Week 11 2

CSC209 Fall 2023

Dr. Demetres (dee) Kostas

November 30, 2023

Announcements

- TA OH for A4
  - see quercus
  - I have OH today
    - this is my last one
- A2 grades to be released
  - before next week
  - the average is quite low
    - but that's pretty normal

Bash/shell scripting

- unique "programming language"
  - even in terms of scripting
- the shell needs to support
  - convenient navigation
    - and execution
    - including I/O redirection

In a sense, you know the language

- you've encountered it already
- just *interactively*
  - starting a `bash` terminal
    - is not that different from
      - starting a python shell
      - interactively
      - typing commands 1-by-1

So what make up bash scripts

- mostly just the commands
  - that you type at a terminal
    - automating these things
- In a sense
  - much like the actions
    - in a Makefile

Why not use something else

- bash is extremely effective
  - at `stdin=/=stdout` redirection
- syntax like:
  - each line starts with a program
  - following things are arguments
- You can join programs
  - through **pipes**
    - using the `|` operator
- You can connect files
  - with `<`, `>` and `>>`

So you know the basic syntax

- each "line"
  - `prog_name arg1 arg2 ...`
- this constitutes using bash
  - as a **shell**
    - because it includes commands
    - to navigate the filesystem
      - *and* start programs

What haven't you seen before

- creating variables!
- capturing command outputs
- conditional flow
  - if/else
- looping
  - while, for, etc.

Something new about the shell

- we've rarely used them
  - but variables play a big part
- e.g. `$PATH`
  - this is what is used
    - to figure out where `ls` and `cd`
      - are located
  - adding directories to this
    - makes them "first-class"

How do you create variables

- focusing on scripting
- it is as simple as a name
  - generally, `UPPER_CASE`
- then a single equal sign
  - and a value

```
MY_VARIABLE=3
echo $MY_VARIABLE
```

Accessing variables

- the special `$` character
  - is used to convert text following
    - that matches a variable
    - into the **text**
      - stored by the variable
  - otherwise error
- Using `${<var-name>}`
  - for ambiguity

gotcha

- what happens when
  - `i = 0`?

Capturing output

- what if we want the text
  - that is output to `stdout`
    * when we run a program
    * to be used in out script
    * not just re-directed
      · to other programs
      · or files?

There are a variety of mechanisms

- we will focus on backticks
  - the ‘ character
- surrounding something in ‘
  - means run what is inside
    * and proceed with the `stdout`
    * in place of the ‘
      · demarcated section

```
LS_OUT=‘ls -l‘
```

More complex language features

- I’ve used
  - the following project as a reference
  - `https://tldp.org/LDP/Bash-Beginners-Guide/html/`

Conditionals

- generally using a program
  - called `test`
- As with most C programs
  - 0 means success
    - for test, it means TRUE!
  - which is kinda awful
- why do you think this is?

if blocks

```
if TEST-COMMANDS; then

CONSEQUENT-COMMANDS

elif MORE-TEST-COMMANDS; then

MORE-CONSEQUENT-COMMANDS

else

ALTERNATE-CONSEQUENT-COMMANDS

fi
```

What should the test commands *exit with*

- they must exit with
  - "success"
    - the value 0
- The other numbers
  - are setup for tracing errors

while loops

```
while THIS_COMMAND_RET_ZERO; do
    COMMANDS_PER_LOOP
done
```

for loops

```
for NAME in LIST_OF_THINGS; do
    COMMANDS_FOR_EACH_THING
done
```

Practical example

- backup all the files

```
for file in ${PATH_TO_FOLDER}/*; do
    echo $file
    cp $file ${DESTINATION_PATH}/${file}.bak
done
```

Some special variables

- the variables
  - `$1`, `$2`, etc.
  - correspond to command line args
  - `argv[1]`, `argv[2]`
    - but for the script!
- The `$?` variable
  - gives us the return code
  - of the last command run

Practical example again

- backup all the files
  - but with args
  - first arg for source
  - second for destination

```
for file in ${1}/*; do
    echo $file
    cp $file ${2}/${file}.bak
done
```

So why not a nicer language

- the real advantage is
  - using **programs** as if
    - they were functions
      - and capturing their `stdout`
- we can do this almost anywhere
  - as this process works by
    - replacing the command
      - by the result of the program

On bash and brackets

- I will not be testing
  - your knowledge of brackets
    - backticks will suffice for output
    - `${<name>}` will suffice for variables

- bash has developed a variety
    - of syntax around brackets
        * that are useful
            · but also cumbersome

Some examples

- Using `[<stuff>]`
    - actually gets replaced
        * by the specific program
            · test – to test conditions
    - useful for conditionals
- Standard `()` are used
    - for a variety of sub-shell
        * operations

Worksheet

`shell-prog.pdf`