Introduction to Shell Expansions

When you are working with the bash shell, you may have noticed that some of the things you want to do might require 2 or 3 additional steps. You might need to make a series of directories or you need to search for a variety of files holding various terms. Often you may have wondered "Is there a better way to do these operations instead of writing 2 or more separate command line calls?" And I am here to tell you it is possible!

One of the most important aspects of bash scripting that you should familiarize yourself with is the power of shell expansion/substitution, which allows you condense commonly accessed, arbitrarily generated, and repeated behavior among your commands. With shell expansions, you can condense your multiple commands down to single lines, allowing you to write scripts of condensed power. That said, understand that these capabilities will only extend as far as your imagination will stretch with them. If I gave you a hammer and the materials to make something, you're not going to make something magnificent unless you put some thought into it.

Following the bash manual [1], there are 8 different shell expansions, 1 of which is a special case. As follows, their order of operations goes Brace, Tilde, Parameter/Variable, Command Substitution, Arithmetic, Process Substitution, Word Splitting, and Filename Expansion. A lower order expansion will not take place before a higher order, and the order of operations should not be confused with literal strings (i.e. apostrophes, and quotation marks) order of operations which while it intertwines with expansions, has a separate ruleset to shell expansion.

Contents

Brace Expansion

Starting with the highest form, brace expansion allows for arbitrary strings to be generated based on the information contained within the braces. In other words, specifying a brace and implementing a pattern will cause strings to be generated of said pattern. In example, you can see that a command like `$echo a{b,c,d}e` would return `abe ace ade`. Another example showcases brace expansion to work via `{x..y[..incr]}` where `x` and `y` are either integers or single characters, and `incr`, an optional increment, is an integer. These allow things like `{a..z}` which would produce the entire alphabet in its lowercase, `{a..z..2}` which would skip every other letter in the alphabet, `{1..10}` would count from 1 to 10 and so on and so forth.

Since this is an arbitrary generation of a string, this process reduces overall processing power into its computation, and holds a number of simple yet wide range applications. Passing a command like `$mkdir {toms, zachs, beths}` would make 3 separate directories, or a command like `$for i in {1..10}` would perform a for loop counting from 1 to 10. However it should be noted that `${}` is not regarded as a brace expansion but another way to write shell substitution and has higher authority over `$()` which may be familiar to you if you have taken an understood algebras order of operations.

```
$mkdir {toms, zachs, beths}
$ls
toms   zachs     beths
This command creates 3 separate
directories within 1
```

Tilde Expansion

Tilde Expansion is nothing as critical as brace expansion but it certainly saves time when it comes to accessing files. Typing a command like `$cd ~/my/path/` would be the same as typing `$cd /path/to/home/directory/my/path`. Do note, that unless you have `$HOME` set, this command would instead be treated as your login name. So for example your username would be used within `$cd ~/my/path` as `$cd username/my/path`. Since it's a login name, it will change relative to whomever is logged in using the command.

```
$ls /root/home
Dir1    Dir2    Dir3
$ls ~
Dir1    Dir2    Dir3
This command demonstrates tilde expansion in referencing a users home directory
```

If using login names or home directories doesn't suit you, you can also use tildes to reference your PWD's, or parent working directories. You can reference your PWD through `~+` or `~-`, to use the current working directory, or the last working directory respectively (The directory your shell was previously residing in since last directory change). I will comment that using `~+` is technically the same as typing `./` as the same result of referencing the current directory is achieved. Keep in mind that this conditional may not apply to all Linux distributions.

Parameter Expansion

Parameter expansion works by taking a parameter or variable specified as `${parameter:attribute}'` and evaluating it out over the given attribute. It should be noted this is different from Brace Expansion, as the later instead of generating, *filters* out data from strings. If a variable `string` is put into the form of say `${string:7:2}` would use the part of the `string` that's 7 characters in, and 2 characters out. This also performs the same operation if the passed variable is an array, which if you are familiar with the C family of programming, strings are simply character arrays.

```
$string=1234567
$echo ${string:3:2}
34
This expansion grabs the string 3 characters in and 2 character out
```

If you wanted to be even more clever with your Parameter Expansions, try playing around with wildcards like `*` or `@`. These wildcards allow you specify how many arguments you want to pick up when working with a variable. Each will behave differently under quoted/unquoted circumstances as noted from Stack Overflow [12]. In simple terms, using `*` will grab all arguments as they appear, whereas `@` will treat each valid whitespace as an argument. Both when unquoted have no difference from the other. Additionally, `$@` is referred to as the default list so when working with loops, this will naturally be reference if no other list is provided.

Command Substitution

Command Substitution is arguably one of the most important expansions you can learn in your entire bash career. The expansion works simply by passing $(command) inside of an already present command. A common use case could be something along the lines of $echo hello $(getUserNameProgram) where the results of the command getUserNameProgram get used as input to the parent command of echo. The reason why it is called substitution is because it creates a copy of your current shell and runs sub commands inside that copy of. Since shell execution is asynchronous to a machines run time, multiple subshells can be started and initialized but remember that putting subshell inside of another subshell may create very lengthy commands. If you would prefer a shorthand for writing command substitutions, you can opt for using `command` syntax as it is less characters to write.

```
$string=zachIsMine
$echo $(cat myFile|grep $string)
```
This would output the all instances where the string "zachIsMine" appeared in the file "myFile"

These expansions save the time and syntax of building larger full blown functions as it follows under Linux's primary developing philosophy [11]. Now that command substitution has been covered, there is an additional caveat that should be noted. In your works, you may discover that writing $(sub) can do the same as ${sub}. Be assured, that there is a major difference between these 2 when it comes to quotations and literals. According to the manual, writing something like ${var}sub would grab whatever is stored in var and try to find a variable that starts with var's contents and ends with sub. You should also remember order of expansions still applies here so using $({var}someCmd) will not do the same as ${var}sub do to some interpretation hijinks. Play around with the commands and literals to see if you can accomplish your idea.

Arithmetic Expansion

Arithmetic expansion, for serving all your arithmetical means! Arithmetical Expressions require the form $((expression)) in order to do any arithmetical operations. Simply put, writing things such as "5 + 3" will not evaluate out to "8" because the shell has not been instructed to. It is a case of where bash is picky with its literals, but for good reason as these would otherwise be interpreted as strings from the get go. This is a design choice of the Linux operating system that I will not digress into as it will be a topic for another time.

Word Splitting

Word splitting allows for the termination of earlier expansions via a registered whitespace character stored in the variable `$IFS`. This is to prevent a command like `$echo hey (zach)` being interpreted as `$echohey(zach)` or any other unpredictable nature. This expansion, which not only helps to take care of things like terminating characters, also handles the special cases with the null inputs. For example, passing `“”` or `‘’` will create empty strings whereas word split expansions on null variables will simply be removed altogether. This is important to note as there may be moments where you may need to generate empty strings or remove input altogether as

Filename Expansion

File Expansion enables the referencing of files via a wildcard expansion. In example, the use of `*` in `$ls ./my/path/*` would aptly list all applicable files under defined directories. In Linux, everything is treated as a file, and the use of extensions is entirely semantic, however each extension triggers how to read the data differently should a program need to. It should be noted that this expansion allows for pattern matching to find values, thanks to its uses of wildcards, allowing you to find files even if you do not directly know their names in the first place.

However, file expansion matching does not work the same as a GNU regular expression (grep) as it is significantly less sophisticated than its grep counterpart. The wildcards will roughly follow as `*` pulls all files, `?` Matches to any ASCII character, and `[]` specifies a range of characters for a scripter to use. Additionally the `[]` characters allow users to specify commonly used whitespace characters (things like spaces, colons, semi-colons, and so on) as `[:character:]` in the event that such delimiters are misinterpreted as something else for other commands.

Special Case - Process Substitution

Process substitution occurs by using the results of one program as the input for another. For instance, use of *‘<(list)’* causes the contained lists output to be used as input for an earlier command and *‘>(list)’* vice versa. This is considered different to shell substitution in that this is a redirection schema coupled with a programs results that does **not** use an instance of the current shell to run in, but rather on an asynchronous action. Additionally this is only supported on file systems that support the use of piping commands, or FIFO’s, therefore the use of this expansion is a special case.

Conclusions

While these explanations hopefully serve to you as a more approachable form of the given documentation on shell expansion, it cannot be stressed enough that these are tools that when used correctly can make or break most script programming. As demonstrated with shell expansions like shell substitution, what you can actually do with them is based on how creative you can get with them. Testing the limits of the shell expansions and your mastery of them can lead to some interesting results and give you more control over your Linux experience.

References

[1] C. Ramey and B. Fox, Bash Reference Manual, 5th ed. GNU Free Software Organization, 2019, pp. 22-34.

[2] R. Moore, "How to read output of sed into a variable", Stack Overflow, 2019. [Online]. Available: https://stackoverflow.com/questions/4014074/how-to-read-output-of-sed-into-a-variable?noredirect=1 . [Accessed: 11- Jan- 2019].

[3] "Extract filename and extension in Bash", Stack Overflow, 2009. [Online]. Available: https://stackoverflow.com/questions/965053/extract-filename-and-extension-in-bash/965072?r=SearchResults#965072 . [Accessed: 11- Jan- 2019].

[4] K. Geles, "Use find to identify filename same as the parent directory name", Stack Overflow, 2019. [Online]. Available: https://stackoverflow.com/questions/54108699/use-find-to-identify-filename-same-as-the-parent-directory-name/54108856?r=SearchResults#54108856 . [Accessed: 11- Jan- 2019].

[5] C. Johnson and J. Varma, Pro Bash programming, 2nd ed. Apress, 2015, p. Chapter 4.

[6] C. Negus, Linux bible, 9th ed. Indianapolis, Indiana: John Wiley & Sons, Inc., 2015.

[7] C. Brad, "Why you Should try Linux Today: 6 compelling reasons", PCWorld, no. 12, pp. 157-167, 2016.

[8] C. Saran, "How Linux has Influenced Modern-day IT", Computer Weekly, pp. 7-9, 2016.

[9] "Download Linux", Linux.org, 2019. [Online]. Available: https://www.linux.org/pages/download/ . [Accessed: 12- Jan- 2019].

[10] "DistroWatch.com: Put the fun back into computing. Use Linux, BSD.", Distrowatch.com, 2019. [Online]. Available: https://distrowatch.com/index.php?dataspan=52 . [Accessed: 12- Jan- 2019].

[11] "Basics of the Unix Philosophy", Catb.org, 2019. [Online]. Available: http://www.catb.org/~esr/writings/taoup/html/ch01s06.html . [Accessed: 17- Jan- 2019].

[12] g. jackman, "Accessing bash command line args $@ vs $*", Stack Overflow, 2012. [Online]. Available: https://stackoverflow.com/questions/12314451/accessing-bash-command-line-args-vs . [Accessed: 17- Jan- 2019].