BIOS 611 HW 2

Brian Gural

2023-09-23

Table of Contents

## Problem 1

**Question**

Suppose you run this code.

alias hello="echo hello world"  
hello

Given that “hello” isn’t a file on the path, describe how to modify your “evaluation strategy” to account for this new behavior.

**Answer**

If the goal of the evaluation strategy is to understand what an *incantation* is doing, we can use use type to interogate the command:

(base) [bgural@longleaf-login5 ~]$ type hello  
hello is aliased to `echo hello world'

This lets us know that it’s *aliased* to this other command, echo hello world, which we can understand with our prior evaluation strategy. I.e. we would assumed the first word is a command, which can be interogated with help and man, followed by the command of interest.

## Problem 2

**Question**

Execute the following code:

alias zz=zz  
zz

Does this produce an error message? If it does, what can you conclude about how alias works?

alias aa=bb  
alias bb=aa  
aa

Given what we have learned above (and without running the code\*), what do you expect this to do?

* You can always run the code, but try to figure this out without doing so.

**Answer**

The alias in alias zz=zz is trying to point to a nonexistent command to create a command, which doesn’t amount to a workable command. It fails since it calls something that was never successfully defined. The error message bash: zz: command not found confirms that it was never defined.

Similarly, the next example of the two commands pointing to each other will fail, since they’re both pointing to unrecognizable, undefined commands. It’ll be like the Spiderman pointing meme, but there are only two and they’re both not real.

## Problem 3

**Question**

Create a file called “experiment.sh” that looks like this:

#!/bin/bash  
echo "argument number one is $1"  
echo "argument number two is $2"  
echo "rest of the arguments ${@:3}"  
echo "all arguments $@"

And invoke it by saying:

chmod u+x ./experiment.sh  
./experiment.sh a b c d e

Describe some new evaluation rules that explain this behavior. Experiment! You don’t need to get this exactly right.

**Answer**

The experiment.sh file is being registered as an executable by chmod u+x ./experiment.sh, so we can call can use it’s file path as a command. So we could add “if the incantation starts with a file path, check to see if it’s pointing to an executable file.” We can check the file itself with code or nano and then the file path. We could confirm that it’s executable with:

ls -l experiment.sh   
-rwxr--r-- 1 bgural rc\_raulab\_psx 137 Sep 23 09:39 experiment.sh

The x is telling us that it is executable.

The file itself is taking trailing arguments from the bash line and printing them back out.

(base) [bgural@longleaf-login5 hw\_2]$ ./experiment.sh this is an example  
argument number one is this  
argument number two is is  
rest of the arguments an example  
all arguments this is an example

## Problem 4

**Question**

The first line of the above script (#!/bin/bash): what do you imagine it does, if anything? What is its purpose? Feel free to Google this one or consult some kind of robot.

**Answer**

The first part, #!, is a statement that amounts to “Hey, use this following interpreter when figuring out how to decipher the following text nonsense as a language. The bit that follows is the path of the interpreter that should be use. Plus, it lets humans who read the code know that it’s a bash script.

## Problem 5

**Question**

Invoke the script like this:

bash ./experiment.sh a 'b c' d

Describe a new evaluation rule to explain the results.

**Answer** The bash opener kind of makes the #!/bin/bash portion of experiment.sh redundant, since it’s specifying that whatever follows should be evaluated as a bash script. Another interesting thing is the 'b c', since it’s being evaluated as a single arguement. This tells us we should add the following to our evaluation rules:

Trailing arguments are separated by spaces, **except when they are encompassed by quotations.** In that case, they are considered a single argument.

## Problem 6

**Question**

Reading documentation is very important for technical work, and we ought not be intimidated by it. Thus, examine the Docker documentation and explain to me the difference between RUN and CMD lines in a Dockerfile.

**Answer**

RUN is for things that you’ve decided you need to be done while building a container, like installing package. These are each layers in the resulting image. They are built on top of each other and are generally cached once run for the first time.

CMD is used to specify what happens once the container is started from the image, like which scripts should be run. They are not built upon each other, instead the last CMD replaces the instructions from the prior.

## Problem 7

**Question**

We use a variety of package managers in this class. Briefly describe each of the following:

apt  
pip  
install.packages

**Answer**

apt is pre-compiled and fast. Offers automatic update notifications, checks if other dependencies are installed or conflicting. Generally safer.

pip doesn’t have the same guardrails that apt has. It can install incompatable software that can brick the machine. Geared towards python exclusive projects. Easier to get more up-to-date packages, can install in virtualenv.

install.packages is an R language package manager. It’ll look for the newest version of a package managed by CRAN. For biologically-minded packages and those managed by Bioconductor, BiocManager::install() can be more convenient.

## Problem 8

**Question**

Get man ls and other manuals to work in the rocker terminal

**Answer**

This is the script I added to Dockerfile to get access to the man command in the rocker terminal:

# Update package list and install man-db  
## This is to see the manual of bash commands while in the Rstudio container  
RUN apt update && apt install -y man-db && rm -rf /var/lib/apt/lists/\*  
  
## Unminimize the system  
RUN yes | unminimize

In a nutshell, it gets an up-to-date list of packages it can install, then installs man-db, says yes to all of it’s prompts, then removes the apt list from earlier to keep the image space tidy.

The next RUN line undoes the minimized installation, giving full access to the man/documentation pages. It also needs “yes” responses, so we use the yes command to pipe an infinite series of affirmations to our needy command. After, we can build and run the image. In the terminal, we have full access to man:

rstudio@03b2cd749247:~$ man ls  
LS(1) User Commands LS(1)  
  
NAME  
 ls - list directory contents  
  
SYNOPSIS  
 ls [OPTION]... [FILE]...

## Problem 9

**Question**

Compare number of lines in find, man, and ls manual pages:

**Answer**

First, I made a shell script and started editing with:

guralbrian@bleep:~/github/spotify\_data$ touch homeworks/hw\_2/q9.sh  
guralbrian@bleep:~/github/spotify\_data$ code homeworks/hw\_2/q9.sh

Then I filled it with the following:

#!/bin/bash  
  
# Temp, empty file to store results  
> results.txt  
  
# Loop through each command and   
# Record number of lines in its man page  
# get manual, pipe into word count by lines, pipe into awk to format the output as command + lines  
for cmd in man ls find; do  
 man $cmd | wc -l | awk -v cmd=$cmd '{print cmd "," $1}' >> results.txt  
done  
  
# Part to sort the contents of the temp results.txt by number of lines  
# Prints the results too, with awk to do the formatting  
sort -t, -k2,2 -g -r results.txt | awk -F, '{print $1 " has " $2 " lines in its man page"}'  
  
# Remove the temporary results file  
rm results.txt

Then I made it executable and ran it, with

guralbrian@bleep:~/github/spotify\_data$ chmod +x homeworks/hw\_2/q9.sh   
guralbrian@bleep:~/github/spotify\_data$ homeworks/hw\_2/q9.sh   
find has 1660 lines in its man page  
man has 699 lines in its man page  
ls has 242 lines in its man page

## Problem 10

**Question**

Get the project repo, docker, and readme.md set up

**Answer**

[Here it is!](https://github.com/guralbrian/spotify_data)