Shells

When you login, Unix presents you with a command line that allows you to execute programs.

The core of Unix itself is actually not much more than a ”library” of code. Programs make calls to routine inside Unix to display information on the terminal, to write to a file, to read from a file, to interact with the network, etc. In fact, the interaction with Unix that you perform on the command line is done through a special program that can read input from the command line, parse it, do I/O redirections, call appropriate commands with appropriate options, etc. It is called a *command shell* (or just *shell* for short).

There are many flavors of shells, split into two families, each family with a different feel, and each member of any given family with different features. The first family is the *Bourne shell family*, with the following shells:

- `sh`, the very basic Bourne shell
- `ksh` the `sh`-compatible Korn shell
- `bash` the GNU Bourne-again shell, with features from both `ksh` and `csh` (see below)

The second family is the *C shell family*, with shells:

- `csh`, the original C shell
- `tcsh`, an extended C shell with more functionality

Typically, when we say that something works for the Bourne shell, it will work for all the shells in the Bourne shell family, and similarly for the C shell.

When you login, the system starts up an initial shell, which is called the *login shell*. You can change which shell is you defauly shell when you login by invoking the `chsh` command at any point.

Since a shell is just a program, you can always switch to a new shell by basically executing this shell. For example, you can start an instance of `ksh` by simply executing it:

```
babbage% ksh
$
```

Since most shells will have different prompts (in my case, my `csh` has a prompt `babbage%`, while `ksh` has a prompt `$`), you can tell in this case that we are in a new shell. We can then issues commands interpreted by this new shell. We will see in the coming lectures some of the differences between the shells. One of them, for example, is I/O redirection, specifically how to handle redirection to an existing file. The Bourne shells will typically overwrite the file, while the C shells will report an error. In a C shell, if you do want to overwrite a file at redirection, you can redirect using the `&!` redirection operator. This operator is not understood by Bourne shells.

What are the roles of the shell? It turns out that the shell is what manages jobs and I/O redirection (cf. Lecture 6). The shells is also in charge of parsing the command line and performing substitutions. This is what we will focus on next.

Shell substitutions

When you enter a line at the shell prompt, before the shell executes the programs that you specify, it will make a pass over the line you entered to perform various kinds of substitutions. Here are the special characters that can be used to guide substitution:

- `*`, `?`, `[...]` these are *wildcard* characters. They are replaced by the appropriate portion of filenames. For instance, the wildcard character `*`, by itself, is replaced by the list of all the files in the current directory. Hence, if you want to search for `foo` in all the files in the current directory, you can type `fgrep 'foo' *`, which the shell will exand into `fgrep 'foo' file1 file2 file3 ...` for all the files in the current directory. You can use wildcards to match part of file names. For instance, `*.txt` will match any filename ending with `.txt`, while `a*b` will match any filename starting with `a` and ending with `b`. Note that these *are not regular expressions*. They use wildcard characters. Another wildcard character is `?`, which can stand for any letter. Hence, `a?c` will be expanded into all the filenames in the current directory that start with `a`, have one character following it and followed by a `b`. The `[abc]` construct can be replaced by any of the characters within the brackets.
- `\` is used to escape the following character. Since some characters have special meaning to the shell, such as `*`, you need to escape them if you want to refer to the actual character. For example, if you have a file called `*` in your current directory, and you want only to search for `foo` in it, then you would need to write `fgrep 'foo' \*`, escaping the special `*` character.
- $*var* where *var* is a shell variables (see next section). This is replaced by the value of the shell variable.

- ‘*command*‘ this is replaced by whatever executing *command* sends to `stdout`.
- `"..."` indicates that within the double-quotes, only variable and command substitutions should occur (i.e., no wildcard expansion).
- `'...'` indicates that within the single-quotes, no substitutions should occur (no wildcard, variable or command substitution). Avoiding substitutions is the reason I told you that the first argument to `grep` (the regular expression) should be enclosed in single-quotes.

As I mentionned, substitutions occur under the hood, after you enter the line at the prompt, and before the shell executes the program specified. How can you tell whether the substitutions is what you were intending? In general, how can you see what substitution is being performed? One trick is to use `echo`. Recall that `echo` simply sends its arguments to `stdout`. Hence, if the command line contains substitutions, `echo` will send the result of the substitutions to `stdout`. For example:

```
babbage% ls
2000FA    HW1       bin       foo       netids    share     test.sh~
2001SP    HW2       f         man       netids~   test.sh   tmp
babbage% echo H*
HW1 HW2
babbage% echo $HOME
/home/cs114
babbage% cat f1
this is a line
babbage% cat f2
H* $HOME
babbage% echo ‘cat f1‘
this is a line
babbage% echo ‘cat f2‘
HW1 HW2 $HOME
```

We see that the result of command substitution is further substituted, but only wildcard expansion is performed. Understanding substitution is slightly tricky when you get to more complicated examples, and some shells differ in such handling. Look into the man pages of the shell that interests you.