CS442
Module 1: Languages: OCaml

University of Waterloo

Winter 2023

1 OCaml in CS442

OCaml is available on `linux.student.cs.uwaterloo.ca` as `ocaml`. In this course, you may use the OCaml standard library, and the `Base`, `Core_Kernel` and `Core` libraries if you would like, but you may not use any others unless you are specifically instructed to do so in an assignment.

OCaml has many useful and fast references online. For the purposes of this course, you should read the Guided Tour (https://dev.realworldocaml.org/guided-tour.html) from Real World OCaml (https://dev.realworldocaml.org/). The rest of the book is also a useful reference, of course, but you're recommended to use it as a reference rather than to read it through, simply because you shouldn't need it.

Real World OCaml recommends using the library `Base`, and in this course, you are recommended, but not required, to do so. You can set up OCaml with `Base` on your own system following the instructions in its installation chapter, or on `linux.student.cs.uwaterloo.ca` by doing the following:

1. Set up `opam` with `opam init` and follow its directions (the defaults should work)
2. Include `opam` in your environment with `eval $(opam env)`
3. Install `Base` and related libraries and tools with `opam install core base utop`
4. Add the following to a file `.ocamlinit` in your home directory, which may be a new file:

```
#use "topfind";;
#thread;;
#require "base";;
#require "core";;
```

Unfortunately, `Base` is a *replacement for the entire standard library of OCaml*, which means that its existence fractures OCaml into two largely incompatible languages: the one using the original standard library, and the one using `Base`. Add onto that that `Base` is an unsearchably vague term, and you have a recipe for confusing documentation. If you use `Base`, you're recommended to search for "Jane Street Base", as Jane Street are the authors of `Base`. These two languages are really more alike than unlike, so you shouldn't have difficulty adapting, but will have to be careful about finding documentation.

2 Restrictions

In this course, *you may not define classes in OCaml*. You may use classes provided by the standard library or allowed libraries, but may not define your own. OCaml is an object-oriented dialect of Caml, which is in turn a dialect of ML, but it is for those ML roots that OCaml was chosen, not for its object orientation. OCaml was chosen over languages without such additions, such as Standard ML, simply because it is more well maintained. Note that records are not classes, and are perfectly fine to use. Generally, object orientation in OCaml is considered

of narrow use, or even an outright mistake, so this restriction shouldn't conflict with anything you find in OCaml documentation.

> **Aside:** Many object-oriented languages don't distinguish records (simple data containers) from classes (object-oriented types) because in these languages, a class is strictly more powerful than a record type. For instance, in C++, although C's record syntax, `struct`, is still supported, `struct`s are just `class`es that are `public` by default. However, the concepts evolved quite separately, and indeed, "purely" object-oriented languages such as Smalltalk don't even have record types!

Real World OCaml recommends the use of the Dune build system, and you're recommended to use it for your own convenience and testing, but we will not be using Dune to build your code. This is because we will be building your code against our own test suites, and we do not wish to require you to learn how to use Dune to build libraries. As a consequence, you *must* name your files as we specify, and *must* name your functions as we specify. You may of course have additional functions, variables, etc, beyond what we demand, as helpers, but you *may not* have additional files for your solution to any assignment question, since we won't be using Dune, so wouldn't know what additional files to compile. You can and should have additional files for testing, but they cannot be a *requirement* of your code; i.e., they cannot be part of your actual code's functionality, only its testing.

3 Testing

To test in the simplest way, just build tests into your normal `.ml` file. However, make sure you remove them or comment them out before submitting!

For better testing, you should write a separate module, i.e., a separate file. It's quite easy to use code from another module. For instance, if you are tasked with writing `a1q1.ml`, and a function named `pushNum`, then if you create a separate file named, e.g., `testa1q1.ml`, you can call `a1q1.ml`'s `pushNum` function as `A1q1.pushNum`. Just make sure you build your test code together with `a1q1.ml`. This is how our own tests will be built, as normal OCaml; you will not normally be expected to write a parser in this course.

Configuring Dune to build a plethora of different tests all against the same main code can be a bit complicated. If you'd rather use `make`, or just compile manually, here is the incantation you need for the above example:

```
ocamlfind ocamlc -package core -linkpkg -thread -o testa1q1 a1q1.ml testa1q1.ml
```

You can generalize this to anything else simply by replacing the exact `.ml` files. This produces an executable named `testa1q1`.

In the above example we use the `Core` package, and for most anything in this course, that should be sufficient, as it links in `Base`, `Stdio`, and most other standard libraries. If you want to be more particular, you can use a comma-separated list of packages:

```
ocamlfind ocamlc -package base,stdio -linkpkg -o testa1q1 a1q1.ml testa1q1.ml
```

Note that in the first example we used `-thread` because `Core` depends on it, but neither `Base` nor `Stdio` do, so it's fine to leave it out in this case. `ocamlc` will warn you if you needed `-thread` but excluded it, and it's always harmless to include it. We won't look at *using* threads in OCaml until the end of the course.

OCaml has C- or C++-like compilation and linking, which is why you need to be specific both about what you're *compiling* against (with `-package`) and what you're *linking* against (with `-linkpkg`).