Bash Interpreter

Contents

Overview

In this module, you will learn about the Bash interpreter, and how the shell can read user input and interpret it within the command line.

The Shell

The Linux OS comes with a **shell**, which provides the user with an interface to the system - this is done through writing and running commands through a *Command Line Interface* (CLI).

There are in fact a number of different shells, such as Bourne, C, and Korn. All of these shells understand and interpret commands for the OS.

The most popular shell is **Bash**, or Bourne-Again Shell, which actually merges the capability of all three of the aforementioned shells.

Reading Commands

When we type commands into the CLI, we can continue the line for as long as we want. When we have finished, we can hit the carriage-return character (`cr`) (typically the enter key) to then run the command.

The carriage-return character is one of multiple special characters recognised by the OS' shell. Once the shell receives the (`cr`), it will *interpret* that as the end of the line and start to run, or parse, the line.

There are a number of special characters, known as metacharacters, the shell can interpret. Lines are read from left to right. Any metacharacter will be actioned as soon as it is read and subsequently substituted back into the line.

For example, let's look at the command below

```
$ ls -l $HOME /tmp/* <CR>
```

There are a number of metacharacters here; `$`, `*` and `<CR>`. Each of these will be read from left to right and actioned as soon as they are recognised from the shell (all of these will be discussed further into the module).

Shell Metacharacters

Line interpretation in the shell relies on special characters. Alongside `<CR>`, other metacharacters include:

- Whitespace (space key): used to tokenise the line (separate words)

- Wildcards (*): (globbing) characters used by programs to group content together
- Quotations (both ' and "): used alongside a metacharacter to tell the shell to interpret it literally, rather than a special character
- Escape Chars (\): used to interpret special chars literally such as when using URIs
- Redirection (<, >, |): used to control the source and destination for command input and output

There are several other characters also recognised by shells; for example

- ; (semicolon) allows separate commands on a single line
- & (ampersand) forks the preceding process into the background (used for chaining independent asynchronous processes)
- &&(AND) logical AND for chaining synchronous processes in sequence so that the second only executes if the first succeeds

Globbing

The wildcard character (*) is used for file globbing, or grouping of files based on filename.

The shell will match the “wildcard” pattern against the filenames at the appropriate place in the file system. For example, look at the file glob below:

```
my*
```

The shell checks against files in the current directory and would then take the input based on what is found. This may result in multiple files such as:

```
my myfile1 myfile2 myprog
```

Ignoring Metacharacters

There will be times where we run a command and need the literal character of a metacharacter - we need to tell the shell that this is the case when it interprets the line.

```
echo *.c
```

There are a few ways we can do this:

- Back stroke (\) - prevents the shell from recognising the next character as a metacharacter and often used to spread a long command over multiple lines
- Single quote (') - prevents all characters in between the single quotes as being recognised as metacharacters
- Double quote (") - prevents all characters in between the single quotes as being recognised as metacharacters besides back strokes, single quotes and dollar signs (used for command substitution)

```
echo '*.c'
echo \*.c
echo "*.c"
```

Variables and Command Substitution

The shell can assign values to variables, and then call these variables with the $ symbol.

```
Log_File=mylog1
echo ${Log_File}
```

Note the use of curled braces for the variable expansion

The above echo command is an example of **variable substitution** - calling the variable into the command with the $ sign and putting it in a standard command.

We can also use **command expansion**, which allows us to use a variable to call a command to be ran.

```
version=$(uname -r)
echo $version
```

Note the use of regular braces for the command expansion

Tab Completion

While typing in the command line it is important to be careful as typos can have unintended consequences, for this reason it's useful to rely upon the shell's ability to `tab complete`. To do this simply press the `tab` key:

- at the highest level (i.e. with no other input) pressing double tab will output everything on your PATH: directories, filenames and binaries
- as soon as you add input you begin to filter the results, and a single press of the tab key will autocomplete the input (a double tab will output a filtered list of suggestions from your PATH)

Tab completion is invaluable in navigating the directory structure competently, quickly and accurately typing commands. It is also possible to extend the tab completion functionality of the shell in order to provide suggestions not just from the PATH but from a particular command's arguments.

History

Each shell has a history that can be seen using the `history` command:

```
$ history
    1  ls
    2  cd /tmp
    3  ls -lah
```

The history can be navigated using the up and down keys.

It is possible to search this history using `reverse-i-search` (**THE MOST USEFUL**) function of the shell. By using `CTRL+r` on the command line the input will change to accept a search string that will automatically search and return the first match from the shell history:

```
$ history
    1  ls
    2  mkdir Repos
    3  cd Repos/
    4  mkdir QA
    ....
   46  git status
   47  git add .
   48  git commit -m "corrected some typos and cleared up a couple points"

$
(reverse-i-search)`comm': git commit -m "corrected some typos and cleared up a
couple points"
```

This can be a huge time saving feature as you cannot be expected to remember everything, and it is better to rely on automation where possible.

Tutorial

File globbing needs a pattern alongside a search. The *pattern* stands for the search string you will type, for example, for filenames beginning with m, type:

```
echo m*
```

OR

```
ls -d m*
```

Run the above commands to see how file globbing would work in your shell.

Exercises

To test your patterns, use one of the commands:
echo *pattern* OR ls -d *pattern>*

where *pattern* stands for the search string you will type, for example, for filenames beginning with m, type:
echo m* OR ls –d m*

Change to the /etc directory, and display files whose names:

- Begin with p
- End with y
- Begin with m and end with d
- Begin with either e or g or m
- Contain an o followed (not necessarily immediately) by a p
- Contain the string conf
- Begin with s and contain an n
- Contain exactly 4 characters
- Contain a digit anywhere in the filename