Unix essentials (and AFNI)

Section 1: Unix essentials

Introduction

Unix is an operating system, as are Windows and Mac OS X. Since Mac OS X, Apple computers are actually running Unix operating systems, but with the Mac OS X graphical user interface on the front end.

When we access Terminal or XQuartz, we are using the command line interface for Unix, similarly to accessing the command prompt in Windows. This allows us to interact with the computer in order to perform many of the same tasks that we can do using the graphical interface (for example, creating/moving/deleting files), but much more powerfully and quickly.

There are a few flavours of command lines for Unix, such as sh, bash, ksh, zsh, and csh. The interface, or shell, that we use here is tcsh (Cyan pronounces it 'tsh' when she's being lazy). The main difference between the shells is the syntax of the different commands, and the syntax used when writing scripts. There are, at minimum, hundreds of commands on most systems.

Basic syntax

N.B. Shell commands are case-sensitive.

Each shell command calls a program. These programs may be built into the operating system (eg. `pwd`, `cd`) or created/installed by a user (eg. shell scripts).

Some of these programs have required arguments (input to the program). For instance, `mkdir` requires a name in order to create a new directory with that name.

Some of these programs have optional arguments allowing the user to run the program in a different manner. For instance, `ls` can be run with the optional argument `-a`, which causes it to list all files in a directory, including files whose name starts with '.' (which are normally hidden and so not listed by `ls`). In many cases, multiple switches may be combined into one. For instance, `ls` can be run with the optional arguments `-a` and `-l` as `ls -a -l`, `ls -l -a`, `ls -al`, or `ls -la`.

The syntax necessary to run many programs from the command-line can be found in their documentation. For most programs built into the shell, the documentation can be viewed using the `man` (*manual)* command (eg. `man ls`).

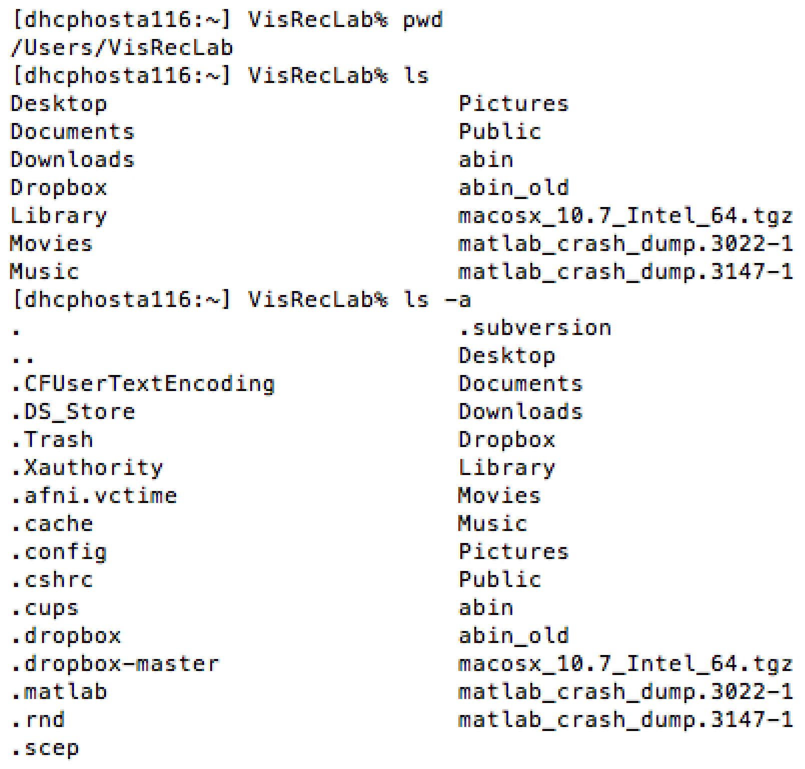
In addition, more extensive documentation about the tcsh scripting language can be found under the manual entry for tcsh (i.e. `man tcsh`), which is also viewable online:
http://www.kitebird.com/csh-tcsh-book/tcsh.html

Navigating the file system

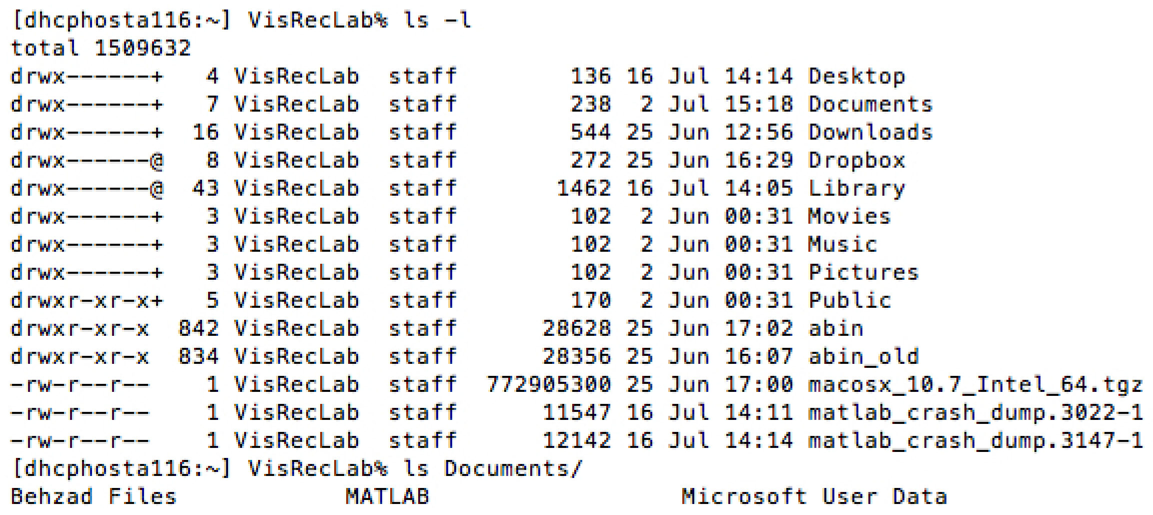
Command	**Description**	**Usage (with some options)**
`pwd` *Print Working Directory*	**“Where am I?”** Display the current working directory (current location within the file system).	`pwd`

`ls`
List
	“What’s here?”
	List the contents of a directory.
	● With no options, list the contents of the current directory.
	● With a filename, list the file and any information about the file.
	● With a directory, list the contents of that directory.
	● With option `-a`, list *all* files in the directory, including files whose name starts with ‘.’ (hidden files).
	● With option -l, list files in the *long* format, which includes information such as the file size, file permissions, and the time that the file was last modified.
		`ls`
		`ls -a`
		`ls -l`
		`ls` *`filename`*
		`ls` *`directory`*

Example usage

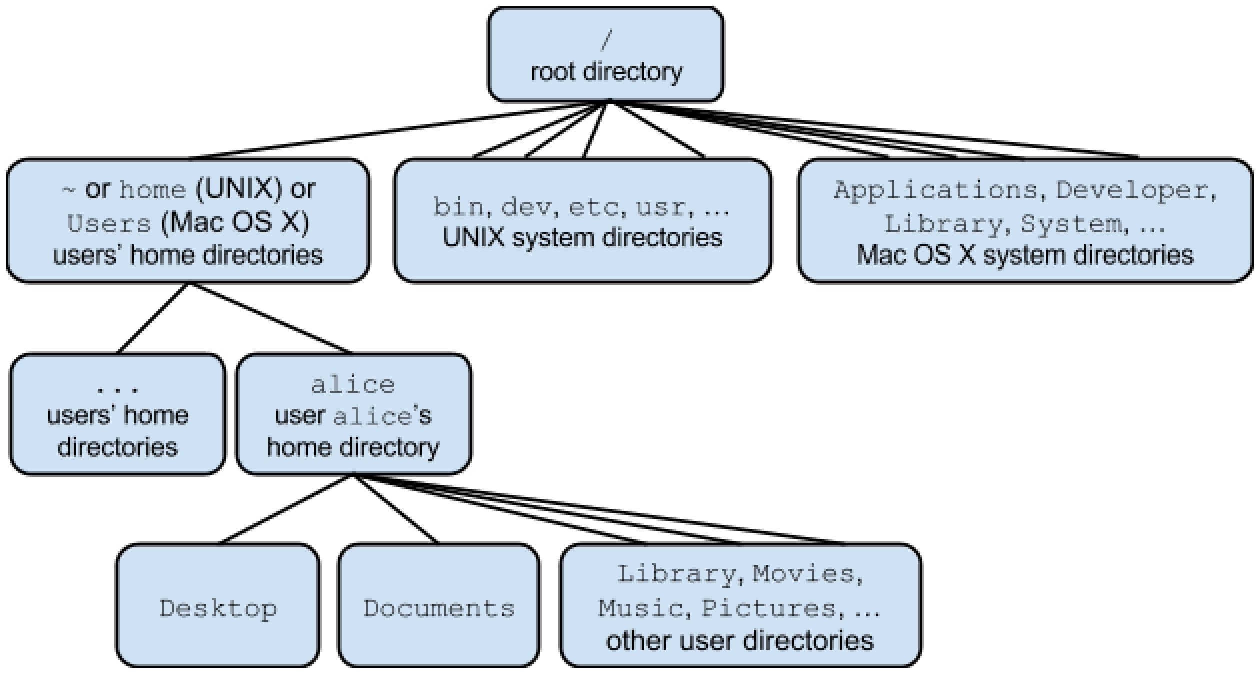


```
[dhcphosta116:~] VisRecLab% pwd
/Users/VisRecLab
[dhcphosta116:~] VisRecLab% ls
Desktop                         Pictures
Documents                       Public
Downloads                       abin
Dropbox                         abin_old
Library                         macosx_10.7_Intel_64.tgz
Movies                          matlab_crash_dump.3022-1
Music                           matlab_crash_dump.3147-1
[dhcphosta116:~] VisRecLab% ls -a
.                               .subversion
..                              Desktop
.CFUserTextEncoding             Documents
.DS_Store                       Downloads
.Trash                          Dropbox
.Xauthority                     Library
.afni.vctime                    Movies
.cache                          Music
.config                         Pictures
.cshrc                          Public
.cups                           abin
.dropbox                        abin_old
.dropbox-master                 macosx_10.7_Intel_64.tgz
.matlab                         matlab_crash_dump.3022-1
.rnd                            matlab_crash_dump.3147-1
.scep
```



```
[dhcphosta116:~] VisRecLab% ls -l
total 1509632
drwx------+   4 VisRecLab  staff        136 16 Jul 14:14 Desktop
drwx------+   7 VisRecLab  staff        238  2 Jul 15:18 Documents
drwx------+  16 VisRecLab  staff        544 25 Jun 12:56 Downloads
drwx------@   8 VisRecLab  staff        272 25 Jun 16:29 Dropbox
drwx------@  43 VisRecLab  staff       1462 16 Jul 14:05 Library
drwx------+   3 VisRecLab  staff        102  2 Jun 00:31 Movies
drwx------+   3 VisRecLab  staff        102  2 Jun 00:31 Music
drwx------+   3 VisRecLab  staff        102  2 Jun 00:31 Pictures
drwxr-xr-x+   5 VisRecLab  staff        170  2 Jun 00:31 Public
drwxr-xr-x  842 VisRecLab  staff      28628 25 Jun 17:02 abin
drwxr-xr-x  834 VisRecLab  staff      28356 25 Jun 16:07 abin_old
-rw-r--r--    1 VisRecLab  staff  772905300 25 Jun 17:00 macosx_10.7_Intel_64.tgz
-rw-r--r--    1 VisRecLab  staff      11547 16 Jul 14:11 matlab_crash_dump.3022-1
-rw-r--r--    1 VisRecLab  staff      12142 16 Jul 14:14 matlab_crash_dump.3147-1
[dhcphosta116:~] VisRecLab% ls Documents/
Behzad Files            MATLAB                  Microsoft User Data
```

The Unix/Mac OS X file system: directory tree


/
root directory
~ or home (UNIX) or Users (Mac OS X)
users' home directories
bin, dev, etc, usr, ...
UNIX system directories
Applications, Developer, Library, System, ...
Mac OS X system directories
...
users' home directories
alice
user alice's home directory
Desktop
Documents
Library, Movies, Music, Pictures, ...
other user directories

Special directories

Root directory

This is the "root" of the directory tree: it contains all other directories in the system. It is designated by `/`. Any file paths that begin with `/` (for example, `/home/alice/Desktop/`) are known as *absolute paths*, because they specify the complete path to the file, beginning at the root of the directory tree (where `/` is used in any other part of a file path, it separates directories).

Home directory

Each user has a home directory, located in `/home` (Unix) or `/Users` (Mac OS X). This is where the user's personal files are usually stored by default, including files related to programs that the user installs. The current user's home directory can be abbreviated as `~` (for example, `/home/alice/Desktop` is equivalent to `~/Desktop` if `alice` is the current user). The home directory of another user can be abbreviated as `~username`, eg. `~bob`.

Enclosing/parent directory

Each directory has an enclosing/parent directory. Within a directory, its parent directory is abbreviated as `..` (for example, within `/home/alice/Desktop`, `/home/alice` can be referred to as `..`, and `/home` can be referred to as `../..`).

Current directory

The current working directory is abbreviated as `.` This is particularly useful for referring to the current directory when its full (absolute) name is long (for example, within `/home/alice/Desktop`, the commands `ls`, `ls /home/alice/Desktop`, and `ls .` are equivalent).

Relative vs. absolute paths

An absolute file path is the absolute location of a file within the file system. It is the more precise description of a file's location, and will not change unless the file is moved. It begins with the root and becomes increasingly specific. For example, `/home/alice/Documents/file.txt` is an absolute path to this file.

A relative file path is the location of a file relative to the user's current location within the file system. For instance, if the current working directory is `/home/alice/Desktop`, then the relative path to the file `/home/alice/Documents/file.txt` is `../Documents/file.txt`.

An analogy to describe the difference between absolute and relative file paths is that the absolute path is a map with a location circled, while a relative path is a series of directions (eg. go straight for 3 blocks, then turn left) leading to that location.

When navigating the file system, absolute and relative paths may be useful in different situations. Depending on the distance between two locations within the file system, either the absolute path or the relative path may be shorter (see the example above). Note that some programs may require an absolute path and will not accept a relative path.

Navigating the file system (continued)

Command	Description	Usage (with some options)

`cd`
Change Directory
	Change the current working directory (current location within the file system).
	● With no options, change to the home directory.
	● With a directory, change to that directory. This can be either a relative or absolute path to the new working directory.
		`cd`
		`cd` *`directory`*

Example usage

```
[dhcphosta116:~] VisRecLab% pwd
/Users/VisRecLab
[dhcphosta116:~] VisRecLab% ls
Desktop                         Pictures
Documents                       Public
Downloads                       abin
Dropbox                         abin_old
Library                         macosx_10.7_Intel_64.tgz
Movies                          matlab_crash_dump.3022-1
Music                           matlab_crash_dump.3147-1
[dhcphosta116:~] VisRecLab% cd Desktop
[dhcphosta116:~/Desktop] VisRecLab% ls
Untitled.jpg
[dhcphosta116:~/Desktop] VisRecLab% cd ../Documents
[dhcphosta116:~/Documents] VisRecLab% ls
Behzad Files            MATLAB                  Microsoft User Data
[dhcphosta116:~/Documents] VisRecLab% cd /
[dhcphosta116:/] VisRecLab% ls
Applications            bin                     opt
Library                 cores                   private
Network                 dev                     sbin
System                  etc                     tmp
User Information        home                    usr
Users                   mach_kernel             var
Volumes                 net
[dhcphosta116:/] VisRecLab% cd System/Library/
[dhcphosta116:/System/Library] VisRecLab% cd ~
[dhcphosta116:~] VisRecLab% pwd
/Users/VisRecLab
```

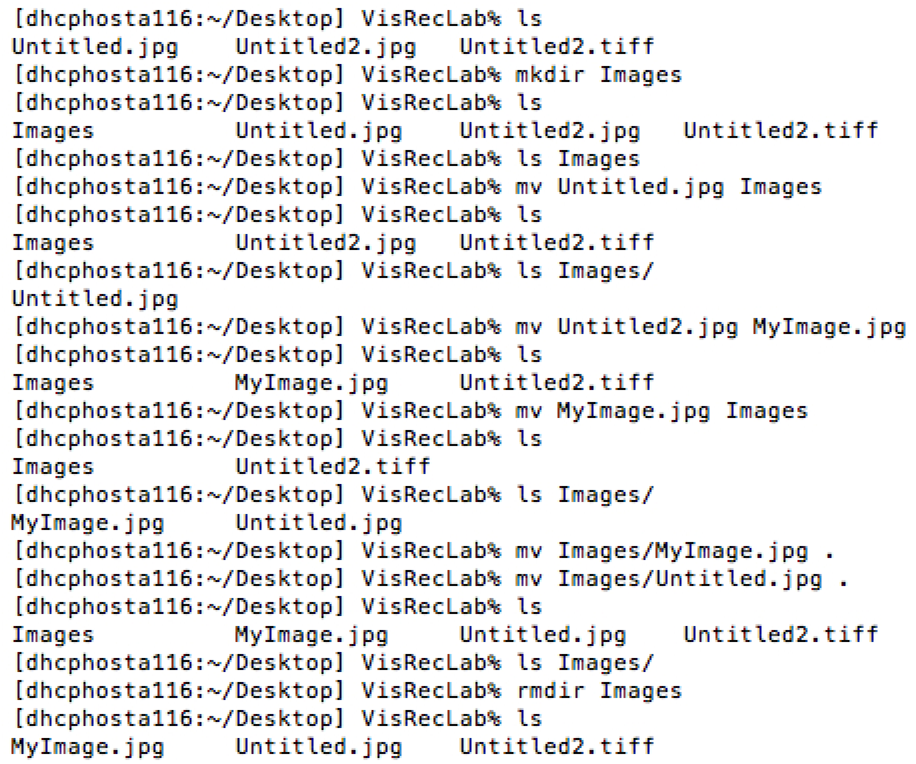
Modifying the directory structure

Command	Description	Usage (with some options)
`mkdir` *Make Directory*	Create a new directory.	`mkdir` *`directory`*

`mv`
Move
	Move a file or directory, or rename a file or directory.
	● With one or more source names of files or directories, and (as the last argument) a directory name that already exists, move all the files and/or directories into that directory.
	● With a directory or file that already exists (as the first argument), and a directory or file name that doesn't exist (as the second argument), rename the file or directory.
		`mv` *`source`* `[...]` *`directory`*
		`mv` *`old_name new_name`*

`rmdir` *Remove Directory*	Delete an empty directory. **N.B. Once deleted, a directory (or file) cannot be easily recovered.**	`rmdir` *`directory`*

Example usage



```
[dhcphosta116:~/Desktop] VisRecLab% ls
Untitled.jpg     Untitled2.jpg   Untitled2.tiff
[dhcphosta116:~/Desktop] VisRecLab% mkdir Images
[dhcphosta116:~/Desktop] VisRecLab% ls
Images           Untitled.jpg    Untitled2.jpg   Untitled2.tiff
[dhcphosta116:~/Desktop] VisRecLab% ls Images
[dhcphosta116:~/Desktop] VisRecLab% mv Untitled.jpg Images
[dhcphosta116:~/Desktop] VisRecLab% ls
Images           Untitled2.jpg   Untitled2.tiff
[dhcphosta116:~/Desktop] VisRecLab% ls Images/
Untitled.jpg
[dhcphosta116:~/Desktop] VisRecLab% mv Untitled2.jpg MyImage.jpg
[dhcphosta116:~/Desktop] VisRecLab% ls
Images           MyImage.jpg     Untitled2.tiff
[dhcphosta116:~/Desktop] VisRecLab% mv MyImage.jpg Images
[dhcphosta116:~/Desktop] VisRecLab% ls
Images           Untitled2.tiff
[dhcphosta116:~/Desktop] VisRecLab% ls Images/
MyImage.jpg      Untitled.jpg
[dhcphosta116:~/Desktop] VisRecLab% mv Images/MyImage.jpg .
[dhcphosta116:~/Desktop] VisRecLab% mv Images/Untitled.jpg .
[dhcphosta116:~/Desktop] VisRecLab% ls
Images           MyImage.jpg     Untitled.jpg    Untitled2.tiff
[dhcphosta116:~/Desktop] VisRecLab% ls Images/
[dhcphosta116:~/Desktop] VisRecLab% rmdir Images
[dhcphosta116:~/Desktop] VisRecLab% ls
MyImage.jpg      Untitled.jpg    Untitled2.tiff
```

Searching for files

When referring to files and directories, most shell commands can use special characters to form patterns. The shell will match these patterns to file and directory names.
For instance, the character `*` in a pattern can correspond to any number of characters in a filename. So `*` is a pattern which matches all files in the current directory. `A*` is a pattern which matches all files that begin with `A`. `*.txt` is a pattern which matches all files in the current directory that end with `.txt`.

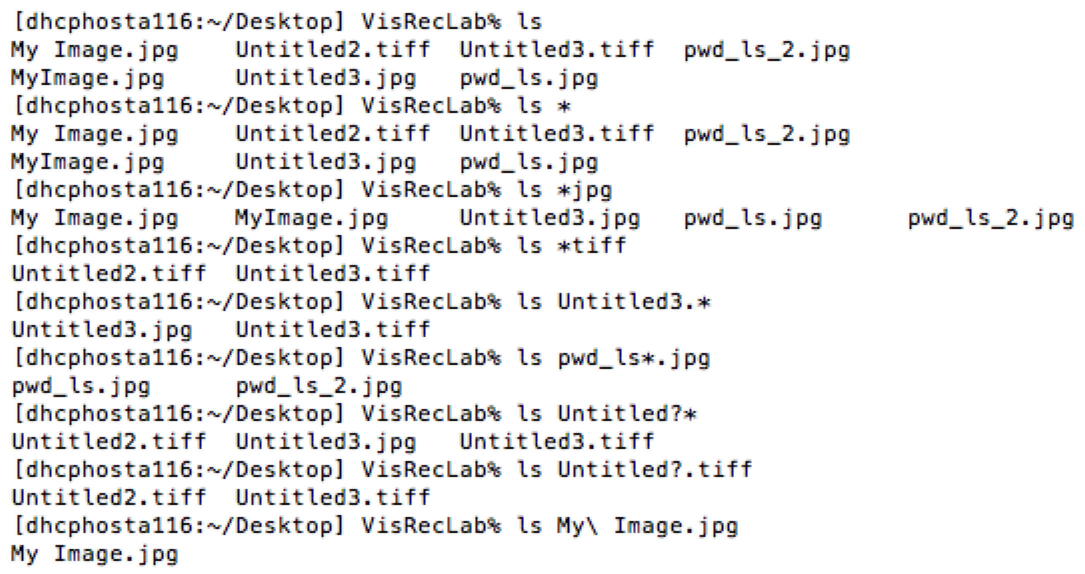
Note that the pattern must directly match the entire filename, and not simply a part of the name. For instance, `A*` will match all files that begin with `A` but not all files which contain `A`.

In addition, the escape character `\`, when used immediately before a special character, allows the shell to match this character as part of a filename. This allows the shell to match filenames containing spaces, for example.

Special characters: Unix globbing

Character	Description	Examples
`*`	Match any number (zero or more) of unknown characters.	`Law*` matches `Law`, `Laws`, or `Lawyer`
`?`	Match exactly one unknown character.	`?at` matches `Cat`, `cat`, `Bat` or `bat`, but not `at` or `spat`
`[`*`characters`*`]`	Match any character within a group of characters.	`[cb]at` matches `cat` or `bat` but not `Cat`, `Bat`, `at`, or `scat`
`[^`*`characters`*`]`	Match any character not in a group of characters.	`[^C]at` matches Bat, bat, or cat but not Cat
`\`*`special_character`*	Match a special character such as `*`, `?`, or a space.	`Law\*` matches `Law*`, `Law\ clerk` matches `Law clerk`

Example usage



```
[dhcphosta116:~/Desktop] VisRecLab% ls
My Image.jpg    Untitled2.tiff  Untitled3.tiff  pwd_ls_2.jpg
MyImage.jpg     Untitled3.jpg   pwd_ls.jpg
[dhcphosta116:~/Desktop] VisRecLab% ls *
My Image.jpg    Untitled2.tiff  Untitled3.tiff  pwd_ls_2.jpg
MyImage.jpg     Untitled3.jpg   pwd_ls.jpg
[dhcphosta116:~/Desktop] VisRecLab% ls *jpg
My Image.jpg    MyImage.jpg     Untitled3.jpg   pwd_ls.jpg      pwd_ls_2.jpg
[dhcphosta116:~/Desktop] VisRecLab% ls *tiff
Untitled2.tiff  Untitled3.tiff
[dhcphosta116:~/Desktop] VisRecLab% ls Untitled3.*
Untitled3.jpg   Untitled3.tiff
[dhcphosta116:~/Desktop] VisRecLab% ls pwd_ls*.jpg
pwd_ls.jpg      pwd_ls_2.jpg
[dhcphosta116:~/Desktop] VisRecLab% ls Untitled?*
Untitled2.tiff  Untitled3.jpg   Untitled3.tiff
[dhcphosta116:~/Desktop] VisRecLab% ls Untitled?.tiff
Untitled2.tiff  Untitled3.tiff
[dhcphosta116:~/Desktop] VisRecLab% ls My\ Image.jpg
My Image.jpg
```

Searching inside files

When searching for a piece of text or a pattern of text inside a file, the `grep` command is very efficient and powerful. Given a pattern, it searches inside the file (or files) line by line. Any lines that contain the pattern are then printed.

Note that it is not necessary for the entire line to match the pattern, but simply for it to contain text that matches the pattern.

`grep` can use standard regular expressions (a type of pattern matching) to search. The syntax for regular expressions can be found in the manual entry for `grep`, and in many other accessible references (such as Wikipedia).

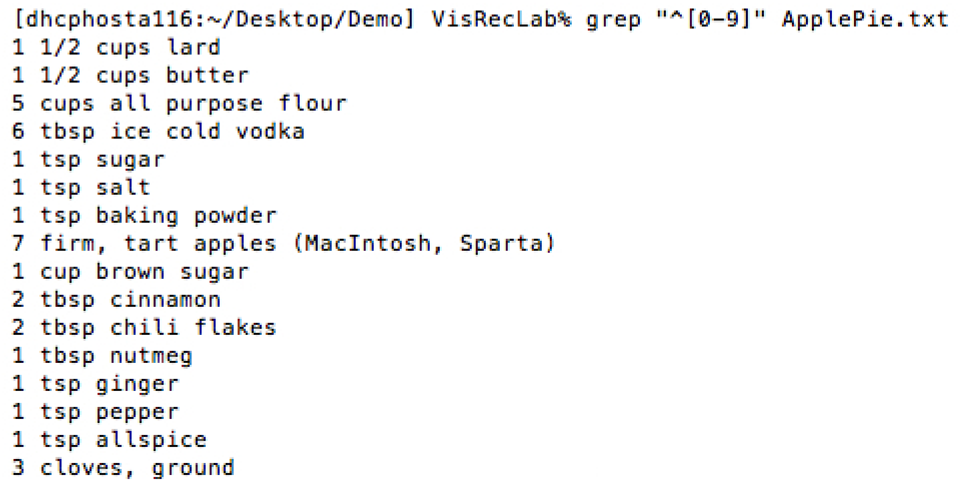
Command	**Description**	**Usage (with some options)**

`grep`	Search for a pattern inside (one or more) files. Display the lines of the file(s) in which the pattern is matched. This pattern can contain *regular expressions* (similar to Unix globbing above).
	● With option `-n`, prefix each line of output with the line number within its input file.
	● With option `-c`, don't print the matching lines; instead, print the number of matching lines.
		`grep` *`pattern filename`* `[...]`
		`grep -n` *`pattern filename`*
		`grep -c` *`pattern filename`*
		`grep -v` *`pattern filename`*

	● With option `-v`, invert the search: print the lines that don't match the pattern.	

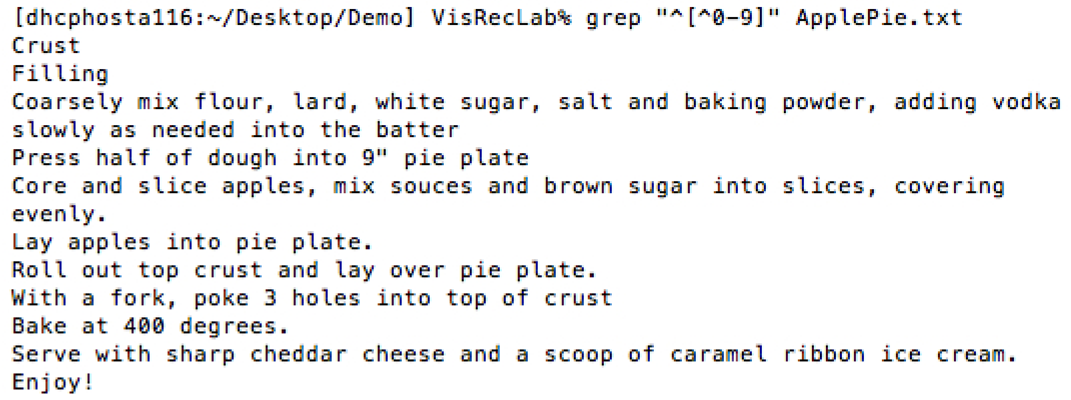
Example usage

What are the ingredients for the recipe?
(find all lines that start with a number)



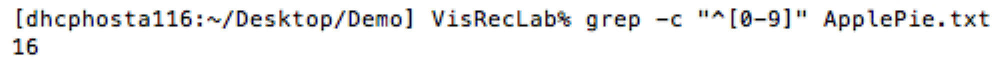
```
[dhcphosta116:~/Desktop/Demo] VisRecLab% grep "^[0-9]" ApplePie.txt
1 1/2 cups lard
1 1/2 cups butter
5 cups all purpose flour
6 tbsp ice cold vodka
1 tsp sugar
1 tsp salt
1 tsp baking powder
7 firm, tart apples (MacIntosh, Sparta)
1 cup brown sugar
2 tbsp cinnamon
2 tbsp chili flakes
1 tbsp nutmeg
1 tsp ginger
1 tsp pepper
1 tsp allspice
3 cloves, ground
```

What are the instructions for the recipe?
(find all non-empty lines that don't start with a number)



```
[dhcphosta116:~/Desktop/Demo] VisRecLab% grep "^[^0-9]" ApplePie.txt
Crust
Filling
Coarsely mix flour, lard, white sugar, salt and baking powder, adding vodka
slowly as needed into the batter
Press half of dough into 9" pie plate
Core and slice apples, mix souces and brown sugar into slices, covering
evenly.
Lay apples into pie plate.
Roll out top crust and lay over pie plate.
With a fork, poke 3 holes into top of crust
Bake at 400 degrees.
Serve with sharp cheddar cheese and a scoop of caramel ribbon ice cream.
Enjoy!
```

How many ingredients are there?
(find the number of lines that start with a number)



```
[dhcphosta116:~/Desktop/Demo] VisRecLab% grep -c "^[0-9]" ApplePie.txt
16
```

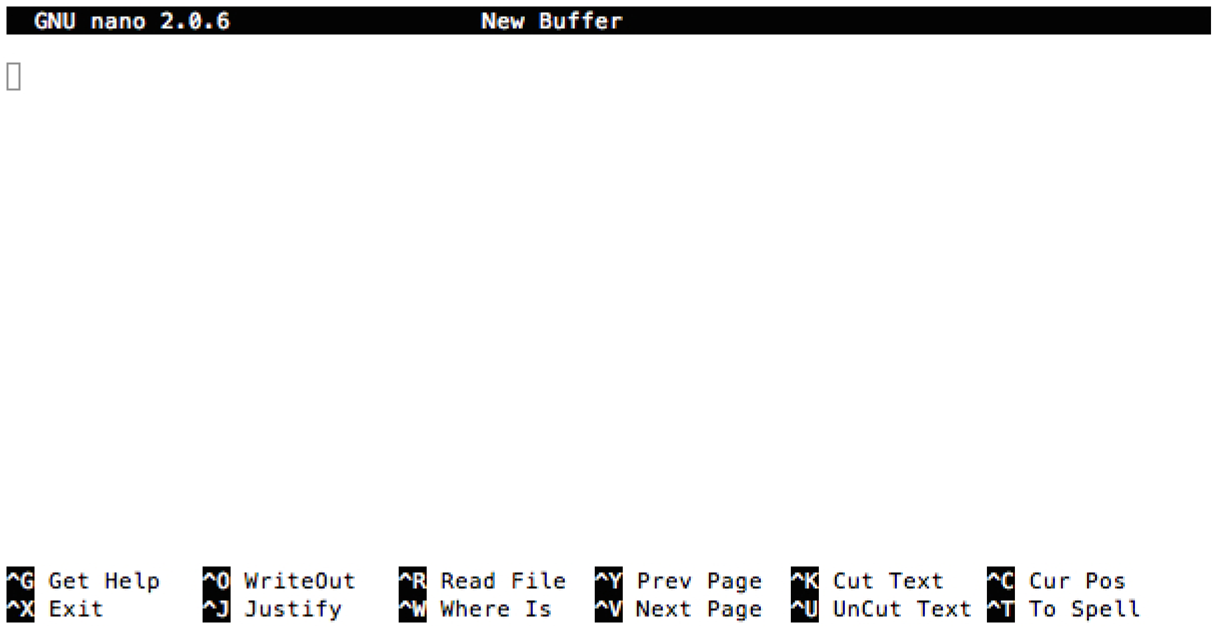
Files and text editing

Some programs do not have a graphical user interface, but instead, run in the shell (possibly with a text user interface). One advantage of working with these programs is speed: not only do they run more quickly without having to support a graphical interface, but they are faster for users because they do not require the mouse or touchpad.

There are numerous text editors used with the command line, including vi/Vim, Emacs, and nano. One advantage of these editors that they allow users to quickly create and edit plain-text files, which are used for programming (including Unix scripting) and by the Unix operating system, among other things.

The editor that we will show here is nano:

- To open nano (and create a new, unnamed file), the command is `nano`.
- To open nano and create a new, named file, the command is `nano` *`new_filename`*.
- To open an existing file in nano, the command is `nano` *`filename`*.



```
  GNU nano 2.0.6                    New Buffer



^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next Page  ^U UnCut Text ^T To Spell
```

- Most "special functions" (eg. cutting, pasting) are a combination of the Ctrl key and a character.
    - For instance, to save a file (*WriteOut*), hold down the Ctrl key and press O.
- Some commonly-used special functions are listed at the bottom of the screen. For a complete list of special functions and for more specific information about nano, see the nano documentation: http://www.nano-editor.org/dist/v2.2/nano.html

Section 2: Combining Unix commands

Standard streams (input/output)

Each shell has three standard streams (communication channels): standard input, standard output, and standard error. Every program that the shell runs uses these streams unless otherwise specified. Standard input is usually the keyboard; standard output and standard error are usually the shell/command-line. This means that, by default, programs receive input through the keyboard, print output to the shell, and print error messages to the shell.

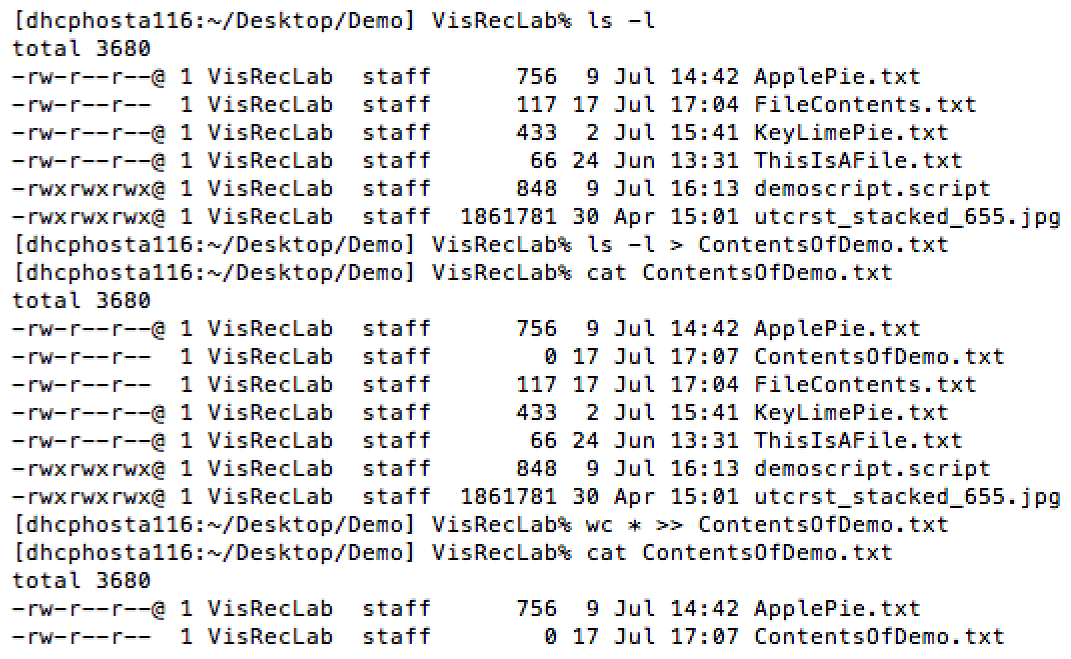
However, it is often useful to redirect these streams. For instance, rather than having the output of a program printed to the shell, it can be redirected so that it is written to a file.
To redirect a stream for a program, type the command for the program and then the way in which you want it to be redirected.

	Redirect standard input	**Redirect standard output**	**Redirect standard error (and output)**

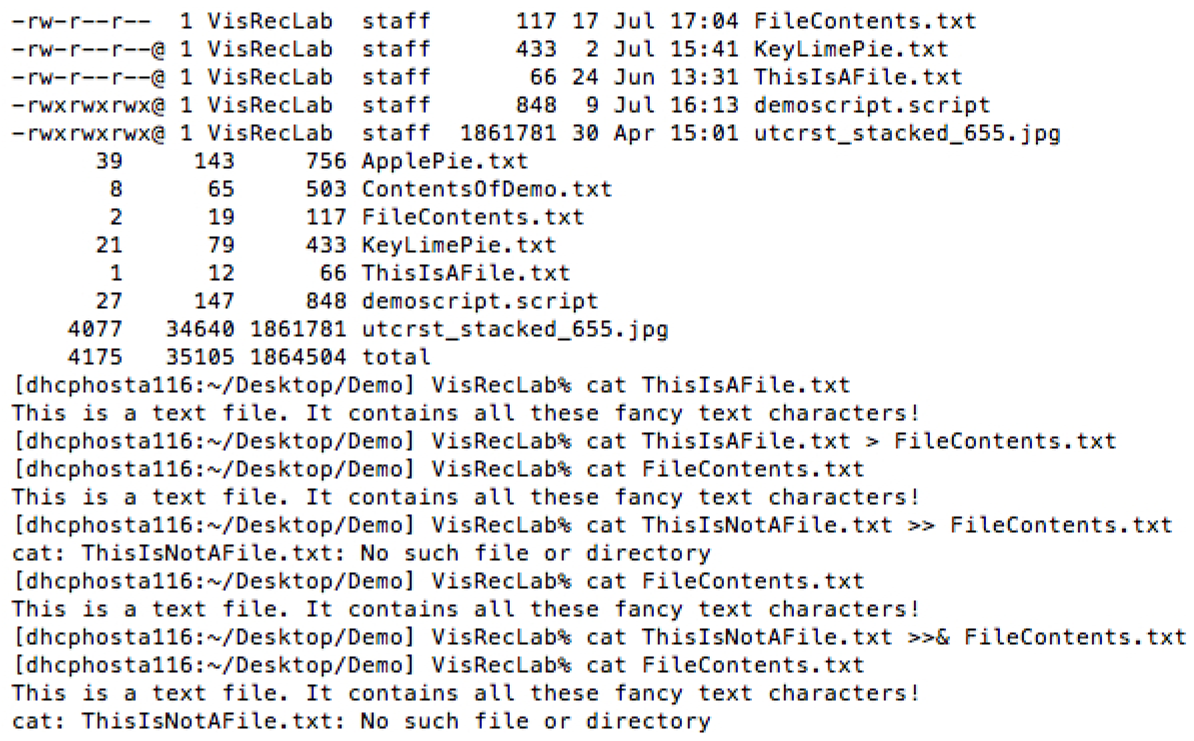
Syntax	`command < file`	`command > file`
		`command >> file`
			`command >& file`
			`command >>& file`

`>` redirects the output to a new file. If this file already exists, it is overwritten.
`>>` appends the output to an already-existing file.
`>&` and `>>&` have the same function, but redirect the standard error in addition to the output.

Example usage



```
[dhcphosta116:~/Desktop/Demo] VisRecLab% ls -l
total 3680
-rw-r--r--@ 1 VisRecLab  staff      756  9 Jul 14:42 ApplePie.txt
-rw-r--r--  1 VisRecLab  staff      117 17 Jul 17:04 FileContents.txt
-rw-r--r--@ 1 VisRecLab  staff      433  2 Jul 15:41 KeyLimePie.txt
-rw-r--r--@ 1 VisRecLab  staff       66 24 Jun 13:31 ThisIsAFile.txt
-rwxrwxrwx@ 1 VisRecLab  staff      848  9 Jul 16:13 demoscript.script
-rwxrwxrwx@ 1 VisRecLab  staff  1861781 30 Apr 15:01 utcrst_stacked_655.jpg
[dhcphosta116:~/Desktop/Demo] VisRecLab% ls -l > ContentsOfDemo.txt
[dhcphosta116:~/Desktop/Demo] VisRecLab% cat ContentsOfDemo.txt
total 3680
-rw-r--r--@ 1 VisRecLab  staff      756  9 Jul 14:42 ApplePie.txt
-rw-r--r--  1 VisRecLab  staff        0 17 Jul 17:07 ContentsOfDemo.txt
-rw-r--r--  1 VisRecLab  staff      117 17 Jul 17:04 FileContents.txt
-rw-r--r--@ 1 VisRecLab  staff      433  2 Jul 15:41 KeyLimePie.txt
-rw-r--r--@ 1 VisRecLab  staff       66 24 Jun 13:31 ThisIsAFile.txt
-rwxrwxrwx@ 1 VisRecLab  staff      848  9 Jul 16:13 demoscript.script
-rwxrwxrwx@ 1 VisRecLab  staff  1861781 30 Apr 15:01 utcrst_stacked_655.jpg
[dhcphosta116:~/Desktop/Demo] VisRecLab% wc * >> ContentsOfDemo.txt
[dhcphosta116:~/Desktop/Demo] VisRecLab% cat ContentsOfDemo.txt
total 3680
-rw-r--r--@ 1 VisRecLab  staff      756  9 Jul 14:42 ApplePie.txt
-rw-r--r--  1 VisRecLab  staff        0 17 Jul 17:07 ContentsOfDemo.txt
```



```
-rw-r--r--  1 VisRecLab  staff      117 17 Jul 17:04 FileContents.txt
-rw-r--r--@ 1 VisRecLab  staff      433  2 Jul 15:41 KeyLimePie.txt
-rw-r--r--@ 1 VisRecLab  staff       66 24 Jun 13:31 ThisIsAFile.txt
-rwxrwxrwx@ 1 VisRecLab  staff      848  9 Jul 16:13 demoscript.script
-rwxrwxrwx@ 1 VisRecLab  staff  1861781 30 Apr 15:01 utcrst_stacked_655.jpg
      39     143     756 ApplePie.txt
       8      65     503 ContentsOfDemo.txt
       2      19     117 FileContents.txt
      21      79     433 KeyLimePie.txt
       1      12      66 ThisIsAFile.txt
      27     147     848 demoscript.script
    4077   34640 1861781 utcrst_stacked_655.jpg
    4175   35105 1864504 total
[dhcphosta116:~/Desktop/Demo] VisRecLab% cat ThisIsAFile.txt
This is a text file. It contains all these fancy text characters!
[dhcphosta116:~/Desktop/Demo] VisRecLab% cat ThisIsAFile.txt > FileContents.txt
[dhcphosta116:~/Desktop/Demo] VisRecLab% cat FileContents.txt
This is a text file. It contains all these fancy text characters!
[dhcphosta116:~/Desktop/Demo] VisRecLab% cat ThisIsNotAFile.txt >> FileContents.txt
cat: ThisIsNotAFile.txt: No such file or directory
[dhcphosta116:~/Desktop/Demo] VisRecLab% cat FileContents.txt
This is a text file. It contains all these fancy text characters!
[dhcphosta116:~/Desktop/Demo] VisRecLab% cat ThisIsNotAFile.txt >>& FileContents.txt
[dhcphosta116:~/Desktop/Demo] VisRecLab% cat FileContents.txt
This is a text file. It contains all these fancy text characters!
cat: ThisIsNotAFile.txt: No such file or directory
```

Pipes

Another useful feature of shells are pipes. Pipes allow the piping of different shell commands by connecting the output stream of one command to the input stream of another command. Pipes are created by putting a bar `|` between two commands. The output of the first command will then be sent to the second command as input. For example, the pipeline `ls -l | wc -l` counts the number of lines produced by `ls -l`.

One common, simple pipeline is piping the lengthy output of a command to a text editor, so that it can be viewed more easily (particularly if the output is long enough that it will not fit in one screen). For example, `ls | less` is useful in a directory with a large number of files.

Pipes can be used to connect multiple commands sequentially. For example, the pipeline `ls -l | tail -n 10 | sort` displays a sorted list of the last 10 lines of output of `ls -l`.

Pipes can also be used to connect both the output stream and error stream of one command, simultaneously, to the input stream of another, by using `|&` instead of just `|`.

Section 3: Unix scripting

Introduction

A shell script is a plain-text file that contains a series of shell commands. When this file is executed/run, the commands are run in sequence.
The purposes of creating a shell script include:

- It allows a sequence of commands to be automated.
  Rather than the user having to type in each command, wait for it to finish running, and then type the next command, as soon as the script is executed, all the commands will be run without the user having to do anything (unless the script is designed to request input from the user).
- It allows a sequence of commands to be saved and re-used.
  If a sequence of commands is going to be run more than once, it can be saved in a script and subsequently, run easily. This saves the user time and effort, prevents mistakes from occurring when attempting to run a sequence of commands again, and allows different users to share a sequence of commands if they want to perform similar tasks.
- It allows the user to control the order in which commands are run (the “control flow” of the script).
  Scripts can contain control flow tools such as loops, which allow a command to be easily run multiple times, and conditionals, which allow different commands to be run in different situations.

Executing a script

Three ways of running a script:

1. `shell script_name`
   Type the name of the shell which will be used to run the script (here, `tcsh`) followed by the name of the script file.
2. `source script_name`
   The `source` command runs the script in `script_name` within the current shell.
3. `script_name`
   Type the name of the script, either as part of an absolute path or relative path. The relative path to a script in the current working directory is `./script_name`.
   When running a script using this method, the first line of the script file should be exactly `#!shell` which specifies the shell that will be used to run the script. Here, the first line would be `#!/bin/tcsh`
   With this method, the script file must also be an executable file (that is, you must have permission to execute it). If it is not, the command `chmod u+x script_name` gives the owner (usually the creator) of the file permission to execute it.

Variables

Variables are labels that are used to refer to values. For example, if the variable `x` is defined to refer to the value `"my_string"`, then each time `x` is used within the script, `"my_string"` will be substituted.

Variables are useful as short forms when reusing values, since the variable name may be simpler, shorter, and/or more descriptive than the value itself.
And if the value is ever changed, it is not necessary to change it each time it is used in the script, since it is always referred to the same way, by the variable name.
They are also useful to indicate where a value will be used in the script without knowing in advance what the value is. For instance, if a script expects the user to give it input when it is run, then the input information can be referred to by a variable, despite not knowing its value.

The syntax for creating/defining a variable that holds "words" (i.e. a string) is
`set` *`variable_name=string`*.
`string` can contain numbers, strings, commands, glob characters, other variable names, etc.
For example, `set x=hello` creates a variable `x` which refers to the string `"hello"`.

The syntax for creating a variable that holds an integer value is `@ variable_name=value`.
`value` can be a single integer, or it can be an arithmetic expression (discussed below).
For example, `@ x=3` creates a variable `x` with refers to the number `3`.

When referring to a variable after it has already been created, the name of the variable must be preceded by a dollar sign `$`. This tells the shell to substitute the value of the variable where its name is used.
For example, a script containing

```
set file_name=MyFile.txt
cat $file_name
```

runs the command `cat MyFile.txt`.

Array variables

Variables can also be created that hold a list of values, rather than just one. These are called "arrays". The syntax for this is `set` *`array_name`*`=(` *`value1`* `[...] )`.
For example, `set x=( hello 10 4 )` creates a variable `x` which refers to a list. The first element in this list is the string `"hello"`; the second element is `10`; and the third element is `4`.

To refer to individual elements of an array, the syntax is *`array_name`*`[`*`index_number`*`]`. The index of the first element is `1`, the second is `2`, etc.
For example, `set x[2]=15` changes the value of the second element of the array `x` to `15`.

Input: Command line arguments

$1, $2, $3, …, ${10}, ${11}, …, $#

In addition to variables that the user can create within the script, there are variables that already created when the script runs, called command line arguments. These are the inputs given to a program as part of the command to run it, separated by spaces.
For example, the command `ls` has no command line arguments, but `ls MyFile` has one, and `ls -l MyFile` has two. As in the case of `ls`, these arguments may be optional, while other programs such as `rm` may require them.

The command line arguments are variables `1`, `2`, …, up to a very large number, where `1` is the first argument, `2` is the second, etc.. They are then referred to as `$1`, `$2`, ….
For example, if the script `add` contains

```
@ sum = $1 + $2
echo The sum is $sum.
```

and is run with the command `add 3 4`, it will print `The sum is 7`.

The total number of command line arguments given to the program is held by the variable `#`, referred to as `$#`.

Exit status

$?

In addition to an output and any errors, most programs also have an exit status. This is a number that indicates whether the program ran successfully. If there was an error, for instance, or if the program terminated early, this may mean that the program was not successful. Usually, an exit status of 0 means that the program succeeded, while an exit status greater than 0 means that it failed.

The exit status of a program is not printed. However, the exit status of the most recently run program is held by the variable `?`, referred to as `$?`.
For example, the script

```
rm MyFile
if ( $? ) then
  echo Failure
else
  echo Success
endif
```

prints "`Success`" if `rm MyFile` ran successfully, or "`Failure`" if `rm MyFile` failed.

To exit a script and specify an exit status, the syntax is `exit[`*`exit_status`*`]`.
For example, to exit the script and indicate that there was an error that caused it to fail, use `exit[1]`.

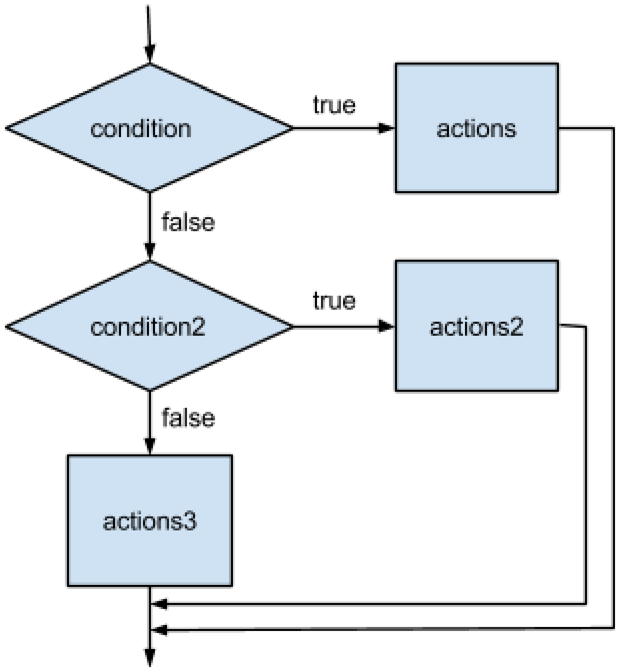
Shell arithmetic

The shell can perform basic arithmetic operations including addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), increment by one (`++`), and decrement by one (`--`).
When performing these operations, the operands must be integer values, and if the result is assigned to a variable, it must be a numerical variable.
For example, `@ x = 354 - 128 + 52 * 5 / 3` assigns the integer value `174` to `x`.

Note that there must be space between each operator and operand in the arithmetic expression.

Control flow: if-statement

```
if (condition) then
  actions
else if (condition2) then
  actions2
else
  actions3
endif
```



If-statements are a type of control flow structure that selects between several different actions depending on whether some condition is true or false.

The most basic type of if-statement has only one block:

```
if (condition) then
  actions
endif
```

This tells the program to check whether the condition is true, and if so, to perform the actions inside the block. If it is not true, these actions are ignored.

You can also add a block that tells the program to ignore these actions if the condition is not true, and instead perform alternative actions:

```
if (condition) then
  actions
else
  alternative_actions
endif
```

If the condition *is* true, however, the alternative actions inside the else-block are ignored.

Lastly, you can add any number of blocks with alternative conditions and alternative actions:

```
if (condition) then
  actions
else if (alternative_condition) then
  alternative_actions
else if (other_alternative_condition) then
  other_alternative_actions
...
else
  final_alternative_actions
endif
```

If the first condition is true, the actions inside the first block are performed and everything else is ignored. If it is not true, the condition inside the first else if-block is checked; if that condition is true, the actions inside that block are performed and everything else is ignored. If that condition is false, the condition inside the next else if-block is checked, etc. Finally, if none of the conditions are true, the actions inside the else-block are performed.

Note that the `else` and `endif` must appear at the beginning of input lines; the first `if` must appear alone on its input line.

Conditions

One way to create a condition is to use an operator that evaluates to true or false, such as a comparison or file inquiry operator.
For example, the script

```
if ( $user == “elvis” ) then
  echo ‘the king lives’
endif
```

checks whether the variable `user` holds a value that is equal to the string `“elvis”`.

Comparison operators

Form: `operand1 operator operand2`

Operator (and opposite operator)	**Checks if...**
== (!=)	`operand1` is equal to `operand2` (`operand1` is equal to `operand2`)
=~ (!~)	`operand1` matches `operand2`, where `operand2` is a glob pattern (`operand1` does not match `operand2`)
< (>)	`operand1` is numerically less than `operand2` (`operand1` is numerically greater than `operand2`)
<= (>=)	`operand1` is numerically less than or equal to `operand2` (`operand1` is numerically greater than or equal to `operand2`)

Some file inquiry operators

Form: `-operator file`

Operator	Checks if...
`r`	The user has permission to read `file`
`w`	The user has permission to write to `file`
`x`	The user has permission to execute `file`
`e`	`file` exists
`s`	`file` is of a non-zero size (and not a directory)
`d`	`file` is a directory

Another way to create a condition is to use the fact that the shell treats 0 as false and any other numerical value as true.
For example, the script

```
if ( $# ) then
  echo Error: insufficient arguments.
endif
```

checks whether the user has given no commandline arguments.

This is particularly useful for deciding which action to take depending on whether a command ran successfully or not, since the exit status of a command is a number that can be used in a condition.
For example, the script (from above)

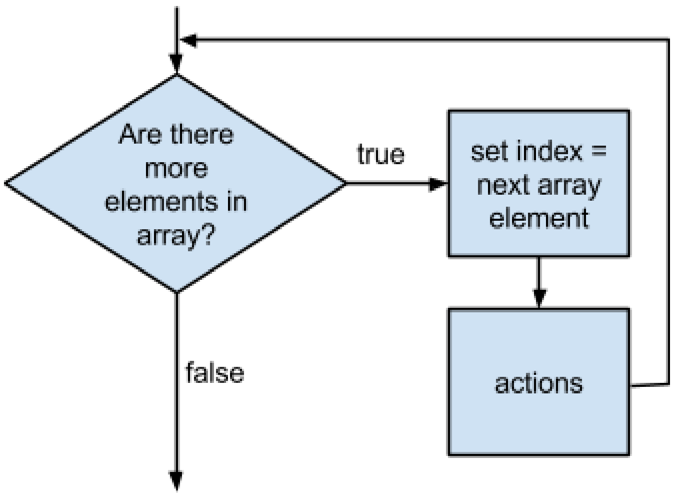
```
rm MyFile
if ( $? ) then
  echo Failure
else
  echo Success
endif
```

prints "`Success`" if `rm MyFile` ran successfully (has an exit status of 0), or "`Failure`" if `rm MyFile` failed (has an exit status of 1).

Finally, multiple conditions can be combined into one larger condition using and (`&&`) and or (`||`).
`( condition1 && condition2 )` forms one larger condition that is only true if both `condition1` *and* `condition2` are true.
`( condition1 || condition2 )` forms one larger condition that is true if either `condition1` *or* `condition2` is true.

Control flow: foreach loop

```
foreach index ( array )
  actions
end
```


Are there more elements in array?
true
set index = next array element
actions
false

foreach loops are a type of control flow structure that allows the user to repeat (loop) a sequence of actions for each element in a list/array. Specifically, this loop successively sets the variable `index` to each element in `array`, and then the actions in this block are performed each time.

For example, the script

```
foreach value ( 1 2 3 four eight 11 )
  echo the current value is $value
end
```

sets the variable `value` to be each element in the array successively, and then prints the current value of `value`. It loops six times, and its output is:

```
the current value is 1
the current value is 2
the current value is 3
the current value is four
the current value is eight
the current value is 11
```

Consider that the array doesn't need to be defined in the foreach loop.
For example, in this script

```
set my_files = *
foreach file ( $my_files )
  echo $file is in this directory
end
```

`my_files` is an array that has already been defined and is then looped over.

Finally, loops can be made more sophisticated using the `continue` and `break` commands: rather than simply looping through all the elements in the array and performing the same actions for each, `continue` causes the loop to continue to the next element in the array without any further actions for the current element, while `break` forces the loop to end without any further actions at all.

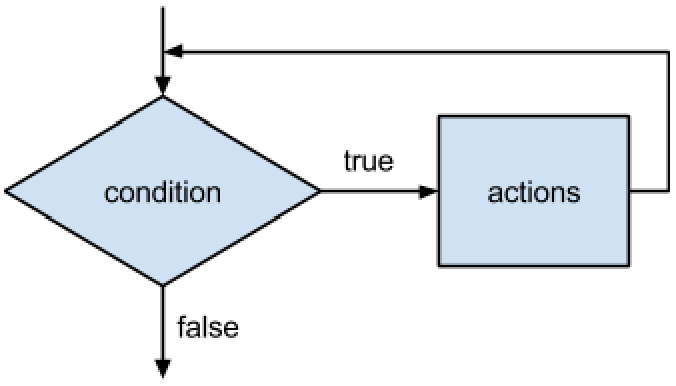
For example, the script

```
foreach arg ( $* )
  if (! -s $arg ) then
     continue
  endif
  sort $arg >> sortedFiles.txt
end
```

loops through all the commandline arguments to the script (`$*`). For each argument, the script checks whether it is the name of a file that exists and is a non-zero size. If not, then the script continues to the next commandline argument (i.e., skips the rest of the lines in the foreach block). If it is, then the script sorts the lines of the file and appends the result to sortedFiles.txt.

Control flow: while loop

```
while ( condition )
  actions
end
```



while loops are another type of looping control flow structure. Unlike foreach loops, while loops are not defined to loop a fixed number of times.
Instead, the while loop checks a condition, and if the condition is true, it performs a series of actions. Once these actions have been performed, it checks the condition again. If the condition remains true, the actions in the while-block are performed again. This is repeated indefinitely until the condition becomes false.

As with foreach loops, `break` and `continue` may be used to terminate or continue the loop.

For example, the script

```
set value = $1
set fact = 1
while ( $value > 0 )
  @ fact = $fact * $value
  @ value -= 1
end
echo $1 factorial is $fact
```

calculates the factorial of a number given as a commandline argument, by multiplying that value by itself less 1, which is then multiplied by itself less 2, etc.
This is a task well-suited to a while loop because the number of times that the multiplication and subtraction are performed depends on the number given as input - that is, when writing the script, we don't know how many times we will need to loop. But we know at what point we should stop the multiplication, which becomes our condition.

Basic UNIX Command Reference

Command	**Description**	**Usage (with some options)**
`pwd` *Print Working Directory*	**“Where am I?”** Display the current working directory (current location within the file system).	`pwd`

`ls`
List
	“What’s here?”
	List the contents of a directory.
	● With no options, list the contents of the current directory.
	● With a filename, list the file and any information about the file.
	● With a directory, list the contents of that directory.
	● With option `-a`, list *all* files in the directory, including files whose name starts with ‘.’ (hidden files).
	● With option -l, list files in the *long* format, which includes information such as the file size, file permissions, and the time that the file was last modified.
		`ls`
		`ls -a`
		`ls -l`
		`ls` *`filename`*
		`ls` *`directory`*

`cd`
Change Directory
	Change the current working directory (current location within the file system).
	● With no options, change to the home directory.
	● With a directory, change to that directory. This can be either a relative or absolute path to the new working directory.
		`cd`
		`cd` *`directory`*

`mkdir` *Make Directory*	Create a new directory.	`mkdir` *`directory`*

`mv`
Move
	Move a file or directory, or rename a file or directory.
	● With one or more source names of files or directories, and (as the last argument) a directory name that already exists, move all the files and/or directories into that directory.
	● With a directory or file that already exists (as the first argument), and a directory or file name that doesn’t exist (as the second argument), rename the file or directory.
		`mv` *`source`* `[...]` *`directory`*
		`mv` *`old_name new_name`*

`rmdir` *Remove Directory*	Delete an empty directory.	`rmdir` *`directory`*

`rm`
Remove
	Delete (one or more) files.
	● With the option `-r`, remove directories (and all their contents) as well as files.
	N.B. Once deleted, a file (or directory) cannot be easily recovered.
		`rm` *`filename`* `[...]`
		`rm -r` *`filename`* `[...]`

`man` *Manual*	Display the manual page that relates to the keyword (such as the name of another command). To navigate within the manual, use the arrow keys. To exit the manual, press `q`.	`man` *`keyword`*

`nano`	Open the nano command-line text editor (and create a new, unnamed file).
	● With a new filename, create a new file with that name.
	● With an existing filename, open that file.
		`nano`
		`nano` *`new_filename`*
		`nano` *`filename`*

`cat`
Concatenate Files
	"What's in this file?"
	Print the contents of a file (as though it were a plain-text file).
	● With a filename, print the contents of the file.
	● With two or more filenames, print the contents of the files one after another.
		`cat` *`filename`*
		`cat` *`file1 file2`* `[...]`

`less`	View the contents of a file (as though it were a plain-text file). To navigate within the file, you can use the arrow keys. To exit the program, press `q`.	`less` *`filename`*

`cp`
Copy
	Create a copy of a file(s).
	● With one or more files as the source arguments, and a directory as the final argument, copy the file(s) into the directory.
	● With an existing file as the first argument, and a new filename as the second argument, copy the file and rename the copy.
	● With the option `-r`, copy directories (and their contents) as well as files.
		`cp` *`filename`* `[...]` *`directory`*
		`cp` *`filename directory/new_name`*
		`cp -r` *`source`* `[...]` *`directory`*

`date`	Display the current system date and time.	`date`

`echo`	Print a string.
	● With the option `-n`, do not print a newline at the end of the output.
		`echo` *`string`*
		`echo -n` *`string`*

`alias`	Replace any use of the command *command* with the *replacement* string.
	`alias ll 'ls -l'` is a common alias which allows the user to use `ll` as an abbreviation of `ls -l`.
	● With no options, print the list of aliases that have already been defined in this shell session.
		`alias` *`command replacement`*
		`alias`

`wc`
Word Count
	Display a word count for a file(s).
	● With no options, print the number of lines, words, and bytes.
	● With option `-l`, print the number of lines.
	● With option `-m`, print the number of characters.
	● With option `-w`, print the number of words.
	● With two or more filenames, print the word count for each file, and then a cumulative word count.
	A line is any group of characters delimited by a newline character.
	A word is any group of characters delimited by whitespace (space, tab, newline, etc.).
		`wc` *`filename`* `[...]`
		`wc -l` *`filename`*
		`wc -m` *`filename`*
		`wc -w` *`filename`*

`head`	Print the first lines of a file(s).
	● With no options, print the first 10 lines.
	● With the option `-n` and an integer `count`, print the first `count` lines.
	● With two or more filenames, print the first lines of *each* file.
		`head` *`filename`* `[...]`
		`head -n` *`count filename`*

`tail`	Display the last 10 lines of a file(s).
	● With no options, print the last 10 lines.
	● With the option `-n` and an integer `count`, print the last `count` lines.
	● With two or more filenames, print the last lines of *each* file.
		`tail` *`filename`* `[...]`
		`tail -n` *`count filename`*

`cut`	Cut out and print columns(s) of a file(s).
	The columns are specified by `list`, a list of numbers and/or ranges (eg. `1, 3-6, 10`). The
		`cut -c` *`list filename`* `[...]`
		`cut -f` *`list filename`*
		`cut -f` *`list`* `-d` *`delimiter filename`*

	ranges are inclusive. If a range has no end (eg. `5-`), it continues to the end of the line. ● With option `-c`, print character(s) at the line position(s) in `list`. ● With option `-f`, print column(s) that are separated by the tab character. ● With option `-f`, `-d`, and a `delimiter` character (eg. space), print column(s) that are separated by the delimiter.	

`sort`	Sort lines of a file(s).
	● With option `-o` and a filename `new_file`, output the sorted result to `new_file`.
	● With option `-c`, check whether the file is already sorted.
	● With option `-b`, ignore leading whitespace.
	● With option `-d`, consider only blanks and alphanumeric characters (eg. not quotes).
	● With option `-f`, ignore case (lower case vs. upper case).
	● With option `-r`, reverse the sort order.
	● With two or more filenames, sort the lines of the files concatenated together.
		`sort` *`filename`* `[...]`
		`sort -o` *`new_file filename`*
		`sort -c` *`filename`*
		`sort -b` *`filename`*
		`sort -d` *`filename`*
		`sort -f` *`filename`*
		`sort -r` *`filename`*

Practice Exercises

Adapted from Alan J. Rosenthal's CSCB09 tutorials
http://mathlab.utsc.utoronto.ca/courses/cscb09w14/tut/
& Bianca Shroeder's CSCB09 labs
http://www.cs.toronto.edu/~bianca/cscb09w13/posted_labs/

Basic Unix commands

Write a Unix command to produce this output:

1. The number of lines in /etc/passwd. (The file "/etc/passwd" lists all user accounts on the system, one per line.)
2. The count of how many users have an 'e' in their user information somewhere (anywhere in their line in /etc/passwd).
3. The contents of the file named: &@$%*
4. The alphabetically-first three lines in the file "abcdef".
5. All file names in the current directory which contain a 'b'.
6. All lines of the file "abcdef" in the current directory which contain a capital 'E'.

Searching for files

Write a "glob" expression which matches all filenames in the current directory with the following properties:

1. filenames beginning with an 'a'
2. filenames ending with a 'c'
3. filenames containing at least one 'x', anywhere in the filename
4. filenames containing one or more digits, anywhere in the file name
5. filenames which end with a dot and then exactly three further characters
6. Given the following directory contents:
   `a.c   abcdef  b.c  b.x   bac`
   For a given triple below, write a glob expression which matches those three file names and not the other two. (Note that one of these is impossible)
   - `a.c, b.c,` and `b.x`
   - `a.c, b.c,` and `bac`
   - `b.c, b.x,` and `bac`
   - `a.c, abcdef,` and `b.c`
   - `a.c, abcdef,` and `b.x`

Unix pipes

The file "hockey_stats.txt" contains 6 columns separated by a single space: (in order) first name of the player, last name, team, position, games played, and total goals.
Create a pipe to perform this task/answer this question based on "hockey_stats.txt":

1. Print out the names of all players sorted alphabetically by first name.
2. How many Toronto players are among the top 10 scorers?
3. Print only lines 10-20 of the file.

4. What is the position of the player with the largest number of games played?
5. How many different teams have players that are listed in the file?

Unix scripting

Write a Unix script to perform this task:

1. Print the message “Hello world”.
2. Add three to the number in the variable 'x' and print the sum.
3. Print the file "file1", or give an appropriate error message if it doesn't exist (hint: a command to print the file won’t succeed if it doesn’t exist).
4. Run the `foo` command. If it succeeds, also run `bar`. (Otherwise, we're done.)
5. For an existing variable 'x', compute $x^2 + 3x + 4$.
6. Get two integers from the command line (as inputs to the program) and add them.
7. Use a for-each loop to print the following four things: the value of the environmental PATH variable, the number of commandline arguments passed to your script, the path to your home directory and the output of the command `pwd`.
8. Use a for-each loop that loops over the commandline arguments passed to your script and outputs the number of characters in each of the command line arguments. (Hint: you can use a pipe and the `wc` command.)