

*Bash* has many features that makes it an attractive interactive environment. We will not talk about those in this course. Rather, we will dive into the scripting aspects. Scripting essentially amounts to writing shell commands that are executed automatically by Unix. This forces us to look at the execution of individual commands.

## Fundamentals of scripting

A script is just a “program” made up of shell comamnds. A shell script is a text file with the following characteristics:

- its first line is *#!/bin/bash*, indicating the shell to use to execute the script,
- it has execute permission.

(It is also possible to execute the script by invoking *bash* on the script, as in *bash scriptfile*.) A script is executed in a subshell.

To exit a script with a specific exit code *n*, use the command *exit n*.

When a script is invoked, the shells sets a number of special variables to hold the arguments passed to the script. The name of the script is held in variable *\$0*. The arguments to the script are held in variables *\$1*, *\$2*, *\$3*, and so on. For argument number 10 and above, use braces (*\${10}*). These variables are called positional parameters. The special variable *\$\** holds the list of arguments. The special variable *\$#* holds the number of arguments passed to the shell. (If no arguments are passed, *\$#* is null.)

Positional parameters cannot be set the way normal variables can. To reset the value of positional parameters, you use the *set* command: the command *set word1 word2 word3* resets the positional parameters as though *word1*, *word2* and *word3* had been passed as arguments to the script. Here is an example. Assume you have a script, invoked with arguments *foo* and *bar*, and the script contains the following lines:

```
echo $2           outputs bar
oldargs=$*       oldargs is "foo bar"
set tarzan jane
echo $2           outputs jane
```

```
set $oldargs      resets original arguments
echo $2          outputs bar
```

## Shell expansions

Unix itself can only execute commands when presented in the form of a sequence of tokens. (Each token can be thought of as a string of characters). The first token represent the command to executed, the subsequent tokens are the arguments to the command. Each command is free to interpret its arguments whatever way it wants.

The main job of the shell is translate a string of characters input by the user (or given by a line of a shell script) into a sequence of tokens for Unix to execute. This process happens in the following order, and I will describe most of them in the remainder of the lecture.

1. brace expansion
2. tilde expansion
3. parameter expansion
4. variable substitution
5. command substitution
6. arithmetic substitution
7. word splitting
8. pathname expansion

For sake of discussion, we will call a sequence of characters separated by spaces a word.

Brace expansion is the process of expanding every word containing a brace expression, of the form  $\{w1,w2,w3\}$ , where each  $w1, w2, w3$  are words (without any space), into a sequence of words where the brace expression is replaced by  $w1, w2, w3$ , respectively. For example,  $abc\{de,fg,hi\}$  expands into the sequence of words  $abcde abcfg abchi$ .

Tilde expansion expands every  $\sim$  into the path to your home directory. The form  $\sim name$  expands into the path to the home directory of user  $name$ .

Variable substitution and parameter expansion substitute the value of variables. Variable substitution replaces every expression  $\$var$  by the value of variable  $var$ . Parameter expansion is a kind of conditional substitution. There are many variations of parameter expansion. The most common

one is to replace every expression of the form  $\${var:-word}$  either by the value of variable *var* if *var* is set and non-null, or by *word* if *var* does not exist, or is null-valued.

Command substitution replaces every expression of the form  $\$(cmd)$ , where *cmd* is a command, by the output of the execution of *cmd*. Hence,  $\$(pwd)$  is replaced by the output of *pwd*, that is, the current working directory.

Arithmetic substitution allows you to perform numerical computations in the shell, instead of using a program such as *bc*. We will return to arithmetic expressions in later lectures.

Word splitting is the process of actually splitting the command line into tokens, at the spaces. Hence, if a substitution occurring earlier in the process substitutes a string with spaces, for example,  $\$FOO$  where variable *FOO* has value *some word*, then that value will be split into two tokens at this step.

Finally, pathname expansion is the process of replacing paths containing wildcard characters (such as *\** and *?*) by the sequence of paths that match the pattern. Recall that *\** matches one or more characters, *?* matches exactly one character,  $[abc]$  matches any character between the brackets (here, *a*, *b*, *c*), and  $[!abc]$  matches any character not in the brackets. (This process is also known as globbing.) Each filename in an expansion is taken as a token. Note that this step happens after word splitting, so that if there are spaces in filenames, each filename is still considered a token.

Sometimes, we want to disable some aspects of this automatic expansion by the shell. By and large, this process is known as quoting. Quoting can take many forms. The simplest form is simply to escape the special characters that have meaning to the shell. For instance, you may want to use the *\$* character without having it interpreted as variable, parameter or command substitution. Similarly for the *&*, *,*, *?*, *\**,  $[$ ,  $]$ ,  $\{$ ,  $\}$  characters. To use such a special character as a literal character, you precede it with a backslash. (This is called escaping a character.) Since a backslash is itself a special character, if you want a literal backslash, you need to escape it as well.

Two alternate forms of quoting exist. The form *'text'* disables any form of expansion between the single quotes, including word splitting. Hence, a line *cmd 'word1 word2'* will split the line into *cmd* and *word1 word2*. The form *"text"* disables expansion between the double quotes, except for variable and command substitution. Hence, *"\*\$foo"* will expand to *\*bar* if variable *foo* has value *bar*. As with single quotes, double quotes also disables word splitting.