OCaml Programming: Correct + Efficient + Beautiful

Michael R. Clarkson et al.

Jun 15, 2021

CONTENTS

Fall 2021 Edition

A textbook on functional programming and data structures in OCaml, with an emphasis on semantics and software engineering.

Based on courses taught by Michael R. Clarkson, Robert L. Constable, Nate Foster, Michael D. George, Dan Grossman, Daniel P. Huttenlocher, Dexter Kozen, Greg Morrisett, Andrew C. Myers, Radu Rugina, and Ramin Zabih.

This work is based on over 20 years worth of course notes and intellectual contributions by the authors named above; teasing out who contributed what is, by now, not an easy task. The primary compiler and author of this work in its form as a unified textbook is Michael R. Clarkson.

For the most recent version of this work, see the most recent CS 3110 course website.

Solutions to the exercises at the end of each chapter are available. Cornell students will have access to them as part of the course. Instructors at other institutions are welcome to contact Michael Clarkson for access.

CHAPTER

ONE

BETTER PROGRAMMING THROUGH OCAML

Do you already know how to program in a mainstream language like Python or Java? Good. This book is for you. It's time to learn how to program better. It's time to learn a functional language, OCaml.

Note: The HTML version of this textbook has about 200 videos embedded in it. The first one is below. The videos usually provide an introduction to material, upon which the textbook then expands.

These videos were produced during pandemic when the Cornell course that uses this textbook had to be asynchronous. The student response to them was overwhelmingly positive, so they are now being made public as part of the textbook. But just so you know, they were not produced by a professional A/V team—just a guy in his basement who was learning as he went.

The videos mostly use the versions of OCaml and its ecosystem that were current in Fall 2020. Current versions you are using are likely to look different from the videos, but don't be alarmed: the underlying ideas are the same. The most visible difference is likely to be the VS Code plugin for OCaml. In Fall 2020 the badly-aging "OCaml and Reason IDE" plugin was still being used. It has since been superceded by the "OCaml Platform" plugin.

The order that the textbook covers topics sometimes differs from the order that the videos cover the topics, simply because the videos originate from lectures. The videos are placed in the textbook nearest to the topic they cover, but that does mean sometimes the videos are not in chronological order.

Functional programming provides a different perspective on programming than what you have experienced so far. Adapating to that perspective requires letting go of old ideas: assignment statements, loops, classes and objects, among others. That won't be easy.

> Nan-in, a Japanese master during the Meiji era (1868-1912), received a university professor who came to inquire about Zen. Nan-in served tea. He poured his visitor's cup full, and then kept on pouring. The professor watched the overflow until he no longer could restrain himself. "It is overfull. No more will go in!" "Like this cup," Nan-in said, "you are full of your own opinions and speculations. How can I show you Zen unless you first empty your cup?"

I believe that learning OCaml will make you a better programmer. Here's why:

- You will experience the freedom of *immutability*, in which the values of so-called "variables" cannot change. Goodbye, debugging.
- You will improve at *abstraction*, which is the practice of avoiding repetition by factoring out commonality. Goodbye, bloated code.
- You will be exposed to a *type system* that you will at first hate because it rejects programs you think are correct. But you will come to love it, because you will humbly realize it was right and your programs were wrong. Goodbye, failing tests.
- You will be exposed to some of the *theory and implementation of programming languages*, helping you to understand the foundations of what you are saying to the computer when you write code. Goodbye, mysterious and magic

> incantations.

All of those ideas can be learned in other contexts and languages. But OCaml provides an incredible opportunity to bundle them all together. **OCaml will change the way you think about programming.**

> "A language that doesn't affect the way you think about programming is not worth knowing."
>
> ---Alan J. Perlis (1922-1990), first recipient of the Turing Award

Moreover, OCaml is beautiful. OCaml is elegant, simple, and graceful. Aesthetics do matter. Code isn't written just to be executed by machines. It's also written to communicate to humans. Elegant code is easier to read and maintain. It isn't necessarily easier to write.

The OCaml code you write can be stylish and tasteful. At first, this might not be apparent. You are learning a new language after all—you wouldn't expect to appreciate Sanskrit poetry on day 1 of Introductory Sanskrit. In fact, you'll likely feel frustrated for awhile as you struggle to express yourself in a new language. So give it some time. After you've mastered OCaml, you might be surprised at how ugly those other languages you already know end up feeling when you return to them.

1.1 The Past of OCaml

Genealogically, OCaml comes from the line of programming languages whose grandfather is Lisp and includes other modern languages such as Clojure, F#, Haskell, and Racket.

OCaml originates from work done by Robin Milner and others at the Edinburgh Laboratory for Computer Science in Scotland. They were working on theorem provers in the late 1970s and early 1980s. Traditionally, theorem provers were implemented in languages such as Lisp. Milner kept running into the problem that the theorem provers would sometimes put incorrect "proofs" (i.e., non-proofs) together and claim that they were valid. So he tried to develop a language that only allowed you to construct valid proofs. ML, which stands for "Meta Language", was the result of that work. The type system of ML was carefully constructed so that you could only construct valid proofs in the language. A theorem prover was then written as a program that constructed a proof. Eventually, this "Classic ML" evolved into a full-fledged programming language.

In the early '80s, there was a schism in the ML community with the French on one side and the British and US on another. The French went on to develop CAML and later Objective CAML (OCaml) while the Brits and Americans developed Standard ML. The two dialects are quite similar. Microsoft introduced its own variant of OCaml called F# in 2005.

Milner received the Turing Award in 1991 in large part for his work on ML. The ACM website for his award includes this praise:

> ML was way ahead of its time. It is built on clean and well-articulated mathematical ideas, teased apart so that they can be studied independently and relatively easily remixed and reused. ML has influenced many practical languages, including Java, Scala, and Microsoft's F#. Indeed, no serious language designer should ignore this example of good design.

1.2 The Present of OCaml

OCaml is a functional programming language. The key linguistic abstraction of functional languages is the mathematical function. A function maps an input to an output; for the same input, it always produces the same output. That is, mathematical functions are *stateless*: they do not maintain any extra information or *state* that persists between usages of the function. Functions are *first-class*: you can use them as input to other functions, and produce functions as output. Expressing everything in terms of functions enables a uniform and simple programming model that is easier to reason about than the procedures and methods found in other families of languages.

Imperative programming languages such as C and Java involve *mutable* state that changes throughout execution. *Commands* specify how to compute by destructively changing that state. Procedures (or methods) can have *side effects* that update state in addition to producing a return value.

The **fantasy of mutability** is that it's easy to reason about: the machine does this, then this, etc.

The **reality of mutability** is that whereas machines are good at complicated manipulation of state, humans are not good at understanding it. The essence of why that's true is that mutability breaks *referential transparency*: the ability to replace an expression with its value without affecting the result of a computation. In math, if $f(x) = y$, then you can substitute y anywhere you see $f(x)$. In imperative languages, you cannot: f might have side effects, so computing $f(x)$ at time t might result in a different value than at time t'.

It's tempting to believe that there's a single state that the machine manipulates, and that the machine does one thing at a time. Computer systems go to great lengths in attempting to provide that illusion. But it's just that: an illusion. In reality, there are many states, spread across threads, cores, processors, and networked computers. And the machine does many things concurrently. Mutability makes reasoning about distributed state and concurrent execution immensely difficult.

Immutability, however, frees the programmer from these concerns. It provides powerful ways to build correct and concurrent programs. OCaml is primarily an immutable language, like most functional languages. It does support imperative programming with mutable state, but we won't use those features until many chapters into the book—in part because we simply won't need them, and in part to get you to quit "cold turkey" from a dependence you might not have known that you had. This freedom from mutability is one of the biggest changes in perspective that OCaml can give you.

1.2.1 The Features of OCaml

OCaml is a *statically-typed* and *type-safe* programming language. A statically-typed language detects type errors at compile time, so that programs with type errors cannot be executed. A type-safe language limits which kinds of operations can be performed on which kinds of data. In practice, this prevents a lot of silly errors (e.g., treating an integer as a function) and also prevents a lot of security problems: over half of the reported break-ins at the Computer Emergency Response Team (CERT, a US government agency tasked with cybersecurity) were due to buffer overflows, something that's impossible in a type-safe language.

Some functional languages, like Python and Racket, are type-safe but *dynamically typed*. That is, type errors are caught only at run time. Other languages, like C and C++, are statically typed but not type safe. There's no guarantee that a type error won't occur an run time. And still other languages, like Java, use a combination of static and dynamic typing to achieve type safety.

OCaml supports a number of advanced features, some of which you will have encountered before, and some of which are likely to be new:

- **Algebraic datatypes:** You can build sophisticated data structures in OCaml easily, without fussing with pointers and memory management. *Pattern matching*—a feature we'll soon learn about that enables examining the shape of a data structure—makes them even more convenient.
- **Type inference:** You do not have to write type information down everywhere. The compiler automatically figures out most types. This can make the code easier to read and maintain.
- **Parametric polymorphism:** Functions and data structures can be parameterized over types. This is crucial for being able to re-use code.
- **Garbage collection:** Automatic memory management relieves you from the burden of memory allocation and deallocation, a common source of bugs in languages such as C.
- **Modules:** OCaml makes it easy to structure large systems through the use of modules. Modules are used to encapsulate implementations behind interfaces. OCaml goes well beyond the functionality of most languages with modules by providing functions (called *functors*) that manipulate modules.

1.2.2 OCaml in Industry

OCaml and other functional languages are nowhere near as popular as Python, C, or Java. OCaml's real strength lies in language manipulation (i.e., compilers, analyzers, verifiers, provers, etc.). This is not surprising, because OCaml evolved from the domain of theorem proving.

That's not to say that functional languages aren't used in industry. There are many industry projects using OCaml and Haskell, among other languages. Yaron Minsky (Cornell PhD '02) even wrote a paper about using OCaml in the financial industry. It explains how the features of OCaml make it a good choice for quickly building complex software that works.

1.3 Look to Your Future

General-purpose languages come and go. In your life you'll likely learn a handful. Today, it's Python and Java. Yesterday, it was Pascal and Cobol. Before that, it was Fortran and Lisp. Who knows what it will be tomorrow? In this fast-changing field you need to be able to rapidly adapt. A good programmer has to learn the principles behind programming that transcend the specifics of any specific language. There's no better way to get at these principles than to approach programming from a functional perspective. Learning a new language from scratch affords the opportunity to reflect along the way about the difference between *programming* and programming *in a language.*

If after OCaml you want to learn more about functional programming, you'll be well prepared. OCaml does a great job of clarifying and simplifying the essence of functional programming in a way that other languages that blend functional and imperative programming (like Scala) or take functional programming to the extreme (like Haskell) do not.

And even if you never code in OCaml again after learning it, you'll still be better prepared for the future. Advanced features of functional languages have a surprising tendency to predict new features of more mainstream languages. Java brought garbage collection into the mainstream in 1995; Lisp had it in 1958. Java didn't have generics until version 5 in 2004; the ML family had it in 1990. First-class functions and type inference have been incorporated into mainstream languages like Java, C#, and C++ over the last 10 years, long after functional languages introduced them.

News Flash!

Python just announced plans to support pattern matching in February 2021.

1.4 A Brief History of CS 3110

This book is the primary textbook for CS 3110 at Cornell University. The course has existed for over two decades and has always taught functional programming, but it has not always used OCaml.

Once upon a time, there was a course at MIT known as 6.001 *Structure and Interpretation of Computer Programs* (SICP). It had a textbook by the same name, and it used Scheme, a functional programming language. Tim Teitelbaum taught a version of the course at Cornell in Fall 1988, following the book rather closely and using Scheme.

CS 212. Dan Huttenlocher had been a TA for 6.001 at MIT; he later became faculty at Cornell. In Fall 1989, he inaugurated CS 212 Modes of Algorithm Expression. Basing the course on SICP, he infused a more rigorous approach to the material. Huttenlocher continued to develop CS 212 through the mid 1990s, using various homegrown dialects of Scheme.

Other faculty began teaching the course regularly. Ramin Zabih had taken 6.001 as a first-year student at MIT. In Spring 1994, having become faculty at Cornell, he taught CS 212. Dexter Kozen (Cornell PhD 1977) first taught the course in Spring 1996. The earliest surviving online record of the course seems to be Spring 1998, which was taught by Greg Morrisett in Dylan; the name of the course had become Structure and Interpretation of Computer Programs.

By Fall 1999, CS 212 had its own lecture notes. As CS 3110 still does, that instance of CS 212 covered functional programming, the substitution and environment models, some data structures and algorithms, and programming language implementation.

CS 312. At that time, the CS curriculum had two introductory programming courses, CS 211 Computers and Programming, and CS 212. Students took one or the other, similar to how students today take either CS 2110 or CS 2112. Then they took CS 410 Data Structures. The earliest surviving online record of CS 410 seems to be from Spring 1998. It covered many data structures and algorithms not covered by CS 212, including balanced trees and graphs, and it used Java as the programming language.

Depending on which course they took, CS 211 or 212, students were entering upper-level courses with different skill sets. After extensive discussions, the faculty chose to make CS 211 required, to rename CS 212 into CS 312 Data Structures and Functional Programming, and to make CS 211 a prerequisite for CS 312. At the same time, CS 410 was eliminated from the curriculum and its contents parceled out to CS 312 and CS 482 Introduction to Analysis of Algorithms. Dexter Kozen taught the final offering of CS 410 in Fall 1999.

Greg Morrisett inaugurated the new CS 312 in Spring 2001. He switched from Scheme to Standard ML. Kozen first taught it in Fall 2001, and Andrew Myers in Fall 2002. Myers began to incorporate material on modular programming from another MIT textbook, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design* by Barbara Liskov and John Guttag. Huttenlocher first taught the course in Spring 2006.

CS 3110. In Fall 2008 two big changes came: the language switched to OCaml, and the university switched to four-digit course numbers. CS 312 became CS 3110. Myers, Huttenlocher, Kozen, and Zabih first taught the revised course in Fall 2008, Spring 2009, Fall 2009, and Fall 2010, respectively. Nate Foster first taught the course in Spring 2012; and Bob Constable and Michael George co-taught for the first time in Fall 2013.

Michael Clarkson (Cornell PhD 2010) first taught the course in Fall 2014, after having first TA'd the course as a PhD student back in Spring 2008. He began to revise the presentation of the OCaml programming material to incorporate ideas by Dan Grossman (Cornell PhD 2003) about a principled approach to learning a programming language by decomposing it into syntax, dynamic, and static semantics. Grossman uses that approach in CSE 341 Programming Languages at the University of Washington and in his popular Programming Languages MOOC.

In Fall 2018 the compilation of this textbook began. It synthesizes the work of over two decades of functional programming instruction at Cornell. In the words of the Cornell Evening Song,

> 'Tis an echo from the walls Of our own, our fair Cornell.

1.5 Summary

This book is about becoming a better programmer. Studying functional programming will help with that. The biggest obstacle in our way is the frustration of speaking a new language, particularly letting go of mutable state. But the benefits will be great: a discovery that programming transcends programming in any particular language or family of languages, an exposure to advanced language features, and an appreciation of beauty.

1.5.1 Terms and concepts

- dynamic typing
- first-class functions
- functional programming languages
- immutability
- Lisp
- ML

- OCaml
- referential transparency
- side effects
- state
- static typing
- type safety

1.5.2 Further reading

- Introduction to Objective Caml, chapters 1 and 2, a freely available textbook that is recommended for this course
- OCaml from the Very Beginning, chapter 1, a relatively inexpensive PDF textbook that is very gentle and recommended for this course
- A guided tour [of OCaml]: chapter 1 of *Real World OCaml*, a book written by some Cornellians that some students might enjoy reading
- The history of Standard ML: though it focuses on the SML variant of the ML language, it's relevant to OCaml
- The value of values: a lecture by the designer of Clojure (a modern dialect of Lisp) on how the time of imperative programming has passed
- The perils of JavaSchools: an essay by the CEO of Stack Overflow on why (my words here) CS 2110 is not enough, and why you need both CS 3110 and CS 3410.
- Teach yourself programming in 10 years: an essay by a Director of Research at Google that puts the time required to become an educated programmer into perspective

1.6 Exercises

Future chapters of this textbook contain exercises as the final section. The exercises are annotated with a difficulty rating:

- One star [★]: easy exercises that should take only a minute or two.
- Two stars [★★]: straightforward exercises that should take a few minutes.
- Three stars [★★★]: exercises that might require anywhere from five to twenty minutes or so.
- Four [★★★★] or more stars: challenging or time-consuming exercises provided for students who want to dig deeper into the material.

It's possible we've misjudged the difficulty of a problem from time to time. Let us know if you think an annotation is off.

Please do not post your solutions to the exercises anywhere, especially not in public repositories where they could be found by search engines. A repository of solutions is available to current students in the course. Instructions for how to access it will be provided elsewhere. Instructors from other universities may also request access.

CHAPTER

TWO

THE BASICS OF OCAML

This chapter will cover some of the basic features of OCaml. But before we dive in to learning OCaml, let's first talk about a bigger idea: learning languages in general.

One of the secondary goals of this course is not just for you to learn a new programming language, but to improve your skills at learning *how to learn* new languages.

There are five essential components to learning a language: syntax, semantics, idioms, libraries, and tools.

Syntax. By *syntax*, we mean the rules that define what constitutes a textually well-formed program in the language, including the keywords, restrictions on whitespace and formatting, punctuation, operators, etc. One of the more annoying aspects of learning a new language can be that the syntax feels odd compared to languages you already know. But the more languages you learn, the more you'll become used to accepting the syntax of the language for what it is, rather than wishing it were different. (If you want to see some languages with really unusual syntax, take a look at APL, which needs its own extended keyboard, and Whitespace, in which programs consist entirely of spaces, tabs, and newlines.) You need to understand syntax just to be able to speak to the computer at all.

Semantics. By *semantics*, we mean the rules that define the behavior of programs. In other words, semantics is about the meaning of a program—what computation a particular piece of syntax represents. Note that although "semantics" is plural in form, we use it as singular. That's similar to "mathematics" or "physics".

There are two pieces to semantics, the *dynamic* semantics of a language and the *static* semantics of a language. The dynamic semantics define the run-time behavior of a program as it is executed or evaluated. The static semantics define the compile-time checking that is done to ensure that a program is legal, beyond any syntactic requirements. The most important kind of static semantics is probably *type checking*: the rules that define whether a program is well typed or not. Learning the semantics of a new language is usually the real challenge, even though the syntax might be the first hurdle you have to overcome. You need to understand semantics to say what you mean to the computer, and you need to say what you mean so that your program performs the right computation.

Idioms. By *idioms*, we mean the common approaches to using language features to express computations. Given that you might express one computation in many ways inside a language, which one do you choose? Some will be more natural than others. Programmers who are fluent in the language will prefer certain modes of expression over others. We could think of this in terms of using the dominant paradigms in the language effectively, whether they are imperative, functional, object oriented, etc. You need to understand idioms to say what you mean not just to the computer, but to other programmers. When you write code idiomatically, other programmers will understand your code better.

Libraries. *Libraries* are bundles of code that have already been written for you and can make you a more productive programmer, since you won't have to write the code yourself. (It's been said that laziness is a virtue for a programmer.) Part of learning a new language is discovering what libraries are available and how to make use of them. A language usually provides a *standard library* that gives you access to a core set of functionality, much of which you would be unable to code up in the language yourself, such as file I/O.

Tools. At the very least any language implementation provides either a compiler or interpreter as a tool for interacting with the computer using the language. But there are other kinds of tools: debuggers; integrated development environments (IDE); and analysis tools for things like performance, memory usage, and correctness. Learning to use tools that are associated with a language can also make you a more productive programmer. Sometimes it's easy to confuse the tool

itself for the language; if you've only ever used Eclipse and Java together for example, it might not be apparent that Eclipse is an IDE that works with many languages, and that Java can be used without Eclipse.

When it comes to learning OCaml in this book, our focus is primarily on semantics and idioms. We'll have to learn syntax along the way, of course, but it's not the interesting part of our studies. We'll get some exposure to the OCaml standard library and a couple other libraries, notably OUnit (a unit testing framework similar to JUnit, HUnit, etc.). Besides the OCaml compiler and build system, the main tool we'll use is the toplevel, which provides the ability to interactively experiment with code.

2.1 The OCaml Toplevel

The *toplevel* is like a calculator or command-line interface to OCaml. It's similar to JShell for Java, or the interactive Python interpreter. The toplevel is handy for trying out small pieces of code without going to the trouble of launching the OCaml compiler. But don't get too reliant on it, because creating, compiling, and testing large programs will require more powerful tools. Some other languages would call the toplevel a *REPL*, which stands for read-eval-print-loop: it reads programmer input, evaluates it, prints the result, and then repeats.

In a terminal window, type `utop` to start the toplevel. Press Control-D to exit the toplevel. You can also enter `#quit;;` and press return. Note that you must type the `#` there: it is in addition to the `#` prompt you already see.

2.1.1 Types and values

You can enter expressions into the OCaml toplevel. End an expression with a double semi-colon `;;` and press the return key. OCaml will then evaluate the expression, tell you the resulting value, and the value's type. For example:

```
# 42;;
- : int = 42
```

Let's dissect that response from utop, reading right to left:

- `42` is the value.
- `int` is the type of the value.
- The value was not given a name, hence the symbol `-`.

You can bind values to names with a `let` definition, as follows:

```
# let x = 42;;
val x : int = 42
```

Again, let's dissect that response, this time reading left to right:

- A value was bound to a name, hence the `val` keyword.
- `x` is the name to which the value was bound.
- `int` is the type of the value.
- `42` is the value.

You can pronounce the entire output as "`x` has type `int` and equals `42`."

2.1.2 Functions

A function can be defined at the toplevel using syntax like this:

```
# let increment x = x+1;;
val increment : int -> int = <fun>
```

Let's dissect that response:

- `increment` is the identifier to which the value was bound.
- `int -> int` is the type of the value. This is the type of functions that take an `int` as input and produce an `int` as output. Think of the arrow `->` as a kind of visual metaphor for the transformation of one value into another value—which is what functions do.
- The value is a function, which the toplevel chooses not to print (because it has now been compiled and has a representation in memory that isn't easily amenable to pretty printing). Instead, the toplevel prints `<fun>`, which is just a placeholder.

Note: `<fun>` itself is not a value. It just indicates an unprintable function value.

You can "call" functions with syntax like this:

```
# increment 0;;
- : int = 1
# increment(21);;
- : int = 22
# increment (increment 5);;
- : int = 7
```

But in OCaml the usual vocabulary is that we "apply" the function rather than "call" it.

Note how OCaml is flexible about whether you write the parentheses or not, and whether you write whitespace or not. One of the challenges of first learning OCaml can be figuring out when parentheses are actually required. So if you find yourself having problems with syntax errors, one strategy is to try adding some parentheses.

2.1.3 Loading code in the toplevel

In addition to allowing you to define functions, the toplevel will also accept *directives* that are not OCaml code but rather tell the toplevel itself to do something. All directives begin with the `#` character. Perhaps the most common directive is `#use`, which loads all the code from a file into the toplevel, just as if you had typed the code from that file into the toplevel.

For example, suppose you create a file named `mycode.ml`. In that file put the following code:

```
let inc x = x + 1
```

Start the toplevel. Try entering the following expression, and observe the error:

```
# inc 3;;
Error: Unbound value inc
Hint: Did you mean incr?
```

The error occurs because the toplevel does not yet know anything about a function named `inc`. Now issue the following directive to the toplevel:

```
# #use "mycode.ml";;
```

Note that the first # character above indicates the toplevel prompt to you. The second # character is one that you type to tell the toplevel that you are issuing a directive. Without that character, the toplevel would think that you are trying to apply a function named `use`.

Now try again:

```
# inc 3;;
- : int = 4
```

2.1.4 Workflow in the toplevel

The best workflow when using the toplevel with code stored in files is:

- Edit the code in the file.
- Load the code in the toplevel with `#use`.
- Interactively test the code.
- Exit the toplevel. **Warning:** do not skip this step.

Tip: Suppose you wanted to fix a bug in your code. It's tempting to not exit the toplevel, edit the file, and re-issue the `#use` directive into the same toplevel session. Resist that temptation. The "stale code" that was loaded from an earlier `#use` directive in the same session can cause surprising things to happen—surprising when you're first learning the language, anyway. So **always exit the toplevel before re-using a file.**

2.2 Compiling OCaml Programs

Using OCaml as a kind of interactive calculator can be fun, but we won't get very far with writing large programs that way. We instead need to store code in files and compile them.

2.2.1 Storing code in files

Open a terminal and use a text editor to create a file called `hello.ml`. Enter the following code into the file:

```
let _ = print_endline "Hello world!"
```

Note: There is no double semicolon `;;` at the end of that line of code. The double semicolon is intended for interactive sessions in the toplevel, so that the toplevel knows you are done entering a piece of code. There's usually no reason to write it in a .ml file.

The `let _ =` above means that we don't care to give a name (hence the "blank" or underscore) to code on the right-hand side of the =.

Save the file and return to the command line. Compile the code:

```
$ ocamlc -o hello.byte hello.ml
```

The compiler is named `ocamlc`. The `-o hello.byte` option says to name the output executable `hello.byte`. The executable contains compiled OCaml bytecode. In addition, two other files are produced, `hello.cmi` and `hello.cmo`. We don't need to be concerned with those files for now. Run the executable:

```
$ ./hello.byte
```

It should print `Hello world!` and terminate.

Now change the string that is printed to something of your choice. Save the file, recompile, and rerun. Try making the code print multiple lines.

This edit-compile-run cycle between the editor and the command line is something that might feel unfamiliar if you're used to working inside IDEs like Eclipse. Don't worry; it will soon become second nature.

Running the compiler directly is good to know how to do, but in larger projects, we want to use the OCaml build system to automatically find and link in libraries. Let's try using it:

```
$ ocamlbuild hello.byte
```

You will get an error from that command. Don't worry; just keep reading this exercise.

The build system is named `ocamlbuild`. The file we are asking it to build is the compiled bytecode `hello.byte`. The build system will automatically figure out that `hello.ml` is the source code for that desired bytecode.

However, the build system likes to be in charge of the whole compilation process. When it sees leftover files generated by a direct call to the compiler, as we did in the previous exercise, it rightly gets nervous and refuses to proceed. If you look at the error message, it says that a script has been generated to clean up from the old compilation. Run that script, and also remove the compiled file:

```
$ _build/sanitize.sh
$ rm hello.byte
```

After that, try building again:

```
$ ocamlbuild hello.byte
```

That should now succeed. There will be a directory `_build` that is created; it contains all the compiled code. That's one benefit of the build system over directly running the compiler: instead of polluting your source directory with a bunch of generated files, they get cleanly created in a separate directory. There's also a file `hello.byte` that is created, and it is actually just a link to "real" file of that name, which is in the `_build` directory.

Now run the executable:

```
$ ./hello.byte
```

You can now easily clean up all the compiled code:

```
$ ocamlbuild -clean
```

That removes the `_build` directory and `hello.byte` link, leaving just your source code.

2.2.2 What about Main?

Unlike C or Java, OCaml programs do not need to have a special function named `main` that is invoked to start the program. The usual idiom is just to have the very last definition in a file serve as the main function that kicks off whatever computation is to be done.

2.3 Expressions

The primary piece of OCaml syntax is the *expression*. Just like programs in imperative languages are primarily built out of *commands*, programs in functional languages are primarily built out of expressions. Examples of expressions include `2+2` and `increment 21`.

The OCaml manual has a complete definition of all the expressions in the language. Though that page starts with a rather cryptic overview, if you scroll down, you'll come to some English explanations. Don't worry about studying that page now; just know that it's available for reference.

The primary task of computation in a functional language is to *evaluate* an expression to a *value*. A value is an expression for which there is no computation remaining to be performed. So, all values are expressions, but not all expressions are values. Examples of values include `2`, `true`, and `"yay!"`.

The OCaml manual also has a definition of all the values, though again, that page is mostly useful for reference rather than study.

Sometimes an expression might fail to evaluate to a value. There are two reasons that might happen:

1. Evaluation of the expression raises an exception.
2. Evaluation of the expression never terminates (e.g., it enters an "infinite loop").

2.3.1 Assertions

The expression `assert e` evaluates `e`. If the result is `true`, nothing more happens, and the entire expression evaluates to a special value called *unit*. The unit value is written `()` and its type is `unit`. But if the result is `false`, an exception is raised.

2.3.2 Operators

Operators can be used to form expressions. OCaml has more or less all the usual operators you would expect in a language from the C or Java family of languages. See the table of all operators in the OCaml manual for details.

Here are two things to watch out for as you begin:

- OCaml deliberately does not support operator overloading. As a consequence, the integer and floating-point operators are distinct. E.g., to add integers, use `+`. To add floating-point numbers, use `+.`.
- There are two equality operators in OCaml, `=` and `==`, with corresponding inequality operators `<>` and `!=`. Operators `=` and `<>` examine *structural* equality whereas `==` and `!=` examine *physical* equality. Until we've studied the imperative features of OCaml, the difference between them will be tricky to explain. See the documentation of `Stdlib.(==)` if you're curious now.

Important: Start training yourself now to use `=` and not to use `==`. This will be difficult if you're coming from a language like Java where `==` is the usual equality operator.

2.3.3 If Expressions

The expression `if e1 then e2 else e3` evaluates to `e2` if `e1` evaluates to `true`, and to `e3` otherwise. We call `e1` the *guard* of the `if` expression.

```
if 3 + 5 > 2 then "yay!" else "boo!"
```

```
- : string = "yay!"
```

Unlike `if-then-else` *statements* that you may have used in imperative languages, `if-then-else` *expressions* in OCaml are just like any other expression; they can be put anywhere an expression can go. That makes them similar to the ternary operator `? :` that you might have used in other languages.

```
4 + (if 'a' = 'b' then 1 else 2)
```

```
- : int = 6
```

`If` expressions can be nested in a pleasant way:

```
if e1 then e2
else if e3 then e4
else if e5 then e6
...
else en
```

You should regard the final `else` as mandatory, regardless of whether you are writing a single `if` expression or a highly nested `if` expression. If you omit it you'll likely get an error message that, for now, is inscrutable:

```
if 2 > 3 then 5
```

```
File "[3]", line 1, characters 14-15:
1 | if 2 > 3 then 5
                  ^
Error: This expression has type int but an expression was expected of type
         unit
       because it is in the result of a conditional with no else branch
```

Syntax. The syntax of an `if` expression:

```
if e1 then e2 else e3
```

The letter `e` is used here to represent any other OCaml expression; it's an example of a *syntactic variable* aka *metavariable*, which is not actually a variable in the OCaml language itself, but instead a name for a certain syntactic construct. The numbers after the letter `e` are being used to distinguish the three different occurrences of it.

Dynamic semantics. The dynamic semantics of an `if` expression:

- if `e1` evaluates to `true`, and if `e2` evaluates to a value `v`, then `if e1 then e2 else e3` evaluates to `v`
- if `e1` evaluates to `false`, and if `e3` evaluates to a value `v`, then `if e1 then e2 else e3` evaluates to `v`.

We call these *evaluation rules*: they define how to evaluate expressions. Note how it takes two rules to describe the evaluation of an `if` expression, one for when the guard is true, and one for when the guard is false. The letter `v` is used here to represent any OCaml value; it's another example of a metavariable. Later we will develop a more mathematical way of expressing dynamic semantics, but for now we'll stick with this more informal style of explanation.

Static semantics. The static semantics of an `if` expression:

- if `e1` has type `bool` and `e2` has type `t` and `e3` has type `t` then `if e1 then e2 else e3` has type `t`

We call this a *typing rule*: it describes how to type check an expression. Note how it only takes one rule to describe the type checking of an `if` expression. At compile time, when type checking is done, it makes no difference whether the guard is true or false; in fact, there's no way for the compiler to know what value the guard will have at run time. The letter `t` here is used to represent any OCaml type; the OCaml manual also has definition of all types (which curiously does not name the base types of the language like `int` and `bool`).

We're going to be writing "has type" a lot, so let's introduce a more compact notation for it. Whenever we would write "`e` has type `t`", let's instead write `e : t`. The colon is pronounced "has type". This usage of colon is consistent with how the toplevel responds after it evaluates an expression that you enter:

```
let x = 42
```

```
val x : int = 42
```

In the above example, variable `x` has type `int`, which is what the colon indicates.

2.3.4 Let Expressions

In our use of the word `let` thus far, we've been making definitions in the toplevel and in `.ml` files. For example,

```
let x = 42;;
```

```
val x : int = 42
```

defines `x` to be 42, after which we can use `x` in future definitions at the toplevel. We'll call this use of `let` a *let definition*.

There's another use of `let` which is as an expression:

```
let x = 42 in x + 1
```

```
- : int = 43
```

Here we're *binding* a value to the name `x` then using that binding inside another expression, `x+1`. We'll call this use of `let` a *let expression*. Since it's an expression it evaluates to a value. That's different than definitions, which themselves do not evaluate to any value. You can see that if you try putting a let definition in place of where an expression is expected:

```
(let x = 42) + 1
```

```
File "[7]", line 1, characters 11-12:
1 | (let x = 42) + 1
               ^
Error: Syntax error: operator expected.
```

Syntactically, a `let` definition is not permitted on the left-hand side of the + operator, because a value is needed there, and definitions do not evaluate to values. On the other hand, a `let` expression would work fine:

```
(let x = 42 in x) + 1
```

```
- : int = 43
```

Another way to understand let definitions at the toplevel is that they are like let expression where we just haven't provided the body expression yet. Implicitly, that body expression is whatever else we type in the future. For example,

```
# let a = "big";;
# let b = "red";;
# let c = a ^ b;;
# ...
```

is understand by OCaml in the same way as

```
let a = "big" in
let b = "red" in
let c = a ^ b in
...
```

That latter series of `let` bindings is idiomatically how several variables can be bound inside a given block of code.

Syntax.

```
let x = e1 in e2
```

As usual, `x` is an identifier. We call `e1` the *binding expression*, because it's what's being bound to `x`; and we call `e2` the *body expression*, because that's the body of code in which the binding will be in scope.

Dynamic semantics.

To evaluate `let x = e1 in e2`:

- Evaluate `e1` to a value `v1`.
- Substitute `v1` for `x` in `e2`, yielding a new expression `e2'`.
- Evaluate `e2'` to a value `v2`.
- The result of evaluating the let expression is `v2`.

Here's an example:

```
    let x = 1 + 4 in x * 3
-->   (evaluate e1 to a value v1)
    let x = 5 in x * 3
-->   (substitute v1 for x in e2, yielding e2')
    5 * 3
-->   (evaluate e2' to v2)
    15
      (result of evaluation is v2)
```

Static semantics.

- If `e1 : t1` and if under the assumption that `x : t1` it holds that `e2 : t2`, then `(let x = e1 in e2) : t2`.

We use the parentheses above just for clarity. As usual, the compiler's type inferencer determines what the type of the variable is, or the programmer could explicitly annotate it with this syntax:

```
let x : t = e1 in e2
```

2.3.5 Scope

`Let` bindings are in effect only in the block of code in which they occur. This is exactly what you're used to from nearly any modern programming language. For example:

```
let x = 42 in
  (* y is not meaningful here *)
  x + (let y = "3110" in
         (* y is meaningful here *)
         int_of_string y)
```

The *scope* of a variable is where its name is meaningful. Variable `y` is in scope only inside of the `let` expression that binds it above.

It's possible to have overlapping bindings of the same name. For example:

```
let x = 5 in
  ((let x = 6 in x) + x)
```

But this is darn confusing, and for that reason, it is strongly discouraged style—much like ambiguous pronouns are discouraged in natural language. Nonetheless, let's consider what that code means.

To what value does that code evaluate? The answer comes down to how `x` is replaced by a value each time it occurs. Here are a few possibilities for such *substitution*:

```
(* possibility 1 *)
let x = 5 in
  ((let x = 6 in 6) + 5)

(* possibility 2 *)
let x = 5 in
  ((let x = 6 in 5) + 5)

(* possibility 3 *)
let x = 5 in
  ((let x = 6 in 6) + 6)
```

The first one is what nearly any reasonable language would do. And most likely it's what you would guess But, **why?**

The answer is something we'll call the *Principle of Name Irrelevance*: the name of a variable shouldn't intrinsically matter. You're used to this from math. For example, the following two functions are the same:

$$f(x) = x^2$$
$$f(y) = y^2$$

It doesn't intrinsically matter whether we call the argument to the function x or y; either way, it's still the squaring function. Therefore, in programs, these two functions should be identical:

```
let f x = x * x
let f y = y * y
```

This principle is more commonly known as *alpha equivalence*: the two functions are equivalent up to renaming of variables, which is also called *alpha conversion* for historical reasons that are unimportant here.

According to the Principle of Name Irrelevance, these two expressions should be identical:

```
let x = 6 in x
let y = 6 in y
```

Therefore, the following two expressions, which have the above expressions embedded in them, should also be identical:

```
let x = 5 in (let x = 6 in x) + x
let x = 5 in (let y = 6 in y) + x
```

But for those to be identical, we **must** choose the first of the three possibilities above. It is the only one that makes the name of the variable be irrelevant.

There is a term commonly used for this phenomenon: a new binding of a variable *shadows* any old binding of the variable name. Metaphorically, it's as if the new binding temporarily casts a shadow over the old binding. But eventually the old binding could reappear as the shadow recedes.

Shadowing is not mutable assignment. For example, both of the following expressions evaluate to 11:

```
let x = 5 in ((let x = 6 in x) + x)
let x = 5 in (x + (let x = 6 in x))
```

Likewise, the following utop transcript is not mutable assignment, though at first it could seem like it is:

```
# let x = 42;;
val x : int = 42
# let x = 22;;
val x : int = 22
```

Recall that every `let` definition in the toplevel is effectively a nested `let` expression. So the above is effectively the following:

```
let x = 42 in
  let x = 22 in
    ... (* whatever else is typed in the toplevel *)
```

The right way to think about this is that the second `let` binds an entirely new variable that just happens to have the same name as the first `let`.

Here is another utop transcript that is well worth studying:

```
# let x = 42;;
val x : int = 42
# let f y = x + y;;
val f : int -> int = <fun>
# f 0;;
: int = 42
# let x = 22;;
val x : int = 22
# f 0;;
- : int = 42  (* x did not mutate! *)
```

To summarize, each let definition binds an entirely new variable. If that new variable happens to have the same name as an old variable, the new variable temporarily shadows the old one. But the old variable is still around, and its value is immutable: it never, ever changes. So even though `let` expressions might superficially look like assignment statements from imperative languages, they are actually quite different.

2.4 Functions

Since OCaml is a functional language, there's a lot to cover about functions. Let's get started.

2.4.1 Function Definitions

The following code

```
let x = 42
```

has an expression in it (`42`) but is not itself an expression. Rather, it is a *definition*. Definitions bind values to names, in this case the value `42` being bound to the name `x`. The OCaml manual describes definitions (see the third major grouping titled "*definition*" on that page), but that manual page is again primarily for reference not for study. Definitions are not expressions, nor are expressions definitions—they are distinct syntactic classes. But definitions can have expressions nested inside them, and vice-versa.

For now, let's focus on one particular kind of definition, a *function definition*. Non-recursive functions are defined like this:

```
let f x = ...
```

Recursive functions are defined like this:

```
let rec f x = ...
```

The difference is just the `rec` keyword. It's probably a bit surprising that you explicitly have to add a keyword to make a function recursive, because most languages assume by default that they are. OCaml doesn't make that assumption, though. (Nor does the Scheme family of languages.)

One of the best known recursive functions is the factorial function. In OCaml, it can be written as follows:

```
(** [fact n] is [n]!.
    Requires: [n >= 0]. *)
let rec fact n = if n = 0 then 1 else n * fact (n - 1)
```

We provided a specification comment above the function to document the precondition (`Requires`) and postcondition (`is`) of the function.

Note that, as in many languages, OCaml integers are not the "mathematical" integers but are limited to a fixed number of bits. The manual specifies that (signed) integers are at least 31 bits, but they could be wider. As architectures have grown, so has that size. In current implementations, OCaml integers are 63 bits. So if you test on large enough inputs, you might begin to see strange results. The problem is machine arithmetic, not OCaml. (For interested readers: why 31 or 63 instead of 32 or 64? The OCaml garbage collector needs to distinguish between integers and pointers. The runtime representation of these therefore steals one bit to flag whether a word is an integer or a pointer.)

Here's another recursive function:

```
(** [pow x y] is [x] to the power of [y].
    Requires: [y >= 0]. *)
let rec pow x y = if y = 0 then 1 else x * pow x (y - 1)
```

Note how we didn't have to write any types in either of our functions: the OCaml compiler infers them for us automatically. The compiler solves this *type inference* problem algorithmically, but we could do it ourselves, too. It's like a mystery that can be solved by our mental power of deduction:

- Since the `if` expression can return `1` in the `then` branch, we know by the typing rule for `if` that the entire `if` expression has type `int`.

- Since the `if` expression has type `int`, the function's return type must be `int`.
- Since `y` is compared to `0` with the equality operator, `y` must be an `int`.
- Since `x` is multiplied with another expression using the `*` operator, `x` must be an `int`.

If we wanted to write down the types for some reason, we could do that:

```
let rec pow (x : int) (y : int) : int = ...
```

The parentheses are mandatory when we write the *type annotations* for `x` and `y`. We will generally leave out these annotations, because it's simpler to let the compiler infer them. There are other times when you'll want to explicitly write down types. One particularly useful time is when you get a type error from the compiler that you don't understand. Explicitly annotating the types can help with debugging such an error message.

Syntax. The syntax for function definitions:

```
let rec f x1 x2 ... xn = e
```

The `f` is a metavariable indicating an identifier being used as a function name. These identifiers must begin with a lowercase letter. The remaining rules for lowercase identifiers can be found in the manual. The names `x1` through `xn` are metavariables indicating argument identifiers. These follow the same rules as function identifiers. The keyword `rec` is required if `f` is to be a recursive function; otherwise it may be omitted.

Note that syntax for function definitions is actually simplified compared to what OCaml really allows. We will learn more about some augmented syntax for function definition in the next couple weeks. But for now, this simplified version will help us focus.

Mutually recursive functions can be defined with the `and` keyword:

```
let rec f x1 ... xn = e1
and g y1 ... yn = e2
```

For example:

```
(** [even n] is whether [n] is even.
    Requires: [n >= 0]. *)
let rec even n =
  n = 0 || odd (n - 1)

(** [odd n] is whether [n] is odd.
    Requires: [n >= 0]. *)
and odd n =
  n <> 0 && even (n - 1);;
```

The syntax for function types is:

```
t -> u
t1 -> t2 -> u
t1 -> ... -> tn -> u
```

The `t` and `u` are metavariables indicating types. Type `t -> u` is the type of a function that takes an input of type `t` and returns an output of type `u`. We can think of `t1 -> t2 -> u` as the type of a function that takes two inputs, the first of type `t1` and the second of type `t2`, and returns an output of type `u`. Likewise for a function that takes `n` arguments.

Dynamic semantics. There is no dynamic semantics of function definitions. There is nothing to be evaluated. OCaml just records that the name `f` is bound to a function with the given arguments `x1..xn` and the given body `e`. Only later, when the function is applied, will there be some evaluation to do.

Static semantics. The static semantics of function definitions:

- For non-recursive functions: if by assuming that `x1 : t1` and `x2 : t2` and … and `xn : tn`, we can conclude that `e : u`, then `f : t1 -> t2 -> ... -> tn -> u`.
- For recursive functions: if by assuming that `x1 : t1` and `x2 : t2` and … and `xn : tn` and `f : t1 -> t2 -> ... -> tn -> u`, we can conclude that `e : u`, then `f : t1 -> t2 -> ... -> tn -> u`.

Note how the type checking rule for recursive functions assumes that the function identifier `f` has a particular type, then checks to see whether the body of the function is well-typed under that assumption. This is because `f` is in scope inside the function body itself (just like the arguments are in scope).

2.4.2 Anonymous Functions

We already know that we can have values that are not bound to names. The integer `42`, for example, can be entered at the toplevel without giving it a name:

```
42
```

```
- : int = 42
```

Or we can bind it to a name:

```
let x = 42
```

```
val x : int = 42
```

Similarly, OCaml functions do not have to have names; they may be *anonymous*. For example, here is an anonymous function that increments its input: `fun x -> x+1`. Here, `fun` is a keyword indicating an anonymous function, `x` is the argument, and `->` separates the argument from the body.

We now have two ways we could write an increment function:

```
let inc x = x + 1
let inc = fun x -> x+1
```

They are syntactically different but semantically equivalent. That is, even though they involve different keywords and put some identifiers in different places, they mean the same thing.

Anonymous functions are also called *lambda expressions*, a term that comes from the *lambda calculus*, which is a mathematical model of computation in the same sense that Turing machines are a model of computation. In the lambda calculus, `fun x -> e` would be written $\lambda x.e$. The λ denotes an anonymous function.

It might seem a little mysterious right now why we would want functions that have no names. Don't worry; we'll see good uses for them later in the course, especially when we study so-called "higher-order programming". In particular, we will often create anonymous functions and pass them as input to other functions.

Syntax.

```
fun x1 ... xn -> e
```

Static semantics.

- If by assuming that `x1 : t1` and `x2 : t2` and … and `xn : tn`, we can conclude that `e : u`, then `fun x1 ... xn -> e : t1 -> t2 -> ... -> tn -> u`.

Dynamic semantics. An anonymous function is already a value. There is no computation to be performed.

2.4.3 Function Application

Here we cover a somewhat simplified syntax of function application compared to what OCaml actually allows.

Syntax.

```
e0 e1 e2 ... en
```

The first expression `e0` is the function, and it is applied to arguments `e1` through `en`. Note that parentheses are not required around the arguments to indicate function application, as they are in languages in the C family, including Java.

Static semantics.

- If `e0 : t1 -> ... -> tn -> u` and `e1 : t1` and … and `en : tn` then `e0 e1 ... en : u`.

Dynamic semantics.

To evaluate `e0 e1 ... en`:

1. Evaluate `e0` to a function. Also evaluate the argument expressions `e1` through `en` to values `v1` through `vn`.

   For `e0`, the result might be an anonymous function `fun x1 ... xn -> e` or a name `f`. In the latter case, we need to find the definition of `f`, which we can assume to be of the form `let rec f x1 ... xn = e`. Either way, we now know the argument names `x1` through `xn` and the body `e`.

2. Substitute each value `vi` for the corresponding argument name `xi` in the body `e` of the function. That substitution results in a new expression `e'`.

3. Evaluate `e'` to a value `v`, which is the result of evaluating `e0 e1 ... en`.

If you compare these evaluation rules to the rules for `let` expressions, you will notice they both involve substitution. This is not an accident. In fact, anywhere `let x = e1 in e2` appears in a program, we could replace it with `(fun x -> e2) e1`. They are syntactically different but semantically equivalent. In essence, `let` expressions are just syntactic sugar for anonymous function application.

2.4.4 Pipeline

There is a built-in infix operator in OCaml for function application called the *pipeline* operator, written `|>`. Imagine that as depicting a triangle pointing to the right. The metaphor is that values are sent through the pipeline from left to right. For example, suppose we have the increment function `inc` from above as well as a function `square` that squares its input. Here are two equivalent ways of writing the same computation:

```
square (inc 5)
5 |> inc |> square
(* both yield 36 *)
```

The latter uses the pipeline operator to send `5` through the `inc` function, then send the result of that through the `square` function. This is a nice, idiomatic way of expressing the computation in OCaml. The former way is arguably not as elegant: it involves writing extra parentheses and requires the reader's eyes to jump around, rather than move linearly from left to right. The latter way scales up nicely when the number of functions being applied grows, where as the former way requires more and more parentheses:

```
5 |> inc |> square |> inc |> inc |> square
square (inc (inc (square (inc 5))))
(* both yield 1444 *)
```

It might feel weird at first, but try using the pipeline operator in your own code the next time you find yourself writing a big chain of function applications.

Since `e1 |> e2` is just another way of writing `e2 e1`, we don't need to state the semantics for `|>`: it's just the same as function application. These two programs are another example of expressions that are syntactically different but semantically equivalent.

2.4.5 Polymorphic Functions

The *identity* function is the function that simply returns its input:

```
let id x = x;;
```

```
val id : 'a -> 'a = <fun>
```

The `'a` is a *type variable*: it stands for an unknown type, just like a regular variable stands for an unknown value. Type variables always begin with a single quote. Commonly used type variables include `'a`, `'b`, and `'c`, which OCaml programmers typically pronounce in Greek: alpha, beta, and gamma.

We can apply the identity function to any type of value we like:

```
# id 42;;
- : int = 42

# id true;;
- : bool = true

# id "bigred";;
- : string = "bigred"
```

Because you can apply `id` to many types of values, it is a *polymorphic* function: it can be applied to many (*poly*) forms (*morph*).

2.4.6 Labeled and Optional Arguments

The type and name of a function usually give you a pretty good idea of what the arguments should be. However, for functions with many arguments (especially arguments of the same type), it can be useful to label them. For example, you might guess that the function `String.sub` returns a substring of the given string (and you would be correct). You could type in `String.sub` to find its type:

```
String.sub;;
```

```
- : string -> int -> int -> string = <fun>
```

But it's not clear from the type how to use it—you're forced to consult the documentation.

OCaml supports labeled arguments to functions. You can declare this kind of function using the following syntax:

```
let f ~name1:arg1 ~name2:arg2 = arg1 + arg2;;
```

```
val f : name1:int -> name2:int -> int = <fun>
```

This function can be called by passing the labeled arguments in either order:

```
f ~name2:3 ~name1:4
```

Labels for arguments are often the same as the variable names for them. OCaml provides a shorthand for this case. The following are equivalent:

```
let f ~name1:name1 ~name2:name2 = name1+name2
let f ~name1 ~name2 = name1 + name2
```

Use of labeled arguments is largely a matter of taste. They convey extra information, but they can also add clutter to types.

The syntax to write both a labeled argument and an explicit type annotation for it is:

```
let f ~name1:(arg1 : int) ~name2:(arg2 : int) = arg1 + arg2
```

It is also possible to make some arguments optional. When called without an optional argument, a default value will be provided. To declare such a function, use the following syntax:

```
let f ?name:(arg1=8) arg2 = arg1 + arg2
```

```
val f : ?name:int -> int -> int = <fun>
```

You can then call a function with or without the argument:

```
f ~name:2 7
```

```
- : int = 9
```

```
f 7
```

```
- : int = 15
```

2.4.7 Partial Application

We could define an addition function as follows:

```
let add x y = x + y
```

```
val add : int -> int -> int = <fun>
```

Here's a rather similar function:

```
let addx x = fun y -> x + y
```

```
val addx : int -> int -> int = <fun>
```

Function `addx` takes an integer `x` as input and returns a *function* of type `int -> int` that will add `x` to whatever is passed to it.

The type of `addx` is `int -> int -> int`. The type of `add` is also `int -> int -> int`. So from the perspective of their types, they are the same function. But the form of `addx` suggests something interesting: we can apply it to just a single argument.

```
let add5 = addx 5
```

```
val add5 : int -> int = <fun>
```

```
add5 2
```

```
- : int = 7
```

It turns out the same can be done with `add`:

```
let add5 = add 5
```

```
val add5 : int -> int = <fun>
```

```
add5 2;;
```

```
- : int = 7
```

What we just did is called *partial application*: we partially applied the function `add` to one argument, even though you would normally think of it as a multi-argument function. This works because the following three functions are *syntactically different* but *semantically equivalent*. That is, they are different ways of expressing the same computation:

```
let add x y = x+y
let add x = fun y -> x+y
let add = fun x -> (fun y -> x+y)
```

So `add` is really a function that takes an argument `x` and returns a function `(fun y -> x+y)`. Which leads us to a deep truth…

2.4.8 Function Associativity

Are you ready for the truth? Here goes…

Every OCaml function takes exactly one argument.

Why? Consider `add`: although we can write it as `let add x y = x + y`, we know that's semantically equivalent to `let add = fun x -> (fun y -> x+y)`. And in general,

```
let f x1 x2 ... xn = e
```

is semantically equivalent to

```
let f =
  fun x1 ->
    (fun x2 ->
       (...
          (fun xn -> e)...))
```

So even though you think of `f` as a function that takes `n` arguments, in reality it is a function that takes 1 argument and returns a function.

The type of such a function

```
t1 -> t2 -> t3 -> t4
```

really means the same as

```
t1 -> (t2 -> (t3 -> t4))
```

That is, function types are *right associative*: there are implicit parentheses around function types, from right to left. The intuition here is that a function takes a single argument and returns a new function that expects the remaining arguments.

Function application, on the other hand, is *left associative*: there are implicit parenthesis around function applications, from left to right. So

```
e1 e2 e3 e4
```

really means the same as

```
((e1 e2) e3) e4
```

The intuition here is that the left-most expression grabs the next expression to its right as its single argument.

2.4.9 Operators as Functions

The addition operator `+` has type `int -> int -> int`. It is normally written *infix*, e.g., `3 + 4`. By putting parentheses around it, we can make it a *prefix* operator:

```
( + )
```

```
- : int -> int -> int = <fun>
```

```
( + ) 3 4;;
```

```
- : int = 7
```

```
let add3 = ( + ) 3
```

```
val add3 : int -> int = <fun>
```

```
add3 2
```

```
- : int = 5
```

The same technique works for any built-in operator.

Normally the spaces are unnecessary. We could write `(+)` or `( + )`, but it is best to include them. Beware of multiplication, which *must* be written as `( * )`, because `(*)` would be parsed as beginning a comment.

We can even define our own new infix operators, for example:

```
let ( ^^ ) x y = max x y
```

And now `2 ^^ 3` evaluates to `3`.

The rules for which punctuation can be used to create infix operators are not necessarily intuitive. Nor is the relative precedence with which such operators will be parsed. So be careful with this usage.

2.5 Documentation

OCaml provides a tool called OCamldoc that works a lot like Java's Javadoc tool: it extracts specially formatted comments from source code and renders them as HTML, making it easy for programmers to read documentation.

2.5.1 How to Document

Here's an example of an OCamldoc comment:

```
(** [sum lst] is the sum of the elements of [lst]. *)
let rec sum lst = ...
```

- The double asterisk is what causes the comment to be recognized as an OCamldoc comment.
- The square brackets around parts of the comment mean that those parts should be rendered in HTML as `type-writer font` rather than the regular font.

Also like Javadoc, OCamldoc supports *documentation tags*, such as `@author`, `@deprecated`, `@param`, `@return`, etc. For example, in the first line of most programming assignments, we ask you to complete a comment like this:

```
(** @author Your Name (your netid) *)
```

For the full range of possible markup inside a OCamldoc comment, see the OCamldoc manual. But what we've covered here is good enough for most documentation that you'll need to write.

2.5.2 What to Document

The documentation style we favor in this book resembles that of the OCaml standard library: concise and declarative. As an example, let's revisit the documentation of `sum`:

```
(** [sum lst] is the sum of the elements of [lst]. *)
let rec sum lst = ...
```

That comment starts with `sum lst`, which is an example application of the function to an argument. The comment continues with the word "is", thus declaratively describing the result of the application. (The word "returns" could be used instead, but "is" emphasizes the mathematical nature of the function.) That description uses the name of the argument, `lst`, to explain the result.

Note how there is no need to add tags to redundantly describe parameters or return values, as is often done with Javadoc. Everything that needs to be said has already been said. We strongly discourage documentation like the following:

```
(** Sum a list.
    @param lst The list to be summed.
    @return The sum of the list. *)
let rec sum lst = ...
```

That poor documentation takes three needlessly hard-to-read lines to say the same thing as the limpid one-line version.

There is one way we might improve the documentation we have so far, which is to explicitly state what happens with empty lists:

```
(** [sum lst] is the sum of the elements of [lst].
    The sum of an empty list is 0. *)
let rec sum lst = ...
```

2.5.3 Preconditions and Postconditions

Here are a few more examples of comments written in the style we favor.

```
(** [lowercase_ascii c] is the lowercase ASCII equivalent of
    character [c]. *)

(** [index s c] is the index of the first occurrence of
    character [c] in string [s].  Raises: [Not_found]
    if [c] does not occur in [s]. *)

(** [random_int bound] is a random integer between 0 (inclusive)
    and [bound] (exclusive).  Requires: [bound] is greater than 0
    and less than 2^30. *)
```

The documentation of `index` specifies that the function raises an exception, as well as what that exception is and the condition under which it is raised. (We will cover exceptions in more detail in the next chapter.) The documentation of `random_int` specifies that the function's argument must satisfy a condition.

In previous courses, you were exposed to the ideas of *preconditions* and *postconditions*. A precondition is something that must be true before some section of code; and a postcondition, after.

The "Requires" clause above in the documentation of `random_int` is a kind of precondition. It says that the client of the `random_int` function is responsible for guaranteeing something about the value of `bound`. Likewise, the first sentence of that same documentation is a kind of postcondition. It guarantees something about the value returned by the function.

The "Raises" clause in the documentation of `index` is another kind of postcondition. It guarantees that the function raises an exception. Note that the clause is not a precondition, even though it states a condition in terms of an input.

Note that none of these examples has a "Requires" clause that says something about the type of an input. If you're coming from a dynamically-typed language, like Python, this could be a surprise. Python programmers frequently document preconditions regarding the types of function inputs. OCaml programmers, however, do not. That's because the compiler itself does the type checking to ensure that you never pass a value of the wrong type to a function. Consider `lowercase_ascii` again: although the English comment helpfully identifies the type of `c` to the reader, the comment does not state a "Requires" clause like this:

```
(** [lowercase_ascii c] is the lowercase ASCII equivalent of [c].
    Requires: [c] is a character. *)
```

Such a comment reads as highly unidiomatic to an OCaml programmer, who would read that comment and be puzzled, perhaps thinking: "Well of course `c` is a character; the compiler will guarantee that. What did the person who wrote that really mean? Is there something they or I am missing?"

2.6 Debugging

Debugging is a last resort when everything else has failed. Let's take a step back and think about everything that comes *before* debugging.

2.6.1 Defenses against Bugs

According to Rob Miller, there are four defenses against bugs:

1. **The first defense against bugs is to make them impossible.**

   Entire classes of bugs can be eradicated by choosing to program in languages that guarantee *memory safety* (that no part of memory can be accessed except through a *pointer* (or reference) that is valid for that region of memory) and *type safety* (that no value can be used in a way inconsistent with its type). The OCaml type system, for example, prevents programs from buffer overflows and meaningless operations (like adding a boolean to a float), whereas the C type system does not.

2. **The second defense against bugs is to use tools that find them.**

   There are automated source-code analysis tools, like FindBugs, which can find many common kinds of bugs in Java programs, and SLAM, which is used to find bugs in device drivers. The subfield of CS known as *formal methods* studies how to use mathematics to specify and verify programs, that is, how to prove that programs have no bugs. We'll study verification later in this course.

   Social methods such as code reviews and pair programming are also useful tools for finding bugs. Studies at IBM in the 1970s-1990s suggested that code reviews can be remarkably effective. In one study (Jones, 1991), code inspection found 65% of the known coding errors and 25% of the known documentation errors, whereas testing found only 20% of the coding errors and none of the documentation errors.

3. **The third defense against bugs is to make them immediately visible.**

   The earlier a bug appears, the easier it is to diagnose and fix. If computation instead proceeds past the point of the bug, then that further computation might obscure where the failure really occurred. *Assertions* in the source code make programs "fail fast" and "fail loudly", so that bugs appear immediately, and the programmer knows exactly where in the source code to look.

4. **The fourth defense against bugs is extensive testing.**

   How can you know whether a piece of code has a particular bug? Write tests that would expose the bug, then confirm that your code doesn't fail those tests. *Unit tests* for a relatively small piece of code, such as an individual function or module, are especially important to write at the same time as you develop that code. Running of those tests should be automated, so that if you ever break the code, you find out as soon as possible. (That's really Defense 3 again.)

After all those defenses have failed, a programmer is forced to resort to debugging.

2.6.2 How to Debug

So you've discovered a bug. What next?

1. **Distill the bug into a small test case.** Debugging is hard work, but the smaller the test case, the more likely you are to focus your attention on the piece of code where the bug lurks. Time spent on this distillation can therefore be time saved, because you won't have to re-read lots of code. Don't continue debugging until you have a small test case!

2. **Employ the scientific method.** Formulate a hypothesis as to why the bug is occurring. You might even write down that hypothesis in a notebook, as if you were in a Chemistry lab, to clarify it in your own mind and keep track of what hypotheses you've already considered. Next, design an experiment to affirm or deny that hypothesis. Run your experiment and record the result. Based on what you've learned, reformulate your hypothesis. Continue until you have rationally, scientifically determined the cause of the bug.

3. **Fix the bug.** The fix might be a simple correction of a typo. Or it might reveal a design flaw that causes you to make major changes. Consider whether you might need to apply the fix to other locations in your code base—for example, was it a copy and paste error? If so, do you need to refactor your code?

4. **Permanently add the small test case to your test suite.** You wouldn't want the bug to creep back into your code base. So keep track of that small test case by keeping it as part of your unit tests. That way, any time you make future changes, you will automatically be guarding against that same bug. Repeatedly running tests distilled from previous bugs is called *regression testing*.

2.6.3 Debugging in OCaml

Here are a couple tips on how to debug—if you are forced into it—in OCaml.

- **Print statements.** Insert a print statement to ascertain the value of a variable. Suppose you want to know what the value of `x` is in the following function:

```
let inc x = x + 1
```

  Just add the line below to print that value:

```
let inc x =
  let () = print_int x in
  x + 1
```

  The Stdlib module contains many other printing statements you can use. We cover some of them in the next section.

- **Function traces.** Suppose you want to see the *trace* of recursive calls and returns for a function. Use the `#trace` directive:

```
# let rec fib x = if x <= 1 then 1 else fib (x - 1) + fib (x - 2);;
# #trace fib;;
```

  If you evaluate `fib 2`, you will now see the following output:

```
fib <-- 2
fib <-- 0
fib --> 1
fib <-- 1
fib --> 1
fib --> 2
```

  To stop tracing, use the `#untrace` directive.

- **Debugger.** OCaml has a debugging tool `ocamldebug`. You can find a tutorial on the OCaml website. Unless you are using Emacs as your editor, you will probably find this tool to be harder to use than just inserting print statements.

2.6.4 Printing

OCaml has built-in printing functions for several of the built-in primitive types: `print_char`, `print_int`, `print_string`, and `print_float`. There's also a `print_endline` function, which is like `print_string`, but also outputs a newline.

Let's look at the types of a couple of those functions:

```
print_endline
```

```
- : string -> unit = <fun>
```

```
print_string
```

```
- : string -> unit = <fun>
```

They both take a string as input and return a value of type `unit`, which we haven't seen before. There is only one value of this type, which is written `()` and is also pronounced "unit". So `unit` is like `bool`, except there is one fewer value of type `unit` than there is of `bool`. Unit is therefore used when you need to take an argument or return a value, but there's no interesting value to pass or return. Unit is often used when you're writing or using code that has side effects. Printing is an example of a side effect: it changes the world and can't be undone.

If you want to print one thing after another, you could sequence some print functions using nested let expressions:

```
let x = print_endline "THIS" in
let y = print_endline "IS" in
print_endline "3110"
```

But the boilerplate of all the `let x = ... in` above is annoying to have to write! We don't really care about giving names to the unit values returned by those printing functions. So there's a special syntax that can be used to chain together multiple functions who return unit. The expression `e1; e2` first evaluates `e1`, which should evaluate to `()`, then discards that value, and evaluates `e2`. So we could rewrite the above code as:

```
print_endline "THIS";
print_endline "IS";
print_endline "3110"
```

And that is far more idiomatic code.

If `e1` does not have type `unit`, then `e1; e2` will give a warning, because you are discarding useful values. If that is truly your intent, you can call the built-in function `ignore : 'a -> unit` to convert any value to `()`:

```
(ignore 3); 5
```

```
- : int = 5
```

2.6.5 Defensive Programming

As we discussed earlier in the section on debugging, one defense against bugs is to make any bugs (or errors) immediately visible. That idea connects with idea of preconditions.

Consider this specification of `random_int`:

```
(** [random_int bound] is a random integer between 0 (inclusive)
    and [bound] (exclusive).  Requires: [bound] is greater than 0
    and less than 2^30. *)
```

If the client of `random_int` passes a value of `bound` that violates the "Requires" clause, such as `-1`, the implementation of `random_int` is free to do anything whatsoever. All bets are off when the client violates the precondition.

But the most helpful thing for `random_int` to do is to immediately expose the fact that the precondition was violated. After all, chances are that the client didn't *mean* to violate it.

So the implementor of `random_int` would do well to check whether the precondition is violated, and if so, raise an exception. Here are three possibilities of that kind of *defensive programming:*

```
(* possibility 1 *)
let random_int bound =
  assert (bound > 0 && bound < 1 lsl 30);
  (* proceed with the implementation of the function *)

(* possibility 2 *)
let random_int bound =
  if not (bound > 0 && bound < 1 lsl 30)
  then invalid_arg "bound";
  (* proceed with the implementation of the function *)

(* possibility 3 *)
let random_int bound =
  if not (bound > 0 && bound < 1 lsl 30)
  then failwith "bound";
  (* proceed with the implementation of the function *)
```

The second possibility is probably the most informative to the client, because it uses the built-in function `invalid_arg` to raise the well-named exception `Invalid_argument`. In fact, that's exactly what the standard library implementation of this function does.

The first possibility is probably most useful when you are trying to debug your own code, rather than choosing to expose a failed assertion to a client.

The third possibility differs from the second only in the name (`Failure`) of the exception that is raised. It might be useful in situations where the precondition involves more than just a single invalid argument.

In this example, checking the precondition is computationally cheap. In other cases, it might require a lot of computation, so the implementer of the function might prefer not to check the precondition, or only to check some inexpensive approximation to it.

Sometimes programmers worry unnecessarily that defensive programming will be too expensive—either in terms of the time it costs them to implement the checks initially, or in the run-time costs that will be paid in checking assertions. These concerns are far too often misplaced. The time and money it costs society to repair faults in software suggests that we could all afford to have programs that run a little more slowly.

Finally, the implementer might even choose to eliminate the precondition and restate it as a postcondition:

```
(** [random_int bound] is a random integer between 0 (inclusive)
    and [bound] (exclusive).  Raises: [Invalid_argument "bound"]
    unless [bound] is greater than 0 and less than 2^30. *)
```

Now instead of being free to do whatever when `bound` is too big or too small, `random_int` must raise an exception. For this function, that's probably the best choice.

In this course, we're not going to force you to program defensively. But if you're savvy, you'll start (or continue) doing it anyway. The small amount of time you spend coding up such defenses will save you hours of time in debugging, making you a more productive programmer.

2.7 Summary

Syntax and semantics are a powerful paradigm for learning a programming language. As we learn the features of OCaml, we're being careful to write down their syntax and semantics. We've seen that there can be multiple syntaxes for expressing the same semantic idea, that is, the same computation.

The semantics of function application is the very heart of OCaml and of functional programming, and it's something we will come back to several times throughout the course to deepen our understanding.

2.7.1 Terms and concepts

- anonymous functions
- assertions
- binding
- binding expression
- body expression
- debugging
- defensive programming
- definitions
- documentation
- dynamic semantics
- evaluation
- expressions
- function application
- function definitions
- identifiers
- idioms
- if expressions
- lambda expressions
- let definition
- let expression
- libraries
- metavariables
- mutual recursion
- pipeline operator
- postcondition
- precondition
- printing
- recursion

- semantics
- static semantics
- substitution
- syntax
- tools
- type checking
- type inference
- values

2.7.2 Further reading

- *Introduction to Objective Caml*, chapter 3
- *OCaml from the Very Beginning*, chapter 2
- *Real World OCaml*, chapter 2

2.8 Exercises

Solutions to exercises are available to students in Cornell's CS 3110. Instructors at other institutions are welcome to contact Michael Clarkson for access.

Exercise: values [★]

What is the type and value of each of the following OCaml expressions?

- `7 * (1 + 2 + 3)`
- `"CS " ^ string_of_int 3110`

Hint: type each expression into the toplevel and it will tell you the answer. Note: `^` is not exponentiation.

Exercise: operators [★★]

Examine the table of all operators in the OCaml manual (you will have to scroll down to find it on that page).

- Write an expression that multiplies `42` by `10`.
- Write an expression that divides `3.14` by `2.0`. *Hint: integer and floating-point operators are written differently in OCaml.*
- Write an expression that computes `4.2` raised to the seventh power. *Note: there is no built-in integer exponentiation operator in OCaml (nor is there in C, by the way), in part because it is not an operation provided by most CPUs.*

Exercise: equality [★]

- Write an expression that compares `42` to `42` using structural equality.
- Write an expression that compares `"hi"` to `"hi"` using structural equality. What is the result?
- Write an expression that compares `"hi"` to `"hi"` using physical equality. What is the result?

Exercise: assert [★]

- Enter `assert true;;` into utop and see what happens.
- Enter `assert false;;` into utop and see what happens.
- Write an expression that asserts 2110 is not (structurally) equal to 3110.

Exercise: if [★]

Write an if expression that evaluates to `42` if `2` is greater than `1` and otherwise evaluates to `7`.

Exercise: double fun [★]

Using the increment function from above as a guide, define a function `double` that multiplies its input by 2. For example, `double 7` would be `14`. Test your function by applying it to a few inputs. Turn those test cases into assertions.

Exercise: more fun [★★]

- Define a function that computes the cube of a floating-point number. Test your function by applying it to a few inputs.
- Define a function that computes the sign (1, 0, or -1) of an integer. Use a nested if expression. Test your function by applying it to a few inputs.
- Define a function that computes the area of a circle given its radius. Test your function with `assert`.

For the latter, bear in mind that floating-point arithmetic is not exact. Instead of asserting an exact value, you should assert that the result is "close enough", e.g., within 1e-5. If that's unfamiliar to you, it would be worthwhile to read up on floating-point arithmetic.

A function that take multiple inputs can be defined just by providing additional names for those inputs as part of the let definition. For example, the following function computes the average of three arguments:

```
let avg3 x y z = (x +. y +. z) /. 3.
```

Exercise: RMS [★★]

Define a function that computes the *root mean square* of two numbers—i.e., $\sqrt{(x^2 + y^2)/2}$. Test your function with `assert`.

Exercise: date fun [★★★]

Define a function that takes an integer `d` and string `m` as input and returns `true` just when `d` and `m` form a *valid date*. Here, a valid date has a month that is one of the following abbreviations: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sept, Oct, Nov, Dec. And the day must be a number that is between 1 and the minimum number of days in that month, inclusive. For example, if the month is Jan, then the day is between 1 and 31, inclusive, whereas if the month is Feb, then the day is between 1 and 28, inclusive.

How terse (i.e., few and short lines of code) can you make your function? You can definitely do this in fewer than 12 lines.

Exercise: fib [★★]

Define a recursive function `fib : int -> int`, such that `fib n` is the `n`th number in the Fibonacci sequence, which is 1, 1, 2, 3, 5, 8, 13, … That is:

- `fib 1 = 1`,
- `fib 2 = 1`, and
- `fib n = fib (n-1) + fib (n-2)` for any `n > 2`.

Test your function in the toplevel.

Exercise: fib fast [★★★]

How quickly does your implementation of `fib` compute the 50th Fibonacci number? If it computes nearly instantaneously, congratulations! But the recursive solution most people come up with at first will seem to hang indefinitely. The problem is that the obvious solution computes subproblems repeatedly. For example, computing `fib 5` requires computing both `fib 3` and `fib 4`, and if those are computed separately, a lot of work (an exponential amount, in fact) is being redone.

Create a function `fib_fast` that requires only a linear amount of work. *Hint:* write a recursive helper function `h : int -> int -> int -> int`, where `h n pp p` is defined as follows:

- `h 1 pp p = p`, and
- `h n pp p = h (n-1) p (pp+p)` for any `n > 1`.

The idea of `h` is that it assumes the previous two Fibonacci numbers were `pp` and `p`, then computes forward `n` more numbers. Hence, `fib n = h n 0 1` for any `n > 0`.

What is the first value of `n` for which `fib_fast n` is negative, indicating that integer overflow occurred?

Exercise: poly types [★★★]

What is the type of each of the functions below? You can ask the toplevel to check your answers

```
let f x = if x then x else x
let g x y = if y then x else x
let h x y z = if x then y else z
let i x y z = if x then y else y
```

Exercise: divide [★★]

Write a function `divide : numerator:float -> denominator:float -> float`. Apply your function.

Exercise: associativity [★★]

Suppose that we have defined `let add x y = x + y`. Which of the following produces an integer, which produces a function, and which produces an error? Decide on an answer, then check your answer in the toplevel.

- `add 5 1`
- `add 5`
- `(add 5) 1`
- `add (5 1)`

Exercise: average [★★]

Define an infix operator `+/.` to compute the average of two floating-point numbers. For example,

- `1.0 +/. 2.0 = 1.5`
- `0. +/. 0. = 0.`

Exercise: hello world [★]

Type the following in utop:

- `print_endline "Hello world!";;`
- `print_string "Hello world!";;`

Notice the difference in output from each.