COMP 2718: Command Line Expansion

By: Dr. Andrew Vardy

Adapted from the notes of Dr. Rod Byrne

Outline

- Expansion
- Arithmetic Expansion
- Brace Expansion
- Parameter Expansion
- Command Substitution
- Double Quotes
- Single Quotes

Expansion

The globbing that we saw in "The File System: Part 3" was a type of **expansion**. The shell performs many times of expansion which transform the text input to whatever command is being executed. For example:

```
$ ls
012.bin  345.bin  abc.txt  def.txt

$ echo *.txt ~/.bashrc
abc.txt def.txt /home/av/.bashrc
```

The `echo` command never sees the * nor the ~ characters. The shell expands them to the appropriate pathnames filenames.

This (now familiar) form of expansion is called **pathname expansion**. But there are several other forms.

Arithmetic Expansion

The shell can calculate arithmetic expressions using this form:

```
$((expression))
```

where the expression consists of arithmetic operations on integers (floating point numbers are not supported). The following operators are available:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division (but remember, since expansion only supports integer arithmetic, results are integers.)
%	Modulo, which simply means, " remainder."
**	Exponentiation

Examples: All of the following yield 4 as the result:

```
$ echo $((2 + 2))

$ echo $((2**2))

$ echo $((2**1 + (1+1))

$ echo $(((10 % 4) + 2))
```

Brace Expansion

Curly braces contain list of possible parts of a string which are expanded by the shell. e.g.

```
$ echo Front-{A,B,C}-Back
Front-A-Back Front-B-Back Front-C-Back
```

The brace expression can contain the following:

- Comma-separated list of strings (example above)
- Range of integers or characters of the form `START..STOP`

Examples:

```
$ echo Number_{1..5}
Number_1 Number_2 Number_3 Number_4 Number_5
```

Using **zero-padding** to maintain the number of digits used:

```
$ echo {001..12}
001 002 003 004 005 006 007 008 009 010 011 012
```

A range of characters in forward, then reverse order:

```
$ echo {A..C} {C..A}
A B C C B A
```

This example generates all possible Canadian postal codes. However, don't try this one as it will take too long to complete:

```
$ echo {A..Z}{0..9}{A..Z}_{0..9}{A..Z}{0..9}
```

How many different strings can this generate?

$$26^3 * 10^3 = 17,576,000$$

Actually, there are some finicky rules that reduce this to about 7.2 million for postal codes that might actually get used.

Parameter Expansion

We will discuss this more later, but the shell maintains variables—sometimes called **parameters**. To access existing parameters, such as `USER` prefix the variable name with '$'. e.g.

```
$ echo $USER
```

Use the following to browse through available parameters in sorted order:

```
$ printenv | sort | less
```

Command Substitution

Command substitution allows us to embed the output of a command into the command line itself.

`$(command)`

```
$ echo $(ls)
```

Of what use is this? Sometimes the arguments to a command need to be determined by another command. For example, to show the attributes of the `cp` command we can do the following:

```
$ ls -l $(which cp)
```

Could we use a pipeline instead?

```
$ which cp | ls -l
```

This doesn't work. Why not? Because `ls` does not accept standard input. See this link for discussion:
`http://unix.stackexchange.com/questions/140522/why-do-some-commands-not-read-from-their-standard-input`

We can wrap any command within a substitution, including entire pipelines. e.g.

```
$ file $(ls -d /usr/bin/* | grep zip)
```

Further examples:

Log an event with a time stamp:

```
$ echo "Started work at " $(date) >> ~/worklog
$ echo "Stopped work at " $(date) >> ~/worklog
```

Create a filename using the time:

```
$ touch $(date +%T).txt
```

Remove the oldest 10 files (dangerous!):

```
$ rm -i $(ls -tr | head -10)
```

Back-quote Syntax

Older shell programs use *back-quotes* instead of the dollar sign and parenthesis:

```
$ ls -l `which cp`
```

`bash` still supports this syntax.

Double Quotes

We saw in "Command Line Parsing" that quotes can be used to enclose desired white space...

```
$ echo "...like        this."
```

Double quotes take away the special meaning of some characters, but not all of them. Therefore they suppress the following:

- Pathname expansion (i.e. globbing/wildcards)
- Tilde expansion (~)
- Brace expansion

However, $, \, and ‘ are still active inside double quotes, so the following expansions are **not suppressed**:

- Parameter expansion
- Arithmetic expansion
- Command substitution

Notice how parameter expansion, arithmetic expansion, and command substitution still occur within this double-quoted argument:

```
$ echo "$USER is number $((10-9)). $(cal)"
av is number 1.    February 2016
Su Mo Tu We Th Fr Sa
    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
```

Single Quotes

To suppress all expansions, use single quotes:

```
$ echo '$USER is number $((10-9)). $(cal)'
$USER is number $((10-9)). $(cal)
```

If you just want to suppress the special meaning of special characters within double-quoted or unquoted strings, use \ to exape them:

```
$ echo "I really meant to say \$USER."
I really meant to say $USER.
```