

CS51 PROBLEM SET 0: GETTING STARTED ON MAC

0. INTRODUCTION

This problem set has several goals.

First, we will walk you through setting up your OCaml development environment. We do this rather than providing automated setup so that you have a sense of what a local development environment (rather than, for example, Cloud9) requires, but it should not take much longer than running a script would.

After your environment is configured, you'll finish up the problem set by writing a first program in OCaml.

This problem set is graded only for submission. Submission means filling in `ps0.ml`, pushing to `GITHUB`, and submitting to `GRADESCOPE`.

There are questions listed throughout the problem set whose answers would be useful for you to know, but written answers to the questions are not required.

1. SETTING UP A DEVELOPMENT ENVIRONMENT

While Macs in the general population are popular in part because of their sleek user interfaces, in the computer science community Macs are prevalent because their operating system is part of the **Unix family**.

Unix is a family of operating systems developed beginning at Bell Labs in the 1970s. Mac OS has many newer user interface features, but some of the original features from Unix will be the most useful in CS 51.

Users of the earliest Unix systems interacted with their computers exclusively through the keyboard, via a `COMMAND LINE`. They would type commands (short programs) that directed the computer to take action, like running a program to switch to a different folder in the file system or to create or delete a text file.

If you took CS 50, you ran commands in the command line on the CS 50 IDE. While you may never have done so before, you can also interact with your Mac via the command line. You will spend a good deal of time at the command line in CS 51, and comfort at the command line, developed over time, will be one of the skills you build in this course.

The first step is to open a `TERMINAL` window. You can do this by searching for the Terminal application or find it in the Applications > Utilities folder.

Date: January 24, 2017.

Question 1. *Open a Terminal. What is the Terminal? What is it for?* □

Let's investigate some programs that are useful for interacting at the command line.

Question 2. *Give a two or three word description of each of the following programs.*

(1) *man* 35

(2) *mkdir*

(3) *cd*

(4) *ls*

(5) *rm*

(6) *touch* 40

□

The *man* and *touch* commands may be new to you; hopefully the others are familiar. The *man* command will be helpful through the semester when you need information about your system or a program. We call these “commands”, but they are actually programs. Thinking of them as programs will be helpful when we start to install new ones, and you should be aware that these programs can be run with many different interesting and powerful options. (Take a look at *man ls*. Type *q* to close.) 45

For more guidance on the command line, check out [this link](#).

Now that you know what the command line is and how to move around, it's time to start installing software. You'll need a number of different things: a version control system called *git*, the OCaml compiler and interpreter, and the OCaml package manager, among others. 50

To install all of these things, you could search for instructions on the web and download .zip files. This is how you may have installed software in the past. Since we're going to be building our own software, however, we care a great deal about keeping careful track of what software we have installed. As such, we're going to use a PACKAGE MANAGER. With a quick web search, figure out what the most widely used package manager is for your system. 55

PACKAGE MANAGER

Question 3. *What's an appropriate package manager for your system? What does a package manager do?* 60 □

There isn't one strictly correct answer to the question above for Mac systems. We recommend using [Homebrew](#), which you'll install as *brew* at the command line.

1.0.1. *Installing brew.* Head to <http://brew.sh> and follow the directions there to install the *brew* command line tool. The standard installation should work. 65

To double check, type following command.

```
$ brew
```

You should see a help menu printed. (Throughout the course, the \$ symbol
70 represents the Unix command line prompt, and indicates commands that should be typed at the Unix command line. You should not type that symbol.)

To make sure that brew can work properly, you should also run this command:

```
$ sudo chown -R $(whoami):admin /usr/local
```

Homebrew places programs it installs into the /usr/local directory. This com-
75 mand makes sure that your system lets you install things there, via brew, by adding your account to the list of that directory's owners. You may need to type your Mac password. (No characters appear when passwords are typed at the command line.)

1.1. **Installing OCaml.** Now that you have a package manager, you can install the
80 software that you will need. Follow the directions that are appropriate from your system.

First, update Homebrew's listing of available packages. Homebrew's develop-
ers recommend updating it twice to be safe.

```
$ brew update
```

```
$ brew update
```

85 At the end of the semester, you will build a program that has a graphical user interface (GUI, pronounced *goo-ey*). OCaml uses the **X system** to display GUIs, but Apple doesn't install X on Mac OS by default anymore. To install it yourself, run:

```
$ brew cask install xquartz
```

You're now ready to install OCaml. Again, we'll need the OCaml graphics
90 libraries at the end of the semester.

```
$ brew install ocaml --with-x11
```

Question 4. *What does --with-x11 do? What is this type of addition to a command called?* □

Finally, we need to install opam, the package manager for OCaml.

95 \$ brew install opam

1.2. **OCaml versions and packages.** You may be asking yourself why we just in-
stalled a package manager specifically for OCaml after installing a general package
manager for your system.

This is a reasonable question, and luckily it also has a reasonable answer. A
100 software project is often developed by many different people, each of whom may be running a different operating system. Many software projects also take advantage of external libraries that have been made available to other developers. Sharing

previously written libraries and packages for specific programming languages across multiple operating system package managers (note that there are 4 different version of this document), and keeping all of the different listings in sync with each other, would be a true challenge of coordination. This complexity arises before considering the fact that several different projects that one developer is working on may each require a different version of the same programming language. 105

ABSTRACTIONS To solve this problem, the developer communities for most popular programming languages have built ABSTRACTIONS between the libraries for and versions of their language and the operating system in the form of language-specific package managers. Each system's version of a language-specific package manager knows how to install libraries on that system, so that the authors of a library need only describe how the single manager should install the library. 110

OPAM is the package manager for OCaml. Some other examples of language-specific package managers are pip for Python, npm for Node.js, and gem for Ruby. 115

Question 5. *What's the benefit of language-specific package managers? In what way do they serve as abstractions?* □

To set up OPAM, run:

```
$ opam init -a 120
```

The official version of OCaml used in CS 51 this year is 4.02.3. This is most likely not the version installed by default by the system package manager. In addition to managing OCaml packages, OPAM can also manage OCaml versions. To install the correct version, run:

```
$ opam switch 4.02.3 125
```

Now that you have the right version, you can install some packages that you will need during the course.

```
$ opam install -y ocamlbuild
$ opam install -y ocamlfind
$ opam install -y ocamlnet 130
$ opam install -y yojson
$ opam install -y merlin
$ opam install -y utop
```

After all of this is done, you should finish the process by running

```
$ eval `opam config env` 135
```

N.B. the command above should have no output. If you see output, the ` character was likely not copied correctly. Type the command manually, including that

character. Note that ``` is the character at the upper left, below the escape key, and not a single quote.

140 **At this point, close anything you are working on and restart your computer.**

2. VERIFICATION

Now that you have OCaml installed, to verify that everything went well, open your system's Terminal and type:

```
$ ocaml
```

145 You should see the following:

```
OCaml version 4.02.3
```

```
#
```

150 This is the OCaml READ-EVAL-PRINT LOOP (REPL), where you can type or paste OCaml code and see the output evaluated. To quit the REPL, press `Ctrl + D` to send an EOF character, or `#quit;;` to call the REPL's quit command.

READ-EVAL-PRINT LOOP

You also installed a more-fully featured OCaml REPL, called `utop`. It has useful features like auto-completion built-in. To make sure that is working, type:

```
$ utop
```

155 It should also identify 4.02.3 as the version of OCaml it is running. It can be quit in the same way as the `ocaml` REPL. Then verify that typing the string "are we there yet?" (including quotes) followed by two semicolons and pressing enter results in something like this.

```
# "are we there yet?";;  
- : string = "are we there yet?"
```

(The `'#'` character represents the OCaml prompt; you don't need to type it.) You just wrote your first piece of OCaml.

160

3. SETTING UP GIT

Before reading this section, you should watch [this video](#) about the version control system `git`.

3.1. **Sign up for GitHub.** We will be using `git`, a popular source control system, to distribute problem sets to you, and you will likewise use `git` to submit your work. The `git` repositories we will work with will be remotely hosted on [GitHub](#), a `git` service offering remote repositories hosted in the cloud.

165

If you don't already have a GitHub account, follow [these instructions](#) to create one.

3.2. Adding an SSH key. Use SSH to let GitHub identify you and your computer. This lets you avoid having to type in your GitHub password every time you push to save your code remotely (which you should do often). To set up SSH authentication, follow these articles from GitHub: 170

- (1) [Checking for existing keys](#)
- (2) [Generating a new key \(if necessary\)](#) 175
- (3) [Telling GitHub about your key.](#)

git should now be fully configured!

4. GETTING THE SOURCE CODE

We hand out problem set distribution code using [GitHub Classroom](#). Every problem set specification will contain a link to the distribution code. When you click the link, GitHub will create a repository for you to use, and, if applicable, for you to share with your problem set partner. 180

4.1. Creating the remote repository. To create your repository for this homework, go to <http://tiny.cc/cs51ps0> and follow the directions about GitHub.

4.2. Cloning the code. You now have a remote repository to store your homework. In addition, you'll also need a local repository so that you can make changes and then git push them to GitHub. 185

Following the directions listed [here](#), clone the remote repository that was just created for you above. We recommend creating a folder to hold all of your problem sets (using `mkdir`). 190

If all went well, you should have seen something like this:

```
Cloning into 'cs51/ps0-student'...
remote: Counting objects: 51, done.
remote: Compressing objects: 100% (51/51), done.
remote: Total 51 (delta 51), reused 51 (delta 51)
Receiving objects: 100% (51/51), 51 MiB | 51 MiB/s, done.
Resolving deltas: 100% (51/51), done.
```

195

Using `cd`, change into the directory that was just cloned. When you run `ls`, you should see `ps0.ml`, `ps0_tests.ml`, `makefile`, and `_tags`, as well as several versions of this document. 200

4.3. Checking for Updates. We may occasionally publish changes to the distribution code after a problem set has already been released. If they occur, the changes will be small, and we may not notify you. If they are large, we will send an email to the course and post on Piazza.

205 `git` should make it painless to keep up the distribution code up-to-date. At the beginning of each assignment, we will ask you to add an extra **REMOTE**, which is a connection between a local repository and a remote repository, where we will upload the distribution code. **REMOTE**

To add this week's extra remote, run

210 `$ git remote add distribution git@github.com:cs51/ps0.git`

This tells `git` to track the remote repository and gives the remote repository the name "distribution".

To check for updates, run:

`$ git pull distribution master`

215 This command checks the remote repository for changes and merges them in, if possible, or alerts you to conflicts if manual merges are required.

5. WRITING YOUR FIRST OCAML PROGRAM

Good job making it this far. We know that getting your setup ready can be frustrating, but it's an important part of the process.

220 It's time to write and submit your first OCaml program. **Your job is to edit `ps0.ml`. The directions are inside.**

First, however, check that everything is working. Type:

`$ make all`

`$./ps0.byte`

225 You should see:

Name: FIRST LAST

Year: Other: I haven't filled it in yet

230 50?: Other: I haven't filled it out yet

I'm excited about!

235 -----

If that worked, **open up `ps0.ml` in your favorite text editor**. (Members of the staff recommend **Sublime Text 3**, and Prof. Shieber recommends Emacs.) **Follow the directions inside.**

6. SUBMITTING YOUR PROBLEM SET

Submitting homework happens in two phases. First, you'll commit your changes to your local git repository and push those changes to the remote GitHub repository. Second, you'll log in to Gradescope and tell us that you want to submit; Gradescope will then retrieve your submission from GitHub so that we can grade it. 240

6.1. **Using git.** When you modify some code, reach a good checkpoint in your coding (e.g., "fixed that bug!"), or finish your work, you can and should "commit" these changes by executing 245

```
$ git commit -am "some message here"
```

at the terminal from the directory in which you're working. **You should commit early and often, and push whenever you finish something important, so that your work is backed up on GitHub in the event of a computing emergency (like a spilled cup of coffee).** The -a flag ("a" stands for "all") here specifies that git should take note of *all* changes made in files that you have previously told git to track. 250

- You can tell git to track a file named `filename` by adding it, as in: 255

```
$ git add filename
```

- Since `ps0.ml` was already added, you don't have to add it again. However, if you were to make a new file, you would have to add it.
- You can add all of the files in your current directory (and subdirectories) to your repository by running 260

```
$ git add -all
```

This is probably the safest thing to do (to ensure you're tracking everything), but you'll probably not want to track the compiled binaries (like the `_build` directory and `ps0.byte` that you'll shortly be generating).

The -m flag ("m" for "message"), followed by some string, specifies a *commit message*, which is a short string describing the changes that have been made since the last commit. This can be useful in keeping track of exactly what changes you make throughout the development process. If you work on projects with other people, they can see what other changes you've made by reading these commit messages, which they (and you) can see by running `git log`. 265 270

Merely committing will store these changes in your computer's local copy of the repository. To "push" this repository's changes online to GitHub, execute:

```
$ git push
```

This will allow you to submit your work to Gradescope (though it doesn't perform the submission process itself; see below). **Equally important: it will also** 275

create a remote backup of your work so that, in the event of your losing access to your computer, you'll still have something to work with and submit. Again, commit and push early and often.

280 6.2. **Submitting on Gradescope.** We'll be using Gradescope, an online coding course management platform, to manage CS51 labs and problem sets this year.

After enrollments have been submitted, you will have received an email from `team@Gradescope.com` with a link to sign up for an account.

Once you have signed in, select CS 51 as the course and look for the appropriate assignment. Choose it and then click "Submit from GitHub."

285 After Gradescope receives your submission, it double checks that your code compiles against our unit testing framework. After this test is complete (it usually takes about 10 seconds), you will receive confirmation that your submission succeeded. Code submissions that do not compile are rejected by the system and count for no credit.

290 **Remember, pushing to GitHub does not complete your submission. You must submit inside Gradescope to complete the homework.**

You can make sure that the right files were submitted by clicking on the "code" tab on the assignment page after you have submitted.