CS51 PROBLEM SET 0: GETTING STARTED ON OLDER WINDOWS SYSTEMS

STUART M. SHIEBER

1. 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, a cloud-based integrated development environment (IDE)) requires, but it should not take much longer than running a script would.

After your environment is configured, you'll download a code file `ps0.ml` using the source control system `git` and the code distribution system Github Classroom, and then write a first program in `OCaml`.

This problem set is graded only for submission. Submission means filling in `ps0.ml` with your code, pushing your code to the git repository you created using the GitHub Classroom link, and submitting to the course's code submission and grading server, Gradescope. 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.

2. Windows 7 Setup

If you are following these instructions, you should be running a version of Windows older than Windows 10, or you should have tried the most up-to-date Windows setup first. If this is not the case, please switch to the `pset0-windows.pdf` document. Otherwise:

Type "system information" into the Windows search bar. Click on the System Information app, and there should be a summary of what architecture you system has (32 bit or 64bit). From here on, install the appropriate version of software that matches your system.

2.1. Install git shell (adapted from CS 109 instructions).

Follow this link: https://git-scm.com/downloads. Click install for Windows. An `.exe` file will download. Run it. Accept all defaults in the GUI installer that opens. A few defaults that you must be **certain** to follow can be found in Figures 1, 2, 3, and 4, which you should reference as you move through the installer screens.

Date: January 23, 2018.

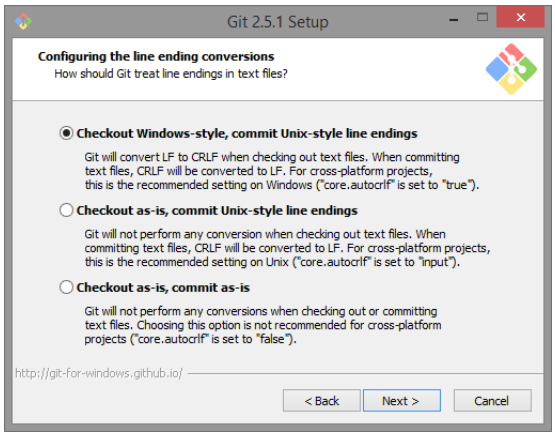

Figure 2. This ensures the proper default line-encoding conversion.

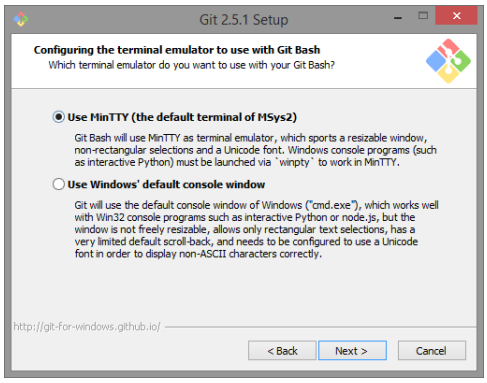

Figure 3. This uses a better terminal emulator than the one shipped on windows.

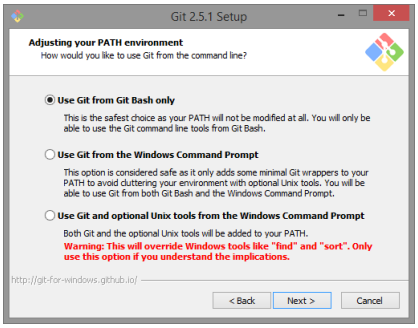

Figure 1. This makes sure that you get a bash shell to use.

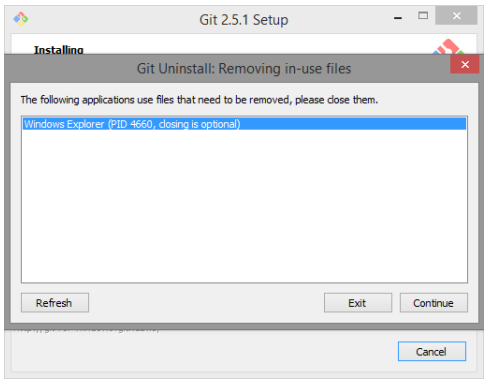

Figure 4. If you see this message, just click continue.

2.2. **Install make.** Follow this link: http://gnuwin32.sourceforge.net/packages/make.htm. Click on the setup link under "Complete package, except source". Another .exe file will download, run that. If you follow all default options, the confirmation page before you install `make` should look like Figure 5.

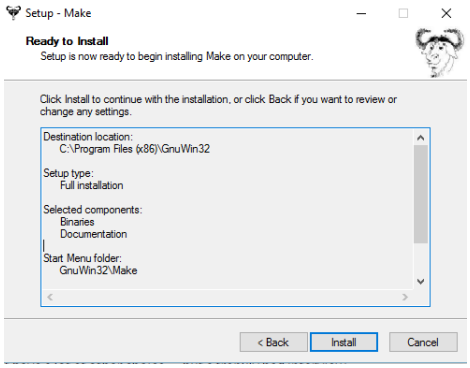

Figure 5. make installer

Note the directory in which `make` will be installed, as you will later need to add that directory to the System Path in Windows. Click "install", then wait for the process to finish. After it completes, go to your Control Panel and open up "System Choices". Click on "Advanced system settings", so that you see the System Properties window, which should look like Figure 6:

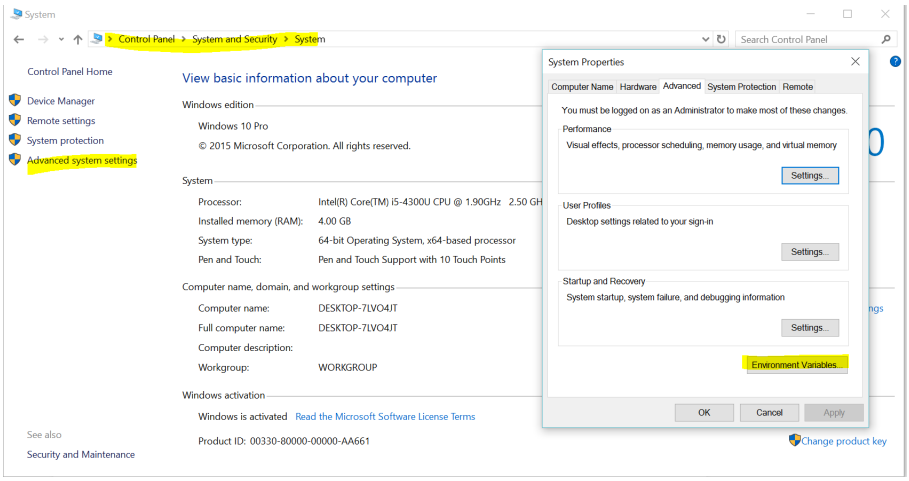

Figure 6. Control panel and system properties

Click on "Environmental Variables", as highlighted at the right of Figure 6. Click on the path variable under System variables and choose to edit it, as highlighted in Figure 7.

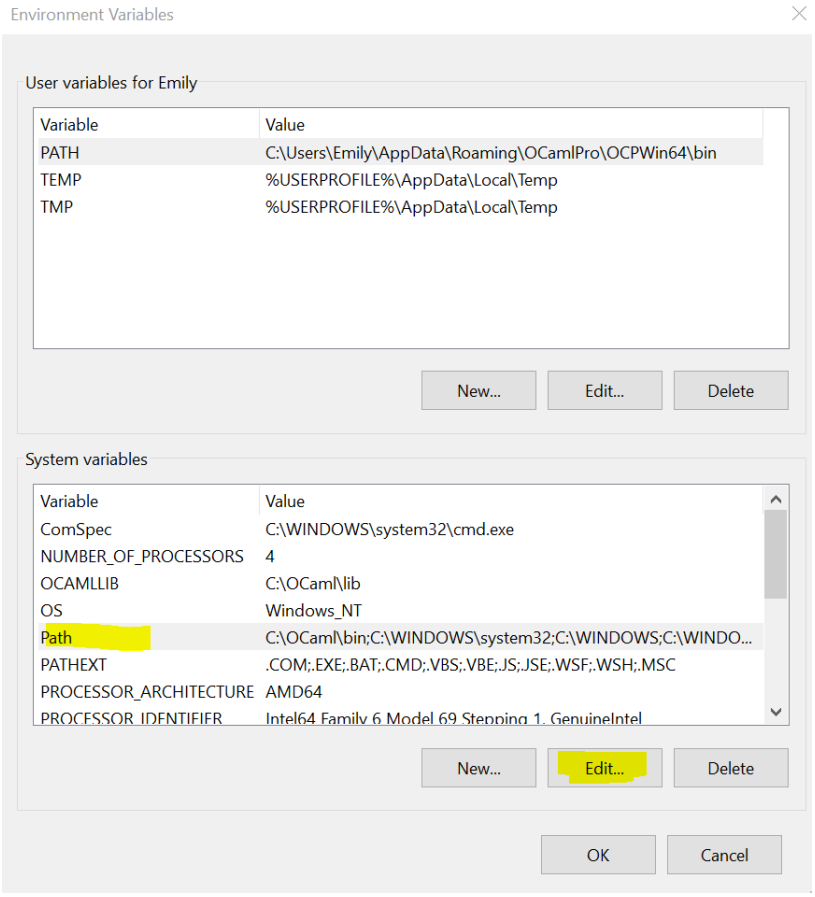

Figure 7. Finding Path environment variable to edit.

A page similar to Figure 8 should pop up.

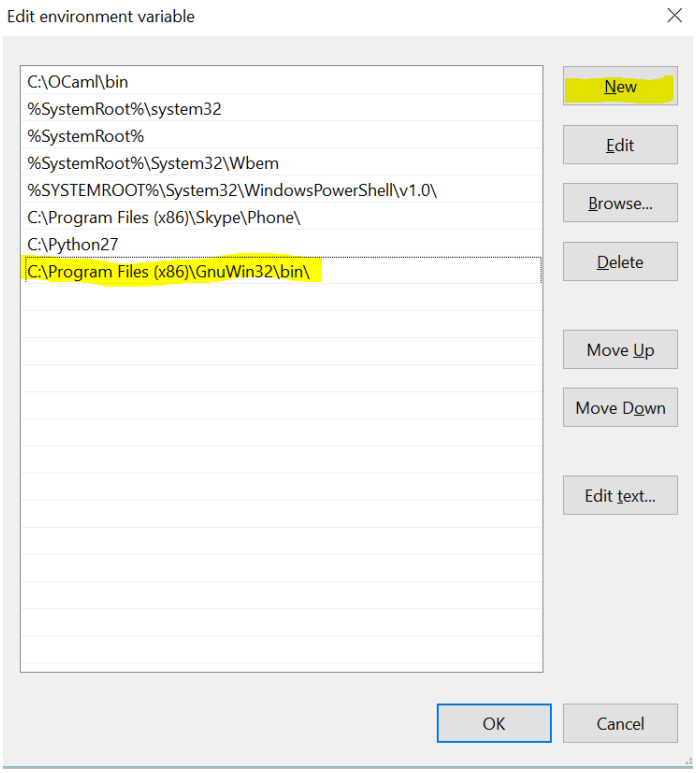

Figure 8. Altering your Windows Path

Add one more path on the bottom by clicking New, as highlighted in Figure 8.[1] For most people, the path that you should paste is highlighted in Figure 8. To ensure that this path is correct, search for make.exe on your computer, to determine where it was saved.

Save and apply all those settings.

[1] As noted by Nicholas on Piazza: "Just a heads up–the path variable editor shown in the Windows setup documentation is new to Windows 10's November 2015 release. In case some out there haven't made the leap yet: you'll need to scroll all the way to the end of the text in the box, add a semicolon if one isn't already there, then type the path."

2.3. **Install OCaml and `opam`.** Go to this link: http://protz.github.io/ocaml-installer/ and install the proper version of OCaml for your machine. (If you have a 64bit system, choose the largest link that says "64bit"; if you have a 32bit system, choose the first smaller link immediately below.) When running the GUI installer, you have to change one default option, where you uncheck cygwin installation:

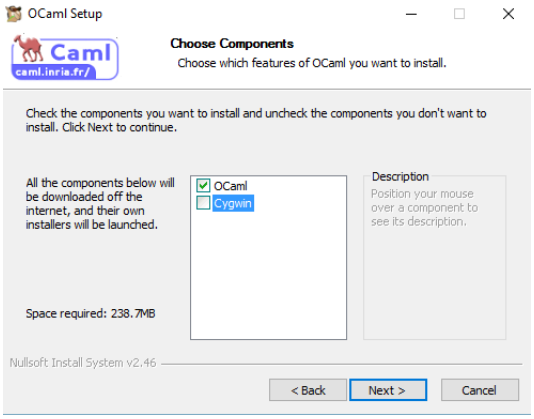

Figure 9. Installing OCaml

The OCaml installer adds to your path automatically, so you don't need to follow the same steps for make. From there, everything should be set up.

2.4. **Testing to see if things work:** If you press the Windows key, and then type `git bash` and run `git bash`, you should be able to navigate around your file structure using Unix commands like `ls`, `cd`, etc.

Proceed through the rest of the problem set. If you run into issues, check Piazza to see if they have been rectified. If not, then post a question of your own tagged with the `problem_set0` folder and the `windows_setup` folder.

2.5. **`OCaml` versions and packages.** You may be asking yourself why we just installed 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 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.

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 ABSTRACTIONS

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.

`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.

Question 1. *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
```

The official version of OCaml used in CS 51 this year is `4.06.0`. This may not be 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 4.06.0
```

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
$ 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`
```

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

3. Verification

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

```
$ ocaml
```

You should see the following:

```
OCaml version 4.06.0

#
```

READ-EVAL-PRINT LOOP

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 type `#quit;;` to call the REPL's quit command.

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
```

It should also identify `4.06.0` 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`.

4. Setting up `git`

Before reading this section, you should watch this video about the version control system `git`.

4.1. **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.

If you don't already have a GitHub account, follow these instructions to create one.

4.2. **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:

(1) Checking for existing keys
(2) Generating a new key (if necessary)
(3) Telling GitHub about your key.

`git` should now be fully configured!

5. 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.

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

5.2. **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.

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`).

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.
```

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.

5.3. **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.

`git` should make it painless to keep up the distribution code up-to-date. At the beginning of
REMOTE 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.

To add this week's extra remote, run

```
$ 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
```

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

6. 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.

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
```

You should see:

```
----------------------------------------
Name: FIRST LAST

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

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

I'm excited about ....!

----------------------------------------
```

If that worked, **open up `ps0.ml` in your favorite text editor** (You may want to look at CS51's text editor guide which provides advice on setting up Sublime Text 3 or Atom for OCaml programming). **Follow the directions inside.**

7. 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.

7.1. **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

```
$ 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.

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

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 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.**

7.2. **Submitting on `Gradescope`.** We'll be using `Gradescope`, an online coding course management platform, to manage many aspects of CS51 labs and problem sets this year – submitting your solutions, automatically testing them, grading them, and providing feedback to you.

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."

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.

We have created a Google Chrome extension that provides helpful messages in case there are any problems with your Gradescope submission. If you use Google Chrome, you should download the CS51 Extension from the Chrome Web Store.

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.