 1;
11          }
12
13 while (<CAL>) {
14      if ($. >= 3) {
15          @F = split;
16          ++$count if $F[0] =~ /\d/;
17
```

To count the Mondays, search for lines that have a digit in **$F[1]**. To count the Tuesdays, search for lines that have a digit in **$F[2]**.

But don't write a chain of seven **elsif**'s. Use an associative array:

```
1 %a = (
2     'Sun',  0,
3     'Mon',  1,
4     'Tue',  2,      #etc.
```

Use **$ARGV[0]** (the first command line argument) as the subscript of the associative array:

```
$a{$ARGV[0]}
```

Use **$a{$ARGV[0]}** (a number in the range 0 through 7 inclusive) as the subscript of the array **@F**:

```
$F[$a{$ARGV[0]}]
```

Use **$F[$a{$ARGV[0]}]** as the left operand o the **=~**.

▲

Parentheses and |

```
1 perl -ne 'print if /prochoice|prolife/'
2 perl -ne 'print if /pro(choice|life)/'
3 perl -ne 'print if /(pro|anti)(choice|life|abortion)/'
4 perl -ne 'print if /^(..)*$/'                          a line of even length
5 perl -ne 'print unless /^(..)*$/'                      a line of odd length
```

[^] wildcard

```
[A-Z]                       any uppercase letter
[]   !"#$%&'()*+,./0123456789:;<=>?@[\^_`abcdefghijklmnopqrstuvwxyz{|}~-]
[^ABCDEFGHIJKLMNOPQRSTUVWXYZ]    any character except an uppercase letter
[^A-Z]                      any character except an uppercase letter
[^ABC]                      any character except an uppercase A, B, or C
[^A]                        any character except an uppercase A
```

```
perl -ne 'print if /q[^u]/i' /usr/dict/words
QED
Qatar

perl -ne 'print if /q$/i' /usr/dict/words
Iraq
```

```
perl -ne 'print if /q([^u]|$)/i' /usr/dict/words
Iraq
QED
Qatar
```
p. 103

See the wildcard abbreviations on p. 104: **\d**, **\dD**, etc.

The difference between [^ and "unless"

```
perl -ne 'print if /[^A]/'
perl -ne 'print unless /A/
ABC
```
every line that contains a character other than **A**
every line that has no **A**
suppose you input this line to the last two **perl***'s*

Search for a number or word that is not part of a longer number or word

To search for the number **100** without finding longer numbers such as **21003**,

```
 1 perl -ne 'print if /100/'
 2 perl -ne 'print if /[^0-9]100[^0-9]/'
 3 perl -ne 'print if /\D100\D/'
 4 perl -ne 'print if /^100$/'
 5 perl -ne 'print if /(^|\D)100(\D|$)/'
```
Does this output every line that contains **100***?*
p. 104

To search for a variable named **max** without finding longer names such as **maximum** or **xmax**,

```
 6 perl -ne 'print if /max/'
 7 perl -ne 'print if /[^A-Za-z0-9_]max[^A-Za-z0-9_]/'
 8 perl -ne 'print if /\Wmax\W/'
 9 perl -ne 'print if /(^|\W)max(\W|$)/'
10 perl -ne 'print if /\bmax\b/'
```
p. 104
word boundary, p. 103

▼ Homework 10.35: find the HIV newsgroups

The network news is divided by topic into *newsgroups*, i.e., bulletin boards; see **man -t nn**. The names of the newsgroups are listed one per line by

```
nngrep -a | grep -v '^Connecting' | more
```

Write one **perl** command to output only the names of each newsgroups whose name contains HIV. Search for uppercase and lowercase. There were HIV newsgroups on November 22, 2006.
▲

Search for a non-printing character

The file **/usr/pub/ascii** shows that the first and last printable characters (except for the tab) are blank and tilde. The wildcard

```
[ -~]
```
one blank after the **[**

would therefore match all of them (except for the tab), and the wildcard

```
[ -~   ]
```
one blank after the **[**, *one tab before the* **]**

would match all of them. To search for lines containing a non-printing character,

```
perl -ne 'print if /[^ -~   ]/'
perl -ne 'print if /[^\x20-\x7E\x09]/'
```
one blank after the ^, one tab before the **]**
p. 104

▼ **Homework 10.36: search for unusual execute permission: pp. 54−55**

The fourth character of every line (except the first) output by **`ls -l`** is a dash or a lowercase **`x`**, right?

```
1$ cd $m46/handout
2$ ls -l | tail +2
drwxr-xr-x   2 mm64     users         10 Jan 31  2001 answer
-r--r--r--   1 mm64     users       1307 Nov 22 13:30 handout10.ms
```

Search for all the lines in the output of **`ls -l /usr/bin | tail +2`** whose fourth character is neither a dash nor a lowercase **`x`**. Use **`unless`**.

▲

What can go wrong with a wildcard with [square brackets]

Never use **[**square brackets**]** unless you are writing more than one character in them:

```
 1 grep '[ABC]'          good
 2 grep '[AB]'           good
 3 grep '[A]'            bad   ☜
 4 grep 'A'              good
```

You need no dash when you write two consecutive characters in **[**square brackets**]**:

```
 5 grep '[A-C]'          good
 6 grep '[A-B]'          bad
 7 grep '[AB]'           good
```

See the top of p. 104:

```
 8 perl -ne 'print if /[ABC]/'
 9 perl -ne 'print if /[A-C]/'
10 perl -ne 'print if /[A\-C]/'
11 perl -ne 'print if /[AC-]/'
```

What the asterisk is for: p. 103

Never write an asterisk in a regular expression without a character or wildcard immediately to the left of it. **`.*`** (i.e., dot star) in a regular expression has the same meaning as **`*`** in the shell language.

```
 1 perl -ne 'print if /^[A-K]/' names > ak          no leading blanks
 2 perl -ne 'print if /^ [A-K]/' names >> ak        one leading blank
 3 perl -ne 'print if /^  [A-K]/' names >> ak       two leading blanks
 4 perl -ne 'print if /^   [A-K]/' names >> ak      three leading blanks

 5 perl -ne 'print if /^ *[A-K]/' names > ak        zero or more leading blanks

 6 perl -ne 'print if /21210040/'                   separated by no characters
 7 perl -ne 'print if /212.10040/'                  separated by one character
 8 perl -ne 'print if /212..10040/'                 separated by two characters
 9 perl -ne 'print if /212...10040/'                separated by three characters
10 perl -ne 'print if /212.{3}10040/'               separated by three characters, p. 103

10 perl -ne 'print if /212.*10040/'                 separated by zero or more characters
11 perl -ne 'print if /212.+10040/'                 separated by one or more characters
12 perl -ne 'print if /212.+?0040/'                 separated by zero or one characters

11 perl -ne 'print if /^anti.*ism$/i' /usr/dict/words
12 perl -ne 'print if /^anti/' /usr/dict/words | perl -ne 'print if /ism$/'
```

```
13 perl -ne 'print if /A/'                    Search for lines containing one or more A's
14 perl -ne 'print if /A.*A/'                 Search for lines containing two or more A's.
15 perl -ne 'print if /A.*A.*A/'              Search for lines containing three or more A's.
16 perl -ne 'print unless /'A.*A.*A/'         Search for lines containing less than three A's.
```

Examples of regular expressions using [^] and *

To output the lines that contain phone numbers in area code 212 (i.e., lines that contain **212** with no digits ahead of them):

```
1 perl -ne 'print if /^[^0-9]*212/'
2 perl -ne 'print if /^\D*212/'                                              p. 104
```

To output the lines in the file **/etc/passwd** with no password,

```
3 perl -ne 'print if /^[^:]*::/' /etc/passwd
4 perl -ne '@F = split(/:/, $_); print if length($F[1]) == 0' /etc/passwd
5 perl -ne '@F = split(/:/); print if length($F[1]) == 0' /etc/passwd
6 perl -ne '@F = split(/:/); print if $F[3] == 2048' /etc/passwd

7 perl -ne 'print if /A/'                                    lines with one or more A's
8 perl -ne 'print if /^[^A]*A[^A]*$/'                        lines with exactly one A
9 perl -ne 'print if /^[^A]*A[^A]*A[^A]*$/'                  lines with exactly two A's
```

```
1 #!/bin/perl
2 Print every line of input that has an odd number of single quotes.
3
4 print "$. $_" unless /^([^']*'[^']*'[^']*)*$/;
```

To output the name of every directory on the machine, one per line, run this **find** command in the background because it takes so long:

```
1$ find / -type d -print > $HOME/find.out &
```

To output the name of every directory on the third level (e.g., **/usr/dict**, **/home1/a**, **/home1/m**, etc.), **grep** for the lines that have exactly two slashes:

```
2$ ps -lx                                    When you no longer see find, then you can...
3$ perl -ne 'print if /^/[^/]*/[^/]*$/' find.out                              wrong
4$ perl -ne 'print if m:^/[^/]*/[^/]*$:' find.out                             right, p. 125
```

Search for the most common bug in C

To output the lines in **prog.c** that contain **if (a = b)**, where **a** and **b** are any expressions,

```
1 perl -ne 'print if /if *(.*=.*)/' prog.c
2 perl -ne 'print if /if *(.*[^=]=[^=].*)/' prog.c
3 perl -ne 'print if /if *(.*[^<>!=]=[^=].*)/' prog.c
```

Allow zero or more blanks between the word **if** and the left parentheses. The first version finds lines such as

```
if (a = b)
```

but unfortunately it also finds lines such as

```
if (a == b)
```

—exacly what we want to avoid. The second version is smart enough to avoid **if (a == b)**, but unfortunately it still finds line such as


```
if (a != b)
if (a <= b)
```

How not to write regular expressions

Here are three ways of doing the same thing. Write only the first one.

```
1 perl -ne 'printr if /moe/'          good
2 perl -ne 'print if /moe.*/'         bad
3 perl -ne 'print if /moe.*$/'        bad
```

Similarly, here are three ways of doing the same thing. Write only the first one.

```
4 perl -ne 'print if /moe/'           good
5 perl -ne 'print if /.*moe/'         bad
6 perl -ne 'print if /^.*moe/'        bad
```

Why aren't the following lines three ways of doing the same thing?

```
7 perl -ne 'print if /212'
8 perl -ne 'print if /\D*212'                    \D means [^0-9], p. 104
9 perl -ne 'print if /^\D*212'
3212                                  suppose you input this line to the last three grep's
```

▼ Homework 10.37: a facetious example

Write a perlscript to output all the lines in **/usr/dict/words** that contain all five vowels in alphabetical order, with each vowel appearing exactly once. ''Y'' is not a vowel. Search for both upper and lowercase. Examples: abstemious, facetious.

▲

Tagged regular expressions: pp. 104−106

Each time you use a regular expression, you automatically give values to the three variables **$‘**, **$&**, and **$’** (pp. 111−112):

```
1 #!/bin/perl -n
2 #Look for lines of input containing a zip code.
3
4 if ($_ =~ /[0-9][0-9][0-9][0-9][0-9]/) {
5     print "Line $. contains the zip code $&.\n";
6     print "The part of line $. before the zip code was $‘.\n";
7     print "The part of line $. after the zip code was $’.\n";
8 }
```

You could change line 4 to

```
if ($_ =~ /\d\d\d\d\d/) {
if ($_ =~ /\d{4}) {
```

In the above example, **$&** was the entire section of the line matched by the regular expression (in this case, the five digits). To store only a part of this section into a variable, put parentheses in the regular expression:

```
1 #!/bin/perl -n
2 #Look for lines of input containing phone number.
3
4 if ($_ =~ /(\d{3})-(\d{3}-\d{4})/) {
5     print "The area code is $1 and the phone number is $2.\n";
6     print "The area code and phone number together are $&.\n";
7 }
```

You can also say (pp. 125−126)

```
1 #scalar context
2 $success_or_failure = ($_ =~ /(\d{3})-(\d{3}-\d{4})/);
3
4 #array context
5 ($areacode, $pohonenumber) = ($_ =~ /(\d{3})-(\d{3}-\d{4})/);
6 ($areacode, $pohonenumber) = /(\d{3})-(\d{3}-\d{4})/;
```

▼ Homework 10.38: don’t be deceived by a longer number

Change the zip code example above so that it matches only those numbers that are exactly five digits long. Require that the zip code be preceded by the start of the line or a non-digit; require that the zip code be followed by the end of the line or a non-digit. Allow nine-digit as well as five-digit zip codes:

/\d{5}(-\d{4})?/ **?** *means zero or one (i.e., optional), p. 104*

▲

▼ Homework 10.39: look for assembly language labels

Write a perlscript with the **-n** option that will input an assembly language file and output every label that is defined. A label is defined by following it with a colon. Just to be safe, look at a sample of your own local assembly language first.

```
$$11:
    .ascii  "Tell me a knock-knock joke.\X0A\X00":1
main:
    subu$sp, 296
    sw  $31, 28($sp)
$32:
    addu$8, $25, 1
    ble $8, 10, $32
```

▲

▼ **Homework 10.40: list the unreferenced labels**

In the above example, the label **$32** was referenced (i.e., mentioned) in the ‘‘branch if less than or equal’’ instruction. Write a perlscript that will output all the labels that are defined but never referenced.

open the input file of assembly language. As you read each line, use each defined label as the subscript in an associative array. Store a zero into each element of the associative array:

```
$a{$label} = 0;
```

When you are done, **close** the file. Then **open** the file again. This time, count how many times the label is referenced: simply increment the value of each array element whenever you encounter a reference to the corresponding label.

```
$a{$label} = $a{$label} + 1;
```

When you are done, loop through the associative array and print every subscript whose value is still zero.

▲

Use $1, $2, $3 inside the regular expression

Output the lines that contain a double character:

```
perl -ne 'print if /aa/' /usr/dict/words
perl -ne 'print if /bb/' /usr/dict/words
perl -ne 'print if /cc/' /usr/dict/words
```

Write **\1** instead of **$1** inside the regular expression (p. 104):

```
perl -ne 'print if /(.)\1/' /usr/dict/words          a better way
accept
too
```

Output the lines that begin with a double character:

```
perl -ne 'print if /^(.)\1/' /usr/dict/words
eel
ooze
```

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

```
perl -ne 'print if /^(.).*\1$/i' /usr/dict/words
algebra
Celtic
```

Output the lines that contain a triple character:

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


▼ Homework 10.41: illegal, immoral, fattening —Alexander Woollcott (1887−1943)

Find all the words in **/usr/dict/words** that start with a lowercase ‘‘i’’, followed by a double letter. Restrict yourself to words of at least six characters. What do most of these words have in common?

▲

Look for more than one pair of characters

```
per -ne 'print if /(.)\1(.)\2/i' /usr/dict/words
raccoon
Tallahassee

per -ne 'print if /(.)\1(.)\2$/i' /usr/dict/words
Tallahassee
tattoo

perl -ne 'print if /(.)\1..*(.)\2/i' /usr/dict/words
ballyhoo
booboo
butterball

perl -ne 'print if /^(.)\1..*(.)\2$/i' /usr/dict/words
eelgrass

perl -ne 'print if /(.)\1.*(.)\2.*(.)\3/i' /usr/dict/words
committee
Mississippi
Tennessee
```

▼ Homework 10.42: repeated strings of characters

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

```
perl -ne 'print if /(...)\1/i' /usr/dict/words
alfalfa
clinging
instantaneous
murmur
```

▲

▼ Homework 10.43: lines composed of two identical parts

Find all the words in **/usr/dict/words** that are composed of three identical parts. Each part must consist of one or more characters. Ignore the difference between upper and lowercase. Use this method:

```
perl -ne 'print if /'^(..*)\1$/' /usr/dict/words
beriberi
ii
murmur
tutu
```

▲

Search for palindromes

Output the three-character palindromes:

```
perl -ne 'print if /^(.).\1$/i' /usr/dict/words
eye
gag
```

Output the four-character palindromes:

```
perl -ne 'print if /^(.)(.)\2\1$/i' /usr/dict/words
peep
toot

perl -ne 'print if /^(.)(.)\1\2$/i' /usr/dict/words      not palindromes
Mimi
papa
perl -ne 'print if /^(..)\1$/i' /usr/dict/words          does the same thing
```

Output the five-character palindromes:

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

▼ **Homework 10.44: six- and seven-character palindromes: hallah, reviver**

Output the six- and seven-character palindromes in the file **`/usr/dict/words`**.

▲

Look for useless mov instructions

Imagine an assembly language where the register names are **`r0`**, **`r1`**, **`r2`**, ..., **`r9`**. The following **`grep`** command outputs the lines in an assembly language file that contain useless **`mov`** instructions, i.e., **`mov r0,r0`** or **`mov r1,r1`**.

```
perl -ne 'print if /mov (r[0-9]),\1/' file.s
```

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 serach for the syntactically correct lines in a file of arithmetic expressions:

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

In fact, you can't even serach for palindromes of any length, only for palindromes of a specific length. For these and other searching tasks, we'll have to use variables as well as regular expressions, or resort to **`yacc`**.

Substitute commands: pp. 174−176

Emulate sed with the -p option: p. 355

The first half of the substitue command (e.g., **old**) is a regular expression.

```
1 #!/bin/perl
2 #Output a copy of the input, with every "old" changed to "new".
3
4 while (<>) {
5     $_ =~ s/old/new/g;
6     print;
7 }
```

```
1 #!/bin/perl -n
2
3 $_ =~ s/old/new/g;
4 print;
```

```
1 #!/bin/perl -p
2
3 $_ =~ s/old/new/g;
```

```
1$ perl -pe '$_ =~ s/old/new/g;' < infile > outfile
2$ perl -pe 's/old/new/g' < infile > outfile
3$ perl -pi -e 's/old/new/g' file
```

edit the **file***: pp. 353−354*

Edit more than one file

```
1 #!/bin/perl -pi
2 #Edit every file given to the perlscript as a command line argument.
3 #See pp. 240, 353.
4
5 s/Acme/Bongdex/g;
6 s/1993/1994/g;
```

```
1$ perlscript *.c
```

"s" calisthenics I

The following commands can also be used in **sed**, **ed**, and **vi**.

```
3
0100
00012345
123456789
```

```
s/^0*//;
```

Remove leading zeros.

```
3
100
12345
123456789
```

`s/^/000000000/;` *Right-justify the numbers with leading zeros (two steps).*

```
0000000003
000000000100
00000000012345
000000000123456789
```

`s/.*(.{9})$/$1/;` *Remove all but last 9 characters; don't need* **$**.

```
000000003
000000100
000012345
123456789
```

`s/.../$&,/g;` *Insert commas every 3 digits.*

```
000,000,003,
000,000,100,
000,012,345,
123,456,789,
```

`s/,$//;` *Remove trailing comma.*

```
000,000,003
000,000,100
000,012,345
123,456,789
```

`s/^[0,]*//;` *Remove leading zeros and commas.*

```
3
100
12,345
123,456,789
```

`s/^/$/;` *Add a leading dollar sign.*
`s/$/.00/;` *Add a trailing* **.00**

```
$3.00
$100.00
$12,345.00
$123,456,789.00
```

`s/^/ /;` *Right-justify the numbers with leading blanks (two steps).*

```
              $3.00
              $100.00
              $12,345.00
              $123,456,789.00
```

```
s/^.*(.{15})$/$1/;
```
Remove all but last 15 characters; don't need **$**.

```
         $3.00
       $100.00
     $12,345.00
$123,456,789.00
```

"s" calisthenics II

```
Garrent Leung
```

```
s/ {2,}/ /g;
```
Reduce space between words to a single blank, p. 103

```
Garrent Leung
```

```
s/(.*) (.*)/$2, $1/;
```

```
Leung, Garrent
```

```
s/ (.).*$/ $1./;
```

```
Leung, G.
```

```
s/^(.)(.*, )(.)/$3$1 $1$2$3/;
```

```
GL Leung, G.
```

```
s/^...//;
```

```
Leung, G.
```

```
s/^(.{6}).*,/$1,/;
```
Truncate the last name to at most 6 characters.

```
Leung, G.
```

```
s/,/------,/;
```
Pad the last name with **-***'s to 6 characters (two steps).*

```
Leung------, G.
```

```
s/^(.{6}).*,/$1,/;
```

```
Leung-, G.
```



```
s/./$& /g;                    Double space.
s/ $//;                       Remove trailing blank.
```

```
L e u n g - ,   G .
```

```
s/(.) /$1/g;                  Remove the double space.
```

```
Leung-, G.
```

More examples of s commands

```
1 perl -pi -e 's/\bmax\b/MAX/g' prog.c
2 perl -pi -e 's/-(\d+)/($1)/g' prog.c
```

Could you change pp. 113−114 to pp. 113−4?

Data transformation

▼ Homework 10.45: perlscript to convert Roman numerals to Arabic numerals

Insert six **s** commands into the following perlscript so that it will also translate Roman numerals that contain subtraction, i.e., **iv**, **ix**, **xl**, **xc**, **cd**, and **cm**. Insert these new commands in the right place in the list. Add the final **g** only if necessary.

```
 1 #!/bin/perl -pi
 2 #Input one roman numeral per line, and output the corresponding
 3 #arabic numeral.
 4 #For example, CCXXXI becomes 231.
 5
 6 chop $_;
 7 $_ =~ tr/a-z/A-Z/;       #p. 194
 8
 9 s/I/+1/g;
10 s/V/+5/;
11
12 s/X/+10/g;
13 s/L/+50/;
14
15 s/C/+100/g;
16 s/D/+500/;
17
18 s/M/+1000/g;
19
20 eval "print $_";
21 print "\n";
```

▲

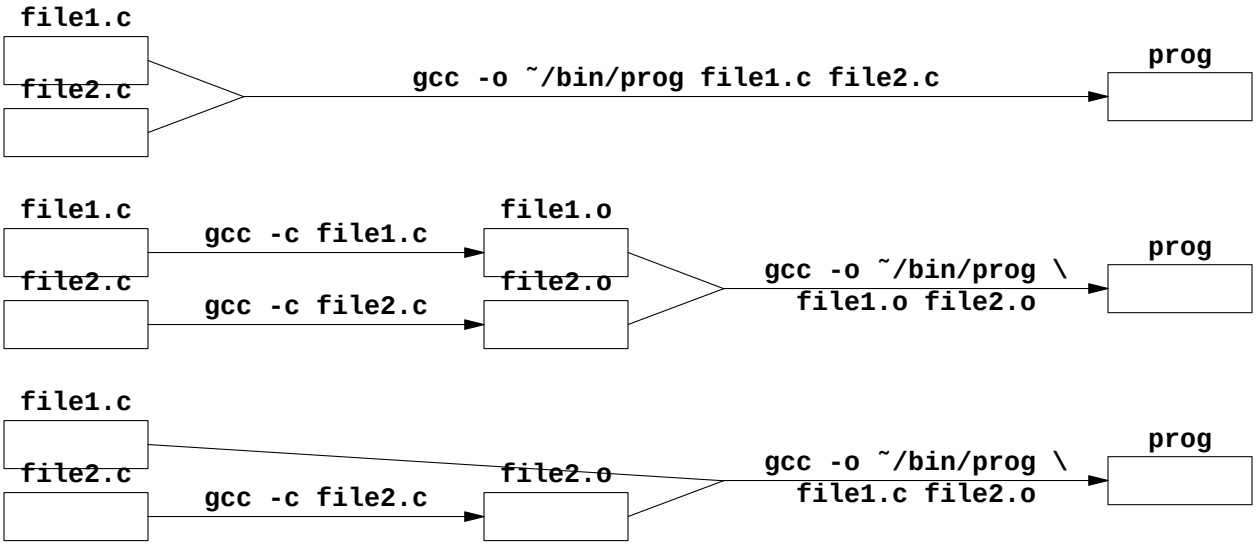
English to Pig Latin

```
#!/bin/perl -pi
#Translate English to lowercase Pig Latin: owercaselay igpay atinlay

#If the word starts with a vowel, prefix a 'w'.
s/\b([aeiou])/w$1/gi;

#Chop off first letter, move it to end, and add "ay".
s/\b([a-z])([a-z]+)/$2$1ay/gi;
```

Great Moments in Computer History: The First Fortran Compiler (1957)

```
 1 #!/bin/perl -pi
 2 #Fully parenthesize Fortran expressions
 3 #using binary + - * / ** (no unary + or -)
 4 #   *        becomes  )*(
 5 #   /        becomes  )/(
 6 #   +        becomes  ))+((
 7 #   -        becomes  ))-((
 8 #Finally, add (( to the start of the line and )) to the end.
 9 #For example,
10 #   A+B      becomes  ((A))+((B))
11 #   A+B*C    becomes  ((A))+((B)*(C))
12 #   A+B*C**D becomes  ((A))+((B)*(C**D))
13 #The algorithm adds more parentheses than are strictly necessary.
14 #It works correctly even if the input already has some parentheses.
15
16 s/\+/))+((/g;
17 s/-/))-((/g;
18
19 #Temporarily change ** to @ so that subsequent substitute commands
20 #will not mistake ** for two multiplication symbols.
21 #Use \* to search for an asterisk.
22 s/\*\*/@/g;
23
24 s/\*/)*(/g;
25 s:/:)/(:g;
26
27 #Now that multiplication has been processed, change exponentiation back to **.
28 s/@/**/g;
29
30 #Add leading (( and trailing )).
31 s/^/((/;
32 s/$/))/;
```


Output the name of every .c file that has not yet been compiled into a .o file




```
1 1$ gcc -o prog file1.c file2.c          Create executable file prog

2 2$ gcc -c file1.c                       Create object file file1.o
3 3$ gcc -c file2.c                       Create object file file2.o
4 4$ gcc -o prog file1.o file2.o          Create executable file prog

5 5$ gcc -o prog file1.c file2.o          Create executable file prog
```

```
 1 #!/bin/perl
 2 #Output the name of every .c file in the current directory that does
 3 #not have a corresponding .o file in the current directory.  See p. 206.
 4
 5 while (<*.o>) {
 6     s/o$/c/;
 7     $a{$_} = 1;
 8 }
 9
10 while (<*.c>) {
11     print "$_\n" if $a{$_} != 1;
12 }
```

Reverse the order of the two operands on each line of assembly language

```
compiler | assembler
compiler | perl -pe 's/ (.*),(.*)$/ $2,$1/' | assembler
```

before	*after*
`mov r0,r1`	`mov r1,r0`
`add r2,_max`	`add _max,r2`
`cmp a,$1000`	`cmp $1000,a`

edit a command line argument of a shellscript

Name this perlscript **chmod** and put it in your **bin** directory to create your own version of the Unix **chmod** command.

```
 1 #!/bin/perl
 2 #User-friendly version of chmod.  The user can write the 9
 3 #permission bits as the 9 characters familiar from the output of ls -l,
 4 #rather than as 3 octal digits.  Sample usage: instead of
 5 #   1$ chmod 644 filename
 6 #just say
 7 #   2$ chmod rw-r--r-- filename
 8 #or even
 9 #   3$ chmod rw-r--r filename
10
11 if (@ARGV != 2) {
12     die "$0: must have two arguments0;
13 }
14 $_ = $ARGV[0];
15
16 #If the user typed less than 9 characters, right-pad the string
17 #with dashes until it is 9 characters long.
18 s/$/---------/;
19 s/^(.{9}).*$/$1/;
20
21 #Why must zero be last?  rwxr---wx
22 s/--x/1/g;
23 s/-w-/2/g;
24 s/-wx/3/g;
25 s/r--/4/g;
26 s/r-x/5/g;
27 s/rw-/6/g;
28 s/rwx/7/g;
29 s/---/0/g;
30
31 chmod(oct($_), $ARGV[1]) || die "$0: $!";
```

▼ Homework 10.46: user friendly search for phone numbers

Write a perlscript named **spelledby2** that takes one phone number as its command line argument and outputs all the words in **/usr/dict/words** that are spelled by that number. For example,

```
1$ spelledby2 7373783
reserve
```

Make sure that there is exactly one command line argument and that it does not contain a zero or a one:

```
if (@ARGV != 1 || $ARGV[0] =~ /[01]/) {
    error message
```

Use a list of **s** commands such as

```
s/2/[ABCabc]/g
```

to transform the first argument from a seven-digit number to a seven-wildcard regular expression. There should also be an **s** command to remove dashes from the phone number; where in the list should it go for the greatest speed? Add a leading **^** and a trailing **$** to the regular expression.

If your phone number spells no words, run **spelledby2** separately on the first three digits and the last four digits, etc. No change to **spelledby2** should be necessary to run it on a number with less than

seven digits.
▲

normalize social security numbers

Haphazard social security numbers such as those in the following file (**ss.data**)

```
134-46-0561
134 46 0562
   134460563
SS Num 134-46-0564
134-46-056
134,46,0565 is my Social Security number.
```

can be fed to the following perlscript (**ss.normal**) to output them in a standard form:

```
 1 #!/bin/perl -pi
 2
 3 #Remove all characters except the digits and the newlines.
 4 s/[^0-9\n]//g;
 5
 6 #If the line does not consist of exactly 9 digits, change it
 7 #to 9 question marks.
 8 s/.*/?????????/ unless /^.{9}/;
 9
10 #Insert the 3 dashes.  Do you really need the $3?
11 s/^(...)(..)(....)/$1-$2-$3/;
```

```
1$ ss.normal < ss.data
134-46-0561
134-46-0562
134-46-0563
134-46-0564
???-??-????
134-46-0565
```

▼ **Homework 10.47: normalize phone numbers**

Suppose you had a file of hapazardly formatted phone numbers. Write a perlscript to normalize them.

(479) 934-0364	(???) 934-0364
(273)750-3644	(???) 750-3644
575-464-4227	(???) 464-4227
(750)-495-1111	(???) 495-1111
239-2222	(???) 239-2222
(212) BAD-NEWS	(212) ???-????
(17-33) 197-750-1117	(???) ???-????
(479)269-1554	(???) 269-1554
(419)069-1554	(419) ???-????

Your perlscript should consist of exactly seven **s** commands:

(1) Remove all the non-digits.

(2) If the line consists of exactly seven characters, add three leading question marks to hold the place of the area code.

(3) If the line consists of exactly three characters, add seven trailing question marks to hold the place of the phone number.

(4) If the line consists of a 1 followed by exactly ten characters, remove the leading 1.

(5) If the line now does not consist of exactly ten characters, change it to ten question marks.

(6) If the first digit of the area code is a 0, or the middle digit of the area code is neither a 1 nor a 0, change the area code to three question marks. Use one **s** command with the **unless** modifier.

(7) If the first digit of the phone number is a 0 or a 1, change the phone number to seven question marks.

(8) Add the parentheses, blank, and dash. Do you really need the **$3**?

▲

Sort playing cards in order of increasing rank

Suppose you have a file of playing cards, one per line. The first column is the rank and the second column is the suit:

```
A   S
2   C
Q   H
J   D
```

```
 1 #!/bin/perl
 2 #Sort playing cards, one per line, in order of increasing rank.
 3 #Ignore the suits.
 4 #Add "14 " to start of every line that begins with "A" or "a".
 5
 6 perl -pe '
 7     s/^/2 / if /^2/;
 8     s/^/3 / if /^3/;
 9     s/^/4 / if /^4/;
10     s/^/5 / if /^5/;
11     s/^/6 / if /^6/;
12     s/^/7 / if /^7/;
13     s/^/8 / if /^8/;
14     s/^/9 / if /^9/;
15     s/^/10 / if /^10/;
16     /s/^/11 / if /^J/i;
17     /s/^/12 / if /^Q/i;
18     /s/^/13 / if /^K/i;
19     /s/^/14 / if /^A/i;
20 ' |
21 sort -n |
22 perl -pe 's/^[^ ]* //'   #Remove everything up to and including 1st blank
```

Could you telescope the list of 14 **s** commands into a single s command by using an associative array?

```
%a = (
    'A', 14,
    'K', 13,
    'Q', 12,    #etc.
```

▼ Homework 10.48: sort whatever you want

Write a perlscript named **customsort** that sorts its lines of input into an order other than alphabetical or numerical. For example, if your input consists of one chemical element per line, sort them in order of increasing atomic number. Just do the first ten elements: **H**, **He**, **Li**, **Be**, **B**, **C**, **N**, **O**, **F**, and **Ne**.

before	*after*
H	**H**
O	**He**
He	**Li**
Li	**B**
B	**O**
O	**O**

▲

A combination of sed and awk

```
#!/bin/perl -i
#Renumber the error messages.  See pp. 174-6 for e option of s command.

$count = 1;

while (<>) {
    s/(\berror\s*\(\s*)\d+(\s*\))/$1 . $count++ . $2/ge;
    print;
}
```

▼ **Homework 10.49: add binary constants to C**

C lets you write integers in decimal, octal, and hex. Add binary:

100	*decimal*
0100	*octal*
0x100	*hex*
0b100	*binary*

Whenever it sees the regular expression **/0b[01]+/**, your perlscript should change it to the corresponding decimal number (or hex number, whichever is easier).

▲

Convert other languages to perl: pp. 377−378

Convert sed to perl

```
#!/bin/sed -f
#This file is a sedscript named little.sed.

s/Acme/Bongdex/g
s/1993/1994/g
```

```
1$ s2p little.sed > perlscript
2$ ls -l perlscript                    Make sure s2p created the perlscript.
3$ chmod a+x perlscript
4$ perlscript < infile > outfile
```

Convert awk to perl

```
#!/bin/awk -f
#This file is an awkscript named little.awk.

      {sum += $1}
END   {print sum}
```

```
1$ a2p little.awk > perlscript
2$ ls -l perlscript                    Make sure a2p created the perlscript.
3$ chmod a+x perlscript
4$ perlscript < infile > outfile
```

find commands

```
1$ find / -type d -print
2$ find / -type f -print
3$ find / -type f -print > find.out &

4$ cd
5$ find . -type d -print
6$ find . -type f -print
7$ find . -type f -name core -print
8$ find . -type f -name '*.c' -print
9$ find . -type f -user abc1234 -print
10$ find . -type f -user abc1234 -o -user def5678 -print
11$ find . -type f '!' -user abc1234 -print
12$ find . -type f -atime 14 -print
13$ find . -type f -atime +14 -print
14$ find . -type f -atime -14 -print
```

Convert find to perl

```
1$ find2perl . -type d -print > perlscript
2$ ls -l perlscript                    Make sure find2perl created the perlscript.
3$ chmod a+x perlscript
4$ perlscript > outfile
```

Convert a C .h file to a perl .ph file: p. 396

```
/* This file is named little.h. */

#define N 10
#define X 20
```

```
1$ h2ph < little.h > little.ph          Use < if you're not the superuser.
2$ ls -l little.ph
```

```
1 #This is the little.ph file created by h2ph.
2 sub N {10;}             #a subroutine that returns the value 10
3 sub X {20;}             #a subroutine that returns the value 20
```

```
1 #!/bin/perl
2 #This is a perlscript that requires the above little.ph file.
3 #"require" in perl means the same thing as "#include" in C.
4
5 require 'little.ph';    #First move little.ph to /usr/local/lib/perl directory.
6
7 print &N, "\n";         #N and X are user-defined subroutines: pp. 4, 50-53.
8 print &X, "\n";
```

Expect

The standard book on Expect is

Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs
by Don Libes; O'Reilly & Associates, 1994; ISBN 1-56592-090-2
http://www.oreilly.com/catalog/expect/
http://www.mel.nist.gov/msidstaff/libes/

The Expect home page is **http://expect.nist.gov/**

Programs that require a dialog

ftp
telnet

nslookup
mail *and the other mail readers*
nn *and the other news readers*
dbx, **gdb**, *and the other interactive debuggers*
ksh *requires the user to press control-z, control-c, etc.*
vi *and the other screen editors*

passwd
fsck

Anonymous ftp

The Tcl home page **http://sunscript.sun.com/** tells where to get Tcl via anonymous **ftp**. Ominously, the **Name** prompt has a space after its final colon, but the **Password** prompt doesn't (Libes pp. 165−166).


```
1$ cd
2$ ftp ftp.sunlabs.com
Connected to rocky.sunlabs.com.
220 rocky FTP server (Version 4.125 Fri May 16 21:40:31 PDT 1997) ready.
Name (ftp.sunlabs.com:mm64): anonymous
331 Guest login ok, send your email address as password.
Password:mark.meretzky@nyu.edu
230- Guest login ok, access restrictions apply.
230- Local time is: Sat Feb 28 09:28:09 1998
230
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd pub/tcl
250 CWD command successful.
ftp> ls -l "| more"
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 307030
-rw-rw-r--   1 19663    51          8261 Jan 23 17:07 README
-rw-r--r--   1 19663    staff      11628 Nov 26 18:29 tcl8.0p2.patch.gz
-rw-r--r--   1 19663    staff    2449543 Nov 26 18:29 tcl8.0p2.tar.Z
-rw-r--r--   1 19663    staff    1522374 Nov 26 18:29 tcl8.0p2.tar.gz
226 Transfer complete.
ftp> binary
200 Type set to I.
ftp> get tcl8.0p2.patch.gz
200 PORT command successful.
150 Opening BINARY mode data connection for tcl8.0p2.patch.gz (11628 bytes).
226 Transfer complete.
11628 bytes received in 0.42 seconds (27 Kbytes/s)
ftp> quit
221 Goodbye.
3$
```

expect and send

The following script conducts the dialog with **telnet**. An **expect** command with an argument means ‘‘wait until I receive the following string’’, or until **$timeout** seconds have passed, or until end-of-file is received from the spawned process, whichever comes first. Therefore even if **Name** is never received, you would (eventually) still progress from line 8 to line 9. We’ll fix this later.

Thanks to the terminal driver, the argument of **send** must contain **\r** instead of the usual Unix **\n**. After all, the key that a human being actually hits is **RETURN**, not **LINEFEED** (Libes pp. 79−80). As in Tcl, the double quotes are unnecessary except when they enclose whitespace.

The **expect** command receives whatever output the child wrote to its **stdout**, **stderr**, or controlling **tty**. After all, all three of these would normally appear on the user’s screen. The **send** command sends input to the child which the child can read from either its **stdin** or controlling **tty**.

See Libes pp. 217−218 for the **--** on line 1.

```
1$ which expect
/opt/sfw/bin/expect
```

```
 1 #!/opt/sfw/bin/expect --
 2 #Get the file tcl8.0p2.patch.gz via anonymous ftp.
 3
 4 cd
 5 set timeout 60             ;#default 10 seconds; -1 wait forever; 0 no wait
 6 spawn ftp ftp.sunlabs.com
 7
 8 expect "Name"              ;#no colon
 9 send "anonymous\r"         ;#carriage return
10
11 expect "Password:"         ;#no space after colon
12 send "mark.meretzky@nyu.edu\r"
13
14 expect "ftp> "
15 send "cd pub/tcl\r"
16
17 expect "ftp> "
18 send "binary\r"
19
20 expect "ftp> "
21 send "get tcl8.0p2.patch.gz\r"
22
23 expect "ftp> "
24 send "quit\r"
25
26 exit 0
```

The interact command: Libes pp. 8−9, 82−83

The **interact** command lets the user take over:

```
 1 #!/opt/sfw/bin/expect --
 2
 3 cd
 4 set timeout 60
 5 spawn ftp ftp.sunlabs.com
 6
 7 expect "Name"
 8 send "anonymous\r"
 9
10 expect "Password:"
11 send "mark.meretzky@nyu.edu\r"
12
13 expect "ftp> "
14 send "cd pub/tcl\r"
15
16 expect "ftp> "
17 send "ls -l \"| more \"\r"
18
19 interact                ;#Put terminal in raw mode, Libes pp. 324, 344.
```

To save the user from having to watch the whole dialog, surround part of the program with a pair of **log_user** commands. The **send_user** command is the same as the Tcl command **puts**, except that **puts** automatically appends a **\n**. (And **send_error** is like **puts stderr**, Libes pp. 187−188).

```
 1 #!/opt/sfw/bin/expect --
 2
 3 cd
 4 send_user "Connecting to ftp.sunlabs.com...\n"    ;#newline
 5 log_user 0
 6 spawn ftp ftp.sunlabs.com
 7
 8 expect "Name"
 9 send "anonymous\r"
10
11 expect "Password:"
12 send "mark.meretzky@nyu.edu\r"
13
14 expect "ftp> "
15 send "cd pub/tcl\r"
16 log_user 1
17
18 interact
```

Carriage return vs. newline: Libes pp. 72, 78−79

```
send_user "hello\n"     ;#just like printf("hello\n"); in a C program
expect_user "hello\n"

send "hello\r"          ;#makes the spawned process receive hello\n
expect "hello\r\n"      ;#from a spawned process that sent hello\n
```

Verify the above send.

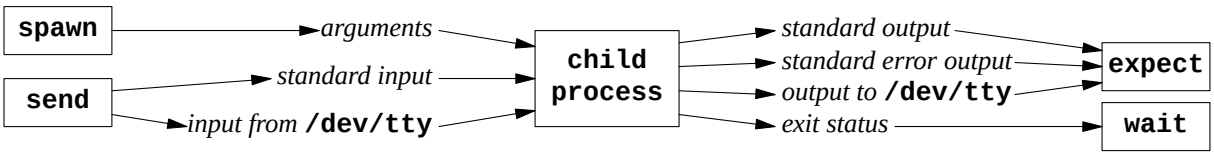
The **`expect`** command with no arguments (Libes pp. 98, 104) reads all of the child's standard output and standard error output and copies it to the standard output of the Expect script. The command terminates when **`eof`** is encountered or when **`$timeout`** seconds have elapsed, whichever comes first. Without the **`expect`** command, you would see nothing.

```
1 #!/opt/sfw/bin/expect --
2
3 spawn od -Ad -cvw1      ;#octal dump the characters, addresses in decimal
4 send "hello\r"
5 expect                  ;#and wait 10 seconds for timeout
6 exit 0
```

The output is

```
spawn od -Ad -cvw1
hello
0000000    h
0000001    e
0000002    l
0000003    l
0000004    o
0000005   \n
```


The I/O redirections performed by Expect: Libes pp. 174−175, 210−211, 291−293



The **spawn** command connects the child process's standard input, standard output, and standard error to a pseudoterminal. To change these defaults, spawn a shell that will overlay itself with the shell **exec** command (Libes pp. 291−293):

```
spawn /bin/sh -c "exec cal 13 1998 2> errorfile"
```

The data directed from a child process into the **expect** command is copied to the standard output of the Expect script, unless you say **log_user 0**.

Please email me if you can run ftp interactively from a perlscript.

Modify the following example so that it drives **ftp** instead of **bc**. Does **ftp** write to its standard output or its **/dev/tty** (or both)? Does **ftp** read from its standard input or its **/dev/tty** (or both)? Will **ftp** flush its output buffer when writing to a pipe, or will it flush only when writing output to a terminal? Can the perlscript direct **ftp**'s output to a pseudoterminal? Does the Perl standard library contain subroutines to do all this? For a discussion of the problems surrounding **Open2** in Perl, see pp. 344−345, 455−457 in the O'Reilly *Programming Perl, 2nd ed.*

Can the perlscript read the **ftp** prompts with the **<>** operator? Do you have to change the input record separator variable **$/**? Or should you read the prompts with the **getc** function?

```
 1 #!/bin/perl
 2 use IPC::Open2;
 3 use FileHandle;
 4
 5 $pid = open2(\*IN, \*OUT, 'bc');
 6 defined $pid || die "$0: $!";
 7 autoflush OUT 1;        #flush automatically after each print OUT
 8
 9 print OUT "1+2\n";
10 $line = <IN>;
11 print $line;
12
13 close IN;
14 close OUT;
15 exit 0;
```

The standard output of the perlscript is

```
3
```

```
 1 #!/opt/sfw/bin/expect --
 2
 3 spawn bc
 4 send "1+2\r"
 5
 6 expect "\n"            ;#As soon as we receive one line of standard output from bc,
 7 close                  ;#send an eof to bc.
 8 exit 0
```


A process can tell the difference

A Unix process can tell if its standard output has been directed to a terminal:

```
if (isatty(1)) {          /* C & C++; and #include <unistd.h> */
if [ -t 1 ]; then         #Bourne shell
if [[ -t 1 ]]; then       #Korn shell
if (-t 1) {               #Perl
if {$tcl_interactive} {   ;#Tcl, Tk, Expect
```

A child spawned by **exec** believes that its standard output has not been redirected to a terminal.

```
1 #!/usr/local/bin/tclsh
2 #Make ls believe its standard output is not directed to a terminal.
3
4 puts [exec ls]
5 exit 0
```

```
curly                                                    single-column output
larry
moe
```

But a child spawned by **spawn** believes that its standard output has been redirected to a terminal.

```
1 #!/opt/sfw/bin/expect --
2 #Make ls believe its standard output is directed to a terminal.
3 spawn ls
4
5 expect
6 exit 0
```

```
curly     larry     moe                                  multi-column output
```

Deliberately bug-prone example

What will this C program print?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 main()
5 {
6     printf ("hello\n");
7     _exit(0);                       /* underscore exit */
8 }
```

```
1$ cc prog.c
2$ a.out              Direct a.out's standard output to screen; it outputs hello.
3$ a.out | cat        Direct a.out's standard output to pipe; it outputs nothing.
```

Expect with a pattern-action pair: Libes p. 75

Let's verify the above claim that a process that outputs **hello\n** will cause an **expect** command to receive **hello\r\n**:


```
 1 #!/opt/sfw/bin/expect --
 2
 3 spawn echo hello
 4
 5 expect "hello\r\n"
 6 send_user "I received hello CR LF.\n"
 7 exit 0
```

Even if the pattern **`hello\r\n`** is never seen, the script will still print the message. That's because an **`expect`** command with a pattern will terminate when the pattern is first seen, or when **`eof`** is encountered, or when **`$timeout`** seconds have elapsed, whichever comes first.

To print the message only if the pattern is seen, give **`expect`** a pattern-action pair:

```
 1 #!/opt/sfw/bin/expect --
 2
 3 spawn echo hello
 4
 5 expect "hello\r\n" {send_user "I received hello CR LF.\n"}
 6 exit 0
```

Syntax for multiple pattern-action pairs: Libes pp. 75−77, 158−160

(1) You're allowed to omit the action of the last pattern-action pair. Therefore it's conventional to list the pattern with the biggest action last, and the patterns with the smallest actions (often error conditions) first. Other than that, it doesn't matter what order you list them in.

```
 1 #Fragment of an expect script that spawns ftp.
 2
 3 expect "ftp> "
 4 send "cd $dirname\r"
 5
 6 expect full_buffer                   {exit 5} \
 7     timeout                          {exit 4} \
 8     eof                              {exit 4} \
 9     "No such file or directory.\r\n" {exit 3} \
10     "Not a directory.\r\n"           {exit 2} \
11     "Permission denied.\r\n"         {exit 1} \
12     "CWD command successful.\r\n"
13
14 #Action for "CWD command was successful.\r\n" goes
15 #here.
```

(2) The pattern **`default`** means **`timeout`** or **`eof`**, whichever happens first (Libes p. 101):

```
 1 expect "ftp> "
 2 send "cd $dirname\r"
 3
 4 expect full_buffer                    {exit 5} \
 5     default                           {exit 4} \
 6     "No such file or directory.\r\n" {exit 3} \
 7     "Not a directory.\r\n"            {exit 2} \
 8     "Permission denied.\r\n"          {exit 1} \
 9     "CWD command successful.\r\n"
10
11 #Action for "CWD command was successful.\r\n" goes
12 #here.
```

(3) Eliminate the backslashes as in the Tcl **`if-elseif`** command:

```
 1 expect full_buffer {
 2         exit 5
 3     } default {
 4         exit 4
 5     } "No such file or directory.\r\n" {
 6         exit 3
 7     } "Not a directory.\r\n" {
 8         exit 2
 9     } "Permission denied.\r\n" {
10         exit 1
11     } "CWD command successful.\r\n"
12
13 #Action for "CWD command was successful.\r\n" goes
14 #here.
```

(4) You can also give the **`expect`** command exactly one argument, of a list of the above arguments. I simply surrounded them with a pair of curly braces:

```
 1 expect {
 2     full_buffer {
 3         exit 5
 4     }
 5
 6     default {
 7         exit 4
 8     }
 9
10     "No such file or directory.\r\n" {
11         exit 3
12     }
13
14     "Not a directory.\r\n" {
15         exit 2
16     }
17
18     "Permission denied.\r\n" {
19         exit 1
20     }
21
22     "CWD command successful.\r\n"
23 }
24
25 #Action for "CWD command was successful.\r\n" goes
26 #here.
```

expect_before and expect_after: Libes pp. 101, 259−268

A conscientious programmer should write a pattern for **`eof`**, **`timeout`**, and **`full_buffer`** in every **`expect`** command. You can avoid this repetition with the **`expect_before`** and **`expect_after`** commands.

Control structure in Expect

`1$` is my shell prompt, and **`?`** is my **`mail`** prompt:

```
2$ mailx
Mail $Revision: 4.2.4.2 $  Type ? for help.
"/usr/spool/mail/mm64": 4 messages 3 unread
    1 abc1234  Sun May 10 10:30  13/293 "Dodsworth"
>U  2 def5678  Sun May 10 11:30  12/290 "help"
 U  3 ghi9012  Sun May 10 11:45  12/288 "Olaf Stapledon"
 U  4 abc1234  Sun May 10 11:46  12/289 "help"
? R2
To: def5678
Subject: Re:  help

~r /home1/m/mm64/helpfile
"/home1/m/mm64/helpfile" 1/29
.
EOT
? d2
? R4
To: abc1234
Subject: Re:  help

~r /home1/m/mm64/helpfile
"/home1/m/mm64/helpfile" 1/29
.
EOT
? d4
? q
3$
```

Since the **mail** prompt **?** is a special character to the **expect** commands in lines 19, 52, and 55, it must be preceded by a ****. And since the **** is a special character in Tcl, it too must be preceded by a ****. See Libes, pp. 73, 121, for the **$** anchor when looking for a prompt.

The array element **$expect_out(buffer)** in line 24 contains all the characters that were received by the most recent successful **expect**—one line of text, ending with **\r\n**.

At line 35, the variable **$letters** contains a list of four elements, each of which is a list of two elements:

```
{1 Dodsworth} {2 help} {3 {Olaf Stapledon}} {4 help}
```

```
 1 #!/opt/sfw/bin/expect --
 2 #Mail a helpfile to everyone whose subject was "help"
 3
 4 spawn mailx
 5
 6 expect {
 7     "No mail for mm64\r\n" {
 8         exit 0
 9     }
10
11     "Type \\? for help.\r\n"
12 }
13 expect "\r\n"                                ;#4 messages, 3 unread
14
15 #Loop until we encounter the ? prompt.
16 set letters {}
```



```
17 while {1} {
18     expect {
19         "\\? $"  {
20             break
21         }
22
23         "\r\n" {
24             set line $expect_out(buffer)
25             set llen [llength $line]
26
27             set subject [lindex $line [expr $llen - 1]]
28             set n       [lindex $line [expr $llen - 8]]
29
30             set letter [list $n $subject]
31             lappend letters $letter
32         }
33     }
34 }
35
36 foreach letter $letters {
37     set subject [lindex $letter 1]
38     if {[string compare $subject "help"] == 0} {
39         set n [lindex $letter 0]
40
41         send "R$n\r"                          ;#Reply
42         expect "\r\n"
43         expect "\r\n"
44         expect "\r\n"
45
46         send "~r /home/m/mm64/helpfile\r"    ;#read file
47         expect "\r\n"
48
49         send ".\r"
50         expect "EOT\r\n"
51
52         expect "\\? $"
53         send "d$n\r"                          ;#delete letter
54
55         expect "\\? $"
56     }
57 }
58
59 send "q\r"                                    ;#quit from mail
60 exit 0
```

The $expect_out array:
Libes pp. 73, 96, 253

Each **`expect`** command takes all the input received since the previous **`expect`** and stores it into **`$expect_out(buffer)`**, replacing the previous contents of this array element. This input includes (and ends with) the characters that were matched by the **`expect`**'s pattern. If the **`expect`** had no pattern, all the input until end-of-file or timeout is copied into **`$expect_out(buffer)`**.

Each **`expect`** also takes only the characters that were matched by the pattern and stores them into **`$expect_out(0,string)`**, replacing the previous contents of this array element. Put no space

alongside the comma.

```
 1 #!/usr/local/bin/expect --
 2
 3 log_user 0
 4 set timeout 60
 5 spawn ftp ftp.sunlabs.com
 6 expect "\r\n"
 7
 8 puts "The entire line (including the CRLF) is \"$expect_out(buffer)\"."
 9 puts "The length of \$expect_out(0,string) is [string length $expect_out(0,string)]
10
11 close
12 log_user 1

   The entire line (including the CRLF) is "Connected to rocky.sunlabs.com.
   ".
   The length of $expect_out(0,string) is 2.
```

Two simple examples of the timeout pattern

Don't wait forever for user input:
Libes pp. 77−78, 112−113

```
#!/bin/sh

echo Please say yes or no.  Defaults to null string after 10 seconds.
answer=‘timed-read‘
```

```
 1 #!/usr/local/bin/expect --
 2 #This script is named timed-read.
 3
 4 expect "\n" {send $expect_out(buffer)}
```

The deluxe version is

```
#!/bin/sh

echo Please say yes or no.  Defaults to no after 60 seconds.
answer=‘timed-read 60 no‘
```



```
 1 #!/usr/local/bin/expect --
 2 #This script is named timed-read.
 3
 4 if {$argc != 2} {
 5     puts stderr "usage: $argv0 timeout default"
 6     exit 2
 7 }
 8
 9 set timeout [lindex $argv 0]
10
11 expect {
12     default {
13         send [lindex $argv 1]
14         exit 1
15     }
16
17     "\n" {
18         send [string trimright $expect_out(buffer) "\n"]
19         exit 0
20     }
21 }
```

Let a non-interactive process run for at most 1 minute:
Libes pp. 98−100

What would go wrong without the **`eval`** in line 7?

If you don't need the messages and exit status, you can simply give the **`expect`** command no arguments. What would go wrong with no **`expect`** command at all after the **`spawn`**?

```
1$ maxtime 60 prog arg1 arg2 arg3
```

```
 1 #!/usr/local/bin/expect --
 2 #This script is named maxtime.
 3
 4 set timeout  [lindex $argv 0]
 5 set progname [lindex $argv 1]
 6
 7 eval spawn [lrange $argv 1 end]
 8
 9 expect {
10     timeout {
11         puts "$progname timed out after $timeout seconds."
12         exit 1
13     }
14
15     eof {
16         puts "$progname finished in less than $timeout seconds."
17         exit 0
18     }
19 }
```


Four notations for patterns

My **expect** interpreter dumped core when I wrote the **-nocase** and **-re** options together.

(1) Exact match (Libes pp. 134−135):

```
expect -ex "a*b"     ;#look for one asterisk between a and b
```

(2) Case-insensitive match (Libes p. 139):

```
expect -nocase "a*b" ;#look for one asterisk between a and b, in either case
```

(3) Glob patterns, as in the shell (Libes pp. 87−94):

```
1$ rm a*b
expect -gl "a*b"     ;#look for anything (or nothing) between a and b
expect "a*b"         ;#-gl is the default
expect -gl "-gl"     ;#look for minus lowercase gl
```

(4) Regular expressions (Libes pp. 107−127):

```
2$ grep 'a.*b'
expect -re "a.*b"    ;#look for anything (or nothing) between a and b
```

Backslashes in Tcl

What would go wrong without the **{**curly braces**}** and backslashes?

```
1 puts {Here are [square brackets].}
2 puts "Here are \[square brackets\]."
3 puts "Here are \[square brackets]."
```

The output is

```
Here are [square brackets].
Here are [square brackets].
Here are [square brackets].
```

The above examples suggest that curly braces are simplest, but curly braces make it impossible to put special characters such as **\t** and **\n** into the string:

```
1 puts {Here\tare [square brackets].}
2 puts "Here\tare \[square brackets\]."
3 puts "Here\tare \[square brackets]."
```

The output is

```
Here\tare [square brackets].
Here    are [square brackets].
Here    are [square brackets].
```


Backslashes in glob patterns and regular expressions:
Libes p. 91

The following command expects a line ending with an uppercase letter from a previously spawned child. The last two characters of the argument of the command are carriage return and newline:

```
expect "\[A-Z]\r\n"                    ;#good
```

The following command expects an uppercase letter followed by the four characters ‘‘backslash lowercase r backslash lowercase n’’. Those four characters are the last four characters of the argument.

```
expect {[A-Z]\r\n}                     ;#bad
```

Look for one backslash:
Libes pp. 91−94

You must write a double backslash to give a single backslash to a command:

```
puts "\\"              ;#Output one backslash (and one newline).
```

You must therefore write a quadruple backslash to give a double backslash to a command:

```
expect -gl "\\\\"     {send "I received one backslash."}
expect -re "\\\\"     {send "I received one backslash."}
expect -ex "\\"       {send "I received one backslash."}

expect -re "\\"       error from the expect command: a single backslash is not a legal regular expression
expect -re "\\\"      lexical error from the expect interpreter: missing closing quote
```

Tagged regular expressions:
Libes pp. 111−112, 115−116, 124−125

```
1$ date
Wed May  6 10:21:50 EDT 1998
```

```
 1 #!/usr/local/bin/expect --
 2
 3 log_user 0
 4 spawn date
 5 expect -re "(...) (...) (..) (..:..:..) (...) (....)\r\n"
 6
 7 puts "\$expect_out(buffer)   == \"$expect_out(buffer)\""
 8 puts "\$expect_out(0,string) == \"$expect_out(0,string)\""
 9
10 puts "\$expect_out(1,string) == \"$expect_out(1,string)\""
11 puts "\$expect_out(2,string) == \"$expect_out(2,string)\""
12 puts "\$expect_out(3,string) == \"$expect_out(3,string)\""
13 puts "\$expect_out(4,string) == \"$expect_out(4,string)\""
14 puts "\$expect_out(5,string) == \"$expect_out(5,string)\""
15 puts "\$expect_out(6,string) == \"$expect_out(6,string)\""
16
17 log_user 1
```

```
$expect_out(buffer)   == "Wed May  6 10:15:09 EDT 1998
"
$expect_out(0,string) == "Wed May  6 10:15:09 EDT 1998
"
$expect_out(1,string) == "Wed"
$expect_out(2,string) == "May"
$expect_out(3,string) == " 6"
$expect_out(4,string) == "10:15:09"
$expect_out(5,string) == "EDT"
$expect_out(6,string) == "1998"
```

To avoid storing the leading blank into **`$expect_out(3,string)`** when the date is a single digit,

```
expect -re "(...) (...) +(\[0-9]+) (..:..:..) (...) (....)\r\n"
```

Expect one line and remove the trailing \r\n:
Libes pp. 112−113, 135−136, 145−147

```
 1 #!/usr/local/bin/expect --
 2
 3 log_user 0
 4 set timeout 60
 5 spawn ftp ftp.sunlabs.com
 6
 7 expect -re "\r\n"
 8 puts "\$expect_out(buffer) == \"$expect_out(buffer)\""
 9
10 set line [string range $expect_out(buffer) 0 [
11     expr [string length $expect_out(buffer)] - 3
12 ]]
13 puts "\$line == \"$line\""
14
15 close
16 log_user 1

$expect_out(buffer) == "Connected to rocky.sunlabs.com.
"
$line == "Connected to rocky.sunlabs.com."
```

If you don't like the placement of the **[**square brackets**]** in lines 10−12, you could write

```
set line [string range $expect_out(buffer) 0 \
    [expr [string length $expect_out(buffer)] - 3]]
```

with a backslash.

The most common use of a tagged regular expression is to remove the **`\r\n`** from the end of each line of input. What would go wrong if we changed the wildcard **`[^\r]`** in line 7 to just a dot?


```
 1 #!/usr/local/bin/expect --
 2
 3 log_user 0
 4 set timeout 60
 5 spawn ftp ftp.sunlabs.com
 6
 7 expect -re "(\[^\r]*)\r\n"
 8 puts "\$expect_out(buffer)   == \"$expect_out(buffer)\""
 9 puts "\$expect_out(0,string) == \"$expect_out(0,string)\""
10 puts "\$expect_out(1,string) == \"$expect_out(1,string)\""
11
12 close
13 log_user 1
```

```
$expect_out(buffer)   == "Connected to rocky.sunlabs.com.
"
$expect_out(0,string) == "Connected to rocky.sunlabs.com.
"
$expect_out(1,string) == "Connected to rocky.sunlabs.com."
```

The order in which the patterns are tried:
Libes p. 190

The patterns are tried in the order in which you write them. For example

```
 1 #!/usr/local/bin/expect --
 2
 3 log_user 0
 4 spawn echo ba
 5
 6 expect {
 7     "a" {puts "Found a."}
 8     "b" {puts "Found b."}
 9 }
10
11 puts "\$expect_out(buffer)   == \"$expect_out(buffer)\""
12 puts "\$expect_out(0,string) == \"$expect_out(0,string)\""
13
14 log_user 1
```

The output shows that the pattern in the above line 8 was never given a chance to match, even though it appeared in the input earlier than the pattern in line 7:

```
Found a.
$expect_out(buffer)   == "ba"
$expect_out(0,string) == "a"
```

To force the patterns to match in the order in which they appear in the input, anchor them with a leading caret (Libes pp. 73−74):


```
 1 #!/usr/local/bin/expect --
 2
 3 log_user 0
 4 spawn echo ba
 5
 6 expect {
 7     "^a" {puts "Found a."}
 8     "^b" {puts "Found b."}
 9 }
10
11 puts "\$expect_out(buffer)   == \"$expect_out(buffer)\""
12 puts "\$expect_out(0,string) == \"$expect_out(0,string)\""
13
14 log_user 1
```

Now the output is

```
Found b.
$expect_out(buffer)   == "b"
$expect_out(0,string) == "b"
```

This caret does not mean ‘‘beginning of line’’ as in **grep**. It means ‘‘beginning of the input that has not yet been read into **$expect_out(buffer)** by a previous successful match’’. For example, the pattern in the following line 12 matches even though the **b** is not at the start of an input line.

```
 1 #!/usr/local/bin/expect --
 2
 3 log_user 0
 4 spawn echo ab
 5
 6 expect "a" {
 7     puts "Found a."
 8     puts "\$expect_out(buffer)   == \"$expect_out(buffer)\""
 9     puts "\$expect_out(0,string) == \"$expect_out(0,string)\""
10 }
11
12 expect "^b" {
13     puts "Found b."
14     puts "\$expect_out(buffer)   == \"$expect_out(buffer)\""
15     puts "\$expect_out(0,string) == \"$expect_out(0,string)\""
16 }
17
18 log_user 1
```

The output is

```
Found a.
$expect_out(buffer)   == "a"
$expect_out(0,string) == "a"
Found b.
$expect_out(buffer)   == "b"
$expect_out(0,string) == "b"
```

Recommendation: write the most specific pattern first.
Libes pp. 189−191

Since the patterns are tried in the order in which you write them, you can simplify the later patterns by writing the most specific ones first.

For example, **`ftp`** responses begin with three-digit numbers. Numbers in the two hundreds indicate success; other numbers indicate failure:

```
ftp> cd pub
250 CWD command successful.
ftp> cd pubb
550 pubb: No such file or directory.
ftp> quit
221 Goodbye.
```

Assume that you've been expecting the input one line at a time. Instead of expecting a response line this way,

```
1 expect {
2     -re "^(\[^2]\[^\r]*)\r\n" {puts "good"}
3     -re     "^(2\[^\r]*)\r\n" {puts "bad"}
4 }
```

you should write the most specific pattern first:

```
1 expect {
2     -re     "^(2\[^\r]*)\r\n" {puts "bad"}
3     -re      "^(\[^\r]*)\r\n" {puts "good"}
4 }
```

The most specific patterns will usually be those for errors, as above. And the actions associated with errors will usually be the shortest:

```
1 expect {
2     "No such file or directory.\r\n" {exit 1}
3     "Permission denied.\r\n"         {exit 2}
4     "\r\n" {
5         do lots
6         and lots
7         and lots of stuff
8     }
9 }
```

Since you can omit the action of the last pattern-action pair (Libes p. 94), put the error patterns first and the normal pattern last to avoid distending the **`expect`** command:

```
1 expect {
2     "No such file or directory.\r\n" {exit 1}
3     "Permission denied.\r\n"         {exit 2}
4     "\r\n"
5 }
6
7 do lots
8 and lots
9 and lots of stuff
```


Silly example of tagged regular expressions

Suppose the user types the following line of words:

```
Happy families are all alike.
```

During the first iteration of the **while** loop, the regular expression in line 14 will match the word **Happy**. During the second iteration, the regular expression in line 10 will match the space after the word **Happy**. During the third iteration, the regular expression in line 14 will match the word **families**. During the second iteration, the regular expression in line 10 will match the space after the word **families**.

But without the caret in line 10, the regular expression in line 10 would match the space after the word **Happy** during the first iteration. In this case, line 10 would set **$expect_out(buffer)** set to **"Happy "**, and would set **$expect_out(0,string)** to **" "**. The word **Happy** would never be matched by the regular expression in line 14.

```
 1 #!/usr/local/bin/expect --
 2 set timeout 60
 3
 4 while {1} {
 5     expect_user {
 6         eof {
 7             break
 8         }
 9
10         -re "^\[^A-Za-z]+" {
11             send_user $expect_out(buffer)
12         }
13
14         -re "^(\[A-Za-z])(\[A-Za-z]*)" {
15             send_user "$expect_out(2,string)$expect_out(1,string)ay"
16         }
17     }
18 }
```

```
appyHay amiliesfay reaay llaay likeaay.
```

A case where Tcl doesn't need the backslash before the dollar sign

You don't need the backslash if the character after the dollar sign is neither a **{**left curly brace nor a character that could start a variable name.

```
puts "\$hello"    ;#Output a dollar sign followed by the word "hello".
puts "hello$"     ;#Output the word "hello" followed by a dollar sign.
```

I forgot my password

It would still work without the **"Login incorrect"** in lines 13−15, but you'd have to wait for the **expect** in line 12 to time out. And without the **wait** in line 14 the zombie children would persist longer (Libes pp. 105−106).

We can speed it up by trying more then one password per **telnet**, until we receive a **Connection closed by foreign host.**

```
1 #!/usr/local/bin/expect --
2
3 set passwords {"bond" "Bond" "bondj" "jbond" "misterbond" "james" "007"}
4
5 set timeout 60
```



```
 6 log_user 0
 7 foreach password $passwords {
 8     spawn telnet hostname.nyu.edu
 9     expect -re "login: $"    {send "loginname\r"}
10     expect -re "Password: $" {send "$password\r"}
11
12     expect {
13         "Login incorrect" {
14             close; wait
15
16             #close
17             #puts "The spawned child's PID number was [lindex [wait] 0]"
18         }
19
20         "Last login:" {
21             puts $password
22             exit 0
23         }
24     }
25 }
26
27 puts "No password worked."
28 exit 1
```

On which hosts is a program running?
Libes p. 122

```
 1 #!/usr/local/bin/expect --
 2
 3 #hostname, loginname, password
 4 set hosts {
 5     {"hostname0.nyu.edu" "loginname0" "password0"}
 6     {"hostname1.nyu.edu" "loginname1" "password1"}
 7     {"hostname2.nyu.edu" "loginname2" "password2"}
 8 }
 9
10 log_user 0
11
12 foreach host $hosts {
13     set hostname [lindex $host 0]
14     spawn telnet $hostname
15     expect -re "login: $"     {send "[lindex $host 1]\r"}
16     expect -re "Password: $"  {send "[lindex $host 2]\r"}
17
18     expect -re "\[%#:\$] $" {send \
19         "ps -al | awk 'NR >= 2 && \$NF == \"prog\"' | wc -l | tr -d ' '\r"
20     }
21     expect "\r\n"    ;#The ps line gets echoed back.
22     expect {
23         "^0\r\n"      {}
24         -re "\[1-9]" {puts $hostname}
25     }
26
27     close; wait
```

```
28 }
29
30 log_user 1
```

There's often a pause before an incorrect password is rejected. Therefore we could speed up the above loop **spawn**'ing many **telnet**'s simultaneously

Spawn id's:
Libes pp. 233−234

The **spawn** command gives a value to the variable **spawn_id**.

```
 1 spawn prog0
 2 set spawn_id0 $spawn_id
 3
 4 spawn prog1
 5 set spawn_id1 $spawn_id
 6
 7 #Go back and expect and send with prog0:
 8 set spawn_id $spawn_id0
 9 expect blah blah blah
10 send blah blah blah
11
12 #Now expect and send with prog1:
13 set spawn_id $spawn_id1
14 expect blah blah blah
15 send blah blah blah
```

Poll many children simultaneously:
Libes pp. 237−238

```
 1 #!/usr/local/bin/expect --
 2
 3 set children {"prog0" "prog1" "prog2" "prog3" "prog4"}
 4 set spawn_ids {}
 5
 6 foreach child $children {
 7     spawn $child
 8     lappend spawn_ids $spawn_id
 9 }
10
11 set timeout 0    ;#Don't let the expect command wait.
12
13 while {1} {
14     foreach spawn_id $spawn_ids {
15         expect -re "whatever" {send "whatever"}
16     }
17 }
```

If different children need different patterns and actions, give the **-i** option to the **expect** command in Libes pp. 247−248.

Sources and destinations

expect	*read from* **spawn***'ed process: Libes pp. 71−72*
send	*write to* **spawn***'ed process: Libes pp. 72−73*

expect_user	*read from standard input: Libes pp. 192−193, 209*
send_user	*write to standard output: Libes pp. 185−187, 209*
send_error	*write to standard error output: Libes pp. 187−192*
expect_tty	*read from* **/dev/tty***: Libes p. 210*
send_tty	*write to* **/dev/tty***: Libes pp. 210−211*

You can even **expect** from and **send** to a file (regular file, terminal, named pipe, etc.): Libes pp. 289−290.

Libes p. 188: the difference between

```
puts "hello"
send_user "hello"
```

(1) **puts** is buffered; **send_user** is unbuffered.

(2) **puts** outputs a trailing newline; **send_user** doesn't.

(3) The output of **send_user** is saved by the **log_file** command (Libes pp. 180−182); the output of **puts** isn't.

Let the script regain control after an interact

Suppose I type

```
One degree is <3.14159/180>
```

in **vi**'s input mode. As soon as I hit the **>**, the line changes to

```
One degree is .01745
```

Since I'm still in input mode, I continue typing

```
One degree is .01745 radians.
```

The **<3.14159/180>** was never input into the file being edited by **vi** (Libes p. 324): it was merely echoed onto the screen by the **-echo** in line 12 (Libes pp. 333−335). I wiped it off the screen by sending a control-L to **vi** in line 14. Of course, you must get out of **vi**'s input mode (line 13) before sending it a control-L.

Another design would have been to have the **<** send a colon to **vi** to move the cursor to the bottom of the screen when typing the input for **bc**.

See Libes pp. 99−100 for the **eval** in line 9.


```
 1 #!/usr/local/bin/expect --
 2
 3 log_user 0
 4 spawn bc
 5 set spawn_id_bc $spawn_id
 6 send "scale = 5\r"
 7 expect "\r\n"
 8
 9 eval spawn vi [lrange $argv 1 end]
10 set spawn_id_vi $spawn_id
11
12 interact -echo -re "<(\[^>]*)>" {
13     send "\033"      ;#ESC to get out of input mode for the control-L.
14     send "\014"      ;#Control-L to refresh the screen
15     send "a"         ;#Go back into input mode.
16
17     set spawn_id $spawn_id_bc
18     send "$interact_out(1,string)\r"
19     expect "\r\n"
20     expect -re "(\[^\r]*)\r\n"
21
22     set spawn_id $spawn_id_vi
23     send "$expect_out(1,string)"
24 }
25
26 log_user 1
```

anonymous ftp

```
1$ whois uwm.edu | more
```
point your browser at **`http://www.uwm.edu`**

Let's use the ''file transfer program'' to get a copy Scott Yanoff's list of interesting Internet resources from his host **`ftp.csd.uwm.edu`** at the Computing Services Division of the University of Wisconsin at Milwaukee. (His home page is **`http://www.cs.uwm.edu/public/yanoff`**). It's a file named **`inet.services.txt`** in the directory **`/pub`**.

```
2$ cd                                   Go to a directory in which you can create a file.
3$ ftp ftp.csd.uwm.edu
Name (ftp.csd.uwm.edu:abc1234): anonymous          Be anonymous.
331 Guest login ok, send ident as password.
Password: abc1234@acf4.nyu.edu          Type your network address.

ftp> pwd                        where are you on ftp.csd.uwm.edu
ftp> cd /pub
ftp> pwd

ftp> ls                         short listing
ftp> dir                        long listing, like ls -l
ftp> dir . "| more"             long listing, like ls -l | more
ftp> dir . listing              Copy long listing into file on acf4.
output to local-file: listing? y
```



```
ftp> control-z                  Stop ftp.
4$ more listing                 Peruse the listing; press space bar or q.
5$ fg                           Start ftp.

ftp> ascii                      ascii before getting text file, binary before binary file
ftp> get inet.services.txt
ftp> quit

6$ ls -l listing inet.services.txt
7$ pr -l60 inet.services.txt | lpr          minus lowercase L sixty
```

The World Wide Web interface to ftp

Instead of giving the **telnet** command, simply point your web browser (lynx, mosaic, netscape, etc.) at the URL

ftp://ftp.csd.uwm.edu/pub/inet.services.txt

▼ Homework 10.50: copy a file to acf4 by anonymous ftp (not to be handed in)

Get the Yanoff list for yourself.

▲

□