Setting up OCaml

This file contains minimal instructions to setup a local installation of OCaml on Linux. See here for instructions on other OSs.

Installing OCaml

First, install opam, the OCaml official package manager:

```
sudo apt install opam
```

Then, you must initialize opam. This installs OCaml and creates a default switch:

```
opam init --bare -a -y
```

Here we assume you will work on the default switch. To check that a switch actually exists:

```
opam switch list
```

In case the previous command shows an empty list, you must manually create a switch:

```
opam switch create lip ocaml-base-compiler.4.14.0
```

This creates a switch for the LIP course with the given version of the OCaml compiler.

The following command updates environment variables, to make OCaml commands available on the current switch:

```
eval $(opam env)
```

Finally, install a few extra OCaml packages:

```
opam install -y dune merlin ocaml-lsp-server odoc ocamlformat utop menhir
ppx_inline_test
```

In particular, this installation includes:

- **dune**, a build system for OCaml projects, similar to make;
- **utop**, a REPL interface for OCaml;
- **Menhir**, a parser generator;
- **ppx_inline_test**, a tool for writing in-line tests in OCaml.

We will use these tools for all the projects of the LIP course.

If you plan to use the emacs editor, run:

```
opam user-setup install
```

First project

As a preliminary step, fork the repository. All the coding activities of this lab must be performed on your fork.

To check that everything is installed correctly, we set up a first project (see here for more detailed instructions).

Once you have forked the repository and cloned it, change directory to the `lip/basics` folder. Then, create a new project called `helloworld` using dune. Below, the lines starting with `>` contain the expected output of the given shell commands:

```
dune init project helloworld
> Success: initialized project component named helloworld
```

This command creates a directory `helloworld` with the following file structure:

```
helloworld/
├── dune-project
├── bin
│   └── dune
│   └── main.ml
├── lib
│   └── dune
├── test
│   ├── dune
│   └── helloworld.ml
└── helloworld.opam
```

To check that the OCaml installation was successful, try to build the project from the `helloworld` directory:

```
cd helloworld
dune build
```

If there are no errors, the output should be empty. Run the project as follows:

```
dune exec helloworld
> Hello, World!
```

We will discuss the project structure in detail in the next exercise. For the moment, note that the `_build` directory contains the output of the `dune build` command. This includes the `main.exe` executable inside the `_build/default/bin/` subdirectory.

In this very first project, all the source code is in `./bin/main.ml`. For more complex projects, we will mainly write our source code in the `lib` directory.

Installing OCaml

This guide will walk you through a minimum installation of OCaml. That includes installing a package manager and the compiler itself. We'll also install some platform tools like a build system, support for your editor, and a few other important ones.

On this page, you'll find installation instructions for Linux, macOS, Windows, and *BSD for recent OCaml versions. For Docker, Linux instructions apply, except when setting up opam.

Note: You'll be installing OCaml and its tools through a command line interface (CLI), or shell

Install opam

OCaml has an official package manager, opam, which allows users to download and install OCaml tools and libraries. Opam also makes it practical to deal with different projects which require different versions of OCaml.

Opam also installs the OCaml compiler. Alternatives exist, but opam is the best way to install OCaml. Although OCaml is available as a package in most Linux distributions, it is often outdated.

To install opam, you can use your system package manager or download the binary distribution. The details are available in these links, but for convenience, we use package distributions:

For macOS

If you're installing with Homebrew:

```
$ brew install opam
```

Or if you're using MacPorts:

```
$ port install opam
```

Note: While it's rather straightforward to install opam using macOS, it's possible you'll run into problems later with Homebrew because it has changed the way it installs. The executable files cannot be found in ARM64, the M1 processor used in newer Macs. Addressing this can be a rather complicated procedure, so we've made a short ARM64 Fix doc explaining this so as not to derail this installation guide.

For Linux

It's preferable to install opam with your system's package manager on Linux, as superuser. On the opam site, find details of all installation methods. A version of opam above 2.0 is packaged in all supported Linux distributions. If you are using an unsupported Linux distribution, please either download a precompiled binary or build opam from sources.

If you are installing in Debian or Ubuntu:

```
$ sudo apt-get install opam
```

If you are installing in Arch Linux:

```
$ sudo pacman -S opam
```

Note: The Debian package for opam, which is also used in Ubuntu, has the OCaml compiler as a recommended dependency. By default, such dependencies are installed. If you want to only install opam without OCaml, you need to run something like this:

```
sudo apt-get install --no-install-recommends opam
```

For Windows

It's easiest to install opam with WinGet:

```
PS C:\> winget install Git.Git OCaml.opam
```

Binary Distribution

If you want the latest release of opam, install it through the binary distribution. On Unix and macOS, you'll need to install the following system packages first: `gcc`, `build-essential`, `curl`, `bubblewrap`, and `unzip`. Note that they might have different names depending on your operating system or distribution. Also, note this script internally calls `sudo`.

The following command will install the latest version of opam that applies to your system:

```
$ bash -c "sh <(curl -fsSL
https://raw.githubusercontent.com/ocaml/opam/master/shell/install.sh)"
```

On Windows, the winget package is maintained by opam's developers and uses the binaries released on GitHub, however you can also install using an equivalent PowerShell script:

```
Invoke-Expression "& { $(Invoke-RestMethod
https://raw.githubusercontent.com/ocaml/opam/master/shell/install.ps1) }"
```

Advanced Windows Users: If you are familiar with Cygwin or WSL2, there are other installation methods described on the OCaml on Windows page.

Initialise opam

After you install opam, you'll need to initialise it. To do so, run the following command, as a normal user. This might take a few minutes to complete.

```
$ opam init -y
```

Note: In case you are running `opam init` inside a Docker container, you will need to disable sandboxing, which is done by running `opam init --disable-sandboxing -y`. This is necessary, unless you run a privileged Docker container.

Make sure you follow the instructions provided at the end of the output of `opam init` to complete the initialisation. Typically, this is:

```
$ eval $(opam env)
```

on Unix, and from the Windows Command Prompt:

```
for /f \"tokens=*\" %i in ('opam env') do @%i
```

or from PowerShell:

```
(& opam env) -split '\r?\n' | ForEach-Object { Invoke-Expression $_ }
```

Opam initialisation may take several minutes. While waiting for its installation and configuration to complete, start reading A Tour of OCaml.

Note: opam can manage something called *switches*. This is key when switching between several OCaml projects. However, in this “getting started” series of tutorials, switches are not needed. If interested, you can read an introduction to opam switches here.

Any problems installing? Be sure to read the latest release notes. You can file an issue at https://github.com/ocaml/opam/issues or https://github.com/ocaml-windows/papercuts/issues.

Install Platform Tools

Now that we've successfully installed the OCaml compiler and the opam package manager, let's install some of the OCaml Platform tools, which you'll need to get the full developer experience in OCaml:

- UTop, a modern interactive toplevel (REPL: Read-Eval-Print Loop)
- Dune, a fast and full-featured build system
- `ocaml-lsp-server` implements the Language Server Protocol to enable editor support for OCaml, e.g., in VS Code, Vim, or Emacs.
- `odoc` to generate documentation from OCaml code
- OCamlFormat to automatically format OCaml code

All these tools can be installed using a single command:

```
$ opam install ocaml-lsp-server odoc ocamlformat utop
```

You're now all set and ready to start hacking.

Check Installation

To check that everything is working properly, you can start the UTop toplevel:

```
$ utop
─────────────┬──────────────────────────────────────────────────────────────┬───────────
─
             │ Welcome to utop version 2.13.1 (using OCaml version 5.1.0)! │
             └──────────────────────────────────────────────────────────────┘
```

```
Type #utop_help for help about using utop.

─( 00:00:00 )─< command 0 >──────────────────────────────────────{ counter: 0 }─
utop #
```

You're now in an OCaml toplevel, and you can start typing OCaml expressions. For instance, try typing `21 * 2;;` at the `#` prompt, then hit `Enter`. You'll see the following:

```
# 21 * 2;;
- : int = 42
```

Congratulations! You've installed OCaml! 🎉

Join the Community

Make sure you join the OCaml community. You'll find many community members on Discuss or Discord. These are great places to ask for help if you have any issues.

Git tutorial

The purpose of this tutorial is to help you set up a minimal working environment to create and push your solutions to the exercises. It also aims to introduce you to some useful tools and resources that offer a smoother and integrated lab workflow.

In the following we assume that:

- You are reading this guide from your own fork of the ocaml-challenge. If you don't know how to fork a repository, see the useful GitHub documentation;
- You are working on a Linux machine. If you are a Windows user, the most straightforward way to run a Linux environment on your Windows system is the Windows Subsystem for Linux, which can be easily installed from the Microsoft Store;
- You have the OCaml compiler installed: check out the installation instructions for Linux.

1. Install `git`

Your Linux distribution most likely comes with `git` preinstalled. You can check this by running:

```
git --version
```

If the previous command returns an error, install `git`:

```
sudo apt update
sudo apt install git
```

2. Link `git` to your GitHub account

To manage your online repositories from the command line you need the GitHub CLI. Follow the installation instructions that suit your Linux distribution.

Once you have `gh` installed, authenticate by running:

```
gh auth login
```

and follow the on-screen procedure carefully.

Now your `git` installation is linked with your GitHub account, however `git` still doesn't know who you are. For this, run the following commands with username and email of your GitHub account as arguments.

```
git config --global user.name <your_username>
git config --global user.email <your@email.com>
```

3. Clone your fork

Cloning downloads a local copy of your fork of the repository on your disk. This is where you edit the code of the exercises using your favorite code editor (we recommend Visual Studio Code together with the OCaml Platform extension).

In a directory of your choice, run the following command with your actual username (and your fork's name in case you named it something other than `ocaml-challenge`) in the URL argument:

```
git clone https://github.com/your_username/ocaml-challenge
```

4. Commit a solution

When you're ready to upload a solution of an exercise to your fork, first run:

```
git commit
```

to record the changes you made to a local commit, then run:

```
git push
```

to transmit the new commit to your remote (i.e. online) fork.

5. Synchronize your fork with `informatica-unica/ocaml-challenge`

To synchronize your fork on your browser, look for the "Sync fork" button in the GitHub page of your fork's repository. Note that syncing on the browser does not affect your local copy of the fork that you cloned earlier.

To synchronize your local copy of the fork with the most recent version of the lab repository, run:

```
git pull
```

This might not work if you have some pending changes not yet committed to your working tree. In this case you can temporarily store away the modified files with:

```
git stash
```

and restore them later on top of the newer commits using:

```
git stash apply
```

Tip: you can always append the `--help` option to any of the above git commands to fully explore their functionality. Also refer to the Git Cheat Sheet for more important commands.

Syncing a fork

Sync a fork of a repository to keep it up-to-date with the upstream repository.

Who can use this feature?

People with write access for a forked repository can sync the fork to the upstream repository.

- Mac
- Windows
- Linux

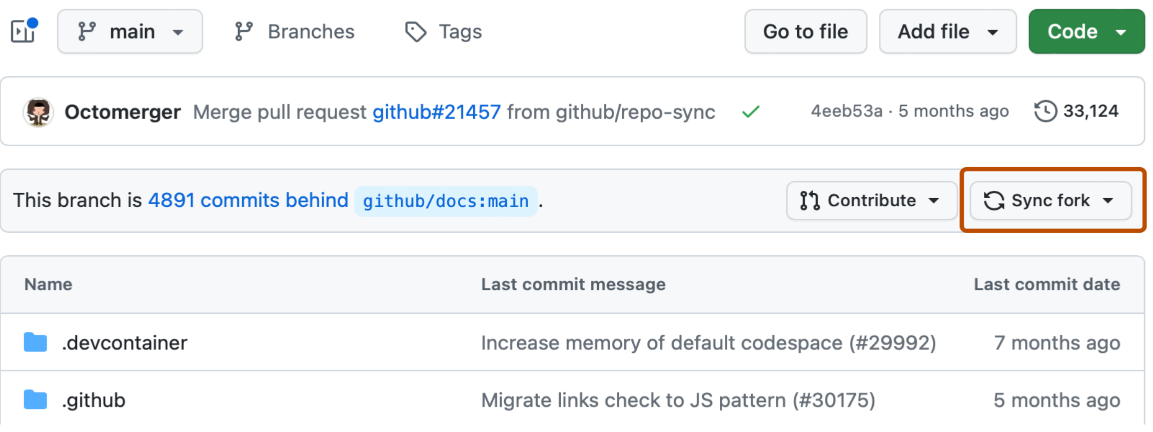
In this article

- Syncing a fork branch from the web UI
- Syncing a fork branch with the GitHub CLI
- Syncing a fork branch from the command line

Syncing a fork branch from the web UI

1. On GitHub, navigate to the main page of the forked repository that you want to sync with the upstream repository.

2. Above the list of files, select the **Sync fork** dropdown menu.



main Branches Tags Go to file Add file Code

Octomerger Merge pull request github#21457 from github/repo-sync 4eeb53a · 5 months ago 33,124

This branch is 4891 commits behind `github/docs:main`.

Contribute Sync fork

Name	Last commit message	Last commit date
.devcontainer	Increase memory of default codespace (#29992)	7 months ago
.github	Migrate links check to JS pattern (#30175)	5 months ago

3. Review the details about the commits from the upstream repository, then click **Update branch**.

If the changes from the upstream repository cause conflicts, GitHub will prompt you to create a pull request to resolve the conflicts.

Syncing a fork branch with the GitHub CLI

GitHub CLI is an open source tool for using GitHub from your computer's command line. When you're working from the command line, you can use the GitHub CLI to save time and avoid switching context. To learn more about GitHub CLI, see "About GitHub CLI."

To update the remote fork from its parent, use the `gh repo sync -b BRANCH-NAME` subcommand and supply your fork and branch name as arguments.

```
gh repo sync owner/cli-fork -b BRANCH-NAME
```

If the changes from the upstream repository cause conflict then the GitHub CLI can't sync. You can set the `--force` flag to overwrite the destination branch.

Syncing a fork branch from the command line

Before you can sync your fork with an upstream repository, you must configure a remote that points to the upstream repository in Git. For more information, see "Configuring a remote repository for a fork."

1. Open Git Bash.

2. Change the current working directory to your local project.

3. Fetch the branches and their respective commits from the upstream repository. Commits to `BRANCH-NAME` will be stored in the local branch `upstream/BRANCH-NAME`.

   ```
   $ git fetch upstream

   > remote: Counting objects: 75, done.

   > remote: Compressing objects: 100% (53/53), done.

   > remote: Total 62 (delta 27), reused 44 (delta 9)

   > Unpacking objects: 100% (62/62), done.
   ```

```
> From https://github.com/ORIGINAL-OWNER/ORIGINAL-REPOSITORY

>  * [new branch]      main     -> upstream/main
```

4. Check out your fork's local default branch - in this case, we use `main`.

```
$ git checkout main

> Switched to branch 'main'
```

5. Merge the changes from the upstream default branch - in this case, `upstream/main` - into your local default branch. This brings your fork's default branch into sync with the upstream repository, without losing your local changes.

```
$ git merge upstream/main

> Updating a422352..5fdff0f

> Fast-forward

>  README                     |    9 -------

>  README.md                  |    7 ++++++

>  2 files changed, 7 insertions(+), 9 deletions(-)

>  delete mode 100644 README

>  create mode 100644 README.md
```

If your local branch didn't have any unique commits, Git will perform a fast-forward. For more information, see Basic Branching and Merging in the Git documentation.

```
$ git merge upstream/main

> Updating 34e91da..16c56ad
> Fast-forward
>  README.md                 |    5 +++--
>  1 file changed, 3 insertions(+), 2 deletions(-)
```

If your local branch had unique commits, you may need to resolve conflicts. For more information, see "Addressing merge conflicts."

Tip: Syncing your fork only updates your local copy of the repository. To update your fork on GitHub.com, you must push your changes.

Working in utop

`utop` is the enhanced REPL (Read-Eval-Print Loop) for OCaml. It is similar to Python's `ipython` or Haskell's `ghci`.

It has many features for testing and debugging your OCaml code efficiently. In this quick tutorial we share some tips on how to get the most out of it.

Getting utop

utop is bundled as an OCaml package. Therefore, we recommend installing it using OCaml's package manager `opam` rather than other available means (e.g. your distro's package manager).

Once you have `opam` installed, utop can be pulled easily:

```
opam install utop
```

Starting utop

Run utop from your terminal by just typing its name:

```
utop
```

If you have some OCaml code in a file called `foo.ml`, you can supply it via the `-init` option to utop.

```
utop -init foo.ml
```

This will compile the source code and import its definitions in the first context.

If you're working inside a dune project, running `dune utop` has the advantage of building the project beforehand so that its definition can be imported in utop.

Tip

Syntax highlighting in utop isn't enabled by default. To enable colors, create a file called `.utoprc` in your home directory and paste the following settings in it - or make up your own colors:

```
profile:                    dark
identifier.foreground:      none
comment.foreground:         x-chocolate1
doc.foreground:             x-light-salmon
constant.foreground:        x-aquamarine
keyword.foreground:         x-cyan1
symbol.foreground:          x-cyan1
string.foreground:          x-light-salmon
char.foreground:            x-light-salmon
quotation.foreground:       x-purple
error.foreground:           x-red
parenthesis.background:     blue
```

Entering expressions

utop uses two semicolons `;;` to mark the end of an expression that's ready for evaluation. Pressing `Enter` after `;;` will submit the expression to the interpreter.

Until you add `;;` to an expression, pressing `Enter` will simply start a new line, and your expression will span multiple lines.

The completion bar

utop also features a completion bar at the bottom of the UI that suggests defined names as you type. You use `Alt+Left arrow` and `Alt+Right arrow` to move the focus to the desired suggestion and `Alt+Down arrow` to enter it in the prompt line.

Tip

One of the best ways to get to know the language and discover the standard library (`List`, `String`, `Set`...) is to experiment with the suggested names and read their types.

```
#show List.fold_left;;
> val fold_left : ('acc -> 'a -> 'acc) -> 'acc -> 'a list -> 'acc
```

Moving around

You can move the cursor, delete, copy and paste text in the prompt while editing an expression just like in any other REPL. Some less obvious inputs:

- The `Up arrow` and `Down arrow` keys let you browse through the history of the expressions you typed in during the current session and in past sessions.
- While editing a multiline expression, use `Ctrl+P` and `Ctrl+N` to the cursor move up and down a line, respectively.
- Like in bash, `Ctrl+A` and `Ctrl+E` jump to the start and the end of the line, and `Ctrl+U` clears any character before the cursor.
- To exit utop safely, press `Ctrl+D` **with an empty prompt line**.

Tracing recursion

Maybe the most useful feature of utop is the ability to *trace* recursive computations. The `#trace` command lets you inspect the step-by-step behavior of a recursive function call.

```
let rec fact n = if n <= 0 then 1 else n * fact (n-1);;
> val fact : int -> int = <fun>

#trace fact;;
> fact is now traced.
```

Now, every time `fact` is called on an `int`, utop prints every recursive subcalls (`<--`) and its partial result (`-->`). An asterisk `*` indicates that the function has been fed all its arguments and enters its body.

```
fact 3;;
```

```
> fact <-- 3
> fact <-- 2
> fact <-- 1
> fact <-- 0
> fact --> 1
> fact --> 1
> fact --> 2
> fact --> 6
> - : int = 6
```

Tracing polymorphic functions

There's a subtlety arising from tracing polymorphic functions: `#trace` won't refine the types of polymorphic parameters at runtime.

```
let rec map f l = match l with [] -> [] | x :: l' -> f x :: map f l';;
> val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

#trace map;;
> map is now traced.
```

The trace of `map succ [1;2;3]` will show a bunch of `<poly>`. Quite annoying.

```
utop # map succ [1;2;3];;
map <-- <fun>
map --> <fun>
map* <-- [<poly>; <poly>; <poly>]
map <-- <fun>
map --> <fun>
map* <-- [<poly>; <poly>]
map <-- <fun>
map --> <fun>
map* <-- [<poly>]
map <-- <fun>
map --> <fun>
map* <-- []
map* --> []
map* --> [<poly>]
map* --> [<poly>; <poly>]
map* --> [<poly>; <poly>; <poly>]
```

One way to work around this is redefining the function with type annotations for the arguments you want to inspect.

```
let rec map (f : int->int) l = match l with [] -> [] | x :: l' -> f x :: map f
l';;
> val map : (int -> int) -> int list -> int list = <fun>

#trace map;;
> map is now traced.
```

Now we can make more sense of the trace output:

```
map succ [1;2;3];;
map <-- <fun>
map --> <fun>
map* <-- [1; 2; 3]
map <-- <fun>
map --> <fun>
map* <-- [2; 3]
map <-- <fun>
map --> <fun>
map* <-- [3]
map <-- <fun>
map --> <fun>
map* <-- []
map* --> []
map* --> [4]
map* --> [3; 4]
map* --> [2; 3; 4]
- : int list = [2; 3; 4]
```

Saving your session

If you wish to save your utop session to the file "foo.ml":

```
#utop_stash "foo.ml"
```

This writes correct OCaml code for the expressions that were successfully evaluated, with messages and errors of the toplevel wrapped in comments.

Loading in existing code

To load in the contents of the OCaml `foo.ml` without leaving utop:

```
#use "foo.ml"
```

This will compile the file and add its definitions to the context.

Using an .ocamlinit

There are some routine expressions that you wish would persist between utop sessions. Typing these expressions every time you quit and reenter utop can be quite tedious.

Fortunately, you can store this initialization code in a file called `.ocamlinit`. If utop is started in the same directory of this file, it will compile it and initialize itself with its contents.