Minimalistic guide to shell scripting syntax

In addition to being an interpreter of user commands, shell also provides a high-level programming language.

#! Specifying a Shell

You can put a special sequence of characters on the first line of a file to tell the operating system which shell should execute the file. Because the operating system checks the initial characters of a program before attempting to exec it, these characters save the system from making an unsuccessful attempt. If # ! are the first two characters of a script, the system interprets the characters that follow as the absolute pathname of the utility that should execute the script. This can be the pathname of any program, not just a shell. The following example specifies that bash should run the script:

```
#!/bin/bash
```

Invoking other programs

In the script you invoke different programs as you would invoke from the shell. For example, echo utility takes multiple arguments and prints all these arguments to the screen.

In file: *we*

```
#!/bin/bash
echo you and me
```

run: *./we*

Variables

You don't declare variables - you just start using them. An assignment statement begins with the variable name, no space, an equals sign, again no space, and then the value to be assigned. To access the values of variables, we use a command-line substitution which begins with a $.

~~`we=you and me`~~

```
we="you and me"
```

~~`echo we`~~

```
echo $we
```

The `$VARIABLE` syntax is a special case of the more general syntax `${VARIABLE}`, in which the variable name is enclosed by ${}. The braces insulate the variable name. Braces are necessary when concatenating a variable value with a string.

Compare:

```
$ PREF=counter
$ WAY=$PREFclockwise
$ FAKE=$PREFfeit
$ echo $WAY $FAKE
```
Prints empty line, because variables $PREFclockwise and $PREFfeit are not initialized

Vs.

```
$ PREF=counter
$ WAY=${PREF}clockwise
$ FAKE=${PREF}feit
$ echo $WAY $FAKE
```
Prints: counterclockwise counterfeit

Substitutions

Shell is scanning for special tokens and substitutes them. "`echo ~`" – the "~" is substituted with your home directory path name.

Some special tokens:

> #output redirection

< #input redirection

HOME

PATH

***, ?** #file name wildcards

| #pipe

How to suppress substitutions

- backslashes suppress the special interpretation of an immediately-following character
- single quotes suppress the interpretation of just about everything inside them, except for other single quotes (that is, there's no way to embed a single quote within single-quoted text)
- double quotes suppress the interpretation of most things inside them. The exceptions are backquotes, backslashes (so you can put a double quote inside double-quoted text), and dollar signs

```
person=alex
echo $person # alex
echo "$person" # alex
echo '$person' # $person
echo \$person # $person
```

Command Substitutions

Command substitution `$(command)` replaces a command with the output of that command.

```
echo $( ls -l )
```

Expressions

An expression is composed of constants, variables, and operators. bash accepts **((**expression**))** as a synonym for *let "expression"* and *expr* with arguments, obviating the need for both quotation marks and dollar signs.

Arithmetic expansion

Arithmetic expansion uses the syntax **`$((expression))`**, evaluates expression, and replaces $((expression)) with the result. You can use $((expression)) as an argument to a command or in place of any numeric value on a command line. You can use arithmetic expansion to display the value of an expression or to assign that value to a variable.

```
echo There are $((60*60*24*365)) seconds in a non-leap year.
```

Arithmetic evaluation

Arithmetic evaluation uses the **`((expression))`** syntax, evaluates expression, and returns a status code. You can use arithmetic evaluation to perform a logical comparison or an assignment.

```
((COUNT = COUNT + 1, VALUE=VALUE*10 +NEW))
```

Logical Evaluation (Conditional Expressions)

The syntax of a conditional expression is **`[[ expression ]]`** where expression is a Boolean (logical) expression. The result of executing this builtin, like the *test* builtin, is a return status. You must surround the [[and]] tokens with whitespace, and place dollar signs before the variables.

While loop

```
echo “Enter password”
read trythis
while [[ “$trythis” != “secret” ]]
do
      echo “Sorry, try again”
      read trythis
done
exit 0
```

Lists

Bash supports one-dimensional array variables. The subscripts are integers with zero-based indexing.

```
NAMES=(max helen sam zach)
echo ${NAMES[2]} #Sam
echo ${#NAMES[@]} #4 – array size
```

For loops

```
read yourname
for name in ${NAMES[@]}
do
      if [[ yourname = name ]]; then
            echo "name exists"
            exit 0
      fi
done
echo "Name not found"

for (( c=1; c<=5; c++ ))
do
   echo "Welcome $c times"
done
```

Positional Parameters

The positional parameters comprise the command name and command line arguments. They are called *positional* because within a shell script, you refer to them by their position on the command line.

$#: Number of Command Line Arguments

$0: Name of the Calling Program

$1 – $n: Command Line Arguments

The first argument on the command line is represented by parameter $1, the second argument by $2, and so on up to $n. For values of *n* over 9, the number must be enclosed within braces. For example, the twelfth command line argument is represented by ${12}.

In file: showargs

```
echo "$0 was called with $# arguments, the first is :$1:."
```

```
$ ./showargs a b c
./showargs was called with 3 arguments, the first is :a:.
```

Exit status

```
$ ls es   #file es exists
```

```
es
$ echo $?
0 #success

$ ls xxx  #file xxx does not exist
ls: xxx: No such file or directory
$ echo $?
1 #failure
```