# Treating Bash as a Language:

Up to now, we've been working with bash as if it were just a series of commands. While that is true, however, there are a number of commands that exist within bash that allow us to use loops, conditionals, and more. In this section, we're going to take a look at these and their syntax:

## Loops: `for`

No language, be it compiled or scripting, would be complete without a loop! The `for` command allows for us to loop over a wordlist (i.e. a series of elements, or words, delimited by whitespace), with the general syntax of:

```
for var in wordlist
do commandlist
done
```

What is happening above is that we are saying: For each element in `wordlist`, execute `commandlist`. To refer to each individual element within the list, you can use `var`. And for what to do within the `commandlist`, you can simply use as many commands with line breaks or semicolon delimited commands too.

Let's make this a bit more real by rewriting `takesArguments.sh` to use a loop instead of printing out each element by its *positional parameter*:

`takesArgumentsLoop.sh`

```
#!/bin/bash

for argument in $@
do
  echo  One of our arguments is:
  echo       $argument

done
```

If we then change the mode of `takesArgumentsLoop.sh` and run it:

```
./takesArgumentsLoop.sh I am an argument
```

> One of our arguments is: I One of our arguments is: am One of our arguments is: an One of our arguments is: argument One of our arguments is: list

It may be the case that one day you wish to take in a string with whitespace as a singular argument. What would happen if you attempted that now:

```
./takesArgumentsLoop.sh I am an "argument list"
```

> One of our arguments is: I One of our arguments is: am One of our arguments is: an One of our arguments is: argument One of our arguments is: list

### Using Strings for Arguments:

`"$@"`

Now, here is where some things can get a bit curious with the `$*` and the `$@` variables. If you wrap the variables in quotations

```
#!/bin/bash

for argument in "$@"
do
  echo  One of our arguments is:
  echo       $argument

done
```

All we've done differently above is wrap the `$@` in quotation marks. Suppose then we had a string input for a command:

```
./loopWithAt.sh I am an "argument list"
```

> One of our arguments is: I One of our arguments is: am One of our arguments is: an One of our arguments is: argument list

By wrapping the `$@` in quotation marks, it then allows for us to treat a quoted value as a singular input.

`"$*"`

What we saw above cannot be said for `$*`. Though they work the same when not quoted, the two variables have different output when inside of quotes:

```
#!/bin/bash

for argument in "$*"
do
  echo  One of our arguments is:
  echo       $argument

done
```

```
./loopWithStar.sh I am an "argument list"
```

> One of our arguments is: I am an argument list

By using the `$*` option, we don't so much treat our variable at `$4` as its own variable, we instead treat the entire argument list as a singular variable.

### C-like Syntax

Bash scripts do allow for us to use c-like syntax. In a file called `clikeLoop.sh` write:

```
#!/bin/bash

for ((i = 0; i < 5; i++))
do
 echo count to $i
done
```

Naturally, this does exactly what we would expect of any for loop:

```
./clikeLoop.sh
```

> count to 0 count to 1 count to 2 count to 3 count to 4

**Note:** Looping will be further discussed in sections 11 and 14.

## 5.6 The `If` Command

Programs could not be anywhere near as robust as they are if it weren't for control flow with if/else commands. The general syntax for the bash if/else construct is:

```
if testExpression
then
  some_commandlist
else
  someOther_commandlist
fi
```

Test expressions will be covered more in the following section, but ultimately follow the general syntax of:

```
[[ a<b ]]
```

Where the condition being tested is written within two square brackets `[[ TEST_CONDITIONS ]]` and return some exit status value. For now, we'll stick to relatively simple tests. Let's write a quick to determine how many directories are in our directory:

```
ls -l | grep ^[d] | wc -l
```

> 18

Now, suppose we wished to write a bash script that would let us know if we ever went over 10 subdirectories within a directory. We could do that by writing a file, `greaterThan10.sh`:

```
#!/bin/bash

directoryCount=$(ls -l | grep ^[d] | wc -l)

if [[ $directoryCount -gt 10 ]]
then
    echo You have more than 10 directories!
else
    echo Not greater than 10
fi
```

Now, if you run this script in a directory with less than 10 directories, you'll wind up with an output:

> Not greater than 10

However, let's see what happens when we run this in a directory with more than ten directories (and to ensure this, enter the following command):

```
mkdir emptyDirectory{1,2,3,4,5,6,7,8,9,10,11}
./greaterThan10.sh
```

> GREATER THAN 10

What if we wished to be a little more discerning? We could create a script that warns us if we have between 5 and 10 subdirectories, and then flat out yells at us if we have more! To do this, we could nest our if/else statements, however, just like many other languages, we can use the bash equivalent of the `else/if`.

So, in the following script, we can extract the command so we're not using it more than once, and we can also spice up our output a bit and add some colors:

```
#!/bin/bash

directoryCount=$(ls -l | grep ^[d] | wc -l)
RED_COLORATION='\033[0;31m' #red color
NO_COLORATION='\033[0m'   #no color
ORANGE_COLORATION='\033[0;33m' #orange color

if [[ $directoryCount -gt 10 ]]
then
    printf "${RED_COLORATION}GREATER THAN 10 DIRECTORIES ${NO_COLORATION} Please rethink your subdirectory solutions!\n"
elif [[ $directoryCount -gt  5 ]]
then
    printf "${ORANGE_COLORATION}Warning! You have $directoryCount directories!${NO_COLORATION} You may wish to reconsider this many\n"
else
    echo Not greater than 10
fi
```

One major takeaway for nested conditionals is that immediately after the `elif` and its condition, we have another `then`.

## 5.8 Shift

You may wish to write a script that works with the positional parameters, but you wish to treat them like a queue. You can shift the positional parameters up in the "queue" by using `shift`:

`shiftExample.sh`

```
#!/bin/bash

echo original values
echo argument0 $0
echo argument1 $1
```

```
echo argument2 $2
echo argument3 $3
echo
echo Shifting
shift
echo argument0 $0
echo argument1 $1
echo argument2 $2
echo argument3 $3
echo
echo Shifting
shift
echo argument0 $0
echo argument1 $1
echo argument2 $2
echo argument3 $3
echo
echo Shifting
shift
echo argument0 $0
echo argument1 $1
echo argument2 $2
echo argument3 $3
```

Now, we can see that we're calling up to the third positional parameter. Let's call the shell script with four, just to see what happens:

```
./shiftExample.sh one two three four
```

> original values argument0 ./shiftExample.sh argument1 one argument2 two argument3 three
>
> Shifting argument0 ./shiftExample.sh argument1 two argument2 three argument3 four
>
> Shifting argument0 ./shiftExample.sh argument1 three argument2 four argument3
>
> Shifting argument0 ./shiftExample.sh argument1 four argument2 argument3

What happened? After every time we print each of the arguments, we shift, dequeuing the value in argument `$1` , and then shifting all other positional arguments to a new position: `n-1`.

Or, you can also use a loop as well, though its execution may look a bit more confusing:

`shiftExample2.sh`

```
#!/bin/bash

echo original values
```

```

for val in $*
do
  echo \$\* = $*
  echo val = $val
  echo argument0 $0
  echo argument1 $1
  echo argument2 $2
  echo argument3 $3
  echo
  echo Shifting
  shift
done
```

We've got a couple more prints in here to see what's going on, but it's still the same information:

```
./shiftExample2.sh one two three four
```

> original values $* = one two three four val = one argument0 ./shiftExample2.sh argument1 one argument2 two argument3 three
>
> Shifting $* = two three four val = two argument0 ./shiftExample2.sh argument1 two argument2 three argument3 four
>
> Shifting $* = three four val = three argument0 ./shiftExample2.sh argument1 three argument2 four argument3
>
> Shifting $* = four val = four argument0 ./shiftExample2.sh argument1 four argument2 argument3
>
> Shifting

## 5.9 Case

The if/else commands let us have conditional statements, but we can also use the `case` command to use pattern matching. The general syntax of `case` is:

```
case (someString) in
matchingPattern) command
  ;;
machingPattern2) command2
  ;;
*) command3 #default
esac
```

## Putting it All Together

Suppose we wished, then, to have a shell script that helped us increment the version of a program. Semantic versioning (or semver) is a general convention for software where a piece of software has an a 3 part version number:

```
find --version
```

> find (GNU findutils) 4.5.11 Copyright (C) 2012 Free Software Foundation, Inc. License GPLv3+: GNU GPL version 3 or later http://gnu.org/licenses/gpl.html. This is free software: you are free to change and redistribute it. There is NO WARRANTY, to the extent permitted by law.
>
> Written by Eric B. Decker, James Youngman, and Kevin Dalley. Features enabled: D_TYPE O_NOFOLLOW(enabled) LEAF_OPTIMISATION FTS(FTS_CWDFD) CBO(level=2)

The version of `find` is `4.5.11`. Semvers follow the pattern `major.minor.patch`, where typically major versions include breaking changes (like the functionality of inputs change), minor versions (new functionality), and patch versions (bug fixes).

Let's write a shell script that matches on the type of version bump we'd like, which would then update the version number within a file (i.e. the program we want to increment). First we'll need a very simple bash script to act as our program:

`versionedProgram.sh`

```
#!/bin/bash

VERSION=0.0.0

echo I am a boring program. My version is $VERSION
```

Now, we could bump the version ourselves, however, opening a file, finding the version, and incrementing it is significantly more labor intensive than just typing a command and passing a file name and a type of version bump. So, in the following file, we'll use case to match on the type of version bumping we want to do, and then use `sed` to find the version. Additionally, we'll want to use the `if` command to determine if the input is even valid:

`versionBump.sh`

```
#!/bin/bash
RED_COLORATION='\033[0;31m' #red color
NO_COLORATION='\033[0m'    #no colori

versionType=$1
```

```
fileName=$2

echo You wish to increment the $versionType version for $fileName

if [ ! -f ./$fileName ]; then
     printf "$fileName ${RED_COLORATION}does not exist.${NO_COLORATION}\n"
     exit
fi



currentVersion=$(sed -n 's/VERSION=//p' $fileName)
echo The current version of $fileName is $currentVersion

case $versionType in
[Mm]ajor)
    echo Major version bump:
    ;;
[Mm]inor)
    echo Minor version bump:
    ;;
[Pp]atch)
    echo Patch version bump:
    ;;
*)
    printf "${RED_COLORATION}Bad version bump input.${NO_COLORATION}\n"
    echo The possible input values for version type are: "Major" \| "Minor" \| "Patch"
    exit
    ;;
esac

newVersion=$(sed -n 's/VERSION=//p' $fileName)
echo File version is now $newVersion
```

So far, we haven't learned to parse the string so that we can extract each of the elements. Not quite yet. However, we'll come back to this program in not too long once we get to arrays!

You may also have noticed above that we used a single bracket in our `if` instead of a double bracket like in the `if` section. Double brackets are a bash construction allowing for compound commands, while single brackets are POSIX, allowing for shell built in commands. You can use shell built in commands in the double brackets, so it may be wise just to use double brackets in bash environments all the time.