Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

Bash Scripting – Introduction

Kurt Schmidt

Dept. of Computer Science, Drexel University

October 4, 2017

Bash Scripting – Introduction

Kurt Schmidt

Intro

Command Execution

Hello, Script!

Conditionals

test – [ ]

[[ ]] Built-in

let – Arithmtic

Control Flow

if-elif-else

Loops

Scripts, Arguments

Functions

Parameter Expansion

Intro

What is a Script?

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

- A sequence of Bash commands
- A Bash program
- Stored as a text file
  - *Interpreted* by the Bash shell

Why Write Scripts

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

- Convenience
- Sequences of oft-performed operations can be placed in a script, executed as a single command
- Shell provides access to many useful utilities
- I have scripts for simple tasks, and scripts for some fairly complex tasks:
  - Rename all files of the form `DSC`*nnnn*`.JPG` to *nnnn*`.jpeg`
  - Organise the flat directory of heavily decorated filenames downloaded from Blackboard into subdirectories by student, restoring the original filenames
  - A testing framework, for grading student programs. Calls individual test cases (themselves scripts) for each program to grade

Shell Scripts vs. C-like Languages

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

- Shells are built for comfort. This comes at a cost
- Shell scripts are generally not as well-suited to large tasks
  - They run more slowly, are more resource-intensive
  - Scripts do not give the programmer nearly the same control over resources
- Languages such as C allow for much more structured programs
- Shell scripts are more difficult to harden against security attacks

Intro

Command Execution

Hello, Script!

Conditionals

test – [ ]

[[ ]] Built-in

let – Arithmtic

Control Flow

if-elif-else

Loops

Scripts, Arguments

Functions

Parameter Expansion

Command Execution

Command Execution

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

Commands in a sequence can be joined several ways:

- Sequenced
- Grouped
- Subshell Group
- Conditional

Command Sequences

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

- Commands to be executed serially
- No direct relationship between them
- Commands can be separated by a newline or ;
  - Note, `\n` is a metacharacter

```
cmd1 ; cmd2 ; cmd 3
```

Grouped Commands

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

`{ cmd1 ; cmd2 ; cmd2 ; }`

- Sequences can be grouped using {}
- List must be terminated by a ;
- Runs in the context of the current shell
- Useful for redirecting I/O
- Return value is the status of the last command executed

```
$ echo a ; echo b ; echo c > out
a
b
$ cat out
c
$ { echo a ; echo b ; echo c ; } > out
$ cat out
a
b
c
```

Grouped Commands for Subshell

`( cmd1 ; cmd2 ; cmd2 )`

- Sequences grouped with `()`
- Also handy for redirecting I/O
- Runs in a subshell
  - Can be run in the background
  - No changes persist
- Return value is the status of the last command executed

```
$ y=30 ; ( y=20 ; echo $y ) ; echo $y
20
30
```

Conditional Execution

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

Operators `&&` and `||`

- Conditional execution depends on the return value of the command on the left
- This value available to the caller (parent shell)
  - Look at special variable `$?` for return status of last command
- Value on [0, 255][1]
  - Zero (`0`) signals success; is true
  - Now you know why we told you to put `return( 0 );` at the end of your programs
  - We enumerate errors (failure), starting at 1
- `&&` and `||` have the same precedence, associate left-to-right

[1] Though I've used a shell that used only 7 bits

Conditional Execution Operators

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

*cmd*1 `&&` *cmd*2

- *cmd*1 is executed first
- If *cmd*1 succeeds, *cmd*2 is executed

*cmd*1 `||` *cmd*2

- *cmd*1 is executed first
- Only if *cmd*1 fails is *cmd*2 executed

```
$ cp file1 file2 && echo "Copy succeeded"
Copy succeeded
$ cp no_such_file file2 2> /dev/null || echo "Copy failed"
Copy failed
```

Bash Scripting – Introduction

Kurt Schmidt

Intro

Command Execution

Hello, Script!

Conditionals

test – [ ]

[[ ]] Built-in

let – Arithmtic

Control Flow

if-elif-else

Loops

Scripts, Arguments

Functions

Parameter Expansion

Hello, Script!

Our First Bash Script

Consider the following file, `hello`

```
#!/bin/bash
echo "Hello, $USER"
exit 0 # success
```

- `#!/bin/bash` – sha-bang
  - First line
  - Identifies the interpreter who is to execute this script
- *Always* quote variables in scripts
  - If you don't think you need to, *quote it anyway!*
- `exit 0` – The return value (status) of the script
- `# success` – Line comment

Running a Script

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

- Can be given explicitly to Bash as a simple input text file:

```
$ bash hello
Hello, kschmidt
```

- Or, we can give it execute permissions, run it as any other utility:

```
$ chmod +x hello
$ hello
hello: command not found
```

- Whoops! The current directory isn't in my PATH (nor should it be). Tell the shell where it is:

```
$ ./hello
Hello, kschmidt
```

Bash Scripting – Introduction

Kurt Schmidt

Intro

Command Execution

Hello, Script!

Conditionals

test – [ ]

[[ ]] Built-in

let – Arithmtic

Control Flow

if-elif-else

Loops

Scripts, Arguments

Functions

Parameter Expansion

Conditionals

Tests for Branches and Loops

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

We need tests for branches and loops

- We've already seen the return value of commands
  - Zero (`0`) is *true* (success)
  - All else is false (signals some failure)
  - Can be negated using `!` before
- There are special utilities and Bash built-ins to provide various tests

`test`, `[ ]`	Provides string, numeric, and file tests
`[[ ]]`	Similar to `[ ]`, but gentler syntax
`let`, `(())`	Provides numeric tests and arithmetic

[ ] – String Tests

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

`[ expr ]`

- Built into Bash
  - But behaves like the disk utility (less than pretty)
  - Only such test available in Bourne shell
- Note, the spaces around the `[ ]` are necessary
- Provides:
  - String tests
  - File tests
  - Numeric tests
  - Logical operators

[ ] – String Tests

- We have the normal binary, relational operators

  < = != >

```
$ [ a = b ] || echo false
false
$ [ jaga < kurt ] && echo true
bash: kurt: No such file or directory
```

- Whoops! < is a shell metacharacter. Needs to be escaped

```
$ [ jaga \< kurt ] && echo true
true
```

[ ] – String Tests

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

We have unary tests for strings

- `-z` True if string is empty
- `-n` True if string is not empty

```
[ -z "$1" ] && echo "Script requires a filename as an argument"
[ -n "$1" ] || echo "Script requires a filename as an argument"
```

[ ] – File Tests

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

Many unary tests for files. Here are a few:[1]

`-e` *file*	True if *file* exists
`-d` *file*	True if *file* is a directory
`-f` *file*	True if *file* is a regular file
`-L` *file*	True if *file* is a symbolic link
`-r` *file*	True if *file* is readable by you
`-w` *file*	True if *file* is writable by you
`-x` *file*	True if *file* is executable by you
`-O` *file*	True if *file* is effectively owned by you

[1] Do a `help test`, or `man test` for more

[ ] – File Tests, 2

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

There are some binary operators for files:

f1 `-nt` *f2*	True if *f1* is newer than *f2*
f1 `-ot` *f2*	True if *f1* is older than *f2*
f1 `-ef` *f2*	True if *f1* is a hard link to *f2* (they are the same file)

```
[ -f "$log" ] && echo "Next status line" >> "$log"
[ -r "$input" ] || echo "I can't read $input"
[ "$f1" -ef "$f2" ] &&
  echo "I can remove $f1 and $f2 will still be there"
```

[ ] – Arithmetic Tests

- We have different relational operators for arithmetic[1]
  - All parameter values are just strings
  - Shell can't tell from context which comparison is meant

```
-lt -le -eq -ne -ge -gt
```

```
$ [ 13 \< 2 ] && echo true
true
$ [ 13 -lt 2 ] || echo false
false
```

[1] The `let` utility makes this prettier

[ ] – Logical Operators

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

! *expr*	NOT - True when *expr* is false, false otherwise
exp1 -a *exp2*	AND - True when both *exp1* and *exp2* are true, false otherwise
exp1 -o *exp2*	OR - False when both *exp1* and *exp2* are false, true otherwise

[[ ]]

Intro

Command Execution

Hello, Script!

Conditionals

test – [ ]

[[ ]] Built-in

let – Arithmtic

Control Flow

if-elif-else

Loops

Scripts, Arguments

Functions

Parameter Expansion

- Supports all the same tests as [ ]
- Is a built-in, so, syntax is gentler
  - Shell metacharacters <, >, etc., don't need to be escaped
  - Shell knows it's in a test
- Mind, install scripts (and makefiles) tend to use Bourne syntax; it's the default for all Unix systems

[[ ]] – New Features

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

- Familiar logical operators
  - ! && ||
- == = are equivalent
- == != treat the right operand as a pattern (glob)

```
$ [[ abcde.f == a*e.? ]] && echo true
true
```

- New operator, =~, treats the right operand as an extended regular expression

```
$ [[ abcde.f =~ a.*e\..? ]] && echo true
true
```

`(( ))` – `let`, Relational Operators

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

- `let` is a Bash built-in
- Bash provides syntactic sugar: `(( ))`[1]
- Treats values stored in parameters as integers[2]
  - *N.B* – Only does integer arithmetic (division)
- Allows you to evaluate relational expressions
  - Same logical operators

`<  <=  ==  !=  >=  >`

```
$ x=13 ; y=87
$ (( x > 7 )) && echo true
true
$ (( x!=0 && y/x >= 6 )) && echo true
true
```

[1] Note, the `$[ ]` form is deprecated

[2] For float arithmetic, see the `bc` utility

`(( ))` – `let`, Arithmetic Operators

`let` can be used to evaluate arithmetic exceptions

- Arithmetic: `** * / % + -`
- Bit-wise: `~ << >> ^ & |`
- Pre- and post-fix increment/decrement: `++ --`
- A C-like ternary operator: `?:`
- Assignment (=), and the usual operator/assignment operators: `+= -= &=`, etc.

(( )) – examples


```
$ x=13
$ echo $(( x+15 ))
28
$ echo $x
13
$ (( y = x*4 ))
$ echo $y
52
$ (( y-=1 ))
$ echo $y
51
$ echo $((x>>2))
3
$ (( 5 && 2 )) && echo true
true
$ (( 5 & 2 )) || echo false
false
$ # Same as in C, Python, etc. Why?
```

Intro

Command Execution

Hello, Script!

Conditionals

test – [ ]

[[ ]] Built-in

let – Arithmtic

Control Flow

if-elif-else

Loops

Scripts, Arguments

Functions

Parameter Expansion

Control Flow

Bash Control Structures

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

We have branching:

- `if`
- `if-else`
- `if-elif-else`
- `case`

And loops:

- `while`
- `until`
- `for`
- `select`

Note: Bash may not care about proper indenting, but your grade may well care

if

`if` *`tests`*`; then` *`cmds`*`; fi`

- *`tests`* is executed
- If the exit status is 0 (success), *`cmds`* is executed

```
if grep Waldo * &> /dev/null ; then
  echo "Found Waldo!"
fi
```

```
if [[ -d "$paris" && -r "$paris" ]] ; then
  echo "I see $paris"'!'
fi
```

```
if (( cats > 3 )) ; then
  echo "Too many cats"
  echo "People will talk"
fi
```

if-else

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

if *tests*; then *cmds*; else *cmds*; fi

```
if grep Waldo * &> /dev/null ; then
  echo "Found Waldo!"
else
  echo "Dude's a slippery one"
fi
```

```
if [[ -d "$paris" && -r "$paris" ]] ; then
  echo "I see $paris"'!'
else
  echo "Might be on the wrong continent"
fi
```

```
if (( cats > 3 )) ; then
  echo "Too many cats"
  echo "People will talk"
else
  echo "You might yet be sane"
fi
```

if-elif-else

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

if *tests*; then *cmds*; {elif cmds;} else *cmds*; fi

```
read grade
if (( grade >= 90 )) ; then
  echo "A"
elif (( grade >= 80 )) ; then
  echo "B"
elif (( grade >= 70 )) ; then
  echo "C"
elif (( grade >= 60 )) ; then
  echo "D"
else
  echo "F"
fi
```

`while` Loop

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

`while` *`tests`*`; do` *`cmds`*`; done`

- *`tests`* is executed
- If the exit status is 0 (success), *`cmds`* is executed
- Execution returns back to *`tests`*, start again

```
i=0
while (( i<=12 )) ; do
  echo $i
  (( i+=1 ))
done
```

```
cat list | while read f ; do
  # Assume list contains one filename per line
  stat "$f"
done
```

for Loop

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

`for` *name* `[in` *list*`]; do` *cmds* `; done`

- Executes *cmds* for each member in *list*
- "$@" used if *list* isn't there

```
$ for i in a b c ; do
>   echo $i
> done
a
b
c
```

```
for id in $(cat userlist) ; do
    # assumes no spaces in userIDs
  echo "Mailing $id..."
  mail -s "Good subject" "$id"@someschool.edu < msg
done
```

Counting `for` Loop

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

- Bash has a C-like for loop:

```
$ for (( i=0; i<3; ++i )) ; do
>   echo $i
> done
0
1
2
```

```
$ for (( i=12; i>0; i-=4 )) ; do
>   echo $i
> done
12
8
4
```

{x..y} – Brace Expansion

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

{x..y[..*inc*]}

- Generates sequences in a natural way

```
$ echo {5..13}
5 6 7 8 9 10 11 12 13
```

```
$ echo {a..g}
a b c d e f g
```

- Brace expansion will pad numbers on the left[1]

```
$ for i in {000..010..2} ; do echo -n "$i " ; done
000 002 004 006 008 010
```

- This is quite handy in loops:

```
for i in {00..05} ; do
  \rm proc${i}.log
done
```

[1] YMMV, depending on your platform

Loops – `continue`, `break`

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

- `break` exits a loop
- `continue` shortcircuits the loop, resumes at the next iteration of the loop

```
$ for i in {1..42} ; do
>   (( i%2 == 0 )) && continue
>   (( i%9 == 0 )) && break
>   echo $i
> done
1
3
5
7
```

case

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

`case` *word* `in` {*pattern* `)` *cmds* `;;`} `esac`

- Selectively execute *cmds* if *word* matches the corresponding *pattern* (glob)
- Commands are separated by `;`
- Cases are separated by `;;`

```
case $opt in
  n ) DRY_RUN=1 ;;
  x ) ECHO=1 ;;
  \? | h | H ) usage() ; exit 1 ;;
  ?) echo "Unkown character ;;
esac
```

select

`select` *name* `[in` *list*`]; do` *cmds* `; done`

- Much like the `for` loop
- Displays enumerated menu of *list*
- Puts user's choice in `name`

```
$ select resp in "This" "That" "Quit" ; do
>   echo "You chose $resp"
>   [ "$resp" == Quit ] && {echo 'bye!' ; break ; }
> done
1) This
2) That
3) Quit
#? 2
You chose That
#? 3
You chose Quit
bye!
```

Intro

Command Execution

Hello, Script!

Conditionals

test – [ ]

[[ ]] Built-in

let – Arithmtic

Control Flow

if-elif-else

Loops

Scripts, Arguments

Functions

Parameter Expansion

Scripts, Arguments

More Bash Parameters

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

These should be familiar:

`$$`	The process ID (pid) of the shell
`$?`	The exit status of the last command

These are handy inside scripts and functions

`$#`	The number of arguments
`$*`	All arguments
`$@`	All arguments (individually quoted)
`${`n`}`	The n^{th} positional argument

Arguments to Scripts

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

- If the script has a proper sha-bang, and the execution bit is set, the script may be invoked directly
- Arguments may be supplied, as with any other commad

```
myScript arg1 arg2 ...
```

- Or, `bash` may be invoked explicitly, and given a script as input
- Arguments to the script would follow the script

```
bash myScript arg1 arg2 ...
```

Arguments to Scripts

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

- Bash, in turn, can take options
- There are two helpful ones for writing and debugging scripts

`-n` Dry run. File is parsed, but commands aren't executed. Check syntax

`-x` Echo on. Commands are echoed to `stderr` as they're executed (after parameter, file, etc. expansion has happened)

```
$ bash -x ./hello
+ echo 'Hello World'
Hello World
+ echo 'My name is kschmidt'
My name is kschmidt
```

Arguments in Scripts

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

- Arguments are positional
- `$0` is the name of the command (how it was invoked)
- In Bourne, `$1` – `$9`
  - Can't access `$10`, it'd be `$1` followed by 0
- In Bash, we can wrap parameters in curly braces, so, `${1}`, `${12}`, etc.

(See `Labs/Bash/args.bash`)

```
echo "Here is how the script was invoked: $0"
echo "Here are the arguments: $*"
echo "This is the number of arguments: $#"
echo "We'll show each arg:"
for a in "$@" ; do echo "$a" ; done
```

Processing Args – `shift`

`shift [`*`cnt`*`]`

- Shifts args to the left *cnt* (default 1) positions
- $1 is gone, $2 → $1, etc.

```
i=0
while [ ! -z "$1" ] ; do
  (( i+=1 ))  # just here to enumerate output
  echo -e "$i\t$1"
   shift   # old $1 is gone
done
```

Processing Args – `getopts`

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

`getopts` *`optstring name`*

- Bash built-in
- Just handles short (single character) options[1]
- Returns SUCCESS when it finds an option
- *`optstring`* is the list of options
  - Options that take an arg are followed by a colon (`:`)
- *`name`* holds the current option
- Index of current arg stored in `$OPTIND`
- If option takes an arg, it is stored in `$OPTARG`

[1] See `getopt` utility

getopts – e.g.

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

```
while getopts "ab:cC" opt; do
  case $opt in
    a  ) echo "option a, at index $OPTIND";;
    b  ) echo "option b with arg = $OPTARG, at index $OPTIND";;
    c | C ) echo "option $opt, at index $OPTIND";;
    ? ) echo "usage: $0 [-a] [b arg] [-c] args..." ; exit 1;;
  esac
done

echo "\$OPTIND = $OPTIND"
shift $(($OPTIND -1)) # shift off the options

echo -e "\nHere are the remaining arguments:"
for i in "$@" ; do
  echo -e "\t$i"
done
```

Intro

Command Execution

Hello, Script!

Conditionals

test – [ ]

[[ ]] Built-in

let – Arithmtic

Control Flow

if-elif-else

Loops

Scripts, Arguments

Functions

Parameter Expansion

Functions

Bash Functions

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

`function` *name* {*body*} [1]
name() {*body*}

- Executed in the same environment
- Arguments to function are handled the same as arguments to a script[2]
- Can be called recursively (see `$FUNCNEST`)
- Built-in `return` *rv* can be used in a function, to return execution (and optional status *rv*) to caller
  - Otherwise, status of last command is used
- Export to make available to subshells:

```
export -f funcname
```

[1] In fact, the curly braces and body can be any compound command
[2] Except that `$0` is unchanged. See `$FUNCNAME`

Functions – e.g.

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

```
function hello
{
  echo "hello $1"
  if [[ -n "$2" && "$2" -gt 1 ]] ; then
    hello $1 $(($2-1))
  fi
}
```

- Called as a script would be:

```
$ hello Vera 3
hello Vera
hello Vera
hello Vera
```

Local Variables in Functions

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

`local` {*var*}

- Defines variable(s) local to function
- Won't step on caller's environment

```
function hello {
  local USER='Elmer Fudd'
  FOO='Hunting Wabbit'
  echo "Hello, $USER, you are $FOO"
}
```

```
$ FOO='Baking Cookies'
$ echo $USER
kschmidt
$ hello
Hello, Elmer Fudd, you are Hunting Wabbit
$ echo $FOO
Hunting Wabbit
$ echo $USER
kschmidt
```

Intro

Command Execution

Hello, Script!

Conditionals

test – [ ]

[[ ]] Built-in

let – Arithmtic

Control Flow

if-elif-else

Loops

Scripts, Arguments

Functions

Parameter Expansion

Parameter Expansion

Unset or Null Parameters

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

Expansions for unset (or null) parameters:

${*param*:-*word*}	Use *word* if *param* is not set or is null
${*param*:=*word*}	Use *word* if *param* is not set or is null, set it to *word*
${*param*:?*word*}	If *param* is not set or is null, print *word* to `stderr`, exit shell (if not interactive)
${*param*:+*word*}	If *param* is *set*, use *word* , otherwise use null

E.g. – Null Parameters

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

```
$ unset foo
$ echo ${foo:-"Hello!"}
Hello!
$ echo $foo

$ echo ${foo:?"Houston, we have a problem"}
bash: foo: Houston, we have a problem
$ echo ${foo:+"We here?"}

$ echo ${foo:='I am de Fault'}
I am de Fault
$ echo $foo
I am de Fault
$ echo ${foo:?"Houston, we have a problem"}
I am de Fault
$ echo ${foo:+"We here?"}
We here?
```

Removing Patterns

Remove leading or trailing Patterns (globs)

${param#pattern}	Remove shortest leading *pattern*
${param##pattern}	Remove longest leading *pattern*
${param%pattern}	Remove shortest trailing *pattern*
${param%%pattern}	Remove longest trailing *pattern*

```
$ f=a^b^c
$ echo ${f#*^}
b^c
$ echo ${f##*^}
c
$ echo ${f%%^*}
a
$ echo ${f%^*}
a^b
```

Parameters – Pattern Substitution

Bash Scripting – Introduction
Kurt Schmidt
Intro
Command Execution
Hello, Script!
Conditionals
test – [ ]
[[ ]] Built-in
let – Arithmtic
Control Flow
if-elif-else
Loops
Scripts, Arguments
Functions
Parameter Expansion

${*param/pattern/string*}

- Substitutes *string* for longest matching *pattern*
  - Again, *pattern* is a glob, not a regular expression
- If *pattern* starts with /, it substitutes all occurrences

```
$ f="The cat sat on the hat"
$ echo ${f/[hcs]at/XXX}
The XXX sat on the hat
$ echo ${f//[hcs]at/XXX}
The XXX XXX on the XXX
```

Parameters – Change Case

${*param<OP>pattern*}

- Change case of characters matched by *pattern*
- *pattern* should not attempt to match a larger string
- If *pattern* is missing, treated as a ?
- ^ ^^ – Convert first (each) matched character to upper
- , ,, – Convert first (each) matched character to lower

```
$ f=SHOUT
$ echo ${f,}
sHOUT
$ echo ${f,,}
shout
$ echo ${f,[SOT]}
sHOUT
$ echo ${f,,[SOT]}
sHoUt
$ f=quiet
$ echo ${f^^}
QUIET
```