1 What is a Shell?

As mentioned in the first article of this series, the heart of the UNIX operating system is the kernel. Users generally don't access the kernel directly, but rather communicate with the kernel through a program known as a shell or command interpreter. For example, the prompt displayed on your screen is displayed by the shell; and when you type a command such as `ls` or `mail`, it is the shell that sees the command and responds. A shell is just a program like any other program, but it does perform a variety of functions. There are many different shells which are in use in the UNIX world. Most users' needs will be well met by whatever default shell is provided by the system they use, but each shell has certain features that may make it more attractive to some users. Some of the shells which are widely used include the Bourne shell (`sh`), the C-shell (`csh`), the modified C-shell (`tcsh` and the Korn shell (`ksh`). Although specifics differ, there are some general services which all shells provide as well as special features which exist in only some of the shells. In the next few sections, I'll present an overview of some these features. As with every command in UNIX, the UNIX `man` command is the place to go for more details.

To find out the name of the shell you are using, you can use the `finger` command, giving your login id as an argument. There should be a line which displays what shell you will get by default when you log into your system. You can change shells by typing the name of the shell you desire. If it is available on your system, you're commands will then be interpreted by that shell, and, for example, you may notice that the prompt you see has changed. To change back to your original shell, type Control-d.

2 Search Path

One of the basic functions that the shell performs is to find the commands you ask it to execute. Every command in UNIX is stored in a file, and the shell keeps a list, known as a search path, of directories in which to search for commands. You can see the directories your shell is using by typing `echo $PATH` or `echo $path`, depending on which shell you are using. When you type a command like `ls` (which displays the names of files in your current directory), the shell looks in each of the directories contained in the search path, and executes the first occurrence of a file called `ls` which it encounters. There is one exception to this rule, however. Some commands are built in to the shell, that is, the shell recognizes them and executes them without resorting to the search path. To determine exactly what is being executed when you type a particular command name, you can use the UNIX command `which`. You can override the search path by giving a fully qualified file name for a command, that is, one which contains one or more slashes (`/`).

3 Filename Abbreviations

Certain characters, sometimes known as magic characters, have special meanings to all the shells, and are expanded to these special meanings before being passed to the program which you invoke through the shell. A filename containing a magic character is sometimes called a wildcard, and the expansion process is referred to as wildcard expansion. It is important to realize that it is the shell which does wildcard expansion, not the individual programs, so that wildcard expansion is available for all programs regardless of their source. In addition, no program ever sees the magic characters you type, unless they are preceded by a backslash (\), or surrounded in single quotes, in which case they lose their "magic" properties. The table below shows some of the magic characters and how they are expanded by the shell.

Character	Meaning
`*`	anything
`?`	a single character
$[c_1\text{-}c_2]$	range of characters
$[c_1c_2\ldots]$	characters within brackets

When you submit a command containing a wildcard to the shell, the magic characters are expanded to match the names of existing files which contain the patterns specified by the wildcard. For example, suppose the following files exist in your current directory:

```
data.old        data3           goodprogram.c
data1           file.c          program.c
data2           firstdata       prog1.c
```

If you wished to use the UNIX program `wc` to count the lines, words and characters in all your C programs (that is, files with the suffix `.c`), you could use the command `wc *.c`. What the `wc` program will actually see in this case is the list of files which match the wildcard, namely `file.c`, `goodprogram.c` `program.c`, and `prog1.c`. To match files with the word data followed by a number, you could use `data[0-9]`; this would match `data1`, `data2` and `data3`, but not `data.old`. To match all files with the word data anywhere in the name, you could use `*data*`, and so on. It bears repeating that the shell expands the wildcard, not the individual programs, so no program ever knows that it was called with a wildcard; the programs simply receive the list of file names which match the wildcard you submit to the shell.

One other special filename character deserves mention. In the C-shell, the tilde (~) is recognized as referring to a home directory. If you use the tilde alone, it refers to your home directory; following the tilde by a username refers to that user's home directory. So, regardless of your current directory, a filename like `~/myfile` refers to the file `myfile` in your home directory, and a filename like `~fred/somefile` refers to the file `somefile` in the home directory of the user named fred.

4 Redirection

By default, most UNIX programs receive their input from the keyboard, and write their output to the terminal screen. Actually, there are three so-called streams which most programs use: standard input (often called `stdin`), standard output (`stdout`), and standard error (`stderr`). One of the services of the shell is assigning other locations for these three streams; this process is known as redirection. A single greater-than sign (`>`) will send standard output to the file whose name follows it, potentially destroying the file if it already exists. Two greater-than signs (`>>`) will have a similar effect, but will append to the output file rather than destroying it. A less-than sign (`<`) will redirect standard input, so that input to a program will come from a named file, instead of the terminal. Like wildcard expansion, redirection is performed by the shell, so it will work for any program which obeys the UNIX convention of using the three streams described above.

You should realize that the redirection described above does not modify the destination of `stderr`; in particular, error messages will still be displayed to the screen even if `stdout` is redirected. While this is clearly desirable for interactive work, when you use redirection to save a non-interactive job's output to a file, you should always make sure that `stderr` is redirected to the file as well. Methods for redirecting `stderr` vary among the different shells - check your online manual pages for the details for the shell you use.

Another valuable form of redirection is a pipe. A pipe allows you to take the standard output of one command, and use it as the standard input to another command. For example, suppose we wished to use the UNIX `ls` command to put the names of the files in the current directory into a file called `myfiles`. The command to use would be `ls > myfiles`. Now suppose we wish to count the number of files using the UNIX command `wc`. One solution would be to call `wc myfile`. But using pipes, there is no need to even create a file to hold the output; `ls`'s standard output can used directly as standard input to `wc` by using a vertical bar (`|`), which is the symbol for a pipe, as in `ls | wc`. Like wildcard expansion, redirection is carried out by the shell, not the individual programs, but there is one difference which is illustrated by the previous example. If a program wants to, it can tell whether or not input has been redirected, unlike wildcard expansion which is completely transparent. It turns out that `ls` has been written to take special action when its output is redirected to a file or through a pipe, namely that it lists files in a single column, instead of using its multicolumn default. So while redirection will work for any program which obeys the standard input/output conventions of UNIX, redirected output is not guaranteed to be identical to the output you would see displayed on the screen.

In the next article, I'll discuss some more services which the shell provides.