UCLA CS35L

Week 3

Wednesday

Reminders

- Start thinking about Week 10 Presentation Topics/Teams
  - Signup link under CCLE – Week 10
- Assignment 3 is longer and due next **Friday** (4/24)

- Anonymous feedback for Daniel - https://forms.gle/tZwuMbALe825DBVn8

Shell Scripting

Interpreted vs Compiled Languages

- Compiled Languages
  - C, C++, Swift, Rust
  - Use a compiler to convert source code into assembly code and then an executable binary
- Advantages
  - Performance
- Disadvantages
  - Portability and speed of deployment

Interpreted vs Compiled Languages

- Interpreted Lanuages
  - Bash, Python, Javascript
  - An interpreter reads and executes the source code line by line
- Advantages
  - Ease of development
  - Usually debugging
  - Portability
- Disadvantage
  - Performance

What are times that you have used a script? Bash or another language?

Interpreted/Scripting Languages vs. Compiled Languages

- Compiled Languages
  - Ex: C/C++, Java
  - Programs are translated from their original source code into object code that is executed by hardware
  - Efficient
  - Work at low level, dealing with bytes, integers, floating points, etc
- Scripting languages
  - Interpreted
  - Interpreter reads program, translates it into internal form, and execute programs

Why use a Shell Scripts

- Simplicity
- Portability
- Ease of development

Basic Shell Constructs

- Shell recognizes three fundamental kinds of commands
  - Built-in commands: Commands that the shell itself executes
  - Shell functions: Self-contained chunks of code, written in shell language
  - External commands

#! in the first line

- When the shell runs a program, it asks the kernel to start a new process and run the given program in that process.
- It knows how to do this for compiled programs but for a script, the kernel will fail, returning a “not executable format file” error.
- To tell the OS how to run the file we specify **#!/usr/bin/bash** so it knows to use the bash interpreter to run the file.
- NOTE – normally # starts a comment line in Bash

Executing a Shell Script

- To execute a shell script (or any file on linux) you can use the command:
    - `./yourFile.sh`
    - If execute permissions are not set, you will get permission denied. How can we fix this?
        - Add permissions with “`chmod +x yourFile.sh`”

Shell Variables

- We assign variables directly:
  - `x=hello`
  - `y=world`
  - Note no space in between the =
- If we want our variable to contain whitespace, we need quotes
  - `z=“hello world”`

Accessing and Using Variables

- Access the variable with $ symbol
    - `echo $a`

- If variable has whitespace (newlines, tabs, etc) then surround with quotes when accessing
    - `echo “$b”`

- Can also easily concatenate string by using them in a new string
    - `comboXY=“$x $y”`

Parameter Expansion

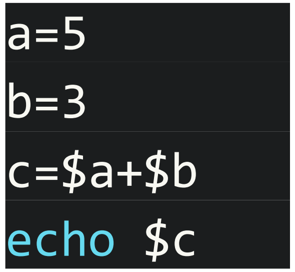
- $x and ${x} are mostly equivalent, but {} is less ambiguous

`z=“$x $y”` is the same as `z=“${x} ${y}”`
but `z=“$xx$y”` is different from `z=“${x}x${y}”`

Why?

Bash Typing

- Bash is a “weakly typed” language. In this specific case, it means:
  - You do not need to declare type for Bash variables
  - Operations usually assume strings. For example, what is the output below:



```
a=5
b=3
c=$a+$b
echo $c
```

Arithmetic Expansion

- By default in bash, all variables are character strings
- But Bash will perform arithmetic if the variable contains only digits

- Use $((…)) for arithmetic operations

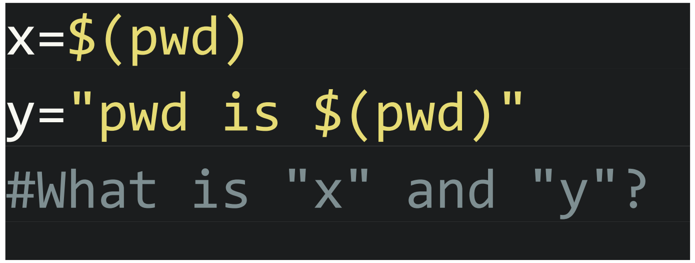
```
a=$((2 + 3))
b=$(($a + 4))
echo $b
```

- NOTE – Arithmetic should not be required for any part of Week 3 Homework

Command Substitution

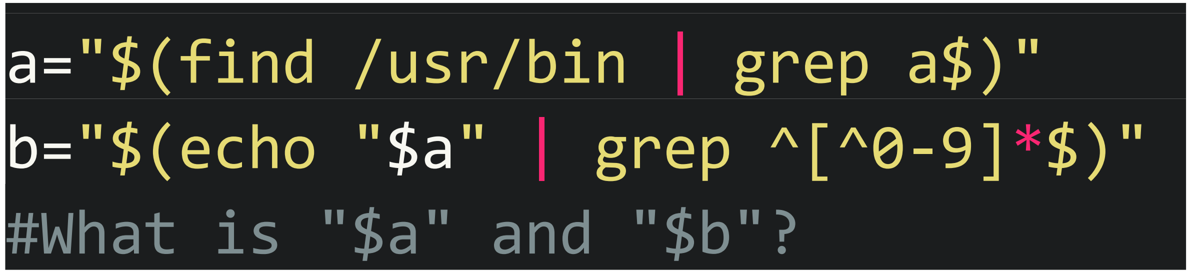
- Using the $(…) syntax we can assign the output of a command in the (…) to a variable

- Example 1



```
x=$(pwd)
y="pwd is $(pwd)"
#What is "x" and "y"?
```

- Example 2



```
a="$(find /usr/bin | grep a$)"
b="$(echo "$a" | grep ^[^0-9]*$)"
#What is "$a" and "$b"?
```

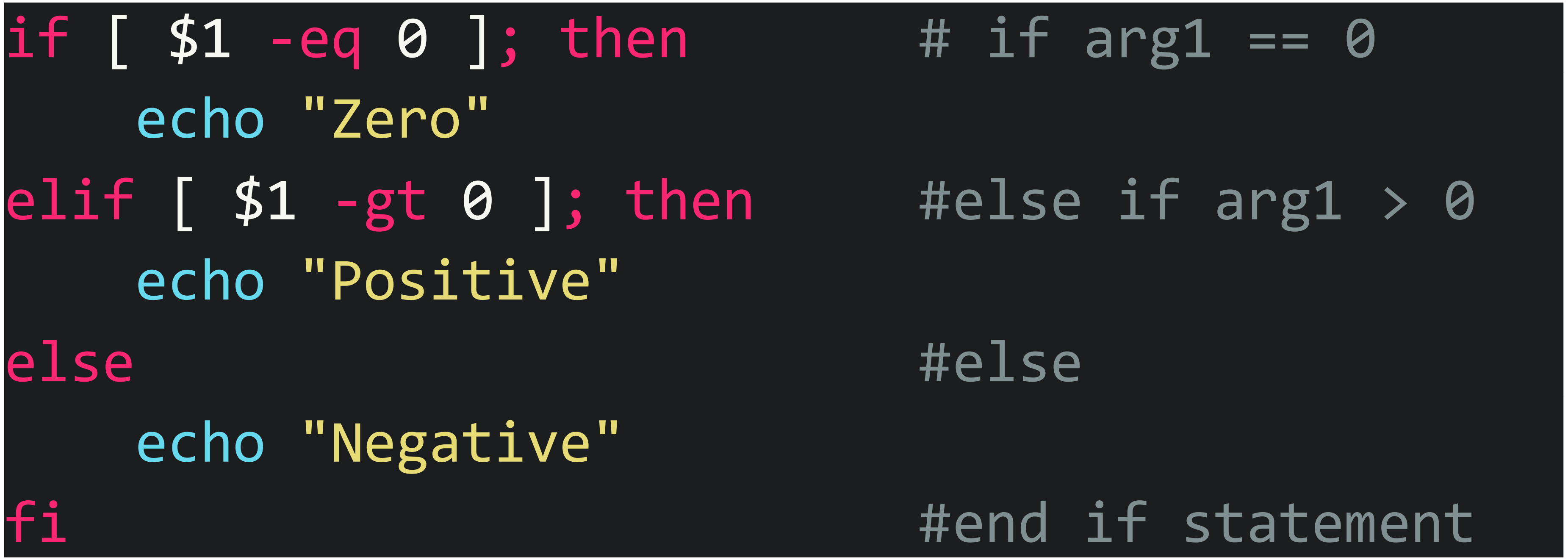
- NOTE – can also be nested $(… $(…))

Built-in shell variables

- Can be accessed from within the shell

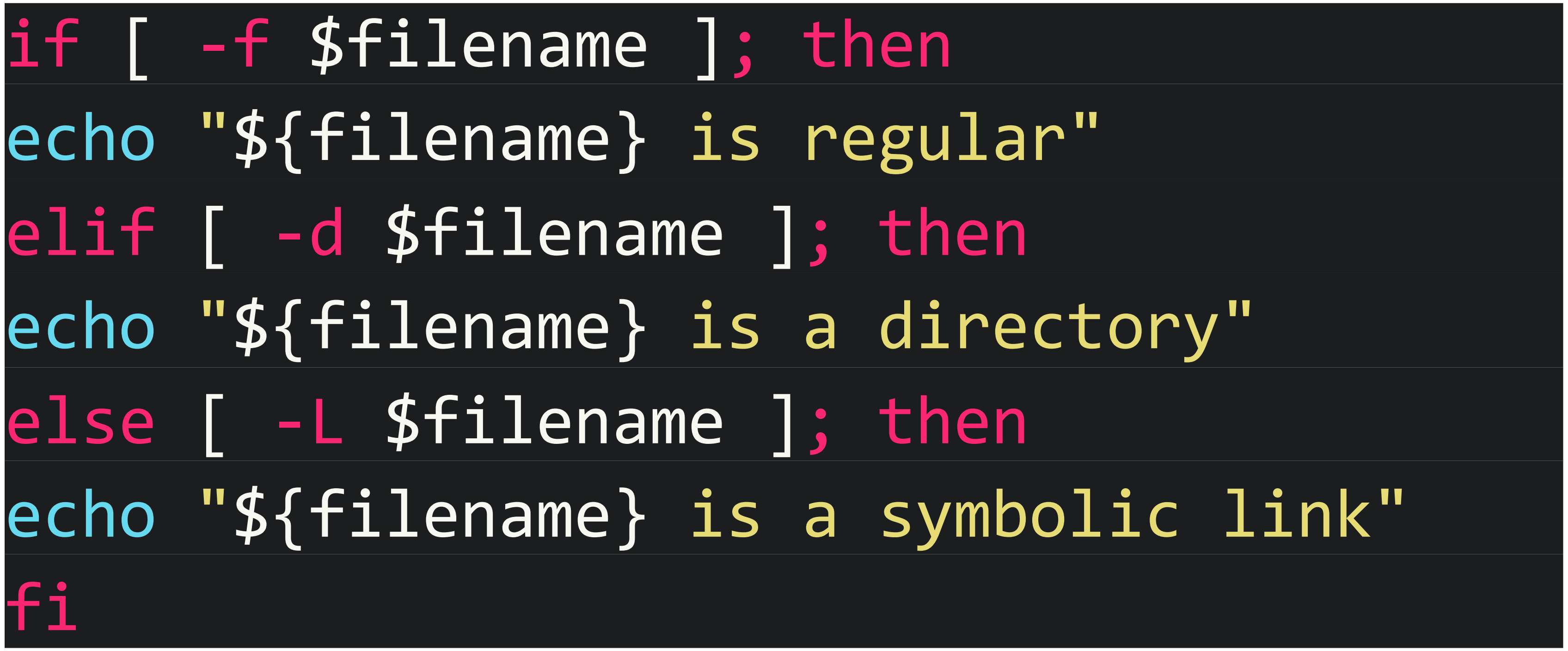
`$#`	Number of arguments provided to script
`$0`	Name of script
`$1, $2, etc`	1st and 2nd argument, etc
`${10}, ${26}, etc`	For arguments greater than 9
`$?`	Exit status of last command
`$$`	Current running process ID

If statement



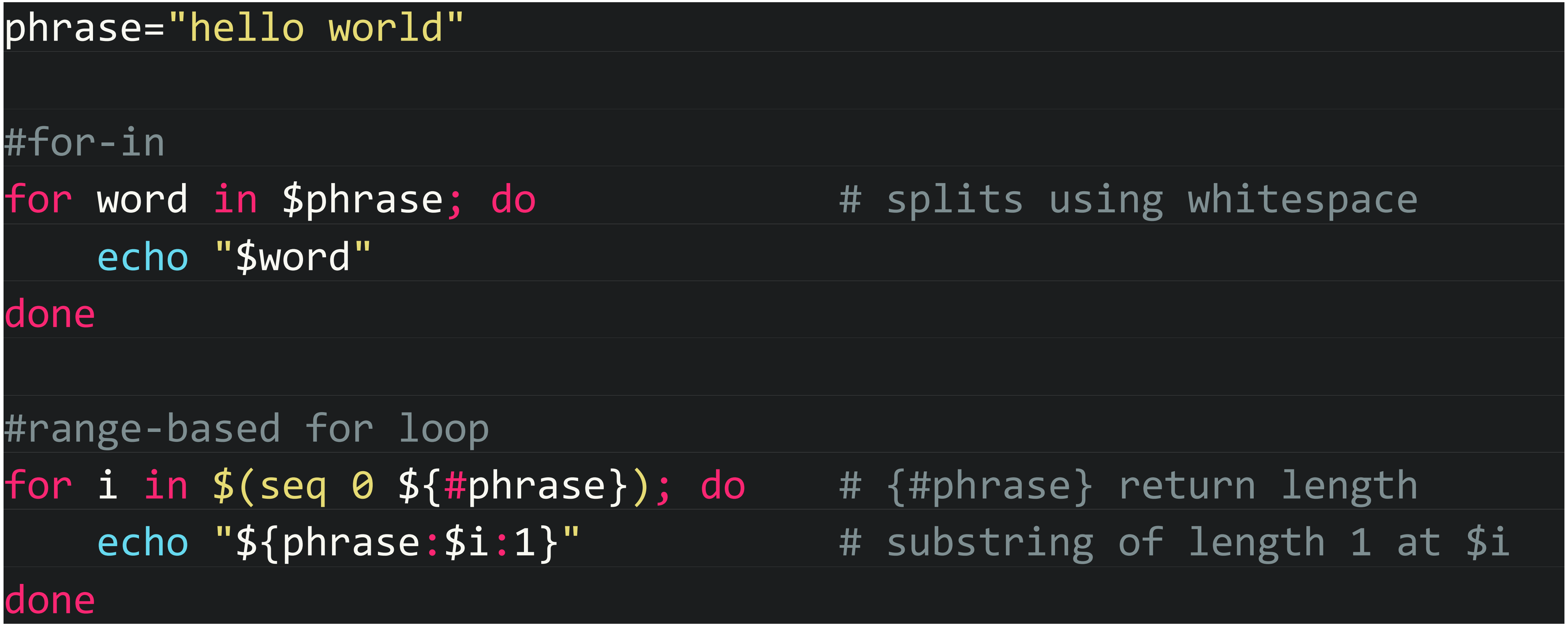
```
if [ $1 -eq 0 ]; then       # if arg1 == 0
    echo "Zero"
elif [ $1 -gt 0 ]; then     #else if arg1 > 0
    echo "Positive"
else                        #else
    echo "Negative"
fi                          #end if statement
```

File Type Checks



```
if [ -f $filename ]; then
echo "${filename} is regular"
elif [ -d $filename ]; then
echo "${filename} is a directory"
else [ -L $filename ]; then
echo "${filename} is a symbolic link"
fi
```

For Loop



```
phrase="hello world"

#for-in
for word in $phrase; do             # splits using whitespace
    echo "$word"
done

#range-based for loop
for i in $(seq 0 ${#phrase}); do    # {#phrase} return length
    echo "${phrase:$i:1}"           # substring of length 1 at $i
done
```

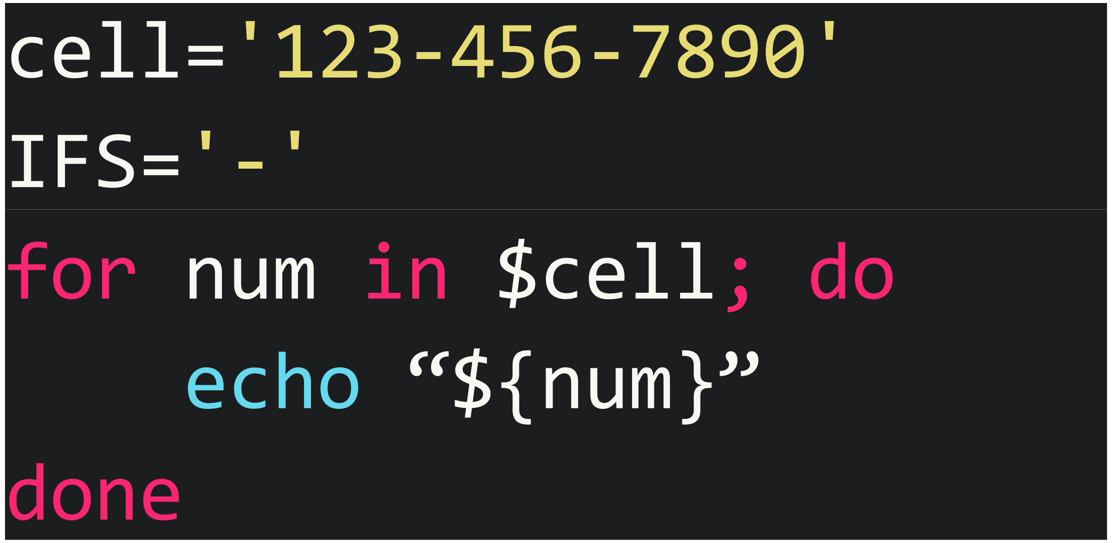
Comparison Operators

- Integer Comparisons
  - -eq, -gt, -lt, -ge, -le
- String Comparisons
  - ==, !=, <, >
- General Reference –

http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html

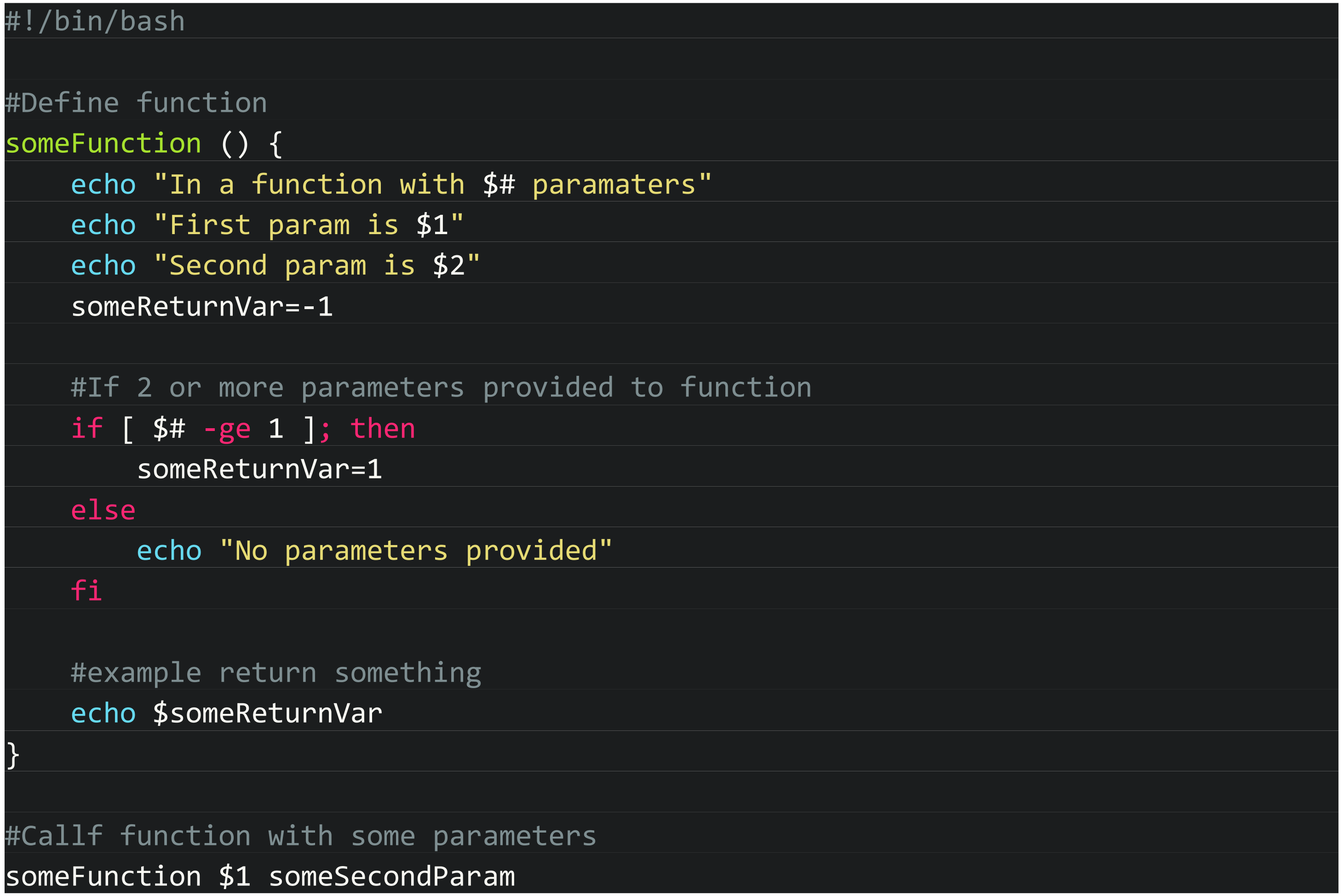
$IFS (Internal Field Separator)

- Built-in variable used to separate fields, like word boundaries
- By default is whitespace (space, tab, and newline)
- But can be modified in the current session
- Check current value with `echo $IFS` will show blank if default
- You can use the command `unset IFS` to restore to default



```
cell='123-456-7890'
IFS='-'
for num in $cell; do
    echo “${num}”
done
```

Functions in Bash



```bash
#!/bin/bash

#Define function
someFunction () {
    echo "In a function with $# paramaters"
    echo "First param is $1"
    echo "Second param is $2"
    someReturnVar=-1

    #If 2 or more parameters provided to function
    if [ $# -ge 1 ]; then
        someReturnVar=1
    else
        echo "No parameters provided"
    fi

    #example return something
    echo $someReturnVar
}

#Callf function with some parameters
someFunction $1 someSecondParam
```

Exiting a Shell Script

- `exit N`
  - Example: `exit 1`

- Will exit the current process.
- The number provided is the “status” that the now terminated process will return
  - 0 is considered a success
  - Non-0 is considered a failure

- So for your homework, `exit 1`, is exiting with a failure status

Shell Scripting – Helpful Links

- POSIX Shell Specification

https://pubs.opengroup.org/onlinepubs/9699919799/utilities/contents.html

- A quick intro (should read at least most of this)

https://www.tutorialspoint.com/unix/shell_scripting.htm

- Bash Academy (more detailed web guide)

https://guide.bash.academy/

- A tutorial more specific to Bash

https://www.tldp.org/LDP/abs/html/index.html

Lab 3 Review

Download the provided HTML page and Run sample commands

- Use wget to obtain file
- Run the provided set of tr commands. Note you need to provide input to the tr command. Something like:
  - `cat hwnwdshw.htm | tr -c 'A-Za-z' '[\n*]’`
  - …
- Pay attention to what the –c and –s option do
- Pay attention to what happens as you pipe, |, more commands
- The final part, `comm -23 - words,` is an example of a very basic English Spell Checker
- Our job is to create a file that can be used for the Hawaiian Spell Checker

Begin to modify the text piece-by-piece

- Feel free to use either sed or tr, or even a mix of both
  - I personally prefer sed, but both will mostly work

- For example – step 1 as specified is to remove all instances of ‘?’, ‘<u>’ and ‘</u>’
- What does the above look like?

Continue Modifying Text

- Next you want to allow text only in the form of: *A*<td*X*>*W*</td>*Z*
  - A is 0 or more spaces
  - X is any characters EXCEPT >
  - W is entirely Hawaiian Characters or Spaces
    - Hawaiiain characters are: p k ' m n w l h a e i o u
    - Note that backticks ` should be converted to a single quote ‘
  - Z is 0 or more spaces
- Come up with a single or multiple commands to filter the text to the above
- Next steps will be to remove everything except W – which is the Hawaiian word. Each W should be on its own line

Finishing Steps

- Follow remaining specs to remove duplicates and sort results
- Things to consider:
  - Some steps can be consolidated into a single command, but it is ok to break the problem into smaller pieces and use multiple commands
  - If you follow the instructions, your script will not catch every single Hawaiiain word. In the lab log, mention what the issue's were.

HW 3 Review

Overview

- Write a Bash Script that prints out all the “poornames” in a directory
    - Non-Recursive Case
    - Recursive Case

- This Bash Script will function more like a typical program you wrote in C, but will use Bash. You will need things like:
    - Loops
    - Conditionals
    - Argument Passing

Name Violations

- Find all violations based on these rules:
  1. A filename component can only contain the 26 upper case alphabets, the 26 lower case alphabets, '.', '-' and '_'
  2. A filename component cannot start with the hyphen -
  3. Except for '.' and '..', a file name component cannot start with the dot .
  4. The length of a filename component cannot exceed 14
- Next find any duplicates
  1. No two files in the same directory can have names that differ only in case. For example, if a directory contains a file 'St._Andrews' then it cannot also contain a file named 'st._anDrEWS'

Non-Recursive Case

- Focus here first – this should be the bulk of the work.

1. `find` all the immediate children of the current directory that violates the name rules
2. Use `grep` and regex to find any name violations
3. Use `sort` and `uniq` to handle any duplicates which have different case
4. Use pipes, |, as needed

Recursive Case

- Use your non-recursive code to solve the recursive case.
    - Use find with the exec option to have your recursive code call your non-recursive function on each directory that you check
        - Example: `find . –type d –exec echo {} >> matches.txt \;`

General Bash Programming Tips

- We don’t have an IDE for Bash, so to debug try to use echo
    - Basically the Bash form of print debugging
    - REMOVE all extra echo’s before submission. It will mess up the autograder
- Try to program with best-practices from C++
    - Break problem down into small problems which will become functions
    - Debug piece-by-piece
- Testing
    - Create your own directory with files that have good and bad names. Test your script on those as you go
    - Make sure you try can match the sample test case in the spec