Problem set 0 [#](https://cs51.io/guides/ps0-instructions/#problem-set-0)

Before we get to the substance of the course, this problem set makes sure that your development environment is set up, that you have access to the OCaml tools, that you’ve got git set up properly, and that you’re generally ready to get to work on the labs and problem sets in the course. We won’t be grading this problem set, so it’s your chance to get all of the kinks out of the development and problem set submission process.

You should start by following the OCaml setup instructions in the “[Setting up your OCaml development environment](https://cs51.io/guides/setup/)” (see below) document. This might take a while, so we suggest getting an early start. Then come back to this document for the rest of the problem set.

You’re back? Great. 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. Now that you’ve done it, you’re ready to write some code.

Creating and cloning the remote repository [#](https://cs51.io/guides/ps0-instructions/#creating-and-cloning-the-remote-repository)

If you aren’t already familiar with the git version control system or you don’t have a github account, follow the instructions in the “[Setting up git](https://cs51.io/guides/git/)” (see below) document.

To create your repository for this homework, go to <http://url.cs51.io/ps0> and follow the directions to create a remote repository for you to use for this problem set. See the “Setup” section of the “Problem set procedures for CS51” document for more detailed information.

You now have a remote repository to store your problem set, and a local clone of that repository. Using cd, enter the directory that was just cloned. When you run ls, you should see the files ps0.ml, ps0\_tests.ml, makefile, and \_tags.

Compiling and running the problem set code [#](https://cs51.io/guides/ps0-instructions/#compiling-and-running-the-problem-set-code)

We’ve provided a makefile that specifies instructions to compile the problem set. When in the root directory for the problem set, you can run the command

% make all

to compile the problem set, generating a file ps0.byte. Then run the compiled file with the command

% ./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 ....!

----------------------------------------

Writing your first OCaml program [#](https://cs51.io/guides/ps0-instructions/#writing-your-first-ocaml-program)

It’s time to write and submit your first OCaml program. Your job is to edit ps0.ml.

Open up ps0.ml in your favorite text editor. (If you don’t already have a favorite text editor, see the “Programmers' editors” handout for our advice on text editors for OCaml programming.) Follow the directions inside ps0.ml.

After you’ve made your changes, you should recompile and run the code. Once it is working to your satisfaction, it’s time to submit the problem set to Gradescope.

Submitting your problem set [#](https://cs51.io/guides/ps0-instructions/#submitting-your-problem-set)

To submit the problem set, you should read and follow the instructions in the “Submission” section of the document “Problem set procedures for CS51”.

# Setting up your OCaml development environment [#](https://cs51.io/guides/setup/#setting-up-your-ocaml-development-environment)

This document should help you get up and running with an OCaml development environment on your computer. We assume you have a Unix-like operating system available, such as MacOS or Linux. (For Windows users, we’ll provide instructions on setting up Linux in a virtual machine.)

## Background on the Unix command line [#](https://cs51.io/guides/setup/#background-on-the-unix-command-line)

Regardless of the operating system of your computer, you’ll be interacting with it using the Unix operating system’s command line.

Unix is a family of operating systems, originally developed at Bell Labs in the 1970s by Ken Thompson and Dennis Ritchie. Operating systems have many newer user interface features, but some of the original features from Unix will be the most useful in CS 51.



#### Ken Thompson and Dennis Ritchie

Ken Thompson (left; 1943–) and Dennis Ritchie (1941–2011), co-developers (with several others) of the Unix operating system, which underlies the design of many of the current operating systems in use today. They were awarded the Turing Award in 1983 for their work on Unix.

Users of the earliest Unix systems interacted with their computers exclusively through the keyboard, via the command line. They would type commands (names of 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, Linux, or Windows machine 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.

As it turns out, macOS and Linux are both built on versions of Unix. On Windows, alternative command line facilities can be used. This common substrate for interacting with the computer makes it possible to work similarly for this course on the various operating systems.

## Accessing the Unix command line [#](https://cs51.io/guides/setup/#accessing-the-unix-command-line)

On macOSOn WindowsOn Linux

## Interacting with the Unix command line [#](https://cs51.io/guides/setup/#interacting-with-the-unix-command-line)

Interacting with the command line starts at the command line prompt, which is typically a % or $ though it may include much other information. We will use the % symbol to represent the Unix command line prompt in our examples. (You should not type that symbol itself at the prompt.)

Let’s investigate some programs that are useful for interacting at the command line. (If you are already familiar with Unix, feel free to skip this section.)

* man – The man command (short for ‘**man**ual’) provides documentation on other commands. Try typing
* % man mkdir

Type q to exit the manual page reader.

* mkdir – The mkdir command (short for ‘**m**a**k**e **dir**ectory’) makes a new directory (or in macOS parlance, folder) in the current directory that you are currently accessing. Try typing
* % mkdir testdirectory
* ls – The ls command (short for ‘**l**i**s**t directory’) lists the contents of the current directory. Try typing
* % ls

It should show the files and subdirectories of your main (‘‘root’') directory, including the testdirectory directory that you just made.

* cd – The cd command (short for ‘**c**hange **d**irectory’) changes the current directory. Its argument specifies the directory to move to. You can give a subdirectory of the current directory to change to that subdirectory, or ‘'..’’ to change to the immediately enclosing directory. Try typing
* % cd testdirectory
* % ls

It should move you to the testdirectory you just made, and list its contents. (There shouldn’t be anything there.) Now

% cd ..

to move back to the root directory.

* rm – The rm command (short for ‘**r**e**m**ove file’) deletes a file. Its argument specifies the file or files to be deleted. Be careful with this one.

We call these ‘‘commands’’, but they are actually programs. Thinking of them as programs will be helpful when you 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 exit the manual page reader.)

For more guidance on the command line, check out [this link](http://swcarpentry.github.io/shell-novice/).

## Package managers [#](https://cs51.io/guides/setup/#package-managers)

Now that you know what the command line is and how to move around, it’s time to start installing software. Your development environment for the course will include the OCaml programming tools (compiler, interpreter, build system, package manager); the git version control system; and a text editor for editing software.

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.

On macOSOn Linux

## Installing software [#](https://cs51.io/guides/setup/#installing-software)

Now that you have a package manager, you can install the other software that you will need.

On macOSOn Linux

## Installing the OCaml tools [#](https://cs51.io/guides/setup/#installing-the-ocaml-tools)

You may be asking yourself why we 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 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. Package managers for particular languages thus serve as an abstraction mechanism over the various operating systems they support.

### **Setting up OPAM**[**#**](https://cs51.io/guides/setup/#setting-up-opam)

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

To set up OPAM, run:

% opam init -a

At this point, close your terminal window and open a new one.

The official version of OCaml used in CS 51 this year is 4.11.1. 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 create 4.11.1

% opam switch 4.11.1

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

% opam install -y graphics

% opam install -y ocamlbuild

% opam install -y ocamlfind

% opam install -y ocamlnet

% opam install -y yojson

% opam install -y merlin

% opam install -y utop

We also have created a special package for CS51 called CS51Utils. Feel free to peruse the CS51Utils package on Github. This is how OCaml distributes third party packages, similar to the ones you installed above. This package contains some useful functions that you can use across the problem sets and labs in the course. You will be introduced to these functions during the labs.

% opam pin add CS51Utils https://github.com/cs51/utils.git -y

After all of this is done, you should finish the process by running the following command, which should have no output:

% eval $(opam env)

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

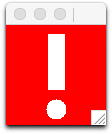
### **Verifying that OCaml graphics is working**[**#**](https://cs51.io/guides/setup/#verifying-that-ocaml-graphics-is-working)

Next, we want to check whether OCaml graphics is properly installed on your computer. To verify that graphics is properly working on your computer, we’ve included a graphics verification tool in CS51Utils. (Thanks to CIS120 at UPenn for help with the following instructions.)

Run the following shell comand.

% cs51-graphics-check

The XQuartz implementation of the X11 window system should run, and a small window with a white exclamation mark on a red background should appear, as in the figure below.



#### An OCaml graphics window

This window should appear if graphics is properly set up.

If so, OCaml graphics is properly installed and you’re all set. If not, proceed with the remaining instructions.

If the graphics window does not work on macOS

### **The OCaml REPL**[**#**](https://cs51.io/guides/setup/#the-ocaml-repl)

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

OCaml version 4.11.1

Findlib has been successfully loaded. Additional directives:

#require "package";; to load a package

#list;; to list the available packages

#camlp4o;; to load camlp4 (standard syntax)

#camlp4r;; to load camlp4 (revised syntax)

#predicates "p,q,...";; to set these predicates

Topfind.reset();; to force that packages will be reloaded

#thread;; to enable threads

#

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 or type #quit;; to call the REPL’s quit command.

#### utop [#](https://cs51.io/guides/setup/#utop)

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

% utop

It should also identify 4.11.1 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.)

Congratulations! You just ran your first OCaml code!

# Setting up git [#](https://cs51.io/guides/git/#setting-up-git)

## Signing up for GitHub [#](https://cs51.io/guides/git/#signing-up-for-github)

The git repositories we will work with will be remotely hosted on [GitHub](https://github.com/), a git service offering remote repositories hosted in the cloud. If you don’t already have a GitHub account, follow [these instructions](https://help.github.com/articles/signing-up-for-a-new-github-account/) to create one.

### **Adding an SSH key**[**#**](https://cs51.io/guides/git/#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](https://help.github.com/articles/checking-for-existing-ssh-keys/)
2. [Generating a new key (if necessary)](https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/)
3. [Telling GitHub about your key.](https://help.github.com/articles/adding-a-new-ssh-key-to-your-github-account/)

Your git should now be fully configured.

## Using git [#](https://cs51.io/guides/git/#using-git)

When you modify some code, reach a good checkpoint in your coding (for example, “fixed bug in function foo”), or finish your work, you can and should commit these changes by executing a git commit command with an appropriate message:

% git commit -am "fixed bug in function foo"

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 further changes to the file, you would have to add it again to commit the further changes.
* 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 a

% git push

This will 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. This will also set things up to submit your work to Gradescope (though it doesn’t perform the submission process itself; more on this later). Again, commit and push early and often.

If you want a succinct reference for your git usage, see Eddie Kohler’s great [guide to Git](http://cs61.seas.harvard.edu/site/ref/git/), as well as the aforementioned [video about git for CS 51](https://www.youtube.com/watch?v=YcenZfJrxvc) by Brian Yu.

## Accessing the distribution code [#](https://cs51.io/guides/git/#accessing-the-distribution-code)

We provide a link to the git repository for each problem set and lab. These links follow the patterns:

* http://url.cs51.io/ps{n}
* http://url.cs51.io/lab{n}

where {n} is the problem set or lab number.

To download all of the materials for a problem set or lab, all you need to do is follow this link to the repository, and on the right of the page above the problem set files, you should see a ‘Clone or Download’ option. Click on this option and copy the ‘https’ link to your clipboard. Open your terminal and navigate to the folder where you want your problem set to be saved. Once there, execute:

% git clone {URL from your clipboard} X

This will download the starter code for Problem Set X in a folder named “X”.

## Working with a partner [#](https://cs51.io/guides/git/#working-with-a-partner)

On partner problem sets, one member of your pair should follow the link on the problem set specification. After clicking this link there is a “Create team” option. Once you enter a name for your team and submit it, your partner can follow the link on the problem set specification and choose that team.

If you want to work solo on a partner problem set, create a team for yourself.

## Getting the source code [#](https://cs51.io/guides/git/#getting-the-source-code)

We hand out problem set distribution code using [GitHub Classroom](http://classroom.github.com/). 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.