SLS Lecture 2 : Shell Part II : Programming, Files and Directories

Contents

Getting our hands dirty.
As powerful as UNIX and the shell are, they are also old and dirty... and have had plenty of time to accumulate some Barnacles.

2.1. Our goals:

1. Basic proficiency in
    - using the shell as a programming language
        - core to automating your UNIX life
            - interactively : one liners
            - scripts : small and large programs
        - navigating and working with the file system
            - running and managing running programs
            - IO redirection and pipes
            - processes
    - understand basics of UNIX credentials
        - user and groups names and ids, file permissions, and who “root” is

sudo rm -f $my_other_dic/$my_another_dic

supreme access ↑

1. Enable you to learn more
    - know enough so that you can experiment
    - know where you can look for more info
    - know enough so you can appreciate and learn from others
        - idioms, tricks, and hacks
        - scripts

First Assignment B is check and need file info

1. Understand how things work so that you have the context for
    - working efficiently with large bodies of ascii files
        - at the command line
        - and in editors like `vim` and `emacs`
        - `make`
        - `git`

2.2. Anatomy of shell command lines

How to have a conversation with the shell

- “prompt” indicates the shell is ready
- Think of command lines like “sentences”
    - sequences of “word” – tokens
        - various characters can be used to split word – easiest are “white space” characters
            - quotes can be use to suppress splitting
    - newline : 0xa – ‘\n’ terminates our sentences – like our period
        - command processing starts
- simple command lines
    - first word is command
    - rest are arguments to the command
- “syntax” will let us compose more complicated “sentences”
    - combine multiple commands into a single command line

NOTES

See book for details

- some “special” characters (“meta-characters”) have special meaning
    - | & ; ( ) < > newline
        - note these will also cause things to be split into words
            - eg things on the right will be treated as a new word
                - sometimes they will also terminate a simple command (thing on the right will be a new command)
- some special words (“reserved”) have special meaning depending on where they occur on the line

2.3. The Shell as a Programming Language

The Glue that holds the UNIX Universe together

Basic Syntax : Notes

- comments
- echo and printf
- simple variable syntax and usage
    - quoting to control splitting and expansion
    - many more fancy variable expansions and special variables : `help variables` but plenty more
- basic programming constructs
    - for lists of words
    - commands and exit codes
        - (())
        - [[]]
        - [ ]
- interactive vs scripts

- history and arrow keys
- scripts
  - ascii file with lines of shell commands
  - mark as executable and put in path or name explicitly

2.4. Shell Programming by example

USE TERMINAL and ILLUSTRATE

- Much of the built in capabilities of the shell is to provide us a programming interface
  - comments
  - set and use variables
  - loops: repeat commands
  - support for basic math
  - exit status – a way to support conditional constructs
    - ((stmt)) : can do math statement and return an exit status
    - [[test]] : can do a bunch of tests and return an exit status
    - the exit status of external commands

2.5. Shell Programming

```
# Comments
# a word that starts with # and everything else till end end of the line is
ignored

# echo – basic built in for outputting
echo hello

# shell variables and variable expansion has lots of subtly but lets start
simple
i=hello

# remember shell expansions will happen to the line prior the "echo" built in
being
# executed

echo $i  # $ -- triggers parameter/variable expansion

# quoting used to control expansion and splitting

x=goodbye and farewell

# single quotes suppresses all expansion and splitting – literal value of each
character
# single quotes cannot contain single quotes
x='goodbye and farewell'
echo $x
```

2.6. Navigating and working with the “File System”

USE TERMINAL and ILLUSTRATE

- Current Working Directory
  - print working directory (`pwd`)
- shell path name expansion : `echo *`
- Externals:
  - `ls` daily bread and butter command
  - `ls -l` see meta data
  - `find` - a even more powerful tool
- Changing Working Directory: `cd`

- Path names: full path vs relative paths
  - `.` and `..`
  - `~` expansion
  - `.` as a way of "hiding" files and directories
- Creating directories: `mkdir`
- Removing directories: `rmdir`
- Creating files : `touch`
- Removing files : `rm`

2.6.1. File and Directories

```
# Current working directory
pwd

# What is in the current directory?
#  Another expansion - Filename expansion the shell does for us.
echo *
echo /*


# ls is a powerfull external command for list contents of directories
ls
man ls
ls -l
ls -lrt
ls -lhrt
ls -1

# change working directory
#
cd /
pwd
ls
cd home
pwd
ls
cd jovyan
cd /home/jovyan
```

Review here

2.7. Variables in detail

- has a name and stores a value
- variables are assigned a value with `name=[value]`
  - Note cannot have spaces between either name `=` or `=` and value
  - by default variable are "string" variables
  - NULL/empty string is a valid value eg. `x=`
  - value of an assignment is subject to the following expansions:
    - tilde
    - parameter and variable expansion
    - quote removal
    - if variable has integer attribute then arithmetic expansion is done
  - one more attributes that can be assigned with the `declare` builtin
  - no need to declare
    - eg. assignment to a non-exiting variable creates it
  - can mark some variables for export (environment variables)
    - "exported" variables are copied via UNIX environment to child processes
      - Environment a set of strings of the form : "NAME=STRING"
      - Bash has the kernel copy the current set of exported variables when it launches a new program. Programs can inspect these values and customize their behavior. (eg. locale)
      - `export <Variable>[=Value]` (can do the same with `declare -x`)
- variable expansion and quotes - `echo $x $y` vs `echo 'echo $x $y'` vs echo "$x y" -echo "$x $y"`
- some special unnamed variables (called parameters)

    - actually variables are technically parameters with names
    - `1` - `n` – positional parameters expand to ordered arguments of script or function
    - `@` and `*` – expands to positional parameters however with differing expansion rules
    - `#` – expands to number of positional parameters
    - `?` – expands to exit status of last command/pipeline
    - others – `-`, `$`, `!`, `0`
- Advanced: `declare`
    - integers
        - `declare -i myvar; myvar=myvar+1; echo $myvar` vs not using the declare -i?
    - indexed arrays
    - associative arrays
    - ref variables
    - other built-ins that are special forms of `declare`
        - `typeset` (same as `declare`)
        - `export` (like using `declare -x`)
        - `readonly` (like using `declare -r`)
        - `+=`
        - `unset`
- things that look like variables but are really fancy expansions
    - `$` actually triggers three types of expansion
        1. parameter expansion
        2. command substitution
        3. arithmetic expansion
    - Parameter expansion
        - simple: `$name` or more explicitly `${name}`
        - many advanced forms: Here are a few that I constantly use
            - setting default value eg. `foo=${foo:-START}`
            - delete prefix eg. `foo=ABCA.txt; echo ${foo##*.}`
            - delete suffix eg. `foo=ABCA.txt; echo ${foo%%.txt}`
            - substitution/deletion eg. `foo=ABCA.txt; echo ${foo//A/O}` or `echo ${foo//A/}`
    - Command substitution
        - very useful run a command and treat its output as a value
        - `foo=$(ls); echo $ls` or `n=$(ls -1 | wc -l); echo $n`
    - Arithmetic expansion: `$(())`, or `(())` or let, or if integer types variables
        - Shell knows how to do basic integer math and some binary/bit operators
            - `++`, `--`, `+`, `-`, `!`, `~`, `**`, `*`, `/`, `%`, `<<`, `>>`
            - `<=`, `>=`, `<`, `>`, `==`, `!=`
            - `&`, `^`, `|`, `&&`, `||`
            - `expr ? expr : expr`
            - `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`
            - number formats:
                - prefix: `0` - octal, `0x` - hex, `[#base]number` – eg. `((x=2#101)); echo $x`

2.8. Shell Commands

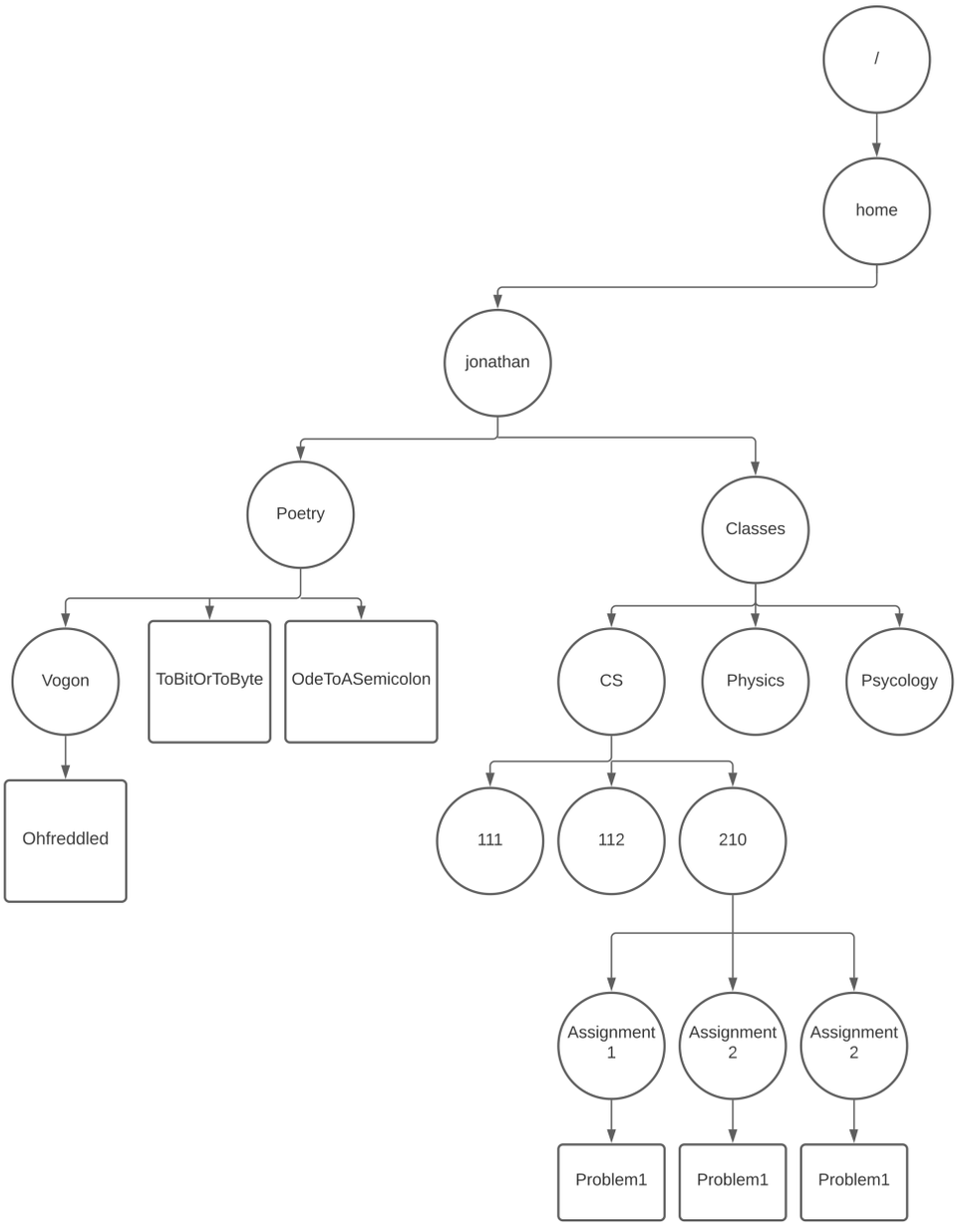
See. https://www.gnu.org/software/bash/manual/bash.html#Shell-Commands 2. Breakdown complex commands into simple commands

- complex commands are a composition of simple commands:
    - pipelines
        - connecting inputs and outputs of commands together using `|`
        - `ls -1 /bin | wc -l`
    - lists

        - one or more pipelines separated by one of: `;`, `&&`, `||` or `&` and terminated by `;` or `;` or `&` or a newline.
            - `echo "about to run ls"; ls; echo "done running ls"`
            - `cmd1 && cmd2` : run `cmd2` if `cmd1` returns “success”
            - `cmd1 || cmd2` : run `cmd2` if `cmd1` returns “not success” aka “failure”
            - `cmd1 & cmd2` : run `cmd1` asynchronously in the background and run `cmd2` “normally”
                - lots more to say about asynchronous commands
    - compound commands: Shell programming constructs (these can span lines)
        - loops: `until`, `while`, `for`, `do`, `done`, `break`, `continue`
            - there are two type of `for` loops
        - conditionals: `if`, `then`, `elif`, `else`, `fi`, `case`, `select`, `(())`, `[[]]`
        - grouping: `()`, `{}`
- functions

2.9. Exercise

Try and recreate the following



By Jonathan Appavoo