OCaml Programming: Correct + Efficient + Beautiful

Michael R. Clarkson et al.

Jun 28, 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.

The title of this book was previously "Functional Programming in OCaml".

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 It has since been superceded by the "OCaml Platform" plugin.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.

2.4.10 Tail Recursion

Consider the following seemingly uninteresting function, which counts from 1 to `n`:

```
(** [count n] is [n], computed by adding 1 to itself [n] times.  That is,
    this function counts up from 1 to [n]. *)
let rec count n =
  if n = 0 then 0 else 1 + count (n - 1)
```

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

Counting to 10 is no problem:

```
count 10
```

```
- : int = 10
```

Counting to 100,000 is no problem either:

```
count 100_000
```

```
- : int = 100000
```

But try counting to 1,000,000 and you'll get the following error:

```
Stack overflow during evaluation (looping recursion?).
```

What's going on here?

The Call Stack. The issue is that the *call stack* has a limited size. You probably learned in one of your introductory programming classes that most languages implement function calls with a stack. That stack contains one element for each function call that has been started but has not yet completed. Each element stores information like the values of local variables and which instruction in the function is currently being executed. When the evaluation of one function body calls another function, a new element is pushed on the call stack, and it is popped off when the called function completes.

The size of the stack is usually limited by the operating system. So if the stack runs out of space, it becomes impossible to make another function call. Normally this doesn't happen, because there's no reason to make that many successive function calls before returning. In cases where it does happen, there's good reason for the operating system to make that program stop: it might be in the process of eating up *all* the memory available on the entire computer, thus harming other programs running on the same computer. The `count` function isn't likely to do that, but this function would:

```
let rec count_forever n = 1 + count_forever n
```

```
val count_forever : 'a -> int = <fun>
```

So the operating system for safety's sake limits the call stack size. That means eventually `count` will run out of stack space on a large enough input. Notice how that choice is really independent of the programming language. So this same issue can and does occur in languages other than OCaml, including Python and Java. You're just less likely to have seen it manifest there, because you probably never wrote quite as many recursive functions in those languages.

Tail Recursion. There is a solution to this issue that was described in a 1977 paper about LISP by Guy Steele. The solution, *tail-call optimization*, requires some cooperation between the programmer and the compiler. The programmer does a little rewriting of the function, which the compiler then notices and applies an optimization. Let's see how it works.

Suppose that a recursive function `f` calls itself then returns the result of that recursive call. Our `count` function does *not* do that:

```
let rec count n =
  if n = 0 then 0 else 1 + count (n - 1)
```

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

Rather, after the recursive call `count (n - 1)`, there is computation remaining: the computer still needs to add `1` to the result of that call.

But we as programmers could rewrite the `count` function so that it does *not* need to do any additional computation after the recursive call. The trick is to create a helper function with an extra parameter:

```
let rec count_aux n acc =
  if n = 0 then acc else count_aux (n - 1) (acc + 1)

let count_tr n = count_aux n 0
```

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

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

Function `count_aux` is almost the same as our original `count`, but it adds an extra parameter named `acc`, which is idiomatic and stands for "accumulator". The idea is that the value we want to return from the function is slowly, with each recursive call, being accumulated in it. The "remaining computation" —the addition of 1— now happens *before* the recursive call not *after*. When the base case of the recursion finally arrives, the function now returns `acc`, where the answer has been accumulated.

But the original base case of 0 still needs to exist in the code somwhere. And it does, as the original value of `acc` that is passed to `count_aux`. Now `count_tr` (we'll get to why the name is "tr" in just a minute) works as a replacement for our original `count`.

At this point we've completed the programmer's responsibility, but it's probably not clear why we went through this effort. After all `count_aux` will still call itself recursively too many times as `count` did, and eventually overflow the stack.

That's where the compiler's responsibility kicks in. A good compiler (and the OCaml compiler is good this way) can notice when a recursive call is in *tail position*, which is a technical way of saying "there's no more computation to be done after it returns". The recursive call to `count_aux` is in tail position; the recursive call to `count` is not. Here they are again so you can compare them:

```
let rec count n =
  if n = 0 then 0 else 1 + count (n - 1)

let rec count_aux n acc =
  if n = 0 then acc else count_aux (n - 1) (acc + 1)
```

Here's why tail position matters: **A recursive call in tail position does not need a new stack frame. It can just reuse the existing stack frame.** That's because there's nothing left of use in the existing stack frame! There's no computation left to be done, so none of the local variables, or next instruction to execute, etc. matter any more. None of that memory ever needs to be read again, because that call is effectively already finished. So instead of wasting space by allocating another stack frame, the compiler "recycles" the space used by the previous frame.

This is the *tail-call optimization*. It can even be applied in cases beyond recursive functions if the calling function's stack frame is suitably compatible with the callee. And, it's a big deal. The tail-call optimization reduces the stack space requirements from linear to constant. Whereas `count` needed $O(n)$ stack frames, `count_aux` needs only $O(1)$, because the same frame gets reused over and over again for each recursive call. And that means `count_tr` actually can count to 1,000,000:

```
count_tr 1_000_000
```

```
- : int = 1000000
```

Finally, why did we name this function `count_tr`? The "tr" stands for *tail recursive*. A tail recursive function is a recursive function whose recursive calls are all in tail position. In other words, it's a function that (unless there are other pathologies) will not exhaust the stack.

The Importance of Tail Recursion. Sometimes beginning functional programmers fixate a bit too much upon it. If all you care about is writing the first draft of a function, you probably don't need to worry about tail recursion. It's pretty easy to make it tail recursive later if you need to, just by adding an accumulator argument. Or maybe you should rethink how you have designed the function. Take `count`, for example: it's kind of dumb. But later we'll see examples that aren't dumb, such as iterating over lists with thousands of elements.

It is important that the compiler support the optimization. Otherwise, the transformation you do to the code as a programmer makes no difference. Indeed, most compilers do support it, at least as an option. Java is a notable exception.

The Recipe for Tail Recursion. In a nutshell, here's how we made a function be tail recursive:

1. Change the function into a helper function. Add an extra argument: the accumulator, often named `acc`.
2. Write a new "main" version of the function that calls the helper. It passes the original base case's return value as the initial value of the accumulator.
3. Change the helper function to return the accumulator in the base case.
4. Change the helper function's recursive case. It now needs to do the extra work on the accumulator argument, before the recursive call. This is the only step that requires much ingenuity.

An Example: Factorial. Let's tranform this factorial function to be tail recursive:

```
(* [fact n] is [n] factorial *)
let rec fact n =
  if n = 0 then 1 else n * fact (n - 1)
```

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

First, we change its name and add an accumulator argument:

```
let rec fact_aux n acc = ...
```

Second, we write a new "main" function that calls the helper with the original base case as the accumulator:

```
let rec fact_tr n = fact_aux n 1
```

Third, we change the helper function to return the accumulator in the base case:

```
if n = 0 then acc ...
```

Finally, we change the recursive case:

```
else fact (n - 1) (n * acc)
```

Putting it all together, we have:

```
let rec fact_aux n acc =
  if n = 0 then acc else fact_aux (n - 1) (n * acc)

let fact_tr n = fact_aux n 1
```

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

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

It was a nice exercise, but maybe not worthwhile. Even before we exhaust the stack space, the computation suffers from integer overflow:

```
fact 50
```

```
- : int = -3258495067890909184
```

To solve that problem, we turn to OCaml's big integer library, Zarith. Here we use a few OCaml features that are beyond anything we've seen so far, but hopefully nothing terribly surprising.

```
#require "zarith.top";;

let rec zfact_aux n acc =
  if Z.equal n Z.zero then acc else zfact_aux (Z.pred n) (Z.mul acc n);;

let zfact_tr n = zfact_aux n Z.one;;

zfact_tr (Z.of_int 50)
```

```
val zfact_aux : Z.t -> Z.t -> Z.t = <fun>
```

```
val zfact_tr : Z.t -> Z.t = <fun>
```

```
- : Z.t = 30414093201713378043612608166064768844377641568960512000000000000
```

If you want you can use that code to compute `zfact_tr 1_000_000` without stack or integer overflow, though it will take several minutes.

The chapter on modules will explain the OCaml features we used above in detail, but for now:

- `#require` loads the library, which provides a module named `Z`. Recall that $\mathbb{Z}$ is the symbol used in mathematics to denote the integers.
- `Z.n` means the name `n` defined inside of `Z`.
- The type `Z.t` is the library's name for the type of big integers.
- We use library values `Z.equal` for equality comparison, `Z.zero` for 0, `Z.pred` for predecessor (i.e., subtracting 1), `Z.mul` for multiplication, `Z.one` for 1, and `Z.of_int` to convert a primitive integer to a big integer.

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
- *Tail Recursion, The Musical*. Tail-call optimization explained in the context of JavaScript with cute 8-bit animations, and Disney songs!

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.

CHAPTER

THREE

DATA

In this chapter, we'll examine some of OCaml's built-in data types, including lists, variants, records, tuples, and options. Many of those are likely to feel familiar from other programming languages. In particular,

- **lists** and **tuples**, might feel similar to Python; and
- **records** and **variants**, might feel similar to `struct` and `enum` types from C or Java.

Because of that familiarity, we call these *standard* data types. We'll learn about *pattern matching*, which is a feature that's less likely to be familiar.

Almost immediately after we learn about lists, we'll pause our study of standard data types to learn about unit testing in OCaml with OUnit, a unit testing framework similar to those you might have used in other languages. OUnit relies on lists, which is why we couldn't cover it before now.

Later in the chapter, we study some OCaml data types that are unlikely to be as familiar from other languages. They include:

- **options**, which are loosely related to `null` in Java;
- **association lists**, which are an amazingly simple implementation of maps (aka dictionaries) based on lists and tuples;
- **algebraic data types**, which are arguably the most important kind of type in OCaml, and indeed are the power behind many of the other built-in types; and
- **exceptions**, which are a special kind of algebraic data type.

3.1 Lists

An OCaml list is a sequence of values all of which have the same type. They are implemented as singly-linked lists. These lists enjoy a first-class status in the language: there is special support for easily creating and working with lists. That's a characteristic that OCaml shares with many other functional languages. Mainstream imperative languages, like Python, have such support these days too. Maybe that's because programmers find it so pleasant to work directly with lists as a first-class part of the language, rather than having to go through a library (as in C and Java).

3.1.1 Building Lists

Syntax. There are three syntactic forms for building lists:

```
[]
e1 :: e2
[e1; e2; ...; en]
```

The empty list is written `[]` and is pronounced "nil", a name that comes from Lisp. Given a list `lst` and element `elt`, we can prepend `elt` to `lst` by writing `elt :: lst`. The double-colon operator is pronounced "cons", a name that comes from an operator in Lisp that constructs objects in memory. "Cons" can also be used as a verb, as in "I will cons an element onto the list." The first element of a list is usually called its *head* and the rest of the elements (if any) are called its *tail*.

The square bracket syntax is convenient but unnecessary. Any list `[e1; e2; ...; en]` could instead be written with the more primitive nil and cons syntax: `e1 :: e2 :: ... :: en :: []`. When a pleasant syntax can be defined in terms of a more primitive syntax within the language, we call the pleasant syntax *syntactic sugar*: it makes the language "sweeter". Transforming the sweet syntax into the more primitive syntax is called *desugaring*.

Because the elements of the list can be arbitrary expressions, lists can be nested as deeply as we like, e.g., `[[[]]; [[1; 2; 3]]]`.

Dynamic semantics.

- `[]` is already a value.
- If `e1` evaluates to `v1`, and if `e2` evaluates to `v2`, then `e1 :: e2` evaluates to `v1 :: v2`.

As a consequence of those rules and how to desugar the square-bracket notation for lists, we have the following derived rule:

- If `ei` evaluates to `vi` for all `i` in `1..n`, then `[e1; ...; en]` evaluates to `[v1; ...; vn]`.

It's starting to get tedious to write "evaluates to" in all our evaluation rules. So let's introduce a shorter notation for it. We'll write `e ==> v` to mean that `e` evaluates to `v`. Note that `==>` is not a piece of OCaml syntax. Rather, it's a notation we use in our description of the language, kind of like metavariables. Using that notation, we can rewrite the latter two rules above:

- If `e1 ==> v1`, and if `e2 ==> v2`, then `e1 :: e2 ==> v1 :: v2`.
- If `ei ==> vi` for all `i` in `1..n`, then `[e1; ...; en] ==> [v1; ...; vn]`.

Static semantics.

All the elements of a list must have the same type. If that element type is `t`, then the type of the list is `t list`. You should read such types from right to left: `t list` is a list of `t`'s, `t list list` is a list of list of `t`'s, etc. The word `list` itself here is not a type: there is no way to build an OCaml value that has type simply `list`. Rather, `list` is a *type constructor*: given a type, it produces a new type. For example, given `int`, it produces the type `int list`. You could think of type constructors as being like functions that operate on types, instead of functions that operate on values.

The type-checking rules:

- `[] : 'a list`
- If `e1 : t` and `e2 : t list` then `e1 :: e2 : t list`. In case the colons and their precedence is confusing, the latter means `(e1 :: e2) : t list`.

In the rule for `[]`, recall that `'a` is a type variable: it stands for an unknown type. So the empty list is a list whose elements have an unknown type. If we cons an `int` onto it, say `2 :: []`, then the compiler infers that for that particular list, `'a` must be `int`. But if in another place we cons a `bool` onto it, say `true :: []`, then the compiler infers that for that particular list, `'a` must be `bool`.

3.1.2 Accessing Lists

Note: The video linked above also uses records and tuples as examples. Those are covered in the *next section* of this book.

There are really only two ways to build a list, with nil and cons. So if we want to take apart a list into its component pieces, we have to say what to do with the list if it's empty, and what to do if it's non-empty (that is, a cons of one element onto some other list). We do that with a language feature called *pattern matching*.

Here's an example of using pattern matching to compute the sum of a list:

```
let rec sum lst =
  match lst with
  | [] -> 0
  | h :: t -> h + sum t
```

```
val sum : int list -> int = <fun>
```

This function says to take the input `lst` and see whether it has the same shape as the empty list. If so, return 0. Otherwise, if it has the same shape as the list `h :: t`, then let `h` be the first element of `lst`, and let `t` be the rest of the elements of `lst`, and return `h + sum t`. The choice of variable names here is meant to suggest "head" and "tail" and is a common idiom, but we could use other names if we wanted. Another common idiom is:

```
let rec sum xs =
  match xs with
  | [] -> 0
  | x :: xs' -> x + sum xs'
```

```
val sum : int list -> int = <fun>
```

That is, the input list is a list of xs (pronounced EX-uhs), the head element is an x, and the tail is xs' (pronounced EX-uhs prime).

Syntactically it isn't necessary to use so many lines to define `sum`. We could do it all on one line:

```
let rec sum xs = match xs with | [] -> 0 | x :: xs' -> x + sum xs'
```

```
val sum : int list -> int = <fun>
```

Or, noting that the first `|` after `with` is optional regardless of how many lines we use, we could also write:

```
let rec sum xs = match xs with [] -> 0 | x :: xs' -> x + sum xs'
```

```
val sum : int list -> int = <fun>
```

The multi-line format is what we'll usually use in this book, because it helps the human eye understand the syntax a bit better. OCaml code formatting tools, though, are moving toward the single-line format whenever the code is short enough to fit on just one line.

Here's another example of using pattern matching to compute the length of a list:

```
let rec length lst =
  match lst with
  | [] -> 0
  | h :: t -> 1 + length t
```

```
val length : 'a list -> int = <fun>
```

Note how we didn't actually need the variable `h` in the right-hand side of the pattern match. When we want to indicate the presence of some value in a pattern without actually giving it a name, we can write _ (the underscore character):

```
let rec length lst =
  match lst with
  | [] -> 0
  | _ :: t -> 1 + length t
```

```
val length : 'a list -> int = <fun>
```

That function is actually built-in as part of the OCaml standard library `List` module. Its name there is `List.length`. That "dot" notation indicates the function named `length` inside the module named `Length`, much like the dot notation used in many other languages.

And here's a third example that appends one list onto the beginning of another list:

```
let rec append lst1 lst2 =
  match lst1 with
  | [] -> lst2
  | h :: t -> h :: append t lst2
```

```
val append : 'a list -> 'a list -> 'a list = <fun>
```

For example, `append [1; 2] [3; 4]` is `[1; 2; 3; 4]`. That function is actually available as a built-in operator `@`, so we could instead write `[1; 2] @ [3; 4]`.

As a final example, we could write a function to determine whether a list is empty:

```
let empty lst =
  match lst with
  | [] -> true
  | h :: t -> false
```

```
val empty : 'a list -> bool = <fun>
```

But there a much better way to write the same function without pattern matching:

```
let empty lst =
  lst = []
```

```
val empty : 'a list -> bool = <fun>
```

Note how all the recursive functions above are similar to doing proofs by induction on the natural numbers: every natural number is either 0 or is 1 greater than some other natural number n, and so a proof by induction has a base case for 0 and an inductive case for $n + 1$. Likewise all our functions have a base case for the empty list and a recursive case for the list that has one more element than another list. This similarity is no accident. There is a deep relationship between induction and recursion; we'll explore that relationship in more detail later in the book.

By the way, there are two library functions `List.hd` and `List.tl` that return the head and tail of a list. It is not good, idiomatic OCaml to apply these directly to a list. The problem is that they will raise an exception when applied to the empty list, and you will have to remember to handle that exception. Instead, you should use pattern matching: you'll then be forced to match against both the empty list and the non-empty list (at least), which will prevent exceptions from being raised, thus making your program more robust.

3.1.3 (Not) Mutating Lists

Lists are immutable. There's no way to change an element of a list from one value to another. Instead, OCaml programmers create new lists out of old lists. For example, suppose we wanted to write a function that returned the same list as its input list, but with the first element (if there is one) incremented by one. We could do that:

```
let inc_first lst =
  match lst with
  | [] -> []
  | h :: t -> h + 1 :: t
```

Now you might be concerned about whether we're being wasteful of space. After all, there are at least two ways the compiler could implement the above code:

1. Copy the entire tail list `t` when the new list is created in the pattern match with cons, such that the amount of memory in use just increased by an amount proportionate to the length of `t`.
2. Share the tail list `t` between the old list and the new list, such that the amount of memory in use does not increase—beyond the one extra piece of memory needed to store `h + 1`.

In fact, the compiler does the latter. So there's no need for concern. The reason that it's quite safe for the compiler to implement sharing is exactly that list elements are immutable. If they were instead mutable, then we'd start having to worry about whether the list I have is shared with the list you have, and whether changes I make will be visible in your list. So immutability makes it easier to reason about the code, and makes it safe for the compiler to perform an optimization.

3.1.4 Pattern Matching with Lists

We saw above how to access lists using pattern matching. Let's look more carefully at this feature.

Syntax.

```
match e with
| p1 -> e1
| p2 -> e2
| ...
| pn -> en
```

Each of the clauses `pi -> ei` is called a *branch* or a *case* of the pattern match. The first vertical bar in the entire pattern match is optional.

The `p`'s here are a new syntactic form called a *pattern*. For now, a pattern may be:

- a variable name, e.g. `x`
- the underscore character _, which is called the *wildcard*
- the empty list `[]`
- `p1 :: p2`
- `[p1; ...; pn]`

No variable name may appear more than once in a pattern. For example, the pattern `x :: x` is illegal. The wildcard may occur any number of times.

As we learn more of data structures available in OCaml, we'll expand the possibilities for what a pattern may be.

Dynamic semantics.

Pattern matching involves two inter-related tasks: determining whether a pattern matches a value, and determining what parts of the value should be associated with which variable names in the pattern. The former task is intuitively about

determining whether a pattern and a value have the same *shape*. The latter task is about determining the *variable bindings* introduced by the pattern. For example, consider the following code:

```
match 1 :: [] with
| [] -> false
| h :: t -> h >= 1 && List.length t = 0
```

```
- : bool = true
```

When evaluating the right-hand side of the second branch, `h` is bound to `1` and `t` is bound to `[]`. Let's write `h->1` to mean the variable binding saying that `h` has value `1`; this is not a piece of OCaml syntax, but rather a notation we use to reason about the language. So the variable bindings produced by the second branch would be `h->1, t->[]`.

Using that notation, here is a definition of when a pattern matches a value and the bindings that match produces:

- The pattern `x` matches any value `v` and produces the variable binding `x->v`.
- The pattern _ matches any value and produces no bindings.
- The pattern `[]` matches the value `[]` and produces no bindings.
- If `p1` matches `v1` and produces a set b_1 of bindings, and if `p2` matches `v2` and produces a set b_2 of bindings, then `p1 :: p2` matches `v1 :: v2` and produces the set $b_1 \cup b_2$ of bindings. Note that `v2` must be a list (since it's on the right-hand side of `::`) and could have any length: 0 elements, 1 element, or many elements. Note that the union $b_1 \cup b_2$ of bindings will never have a problem where the same variable is bound separately in both b_1 and b_2 because of the syntactic restriction that no variable name may appear more than once in a pattern.
- If for all `i` in `1..n`, it holds that `pi` matches `vi` and produces the set b_i of bindings, then `[p1; ...; pn]` matches `[v1; ...; vn]` and produces the set $\bigcup_i b_i$ of bindings. Note that this pattern specifies the exact length the list must be.

Now we can say how to evaluate `match e with p1 -> e1 | ... | pn -> en`:

- Evaluate `e` to a value `v`.
- Match `v` against `p1`, then against `p2`, and so on, in the order they appear in the match expression.
- If `v` does not match against any of the patterns, then evaluation of the match expression raises a `Match_failure` exception. We haven't yet discussed exceptions in OCaml, but you're surely familiar with them from other languages. We'll come back to exceptions near the end of this chapter, after we've covered some of the other built-in data structures in OCaml.
- Otherwise, stop trying to match at the first time a match succeeds against a pattern. Let `pi` be that pattern and let b be the variable bindings produced by matching `v` against `pi`.
- Substitute those bindings inside `ei`, producing a new expression `e'`.
- Evaluate `e'` to a value `v'`.
- The result of the entire match expression is `v'`.

For example, here's how this match expression would be evaluated:

```
match 1 :: [] with
| [] -> false
| h :: t -> h = 1 && t = []
```

```
- : bool = true
```

- `1 :: []` is already a value.
- `[]` does not match `1 :: []`.

- `h :: t` does match `1 :: []` and produces variable bindings {`h->1`,`t->[]`}, because:
  - `h` matches `1` and produces the variable binding `h->1`.
  - `t` matches `[]` and produces the variable binding `t->[]`.
- Substituting {`h->1`,`t->[]`} inside `h = 1 && t = []` produces a new expression `1 = 1 && [] = []`.
- Evaluating `1 = 1 && [] = []` yields the value `true`. We omit the justification for that fact here, but it follows from other evaluation rules for built-in operators and function application.
- So the result of the entire match expression is `true`.

Static semantics.

- If `e : ta` and for all `i`, it holds that `pi : ta` and `ei : tb`, then `(match e with p1 -> e1 | ... | pn -> en) : tb`.

That rule relies on being able to judge whether a pattern has a particular type. As usual, type inference comes into play here. The OCaml compiler infers the types of any pattern variables as well as all occurrences of the wildcard pattern. As for the list patterns, they have the same type-checking rules as list expressions.

Additional Static Checking.

In addition to that type-checking rule, there are two other checks the compiler does for each match expression.

First, **exhaustiveness:** the compiler checks to make sure that there are enough patterns to guarantee that at least one of them matches the expression `e`, no matter what the value of that expression is at run time. This ensures that the programmer did not forget any branches. For example, the function below will cause the compiler to emit a warning:

```
let head lst = match lst with h :: _ -> h
```

```
File "[12]", line 1, characters 15-41:
```

```
1 | let head lst = match lst with h :: _ -> h
```

```
                   ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
Warning 8: this pattern-matching is not exhaustive.
```

```
Here is an example of a case that is not matched:
```

```
[]
```

```
val head : 'a list -> 'a = <fun>
```

By presenting that warning to the programmer, the compiler is helping the programmer to defend against the possibility of `Match_failure` exceptions at runtime.

Note: Sorry about how the output from the cell above gets split into many lines in the HTML. That is currently an open issue with JupyterBook, the framework used to build this book.

Second, **unused branches:** the compiler checks to see whether any of the branches could never be matched against because one of the previous branches is guaranteed to succeed. For example, the function below will cause the compiler to emit a warning:

```
let rec sum lst =
  match lst with
  | h :: t -> h + sum t
  | [ h ] -> h
  | [] -> 0
```

```
File "[13]", line 4, characters 4-9:
```

```
4 |   | [ h ] -> h
```

```
        ^^^^^
```

```
Warning 11: this match case is unused.
```

```
val sum : int list -> int = <fun>
```

The second branch is unused because the first branch will match anything the second branch matches.

Unused match cases are usually a sign that the programmer wrote something other than what they intended. So by presenting that warning, the compiler is helping the programmer to detect latent bugs in their code.

Here's an example of one of the most common bugs that causes an unused match case warning. Understanding it is also a good way to check your understanding of the dynamic semantics of match expressions:

```
let length_is lst n =
  match List.length lst with
  | n -> true
  | _ -> false
```

```
File "[14]", line 4, characters 4-5:
```

```
4 |   | _ -> false
```

```
        ^
```

```
Warning 11: this match case is unused.
```

```
val length_is : 'a list -> 'b -> bool = <fun>
```

The programmer was thinking that if the length of `lst` is equal to `n`, then this function will return `true`, and otherwise will return `false`. But in fact this function *always* returns `true`. Why? Because the pattern variable `n` is distinct from the function argument `n`. Suppose that the length of `lst` is 5. Then the pattern match becomes: `match 5 with n -> true | _ -> false`. Does `n` match 5? Yes, according to the rules above: a variable pattern matches any value and here produces the binding `n->5`. Then evaluation applies that binding to `true`, substituting all occurrences of `n` inside of `true` with 5. Well, there are no such occurrences. So we're done, and the result of evaluation is just `true`.

What the programmer really meant to write was:

```
let length_is lst n =
  match List.length lst with
  | m -> m = n
```

```
val length_is : 'a list -> int -> bool = <fun>
```

or better yet:

```
let length_is lst n =
  List.length lst = n
```

```
val length_is : 'a list -> int -> bool = <fun>
```

3.1.5 Deep Pattern Matching

Patterns can be nested. Doing so can allow your code to look deeply into the structure of a list. For example:

- `_ :: []` matches all lists with exactly one element
- `_ :: _` matches all lists with at least one element
- `_ :: _ :: []` matches all lists with exactly two elements
- `_ :: _ :: _ :: _` matches all lists with at least three elements

3.1.6 Immediate Matches

When you have a function that immediately pattern-matches against its final argument, there's a nice piece of syntactic sugar you can use to avoid writing extra code. Here's an example: instead of

```
let rec sum lst =
  match lst with
  | [] -> 0
  | h :: t -> h + sum t
```

```
val sum : int list -> int = <fun>
```

you can write

```
let rec sum = function
  | [] -> 0
  | h :: t -> h + sum t
```

```
val sum : int list -> int = <fun>
```

The word `function` is a keyword. Notice that we're able to leave out the line containing `match` as well as the name of the argument, which was never used anywhere else but that line. In such cases, though, it's especially important in the specification comment for the function to document what that argument is supposed to be, since the code no longer gives it a descriptive name.

3.1.7 OCamldoc and List Syntax

OCamldoc is a documentation generator similar to Javadoc. It extracts comments from source code and produces HTML (as well as other output formats). The standard library web documentation for the List module is generated by OCamldoc from the standard library source code for that module, for example.

> **Warning:** There is a syntactic convention with square brackets in OCamldoc that can be confusing with respect to lists.
>
> In an OCamldoc comment, source code is surrounded by square brackets. That code will be rendered in typewriter face and syntax-highlighted in the output HTML. The square brackets in this case do not indicate a list.

For example, here is the comment for `List.hd` in the standard library source code:

```
(** Return the first element of the given list. Raise
   [Failure "hd"] if the list is empty. *)
```

The `[Failure "hd"]` does not mean a list containing the exception `Failure "hd"`. Rather it means to typeset the expression `Failure "hd"` as source code, as you can see here.

This can get especially confusing when you want to talk about lists as part of the documentation. For example, here is a way we could rewrite that comment:

```
(** [hd lst] returns the first element of [lst].
    Raises [Failure "hd"] if [lst = []]. *)
```

In `[lst = []]`, the outer square brackets indicate source code as part of a comment, whereas the inner square brackets indicate the empty list.

3.1.8 List Comprehensions

Some languages, including Python and Haskell, have a syntax called *comprehension* that allows lists to be written somewhat like set comprehensions from mathematics. The earliest example of comprehensions seems to be the functional language NPL, which was designed in 1977.

OCaml doesn't have built-in syntactic support for comprehensions. Though some extensions were developed, none seem to be supported any longer. The primary tasks accomplished by comprehensions (filtering out some elements, and transforming others) are actually well-supported already by *higher-order programming*, which we'll study in a later chapter, and the pipeline operator, which we've already learned. So an additional syntax for comprehensions was never really needed.

3.1.9 Tail Recursion

Recall that a function is *tail recursive* if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call. Consider these two implementations, `sum` and `sum_tr` of summing a list:

```
let rec sum (l : int list) : int =
  match l with
  | [] -> 0
  | x :: xs -> x + (sum xs)

let rec sum_plus_acc (acc : int) (l : int list) : int =
  match l with
  | [] -> acc
  | x :: xs -> sum_plus_acc (acc + x) xs

let sum_tr : int list -> int =
  sum_plus_acc 0
```

```
val sum : int list -> int = <fun>
```

```
val sum_plus_acc : int -> int list -> int = <fun>
```

```
val sum_tr : int list -> int = <fun>
```

Observe the following difference between the `sum` and `sum_tr` functions above: In the `sum` function, which is not tail recursive, after the recursive call returned its value, we add `x` to it. In the tail recursive `sum_tr`, or rather in `sum_plus_acc`, after the recursive call returns, we immediately return the value without further computation.

If you're going to write functions on really long lists, tail recursion becomes important for performance. So when you have a choice between using a tail-recursive vs. non-tail-recursive function, you are likely better off using the tail-recursive function on really long lists to achieve space efficiency. For that reason, the List module documents which functions are tail recursive and which are not.

But that doesn't mean that a tail-recursive implementation is strictly better. For example, the tail-recursive function might be harder to read. (Consider `sum_plus_acc`.) Also, there are cases where implementing a tail-recursive function entails having to do a pre- or post-processing pass to reverse the list. On small to medium sized lists, the overhead of reversing the list (both in time and in allocating memory for the reversed list) can make the tail-recursive version less time efficient. What constitutes "small" vs. "big" here? That's hard to say, but maybe 10,000 is a good estimate, according to the standard library documentation of the `List` module.

Here is a useful tail-recursive function to produce a long list:

```
(** [from i j l] is the list containing the integers from [i] to [j],
    inclusive, followed by the list [l].
    Example:  [from 1 3 [0] = [1;2;3;0]] *)
let rec from i j l = if i > j then l else from i (j - 1) (j :: l)

(** [i -- j] is the list containing the integers from [i] to [j], inclusive. *)
let ( -- ) i j = from i j []

let longlist = 0 -- 1_000_000
```

```
val from : int -> int -> int list -> int list = <fun>
```

```
val ( -- ) : int -> int -> int list = <fun>
```

```
val longlist : int list =
  [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19; 20;
   21; 22; 23; 24; 25; 26; 27; 28; 29; 30; 31; 32; 33; 34; 35; 36; 37; 38;
   39; 40; 41; 42; 43; 44; 45; 46; 47; 48; 49; 50; 51; 52; 53; 54; 55; 56;
   57; 58; 59; 60; 61; 62; 63; 64; 65; 66; 67; 68; 69; 70; 71; 72; 73; 74;
   75; 76; 77; 78; 79; 80; 81; 82; 83; 84; 85; 86; 87; 88; 89; 90; 91; 92;
   93; 94; 95; 96; 97; 98; 99; 100; 101; 102; 103; 104; 105; 106; 107; 108;
   109; 110; 111; 112; 113; 114; 115; 116; 117; 118; 119; 120; 121; 122; 123;
   124; 125; 126; 127; 128; 129; 130; 131; 132; 133; 134; 135; 136; 137; 138;
   139; 140; 141; 142; 143; 144; 145; 146; 147; 148; 149; 150; 151; 152; 153;
   154; 155; 156; 157; 158; 159; 160; 161; 162; 163; 164; 165; 166; 167; 168;
   169; 170; 171; 172; 173; 174; 175; 176; 177; 178; 179; 180; 181; 182; 183;
   184; 185; 186; 187; 188; 189; 190; 191; 192; 193; 194; 195; 196; 197; 198;
   199; 200; 201; 202; 203; 204; 205; 206; 207; 208; 209; 210; 211; 212; 213;
   214; 215; 216; 217; 218; 219; 220; 221; 222; 223; 224; 225; 226; 227; 228;
   229; 230; 231; 232; 233; 234; 235; 236; 237; 238; 239; 240; 241; 242; 243;
   244; 245; 246; 247; 248; 249; 250; 251; 252; 253; 254; 255; 256; 257; 258;
```

(continues on next page)

(continued from previous page)

```
   259; 260; 261; 262; 263; 264; 265; 266; 267; 268; 269; 270; 271; 272; 273;
   274; 275; 276; 277; 278; 279; 280; 281; 282; 283; 284; 285; 286; 287; 288;
   289; 290; 291; 292; 293; 294; 295; 296; 297; 298; ...]
```

It would be worthwhile to study the definition of `--` to convince yourself that you understand (i) how it works and (ii) why it is tail recursive.

3.2 Variants

A *variant* is a data type representing a value that is one of several possibilities. At their simplest, variants are like enums from C or Java:

```
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
let d = Tue
```

```
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

```
val d : day = Tue
```

The individual names of the values of a variant are called *constructors* in OCaml. In the example above, the constructors are `Sun`, `Mon`, etc. This is a somewhat different use of the word constructor than in C++ or Java.

For each kind of data type in OCaml, we've been discussing how to build and access it. For variants, building is easy: just write the name of the constructor. For accessing, we use pattern matching. For example:

```
let int_of_day d =
  match d with
  | Sun -> 1
  | Mon -> 2
  | Tue -> 3
  | Wed -> 4
  | Thu -> 5
  | Fri -> 6
  | Sat -> 7
```

```
val int_of_day : day -> int = <fun>
```

There isn't any kind of automatic way of mapping a constructor name to an `int`, like you might expect from languages with enums.

Syntax.

Defining a variant type:

```
type t = C1 | ... | Cn
```

The constructor names must begin with an uppercase letter. OCaml uses that to distinguish constructors from variable identifiers.

The syntax for writing a constructor value is simply its name, e.g., `C`.

Dynamic semantics.

- A constructor is already a value. There is no computation to perform.

Static semantics.

- If `t` is a type defined as `type t = ... | C | ...`, then `C : t`.

3.2.1 Scope

Suppose there are two types defined with overlapping constructor names, for example,

```
type t1 = C | D
type t2 = D | E
let x = D
```

```
type t1 = C | D
```

```
type t2 = D | E
```

```
val x : t2 = D
```

When `D` appears after these definitions, to which type does it refer? That is, what is the type of `x` above? The answer is that the type defined later wins. So `x : t2`. That is potentially surprising to programmers, so within any given scope (e.g., a file or a module, though we haven't covered modules yet) it's idiomatic whenever overlapping constructor names might occur to prefix them with some distinguishing character. For example, suppose we're defining types to represent Pokémon:

```
type ptype =
  TNormal | TFire | TWater

type peff =
  ENormal | ENotVery | ESuper
```

```
type ptype = TNormal | TFire | TWater
```

```
type peff = ENormal | ENotVery | ESuper
```

Because "Normal" would naturally be a constructor name for both the type of a Pokémon and the effectiveness of a Pokémon attack, we add an extra character in front of each constructor name to indicate whether it's a type or an effectiveness.

3.2.2 Pattern Matching

Each time we introduced a new kind of data type, we need to introduce the new patterns associated with it. For variants, this is easy. We add the following new pattern form to the list of legal patterns:

- a constructor name `C`

And we extend the definition of when a pattern matches a value and produces a binding as follows:

- The pattern `C` matches the value `C` and produces no bindings.

Note: Variants are considerably more powerful than what we have seen here. We'll return to them again soon.

3.3 Unit Testing with OUnit

Note: This section is a bit of a detour from our study of data types, but it's a good place to take the detour: we now know just enough to understand how unit testing can be done in OCaml, and there's no good reason to wait any longer to learn about it.

Using the toplevel to test functions will only work for very small programs. Larger programs need *test suites* that contain many *unit tests* and can be re-run every time we update our code base. A unit test is a test of one small piece of functionality in a program, such as an individual function.

We've now learned enough features of OCaml to see how to do unit testing with a library called OUnit. It is a unit testing framework similar to JUnit in Java, HUnit in Haskell, etc. The basic workflow for using OUnit is as follows:

- Write a function in a file `f.ml`. There could be many other functions in that file too.
- Write unit tests for that function in a separate file `f_test.ml`. That exact name, with an underscore and "test" is not actually essential, but is a convention we'll often follow.
- Build and run `f_test` to execute the unit tests.

The OUnit documentation is available on Github.

3.3.1 An Example of OUnit

The following example shows you how to create an OUnit test suite. There are some things in the example that might at first seem mysterious; they are discussed in the next section.

Create a file named `sum.ml`, and put the following code into it:

```
let rec sum = function
  | [] -> 0
  | x :: xs -> x + sum xs
```

Now create a second file named `sum_test.ml`, and put this code into it:

```
open OUnit2
open Sum

let tests = "test suite for sum" >::: [
  "empty" >:: (fun _ -> assert_equal 0 (sum []));
  "singleton" >:: (fun _ -> assert_equal 1 (sum [1]));
  "two_elements" >:: (fun _ -> assert_equal 3 (sum [1; 2]));
]

let _ = run_test_tt_main tests
```

Depending on your editor and its configuration, you probably now see some "Unbound module" errors about OUnit2 and Sum. Your code is actually correct, but your editor doesn't know how to understand it yet. To fix that, create a file in the same directory and name it `.merlin`. (Note the leading dot in that filename.) In that file put the following:

```
B _build
PKG ounit2
```

The first line tells Merlin to look for compiled code from other source files inside the `_build` directory, which is where OCamlbuild places compiled code. The second line tells Merlin to look for the OUnit2 package.

After you save the `.merlin` file, the error about `OUnit2` should be gone, though you might need to cause your editor to re-check the `sum_test.ml` file to get rid of it. You can close and re-open the window, or make a trivial change in the file (e.g., add then delete a space) to make that tappen.

But the error about `Sum` will still be there, because you haven't yet compiled that file. To do that, run this command:

```
$ ocamlbuild -pkgs ounit2 sum_test.byte
```

Now the error about `Sum` should be gone, though again you might need to convince your editor to re-check the file.

Finally, you can run the test suite:

```
$ ./sum_test.byte
```

You will get a response something like this:

```
...
Ran: 3 tests in: 0.12 seconds.
OK
```

Now suppose we modify `sum.ml` to introduce a bug by changing the code in it to the following:

```
let rec sum = function
  | [] -> 1 (* bug *)
  | x :: xs -> x + sum xs
```

If rebuild and re-execute `sum_test.byte`, all test cases now fail. The output tells us the names of the failing cases. Here's the beginning of the output, in which we've replaced some strings that will be dependent on your own local computer with `...`:

```
FFF
============================================================================
Error: test suite for sum:2:two_elements.

File ".../_build/oUnit-test suite for sum-...#01.log", line 8, characters 1-1:
Error: test suite for sum:2:two_elements (in the log).

Called from unknown location

not equal
----------------------------------------------------------------------------
```

The first line of that output

```
FFF
```

tells us that OUnit ran three test cases and all three failed.

The next interesting line

```
Error: test suite for sum:2:two_elements.
```

tells us that in the test suite named `test suite for sum` the test case at index 2 named `two_elements` failed. The rest of the output for that test case is not particularly interesting; let's ignore it for now.

3.3.2 Explanation of the OUnit Example

Let's study more carefully what we just did in the previous section. In the test file, `open OUnit2` brings into scope the many definitions in OUnit2, which is version 2 of the OUnit framework. And `open Sum` brings into scope the definitions from `sum.ml`. We'll learn more about scope and the `open` keyword later in a later chapter.

Then we created a list of test cases:

```
[
  "empty"  >:: (fun _ -> assert_equal 0 (sum []));
  "one"    >:: (fun _ -> assert_equal 1 (sum [1]));
  "onetwo" >:: (fun _ -> assert_equal 3 (sum [1; 2]));
]
```

Each line of code is a separate test case. A test case has a string giving it a descriptive name, and a function to run as the test case. In between the name and the function we write `>::`, which is a custom operator defined by the OUnit framework. Let's look at the first function from above:

```
fun _ -> assert_equal 0 (sum [])
```

Every test case function receives as input a parameter that OUnit calls a *test context*. Here (and in many of the test cases we write) we don't actually need to worry about the context, so we use the underscore to indicate that the function ignores its input. The function then calls `assert_equal`, which is a function provided by OUnit that checks to see whether its two arguments are equal. If so the test case succeeds. If not, the test case fails.

Then we created a test suite:

```
let tests = "test suite for sum" >::: [
  "empty" >:: (fun _ -> assert_equal 0 (sum []));
  "singleton" >:: (fun _ -> assert_equal 1 (sum [1]));
  "two_elements" >:: (fun _ -> assert_equal 3 (sum [1; 2]));
]
```

The `>:::` operator is another custom OUnit operator. It goes between the name of the test suite and the list of test cases in that suite.

Then we ran the test suite:

```
let _ = run_test_tt_main tests
```

The function `run_test_tt_main` is provided by OUnit. It runs a test suite and prints the results of which test cases passed vs. which failed to standard output. The use of `let _ =` here indicates that we don't care what value the function returns; it just gets discarded.

Finally, when we compiled the test file, we linked in the OUnit2 package:

```
$ ocamlbuild -pkgs ounit2 sum_test.byte
```

If you get tired of typing the `pkgs ounit2` part of that, you can instead create a file named `_tags` (note the underscore) in the same directory and put the following into it:

```
true: package(ounit2)
```

Now Ocamlbuild will automatically link in OUnit2 everytime you compile in this directory, without you having to give the `pkgs` flag. The tradeoff is that you now have to pass a different flag to Ocamlbuild:

```
$ ocamlbuild -use-ocamlfind sum_test.byte
```

And you will continue having to pass that flag as long as the `_tags` file exists. Why is this any better? If there are many packages you want to link, with the tags file you end up having to pass only one option on the command line, instead of many.

3.3.3 Improving OUnit Output

In our example with the buggy implementation of `sum`, we got the following output:

```
==============================================================================
Error: test suite for sum:2:two_elements.

File ".../_build/oUnit-test suite for sum-...#01.log", line 8, characters 1-1:
Error: test suite for sum:2:two_elements (in the log).

Called from unknown location

not equal
------------------------------------------------------------------------------
```

Let's see how to improve that output to be a little more informative.

Stack Traces

The `Called from an unknown location` indicates OCaml was unable to provide a stack trace. That happened because, by default, stack traces are disabled. We can enable them by compiling the code with the debug tag:

```
$ ocamlbuild -pkgs ounit2 -tag debug sum_test.byte
$ ./sum_test.byte

==============================================================================
Error: test suite for sum:2:two_elements.

File "/Users/clarkson/tmp/sum/_build/oUnit-test suite for sum-...#01.log", line 9,␣
 ↪characters 1-1:
Error: test suite for sum:2:two_elements (in the log).

Raised at file "src/oUnitAssert.ml", line 45, characters 8-27
Called from file "src/oUnitRunner.ml", line 46, characters 13-26

not equal
------------------------------------------------------------------------------
```

Now we see the stack trace that resulted from `assert_equal` raising an exception. You'll probably agree that stack trace isn't very informative though: what matters is which test case fails, not which files in the implementation of OUnit were involved in raising the exception. And we could already identify the failing test case from the first line of output. (It's the test case named `two_elements`, which at position 2 in the test suite named `test suite for sum`.)

So we don't usually bother enabling stack traces for OUnit test suites. Nonetheless, it could occasionally be useful if your *own* code is raising exceptions that you want to track down.

Output Values

The `not equal` in the OUnit output means that `assert_equal` discovered the two values passed to it in that test case were not equal. That's not so informative: we'd like to know *why* they're not equal. In particular, we'd like to know what the actual output produced by `sum` was for that test case. To find out, we need to pass an additional argument to `assert_equal`. That argument, whose label is `printer`, should be a function that can transform the outputs to strings. In this case, the outputs are integers, so `string_of_int` from the Stdlib module will suffice. We modify the test suite as follows:

```
let tests = "test suite for sum" >::: [
  "empty" >:: (fun _ -> assert_equal 0 (sum []) ~printer:string_of_int);
  "singleton" >:: (fun _ -> assert_equal 1 (sum [1]) ~printer:string_of_int);
  "two_elements" >:: (fun _ -> assert_equal 3 (sum [1; 2]) ~printer:string_of_int);
]
```

And now we get more informative output:

```
==============================================================================
Error: test suite for sum:2:two_elements.

File "/Users/clarkson/tmp/sum/_build/oUnit-test suite for sum-sauternes#01.log", line␣
 ↪8, characters 1-1:
Error: test suite for sum:2:two_elements (in the log).

Called from unknown location

expected: 3 but got: 4
------------------------------------------------------------------------------
```

That output means that the test named `two_elements` asserted the equality of `3` and `4`. The expected output was `3` because that was the first input to `assert_equal`, and that function's specification says that in `assert_equal x y`, the output you (as the tester) are expecting to get should be `x`, and the output the function being tested actually produces should be `y`.

Notice how our test suite is accumulating a lot of redundant code. In particular, we had to add the `printer` argument to several lines. Let's improve that code by factoring out a function that constructs test cases:

```
let make_sum_test name expected_output input =
  name >:: (fun _ -> assert_equal expected_output (sum input) ~printer:string_of_int)

let tests = "test suite for sum" >::: [
  make_sum_test "empty" 0 [];
  make_sum_test "singleton" 1 [1];
  make_sum_test "two_elements" 3 [1; 2];
]
```

For output types that are more complicated than integers, you will end up needing to write your own functions to pass to `printer`. This is similar to writing `toString()` methods in Java: for complicated types you invent yourself, the language doesn't know how to render them as strings. You have to provide the code that does it.

3.3.4 Testing for Exceptions

We have a little more of OCaml to learn before we can see how to test for exceptions. You can peek ahead to *the section on exceptions* if you want to know now.

3.3.5 Test-Driven Development

Testing doesn't have to happen strictly after you write code. In *test-driven development* (TDD), testing comes first! It emphasizes *incremental* development of code: there is always something that can be tested. Testing is not something that happens after implementation; instead, *continuous testing* is used to catch errors early. Thus, it is important to develop unit tests immediately when the code is written. Automating test suites is crucial so that continuous testing requires essentially no effort.

Here's an example of TDD. We deliberately choose an exceedingly simple function to implement, so that the process is clear. Suppose we are working with a datatype for days:

```
type day = Sunday | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday
```

And we want to write a function `next_weekday : day -> day` that returns the next weekday after a given day. We start by writing the most basic, broken version of that function we can:

```
let next_weekday d = failwith "Unimplemented"
```

Note: The built-in function `failwith` raises an exception along with the error message passed to the function.

Then we write the simplest unit test we can imagine. For example, we know that the next weekday after Monday is Tuesday. So we add a test:

```
let tests = "test suite for next_weekday" >::: [
  "tue_after_mon"  >:: (fun _ -> assert_equal (next_weekday Monday) Tuesday);
]
```

Then we run the OUnit test suite. It fails, as expected. That's good! Now we have a concrete goal, to make that unit test pass. We revise `next_weekday` to make that happen:

```
let next_weekday d =
  match d with
  | Monday -> Tuesday
  | _ -> failwith "Unimplemented"
```

We compile and run the test; it passes. Time to add some more tests. The simplest remaining possibilities are tests involving just weekdays, rather than weekends. So let's add tests for weekdays.

```
let tests = "test suite for next_weekday" >::: [
  "tue_after_mon"  >:: (fun _ -> assert_equal (next_weekday Monday) Tuesday);
  "wed_after_tue"  >:: (fun _ -> assert_equal (next_weekday Tuesday) Wednesday);
  "thu_after_wed"  >:: (fun _ -> assert_equal (next_weekday Wednesday) Thursday);
  "fri_after_thu"  >:: (fun _ -> assert_equal (next_weekday Thursday) Friday);
]
```

We compile and run the tests; many fail. That's good! We add new functionality:

```
  let next_weekday d =
    match d with
    | Monday -> Tuesday
    | Tuesday -> Wednesday
    | Wednesday -> Thursday
    | Thursday -> Friday
    | _ -> failwith "Unimplemented"
```

We compile and run the tests; they pass. At this point we could move on to handling weekends, but we should first notice something about the tests we've written: they involve repeating a lot of code. In fact, we probably wrote them by copying-and-pasting the first test, then modifying it for the next three. That's a sign that we should *refactor* the code. (As we did before with the `sum` function we were testing.)

Let's abstract a function that creates test cases for `next_weekday`:

```
let make_next_weekday_test name expected_output input=
  name >:: (fun _ -> assert_equal expected_output (next_weekday input))

let tests = "test suite for next_weekday" >::: [
  make_next_weekday_test "tue_after_mon" Tuesday Monday;
  make_next_weekday_test "wed_after_tue" Wednesday Tuesday;
  make_next_weekday_test "thu_after_wed" Thursday Wednesday;
  make_next_weekday_test "fri_after_thu" Friday Thursday;
]
```

Now we finish the testing and implementation by handling weekends. First we add some test cases:

```
  ...
  make_next_weekday_test "mon_after_fri" Monday Friday;
  make_next_weekday_test "mon_after_sat" Monday Saturday;
  make_next_weekday_test "mon_after_sun" Monday Sunday;
  ...
```

Then we finish the function:

```
let next_weekday d =
  match d with
  | Monday -> Tuesday
  | Tuesday -> Wednesday
  | Wednesday -> Thursday
  | Thursday -> Friday
  | Friday -> Monday
  | Saturday -> Monday
  | Sunday -> Monday
```

Of course, most people could write that function without errors even if they didn't use TDD. But rarely do we implement functions that are so simple.

Process. Let's review the process of TDD:

- Write a failing unit test case. Run the test suite to prove that the test case fails.
- Implement just enough functionality to make the test case pass. Run the test suite to prove that the test case passes.
- Improve code as needed. In the example above we refactored the test suite, but often we'll need to refactor the functionality being implemented.
- Repeat until you are satisfied that the test suite provides evidence that your implementation is correct.

3.4 Records and Tuples

Singly-linked lists are a great data structure, but what if you want a fixed number of elements, instead of an unbounded number? Or what if you want the elements to have distinct types? Or what if you want to access the elements by name instead of by number? Lists don't make any of those possibilities easy. Instead, OCaml programmers use records and tuples.

3.4.1 Records

A *record* is a composite of other types of data, each of which is named. OCaml records are much like structs in C. Here's an example of a record type definition `mon` for a Pokémon, re-using the `ptype` definition from the *variants* section:

```
type ptype = TNormal | TFire | TWater
type mon = {name : string; hp : int; ptype : ptype}
```

```
type ptype = TNormal | TFire | TWater
```

```
type mon = { name : string; hp : int; ptype : ptype; }
```

This type defines a record with three *fields* named `name`, `hp` (hit points), and `ptype`. The type of each of those fields is also given. Note that `ptype` can be used as both a type name and a field name; the *namespace* for those is distinct in OCaml.

To build a value of a record type, we write a record expression, which looks like this:

```
{name = "Charmander"; hp = 39; ptype = TFire}
```

```
- : mon = {name = "Charmander"; hp = 39; ptype = TFire}
```

So in a type definition we write a colon between the name and the type of a field, but in an expression we write an equals sign.

To access a record and get a field from it, we use the dot notation that you would expect from many other languages. For example:

```
let c = {name = "Charmander"; hp = 39; ptype = TFire};;
c.hp
```

```
val c : mon = {name = "Charmander"; hp = 39; ptype = TFire}
```

```
- : int = 39
```

It's also possible to use pattern matching to access record fields:

```
match c with {name = n; hp = h; ptype = t} -> h
```

```
- : int = 39
```

The `n`, `h`, and `t` here are pattern variables. There is a syntactic sugar provided if you want to use the same name for both the field and a pattern variable:

```
match c with {name; hp; ptype} -> hp
```

```
- : int = 39
```

Here, the pattern `{name; hp; ptype}` is sugar for `{name = name; hp = hp; ptype = ptype}`. In each of those subexpressions, the identifier appearing on the left-hand side of the equals is a field name, and the identifier appearing on the right-hand side is a pattern variable.

Syntax.

A record expression is written:

```
{f1 = e1; ...; fn = en}
```

The order of the `fi=ei` inside a record expression is irrelevant. For example, `{f = e1; g = e2}` is entirely equivalent to `{g = e2; f = e1}`.

A field access is written:

```
e.f
```

where `f` must be an identifier of a field name, not an expression. That restriction is the same as in any other language with similar features——for example, Java field names. If you really do want to *compute* which identifier to access, then actually you want a different data structure: a *map* (also known by many other names: a *dictionary* or *association list* or *hash table* etc., though there are subtle differences implied by each of those terms.)

Dynamic semantics.

- If for all `i` in `1..n`, it holds that `ei ==> vi`, then `{f1 = e1; ...; fn = en} ==> {f1 = v1; ...; fn = vn}`.
- If `e ==> {...; f = v; ...}` then `e.f ==> v`.

Static semantics.

A record type is written:

```
{f1 : t1; ...; fn : tn}
```

The order of the `fi:ti` inside a record type is irrelevant. For example, `{f : t1; g : t2}` is entirely equivalent to `{g:t2;f:t1}`.

Note that record types must be defined before they can be used. This enables OCaml to do better type inference than would be possible if record types could be used without definition.

The type checking rules are:

- If for all `i` in `1..n`, it holds that `ei : ti`, and if `t` is defined to be `{f1 : t1; ...; fn : tn}`, then `{f1 = e1; ...; fn = en} : t`. Note that the set of fields provided in a record expression must be the full set of fields defined as part of the record's type (but see below regarding *record copy*).
- If `e : t1` and if `t1` is defined to be `{...; f : t2; ...}`, then `e.f : t2`.

Record copy.

Another syntax is also provided to construct a new record out of an old record:

```
{e with f1 = e1; ...; fn = en}
```

This doesn't mutate the old record. Rather, it constructs a new record with new values. The set of fields provided after the `with` does not have to be the full set of fields defined as part of the record's type. In the newly-copied record, any field not provided as part of the `with` is copied from the old record.

The dynamic and static semantics of this are what you might expect, though they are tedious to write down mathematically.

Pattern matching.

We add the following new pattern form to the list of legal patterns:

- `{f1 = p1; ...; fn = pn}`

And we extend the definition of when a pattern matches a value and produces a binding as follows:

- If for all `i` in `1..n`, it holds that `pi` matches `vi` and produces bindings b_i, then the record pattern `{f1 = p1; ...; fn = pn}` matches the record value `{f1 = v1; ...; fn = vn; ...}` and produces the set $\bigcup_i b_i$ of bindings. Note that the record value may have more fields than the record pattern does.

As a syntactic sugar, another form of record pattern is provided: `{f1; ...; fn}`. It is desugared to `{f1 = f1; ...; fn = fn}`.

3.4.2 Tuples

Like records, *tuples* are a composite of other types of data. But instead of naming the *components*, they are identified by position. Here are some examples of tuples:

```
(1, 2, 10)
(true, "Hello")
([1; 2; 3], (0.5, 'X'))
```

A tuple with two components is called a *pair*. A tuple with three components is called a *triple*. Beyond that, we usually just use the word *tuple* instead of continuing a naming scheme based on numbers.

Tip: Beyond about three components, it's arguably better to use records instead of tuples, because it becomes hard for a programmer to remember which component was supposed to represent what information.

Building of tuples is easy: just write the tuple, as above. Accessing again involves pattern matching, for example:

```
match (1, 2, 3) with (x, y, z) -> x + y + z
```

```
- : int = 6
```

Syntax.

A tuple is written

```
(e1, e2, ..., en)
```

The parentheses are not entirely mandatory —often your code can successfully parse without them— but they are usually considered to be good style to include.

Dynamic semantics.

- If for all `i` in `1..n` it holds that `ei ==> vi`, then `(e1, ..., en) ==> (v1, ..., vn)`.

Static semantics.

Tuple types are written using a new type constructor `*`, which is different than the multiplication operator. The type `t1 * ... * tn` is the type of tuples whose first component has type `t1`, ..., and nth component has type `tn`.

- If for all `i` in `1..n` it holds that `ei : ti`, then `(e1, ..., en) : t1 * ... * tn`.

Pattern matching.

We add the following new pattern form to the list of legal patterns:

- `(p1, ..., pn)`

The parentheses are again not entirely mandatory but usually are idimoatic to include.

And we extend the definition of when a pattern matches a value and produces a binding as follows:

- If for all `i` in `1..n`, it holds that `pi` matches `vi` and produces bindings b_i, then the tuple pattern `(p1, ..., pn)` matches the tuple value `(v1, ..., vn)` and produces the set $\bigcup_i b_i$ of bindings. Note that the tuple value must have exactly the same number of components as the tuple pattern does.

3.4.3 Variants vs. Tuples and Records

Note: The second video above uses more advanced examples of variants that will be studied in a *later section*.

The big difference between variants and the types we just learned (records and tuples) is that a value of a variant type is *one of* a set of possibilities, whereas a value of a tuple or record type provides *each of* a set of possibilities. Going back to our examples, a value of type `day` is **one of** `Sun` or `Mon` or etc. But a value of type `mon` provides **each of** a `string` and an `int` and `ptype`. Note how, in those previous two sentences, the word "or" is associated with variant types, and the word "and" is associated with tuple and record types. That's a good clue if you're ever trying to decide whether you want to use a variant, or a tuple or record: if you need one piece of data *or* another, you want a variant; if you need one piece of data *and* another, you want a tuple or record.

One-of types are more commonly known as *sum types*, and each-of types as *product types*. Those names come from set theory. Variants are like disjoint union, because each value of a variant comes from one of many underlying sets (and thus far each of those sets is just a single constructor hence has cardinality one). Disjoint union is indeed sometimes written with a summation operator Σ. Tuples/records are like Cartesian product, because each value of a tuple or record contains a value from each of many underlying sets. Cartesian product is usually written with a product operator, $\times$ or Π.

3.5 Advanced Pattern Matching

Here are some additional pattern forms that are useful:

- `p1 | ... | pn`: an "or" pattern; matching against it succeeds if a match succeeds against any of the individual patterns `pi`, which are tried in order from left to right. All the patterns must bind the same variables.
- `(p : t)`: a pattern with an explicit type annotation.
- `c`: here, `c` means any constant, such as integer literals, string literals, and booleans.
- `'ch1'..'ch2'`: here, `ch` means a character literal. For example, `'A'..'Z'` matches any uppercase letter.
- `p when e`: matches `p` but only if `e` evaluates to `true`.

You can read about all the pattern forms in the manual.

3.5.1 Pattern Matching with Let

The syntax we've been using so far for let expressions is, in fact, a special case of the full syntax that OCaml permits. That syntax is:

```
let p = e1 in e2
```

That is, the left-hand side of the binding may in fact be a pattern, not just an identifier. Of course, variable identifiers are on our list of valid patterns, so that's why the syntax we've studied so far is just a special case.

Given this syntax, we revisit the semantics of let expressions.

Dynamic semantics.

To evaluate `let p = e1 in e2`:

1. Evaluate `e1` to a value `v1`.
2. Match `v1` against pattern `p`. If it doesn't match, raise the exception `Match_failure`. Otherwise, if it does match, it produces a set b of bindings.
3. Substitute those bindings b in `e2`, yielding a new expression `e2'`.
4. Evaluate `e2'` to a value `v2`.
5. The result of evaluating the let expression is `v2`.

Static semantics.

- If all the following hold then `(let p = e1 in e2) : t2`:
  - `e1 : t1`
  - the pattern variables in `p` are `x1..xn`
  - `e2 : t2` under the assumption that for all `i` in `1..n` it holds that `xi : ti`,

Let definitions.

As before, a let definition can be understood as a let expression whose body has not yet been given. So their syntax can be generalized to

```
let p = e
```

and their semantics follow from the semantics of let expressions, as before.

3.5.2 Pattern Matching with Functions

The syntax we've been using so far for functions is also a special case of the full syntax that OCaml permits. That syntax is:

```
let f p1 ... pn = e1 in e2    (* function as part of let expression *)
let f p1 ... pn = e           (* function definition at toplevel *)
fun p1 ... pn -> e            (* anonymous function *)
```

The truly primitive syntactic form we need to care about is `fun p -> e`. Let's revisit the semantics of anonymous functions and their application with that form; the changes to the other forms follow from those below:

Static semantics.

- Let `x1..xn` be the pattern variables appearing in `p`. If by assuming that `x1 : t1` and `x2 : t2` and … and `xn : tn`, we can conclude that `p : t` and `e :u`, then `fun p -> e : t -> u`.

- The type checking rule for application is unchanged.

Dynamic semantics.

- The evaluation rule for anonymous functions is unchanged.
- To evaluate `e0 e1`:
  1. Evaluate `e0` to an anonymous function `fun p -> e`, and evaluate `e1` to value `v1`.
  2. Match `v1` against pattern `p`. If it doesn't match, raise the exception `Match_failure`. Otherwise, if it does match, it produces a set b of bindings.
  3. Substitute those bindings b in `e`, yielding a new expression `e'`.
  4. Evaluate `e'` to a value `v`, which is the result of evaluating `e0 e1`.

3.5.3 Pattern Matching Examples

Here are several ways to get a Pokemon's hit points:

```
(* Pokemon types *)
type ptype = TNormal | TFire | TWater

(* A record to represent Pokemon *)
type mon = { name : string; hp : int; ptype : ptype }

(* OK *)
let get_hp m = match m with { name = n; hp = h; ptype = t } -> h

(* better *)
let get_hp m = match m with { name = _; hp = h; ptype = _ } -> h

(* better *)
let get_hp m = match m with { name; hp; ptype } -> hp

(* better *)
let get_hp m = match m with { hp } -> hp

(* best *)
let get_hp m = m.hp
```

```
type ptype = TNormal | TFire | TWater
```

```
type mon = { name : string; hp : int; ptype : ptype; }
```

```
val get_hp : mon -> int = <fun>
```

```
val get_hp : mon -> int = <fun>
```

```
val get_hp : mon -> int = <fun>
```

```
val get_hp : mon -> int = <fun>
```

```
val get_hp : mon -> int = <fun>
```

Here's how to get the first and second components of a pair:

```
let fst (x, _) = x

let snd (_, y) = y
```

```
val fst : 'a * 'b -> 'a = <fun>
```

```
val snd : 'a * 'b -> 'b = <fun>
```

Both `fst` and `snd` are actually already defined for you in the standard library.

Finally, here are several ways to get the 3rd component of a triple:

```
(* OK *)
let thrd t = match t with x, y, z -> z

(* good *)
let thrd t =
  let x, y, z = t in
  z

(* better *)
let thrd t =
  let _, _, z = t in
  z

(* best *)
let thrd (_, _, z) = z
```

```
val thrd : 'a * 'b * 'c -> 'c = <fun>
```

```
val thrd : 'a * 'b * 'c -> 'c = <fun>
```

```
val thrd : 'a * 'b * 'c -> 'c = <fun>
```

```
val thrd : 'a * 'b * 'c -> 'c = <fun>
```

The standard library does not define any functions for triples, quadruples, etc.

3.6 Type Synonyms

A *type synonym* is a new name for an already existing type. For example, here are some type synonyms that might be useful in representing some types from linear algebra:

```
type point = float * float
type vector = float list
type matrix = float list list
```

```
type point = float * float
```

```
type vector = float list
```

```
type matrix = float list list
```

Anywhere that a `float * float` is expected, you could use `point`, and vice-versa. The two are completely exchangeable for one another. In the following code, `get_x` doesn't care whether you pass it a value that is annotated as one vs. the other:

```
let get_x = fun (x, _) -> x

let p1 : point = (1., 2.)
let p2 : float * float = (1., 3.)

let a = get_x p1
let b = get_x p2
```

```
val get_x : 'a * 'b -> 'a = <fun>
```

```
val p1 : point = (1., 2.)
```

```
val p2 : float * float = (1., 3.)
```

```
val a : float = 1.
```

```
val b : float = 1.
```

Type synonyms are useful because they let us give descriptive names to complex types. They are a way of making code more self-documenting.

3.7 Options

Suppose you want to write a function that *usually* returns a value of type `t`, but *sometimes* returns nothing. For example, you might want to define a function `list_max` that returns the maximum value in a list, but there's not a sensible thing to return on an empty list:

```
let rec list_max = function
  | [] -> ???
  | h :: t -> max h (list_max t)
```

There are a couple possibilities to consider:

- Return `min_int`? But then `list_max` will only work for integers— not floats or other types.
- Raise an exception? But then the user of the function has to remember to catch the exception.
- Return `null`? That works in Java, but by design OCaml does not have a `null` value. That's actually a good thing: null pointer bugs are not fun to debug.

Note: Sir Tony Hoare calls his invention of `null` a "billion-dollar mistake".

In addition to those possibilities, OCaml provides something even better called an *option*. (Haskellers will recognize options as the Maybe monad.)

You can think of an option as being like a closed box. Maybe there's something inside the box, or maybe box is empty. We don't know which until we open the box. If there turns out to be something inside the box when we open it, we can

take that thing out and use it. Thus, options provide a kind of "maybe type," which ultimately is a kind of one-of type: the box is in one of two states, full or empty.

In `list_max` above, we'd like to metaphorically return a box that's empty if the list is empty, or a box that contains the maximum element of the list if the list is non empty.

Here's how we create an option that is like a box with `42` inside it:

```
Some 42
```

```
- : int option = Some 42
```

And here's how we create an option that is like an empty box:

```
None
```

```
- : 'a option = None
```

The `Some` means there's something inside the box, and it's `42`. The `None` means there's nothing inside the box.

As for the types we see above, `t option` is a type for every type `t`, much like `t list` is a type for every type `t`. Values of type `t option` might contain a value of type `t`, or they might contain nothing. `None` has type `'a option` because it's unknown what the type is of the thing inside, as there isn't anything inside.

You can access the contents of an option value `e` using pattern matching. Here's a function that extracts an `int` from an option, if there is one inside, and converts it to a string:

```
let extract o =
  match o with
  | Some i -> string_of_int i
  | None -> "";;
```

```
val extract : int option -> string = <fun>
```

And here are a couple of example usages of that function:

```
extract (Some 42);;
extract None;;
```

```
- : string = "42"
```

```
- : string = ""
```

Here's how we can write `list_max` with options:

```
let rec list_max = function
  | [] -> None
  | h :: t -> begin
      match list_max t with
        | None -> Some h
        | Some m -> Some (max h m)
      end
```

```
val list_max : 'a list -> 'a option = <fun>
```

Tip: The `begin`..`end` wrapping the nested pattern match above is not strictly required here but is not a bad habit, as it will head off potential syntax errors in more complicated code. The keywords `begin` and `end` are equivalent to `(` and `)`.

In Java, every object reference is implicitly an option. Either there is an object inside the reference, or there is nothing there. That "nothing" is represented by the value `null`. Java does not force programmers to explicitly check for the null case, which leads to null pointer exceptions. OCaml options force the programmer to include a branch in the pattern match for `None`, thus guaranteeing that the programmer thinks about the right thing to do when there's nothing there. So we can think of options as a principled way of eliminating `null` from the language. Using options is usually considered better coding practice than raising exceptions, because it forces the caller to do something sensible in the `None` case.

Syntax and semantics of options.

- `t option` is a type for every type `t`.
- `None` is a value of type `'a option`.
- `Some e` is an expression of type `t option` if `e : t`. If `e ==> v` then `Some e ==> Some v`

3.8 Association Lists

A *dictionary* is a data structure that maps *keys* to *values*. One easy implementation of a dictionary is an *association list*, which is a list of pairs. Here, for example, is an association list that maps some shape names to the number of sides they have:

```
let d = [("rectangle", 4); ("nonagon", 9); ("icosagon", 20)]
```

```
val d : (string * int) list =
  [("rectangle", 4); ("nonagon", 9); ("icosagon", 20)]
```

Note that an association list isn't so much a built-in data type in OCaml as a combination of two other types: lists and pairs.

Here are two functions that implement insertion and lookup in an association list:

```
(** [insert k v lst] is an association list that binds key [k] to value [v]
    and otherwise is the same as [lst] *)
let insert k v lst = (k, v) :: lst

(** [find k lst] is [Some v] if association list [lst] binds key [k] to
    value [v]; and is [None] if [lst] does not bind [k]. *)
let rec lookup k = function
| [] -> None
| (k', v) :: t -> if k = k' then Some v else lookup k t
```

```
val insert : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
```

```
val lookup : 'a -> ('a * 'b) list -> 'b option = <fun>
```

The `insert` function simply adds a new map from a key to a value at the front of the list. It doesn't bother to check whether the key is already in the list. The `lookup` function looks through the list from left to right. So if there did happen to be multiple maps for a given key in the list, only the most recently inserted one would be returned.

Insertion in an association list is therefore constant time, and lookup is linear time. Although there are certainly more efficient implementations of dictionaries—and we'll study some later in this course—association lists are a very easy and

useful implementation for small dictionaries that aren't performance critical. The OCaml standard library has functions for association lists in the `List` module; look for `List.assoc` and the functions below it in the documentation. What we just wrote as `lookup` is actually already defined as `List.assoc_opt`. There is no pre-defined `insert` function in the library because it's so trivial just to cons a pair on.

3.9 Algebraic Data Types

Thus far, we have seen variants simply as enumerating a set of constant values, such as:

```
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

type ptype = TNormal | TFire | TWater

type peff = ENormal | ENotVery | Esuper
```

But variants are far more powerful that this.

3.9.1 Variants that Carry Data

As a running example, here is a variant type `shape` that does more than just enumerate values:

```
type point = float * float
type shape =
  | Point of point
  | Circle of point * float (* center and radius *)
  | Rect of point * point (* lower-left and upper-right corners *)
```

```
type point = float * float
```

```
type shape = Point of point | Circle of point * float | Rect of point * point
```

This type, `shape`, represents a shape that is either a point, a circle, or a rectangle. A point is represented by a constructor `Point` that *carries* some additional data, which is a value of type `point`. A circle is represented by a constructor `Circle` that carries a pair of type `point * float`, which according to the comment represents the center of the circle and its radius. A rectangle is represented by a constructor `Rect` that carries a pair of type `point*point`.

Here are a couple functions that use the `shape` type:

```
let area = function
  | Point _ -> 0.0
  | Circle (_, r) -> Float.pi *. (r ** 2.0)
  | Rect ((x1, y1), (x2, y2)) ->
      let w = x2 -. x1 in
      let h = y2 -. y1 in
      w *. h

let center = function
  | Point p -> p
  | Circle (p, _) -> p
  | Rect ((x1, y1), (x2, y2)) -> ((x2 +. x1) /. 2.0, (y2 +. y1) /. 2.0)
```

```
val area : shape -> float = <fun>
```

```
val center : shape -> point = <fun>
```

The `shape` variant type is the same as those we've seen before in that it is defined in terms of a collection of constructors. What's different than before is that those constructors carry additional data along with them. Every value of type `shape` is formed from exactly one of those constructors. Sometimes we call the constructor a *tag*, because it tags the data it carries as being from that particular constructor.

Variant types are sometimes called *tagged unions*. Every value of the type is from the set of values that is the union of all values from the underlying types that the constructor carries. For the `shape` type, every value is tagged with either `Point` or `Circle` or `Rect` and carries a value from the set of all `point` valued unioned with the set of all `point * float` values unioned with the set of all `point * point` values.

Another name for these variant types is an *algebraic data type*. "Algebra" here refers to the fact that variant types contain both sum and product types, as defined in the previous lecture. The sum types come from the fact that a value of a variant is formed by *one of* the constructors. The product types come from that fact that a constructor can carry tuples or records, whose values have a sub-value from *each of* their component types.

Using variants, we can express a type that represents the union of several other types, but in a type-safe way. Here, for example, is a type that represents either a `string` or an `int`:

```
type string_or_int =
  | String of string
  | Int of int
```

```
type string_or_int = String of string | Int of int
```

If we wanted to, we could use this type to code up lists (e.g.) that contain either strings or ints:

```
type string_or_int_list = string_or_int list

let rec sum : string_or_int list -> int = function
  | [] -> 0
  | String s :: t -> int_of_string s + sum t
  | Int i :: t -> i + sum t

let lst_sum = sum [String "1"; Int 2]
```

```
type string_or_int_list = string_or_int list
```

```
val sum : string_or_int list -> int = <fun>
```

```
val lst_sum : int = 3
```

Variants thus provide a type-safe way of doing something that might before have seemed impossible.

Variants also make it possible to discriminate which tag a value was constructed with, even if multiple constructors carry the same type. For example:

```
type t = Left of int | Right of int
let x = Left 1
let double_right = function
  | Left i -> i
  | Right i -> 2 * i
```

```
type t = Left of int | Right of int
```

```
val x : t = Left 1
```

```
val double_right : t -> int = <fun>
```

3.9.2 Syntax and Semantics

Syntax.

To define a variant type:

```
type t = C1 [of t1] | ... | Cn [of tn]
```

The square brackets above denote that `of ti` is optional. Every constructor may individually either carry no data or carry data. We call constructors that carry no data *constant*; and those that carry data, *non-constant*.

To write an expression that is a variant:

```
C e
```

Or:

```
C
```

depending on whether the constructor name `C` is non-constant or constant.

Dynamic semantics.

- If `e==>v` then `C e ==> C v`, assuming `C` is non-constant.
- `C` is already a value, assuming `C` is constant.

Static semantics.

- If `t = ... | C | ...` then `C : t`.
- If `t = ... | C of t' | ...` and if `e : t'` then `C e : t`.

Pattern matching.

We add the following new pattern form to the list of legal patterns:

- `C p`

And we extend the definition of when a pattern matches a value and produces a binding as follows:

- If `p` matches `v` and produces bindings b, then `C p` matches `C v` and produces bindings b.

3.9.3 Catch-all Cases

One thing to beware of when pattern matching against variants is what *Real World OCaml* calls "catch-all cases". Here's a simple example of what can go wrong. Let's suppose you write this variant and function:

```
type color = Blue | Red

(* a thousand lines of code in between *)

let string_of_color = function
  | Blue -> "blue"
  | _ -> "red"
```

```
type color = Blue | Red
```

```
val string_of_color : color -> string = <fun>
```

Seems fine, right? But then one day you realize there are more colors in the world. You need to represent green. So you go back and add green to your variant:

```
type color = Blue | Red | Green

(* a thousand lines of code in between *)

let string_of_color = function
  | Blue -> "blue"
  | _ -> "red"
```

```
type color = Blue | Red | Green
```

```
val string_of_color : color -> string = <fun>
```

But because of the thousand lines of code in between, you forget that `string_of_color` needs updating. And now, all the sudden, you are red-green color blind:

```
string_of_color Green
```

```
- : string = "red"
```

The problem is the *catch-all* case in the pattern match inside `string_of_color`: the final case that uses the wildcard pattern to match anything. Such code is not robust against future changes to the variant type.

If, instead, you had originally coded the function as follows, life would be better:

```
let string_of_color = function
  | Blue -> "blue"
  | Red  -> "red"
```

```
File "[9]", lines 1-3, characters 22-17:
```

```
1 | ......................function
```

```
2 |   | Blue -> "blue"
```

```
3 |   | Red  -> "red"
```

```
Warning 8: this pattern-matching is not exhaustive.
```

```
Here is an example of a case that is not matched:
```

```
Green
```

```
val string_of_color : color -> string = <fun>
```

The OCaml type checker now alerts you that you haven't yet updated `string_of_color` to account for the new constructor.

The moral of the story is: catch-all cases lead to buggy code. Avoid using them.

3.9.4 Recursive Variants

Variant types may mention their own name inside their own body. For example, here is a variant type that could be used to represent something similar to `int list`:

```
type intlist = Nil | Cons of int * intlist

let lst3 = Cons (3, Nil)   (* similar to 3 :: [] or [3]*)
let lst123 = Cons(1, Cons(2, lst3)) (* similar to [1; 2; 3] *)

let rec sum (l : intlist) : int=
  match l with
  | Nil -> 0
  | Cons (h, t) -> h + sum t

let rec length : intlist -> int = function
  | Nil -> 0
  | Cons (_, t) -> 1 + length t

let empty : intlist -> bool = function
  | Nil -> true
  | Cons _ -> false
```

```
type intlist = Nil | Cons of int * intlist
```

```
val lst3 : intlist = Cons (3, Nil)
```

```
val lst123 : intlist = Cons (1, Cons (2, Cons (3, Nil)))
```

```
val sum : intlist -> int = <fun>
```

```
val length : intlist -> int = <fun>
```

```
val empty : intlist -> bool = <fun>
```

Notice that in the definition of `intlist`, we define the `Cons` constructor to carry a value that contains an `intlist`. This makes the type `intlist` be *recursive*: it is defined in terms of itself.

Types may be mutually recursive if you use the `and` keyword:

```
type node = {value : int; next : mylist}
and mylist = Nil | Node of node
```

```
type node = { value : int; next : mylist; }
and mylist = Nil | Node of node
```

Any such mutual recursion must involve at least one variant or record type that the recursion "goes through". For example, the following is not allowed:

```
type t = u and u = t
```

```
File "[12]", line 1, characters 0-10:
1 | type t = u and u = t
    ^^^^^^^^^^
Error: The definition of t contains a cycle:
       u
```

But this is:

```
type t = U of u and u = T of t
```

```
type t = U of u
and u = T of t
```

Record types may also be recursive:

```
type node = {value : int; next : node}
```

```
type node = { value : int; next : node; }
```

But plain old type synonyms may not be:

```
type t = t * t
```

```
File "[15]", line 1, characters 0-14:
1 | type t = t * t
    ^^^^^^^^^^^^^^
Error: The type abbreviation t is cyclic
```

Although `node` is a legal type definition, there is no way to construct a value of that type because of the circularity involved: to construct the very first `node` value in existence, you would already need a value of type `node` to exist. Later, when we cover imperative features, we'll see a similar idea used (but successfully) for mutable linked lists.

3.9.5 Parameterized Variants

Variant types may be *parameterized* on other types. For example, the `intlist` type above could be generalized to provide lists (coded up ourselves) over any type:

```
type 'a mylist = Nil | Cons of 'a * 'a mylist

let lst3 = Cons (3, Nil)   (* similar to [3] *)
let lst_hi = Cons ("hi", Nil)   (* similar to ["hi"] *)
```

```
type 'a mylist = Nil | Cons of 'a * 'a mylist
```

```
val lst3 : int mylist = Cons (3, Nil)
```

```
val lst_hi : string mylist = Cons ("hi", Nil)
```

Here, `mylist` is a *type constructor* but not a type: there is no way to write a value of type `mylist`. But we can write value of type `int mylist` (e.g., `lst3`) and `string mylist` (e.g., `lst_hi`). Think of a type constructor as being like a function, but one that maps types to types, rather than values to value.

Here are some functions over `'a mylist`:

```
let rec length : 'a mylist -> int = function
  | Nil -> 0
  | Cons (_, t) -> 1 + length t

let empty : 'a mylist -> bool = function
  | Nil -> true
  | Cons _ -> false
```

```
val length : 'a mylist -> int = <fun>
```

```
val empty : 'a mylist -> bool = <fun>
```

Notice that the body of each function is unchanged from its previous definition for `intlist`. All that we changed was the type annotation. And that could even be omitted safely:

```
let rec length = function
  | Nil -> 0
  | Cons (_, t) -> 1 + length t

let empty = function
  | Nil -> true
  | Cons _ -> false
```

```
val length : 'a mylist -> int = <fun>
```

```
val empty : 'a mylist -> bool = <fun>
```

The functions we just wrote are an example of a language feature called **parametric polymorphism**. The functions don't care what the `'a` is in `'a mylist`, hence they are perfectly happy to work on `int mylist` or `string mylist` or any other `(whatever) mylist`. The word "polymorphism" is based on the Greek roots "poly" (many) and "morph" (form). A value of type `'a mylist` could have many forms, depending on the actual type `'a`.

As soon, though, as you place a constraint on what the type `'a` might be, you give up some polymorphism. For example,

```
let rec sum = function
  | Nil -> 0
  | Cons (h, t) -> h + sum t
```

```
val sum : int mylist -> int = <fun>
```

The fact that we use the `( + )` operator with the head of the list constrains that head element to be an `int`, hence all elements must be `int`. That means `sum` must take in an `int mylist`, not any other kind of `'a mylist`.

It is also possible to have multiple type parameters for a parameterized type, in which case parentheses are needed:

```
type ('a, 'b) pair = {first : 'a; second : 'b}
let x = {first = 2; second = "hello"}
```

```
type ('a, 'b) pair = { first : 'a; second : 'b; }
```

```
val x : (int, string) pair = {first = 2; second = "hello"}
```

3.9.6 Polymorphic Variants

Thus far, whenever you've wanted to define a variant type, you have had to give it a name, such as `day`, `shape`, or `'a mylist`:

```
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

type shape =
  | Point of point
  | Circle of point * float
  | Rect of point * point

type 'a mylist = Nil | Cons of 'a * 'a mylist
```

```
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

```
type shape = Point of point | Circle of point * float | Rect of point * point
```

```
type 'a mylist = Nil | Cons of 'a * 'a mylist
```

Occasionally, you might need a variant type only for the return value of a single function. For example, here's a function `f` that can either return an `int` or ∞; you are forced to define a variant type to represent that result:

```
type fin_or_inf = Finite of int | Infinity

let f = function
  | 0 -> Infinity
  | 1 -> Finite 1
  | n -> Finite (-n)
```

```
type fin_or_inf = Finite of int | Infinity
```

```
val f : int -> fin_or_inf = <fun>
```

The downside of this definition is that you were forced to defined `fin_or_inf` even though it won't be used throughout much of your program.

There's another kind of variant in OCaml that supports this kind of programming: *polymorphic variants*. Polymorphic variants are just like variants, except:

1. You don't have declare their type or constructors before using them.
2. There is no name for a polymorphic variant type. (So another name for this feature could have been "anonymous variants".)
3. The constructors of a polymorphic variant start with a backquote character.

Using polymorphic variants, we can rewrite `f`:

```
let f = function
  | 0 -> `Infinity
  | 1 -> `Finite 1
  | n -> `Finite (-n)
```

```
val f : int -> [> `Finite of int | `Infinity ] = <fun>
```

This type says that `f` either returns `` `Finite n `` for some `n : int` or `` `Infinity ``. The square brackets do not denote a list, but rather a set of possible constructors. The > sign means that any code that pattern matches against a value of that type must *at least* handle the constructors `` `Finite `` and `` `Infinity ``, and possibly more. For example, we could write:

```
match f 3 with
  | `NegInfinity -> "negative infinity"
  | `Finite n -> "finite"
  | `Infinity -> "infinite"
```

```
- : string = "finite"
```

It's perfectly fine for the pattern match to include constructors other than `` `Finite `` or `` `Infinity ``, because `f` is guaranteed never to return any constructors other than those.

There are other, more compelling uses for polymorphic variants that we'll see later in the course. They are particularly useful in libraries. For now, we generally will steer you away from extensive use of polymorphic variants, because their types can become difficult to manage.

3.9.7 Built-in Variants

OCaml's built-in list data type is really a recursive, parameterized variant. It is defined as follows:

```
type 'a list = [] | ( :: ) of 'a * 'a list
```

So `list` is really just a type constructor, with (value) constructors `[]` (which we pronounce "nil") and `::` (which we pronounce "cons").

OCaml's built-in option data type is also really a parameterized variant. It's defined as follows:

```
type 'a option = None | Some of 'a
```

So `option` is really just a type constructor, with (value) constructors `None` and `Some`.

You can see both `list` and `option` defined in the core OCaml library.

3.10 Exceptions

OCaml has an exception mechanism similar to many other programming languages. A new type of OCaml exception is defined with this syntax:

```
exception E of t
```

where `E` is a constructor name and `t` is a type. The `of t` is optional. Notice how this is similar to defining a constructor of a variant type. For example:

```
exception A
exception B
exception Code of int
exception Details of string
```

```
exception A
```

```
exception B
```

```
exception Code of int
```

```
exception Details of string
```

To create an exception value, use the same syntax you would for creating a variant value. Here, for example, is an exception value whose constructor is `Failure`, which carries a `string`:

```
Failure "something went wrong"
```

```
- : exn = Failure "something went wrong"
```

This constructor is pre-defined in the standard library (scroll down to "predefined exceptions") and is one of the more common exceptions that OCaml programmers use.

To raise an exception value `e`, simply write

```
raise e
```

There is a convenient function `failwith : string -> 'a` in the standard library that raises `Failure`. That is, `failwith s` is equivalent to `raise (Failure s)`.

To catch an exception, use this syntax:

```
try e with
| p1 -> e1
| ...
| pn -> en
```

The expression `e` is what might raise an exception. If it does not, the entire `try` expression evaluates to whatever `e` does. If `e` does raise an exception value `v`, that value `v` is that matched against the provide patterns, exactly like `match` expression.

3.10.1 Exceptions are Extensible Variants

All exception values have type `exn`, which is a variant defined in the core. It's an unusual kind of variant, though, called an *extensible* variant, which allows new constructors of the variant to be defined after the variant type itself is defined. See the OCaml manual for more information about extensible variants if you're interested.

3.10.2 Exception Semantics

Since they are just variants, the syntax and semantics of exceptions is already covered by the syntax and semantics of variants—with one exception (pun intended), which is the dynamic semantics of how exceptions are raised and handled.

Dynamic semantics. As we originally said, every OCaml expression either

- evaluates to a value
- raises an exception
- or fails to terminate (i.e., an "infinite loop").

So far we've only presented the part of the dynamic semantics that handles the first of those three cases. What happens when we add exceptions? Now, evaluation of an expression either produces a value or produces an *exception packet*. Packets are not normal OCaml values; the only pieces of the language that recognizes them are `raise` and `try`. The

exception value produced by (e.g.) `Failure "oops"` is part of the exception packet produced by `raise (Failure "oops")`, but the packet contains more than just the exception value; there can also be a stack trace, for example.

For any expression `e` other than `try`, if evaluation of a subexpression of `e` produces an exception packet `P`, then evaluation of `e` produces packet `P`.

But now we run into a problem for the first time: what order are subexpressions evaluated in? Sometimes the answer to that question is provided by the semantics we have already developed. For example, with let expressions, we know that the binding expression must be evaluated before the body expression. So the following code raises `A`:

```
let _ = raise A in raise B;;
```

```
Exception: A.
Called from Toploop.load_lambda in file "toplevel/toploop.ml", line 212, characters␣
↪17-27
```

And with functions, the argument must be evaluated before the function. So the following code also raises `A`, in addition to producing some compiler warnings that the first expression will never actually be applied as a function to an argument:

```
(raise B) (raise A)
```

It makes sense that both those pieces of code would raise the same exception, given that we know `let x = e1 in e2` is syntactic sugar for `(fun x -> e2) e1`.

But what does the following code raise as an exception?

```
(raise A, raise B)
```

The answer is nuanced. The language specification does not stipulate what order the components of pairs should be evaluated in. Nor did our semantics exactly determine the order. (Though you would be forgiven if you thought it was left to right.) So programmers actually cannot rely on that order. The current implementation of OCaml, as it turns out, evaluates right to left. So the code above actually raises `B`. If you really want to force the evaluation order, you need to use let expressions:

```
let a = raise A in
let b = raise B in
(a, b)
```

```
Exception: A.
Called from Toploop.load_lambda in file "toplevel/toploop.ml", line 212, characters␣
↪17-27
```

That code is guaranteed to raise `A` rather than `B`.

One interesting corner case is what happens when a raise expression itself has a subexpression that raises:

```
exception C of string;;
exception D of string;;
raise (C (raise (D "oops")))
```

```
exception C of string
```

```
exception D of string
```

```
Exception: D "oops".
Called from Toploop.load_lambda in file "toplevel/toploop.ml", line 212, characters␣
↪17-27
```

That code ends up raising `D`, because the first thing that has to happen is to evaluate `C (raise (D "oops"))` to a value. Doing that requires evaluating `raise (D "oops")` to a value. Doing that causes a packet containing `D "oops"` to be produced, and that packet then propagates and becomes the result of evaluating `C (raise (D "oops"))`, hence the result of evaluating `raise (C (raise (D "oops")))`.

Once evaluation of an expression produces an exception packet `P`, that packet propagates until it reaches a `try` expression:

```
try e with
| p1 -> e1
| ...
| pn -> en
```

The exception value inside `P` is matched against the provided patterns using the usual evaluation rules for pattern matching—with one exception (again, pun intended). If none of the patterns matches, then instead of producing `Match_failure` inside a new exception packet, the original exception packet `P` continues propagating until the next `try` expression is reached.

3.10.3 Pattern Matching

There is a pattern form for exceptions. Here's an example of its usage:

```
match List.hd [] with
  | [] -> "empty"
  | _ :: _ -> "nonempty"
  | exception (Failure s) -> s
```

```
- : string = "hd"
```

Note that the code is above is just a standard `match` expression, not a `try` expression. It matches the value of `List.hd []` against the three provided patterns. As we know, `List.hd []` will raise an exception containing the value `Failure "hd"`. The *exception pattern* `exception (Failure s)` matches that value. So the above code will evaluate to `"hd"`.

In general, exception patterns are a kind of syntactic sugar. Consider this code:

```
match e with
  | p1 -> e1
  | ...
  | pn -> en
```

Some of the patterns `p1..pn` could be exception patterns of the form `exception q`. Let `q1..qn` be that subsequence of patterns (without the `exception` keyword), and let `r1..rm` be the subsequence of non-exception patterns. Then we can rewrite the code as:

```
match
  try e with
    | q1 -> e1
    | ...
    | qn -> en
with
  | r1 -> e1
  | ...
  | rm -> em
```

Which is to say: try evaluating `e`. If it produces an exception packet, use the exception patterns from the original match expression to handle that packet. If it doesn't produce an exception packet but instead produces a non-exception value, use the non-exception patterns from the original match expression to match that value.

3.10.4 Exceptions and OUnit

If it is part of a function's specification that it raises an exception, you might want to write OUnit tests that check whether the function correctly does so. Here's how to do that:

```
open OUnit2

let tests = "suite" >::: [
    "empty" >:: (fun _ -> assert_raises (Failure "hd") (fun () -> List.hd []));
  ]

let _ = run_test_tt_main tests
```

The expression `assert_raises exc (fun () -> e)` checks to see whether expression `e` raises exception `exc`. If so, the OUnit test case succeeds, otherwise it fails.

Tip: A common error is to forget the `(fun () -> ...)` around `e`. If you do, the OUnit test case will fail, and you will likely be confused as to why. The reason is that, without the extra anonymous function, the exception is raised before `assert_raises` ever gets a chance to handle it.

3.11 Example: Trees

Trees are a very useful data structure. A *binary tree*, as you'll recall from CS 2110, is a node containing a value and two children that are trees. A binary tree can also be an empty tree, which we also use to represent the absence of a child node.

3.11.1 Representation with Tuples

Here is a definition for a binary tree data type:

```
type 'a tree =
| Leaf
| Node of 'a * 'a tree * 'a tree
```

```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree
```

A node carries a data item of type `'a` and has a left and right subtree. A leaf is empty. Compare this definition to the definition of a list and notice how similar their structure is:

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

```
type 'a mylist =
  | Nil
  | Cons of 'a * 'a mylist
```

The only essential difference is that `Cons` carries one sublist, whereas `Node` carries two subtrees.

Here is code that constructs a small tree:

```
(* the code below constructs this tree:
         4
       /   \
      2     5
```

(continues on next page)

(continued from previous page)

```
     / \   / \
    1   3 6   7
*)
let t =
  Node(4,
    Node(2,
      Node(1, Leaf, Leaf),
      Node(3, Leaf, Leaf)
    ),
    Node(5,
      Node(6, Leaf, Leaf),
      Node(7, Leaf, Leaf)
    )
  )
```

```
val t : int tree =
  Node (4, Node (2, Node (1, Leaf, Leaf), Node (3, Leaf, Leaf)),
   Node (5, Node (6, Leaf, Leaf), Node (7, Leaf, Leaf)))
```

The *size* of a tree is the number of nodes in it (that is, `Node`s, not `Leaf`s). For example, the size of tree `t` above is 7. Here is a function function `size : 'a tree -> int` that returns the number of nodes in a tree:

```
let rec size = function
  | Leaf -> 0
  | Node (_, l, r) -> 1 + size l + size r
```

3.11.2 Representation with Records

Next, let's revise our tree type to use use a record type to represent a tree node. In OCaml we have to define two mutually recursive types, one to represent a tree node, and one to represent a (possibly empty) tree:

```
type 'a tree =
  | Leaf
  | Node of 'a node

and 'a node = {
  value: 'a;
  left: 'a tree;
  right: 'a tree
}
```

```
type 'a tree = Leaf | Node of 'a node
and 'a node = { value : 'a; left : 'a tree; right : 'a tree; }
```

Here's an example tree:

```
(* represents
     2
    / \
   1   3  *)
let t =
  Node {
    value = 2;
    left = Node {value = 1; left = Leaf; right = Leaf};
```

(continues on next page)

(continued from previous page)

```
    right = Node {value = 3; left = Leaf; right = Leaf}
  }
```

```
val t : int tree =
  Node
   {value = 2; left = Node {value = 1; left = Leaf; right = Leaf};
    right = Node {value = 3; left = Leaf; right = Leaf}}
```

We can use pattern matching to write the usual algorithms for recursively traversing trees. For example, here is a recursive search over the tree:

```
(** [mem x t] is whether [x] is a value at some node in tree [t]. *)
let rec mem x = function
  | Leaf -> false
  | Node {value; left; right} -> value = x || mem x left || mem x right
```

```
val mem : 'a -> 'a tree -> bool = <fun>
```

The function name `mem` is short for "member"; the standard library often uses a function of this name to implement a search through a collection data structure to determine whether some element is a member of that collection.

Here's a function that computes the *preorder* traversal of a tree, in which each node is visited before any of its children, by constructing a list in which the values occur in the order in which they would be visited:

```
let rec preorder = function
  | Leaf -> []
  | Node {value; left; right} -> [value] @ preorder left @ preorder right
```

```
val preorder : 'a tree -> 'a list = <fun>
```

```
preorder t
```

```
- : int list = [2; 1; 3]
```

Although the algorithm is beautifully clear from the code above, it takes quadratic time on unbalanced trees because of the `@` operator. That problem can be solved by introducing an extra argument `acc` to accumulate the values at each node, though at the expense of making the code less clear:

```
let preorder_lin t =
  let rec pre_acc acc = function
    | Leaf -> acc
    | Node {value; left; right} -> value :: (pre_acc (pre_acc acc right) left)
  in pre_acc [] t
```

```
val preorder_lin : 'a tree -> 'a list = <fun>
```

The version above uses exactly one `::` operation per `Node` in the tree, making it linear time.

3.12 Example: Natural Numbers

We can define a recursive variant that acts like numbers, demonstrating that we don't really have to have numbers built into OCaml! (For sake of efficiency, though, it's a good thing they are.)

A *natural number* is either *zero* or the *successor* of some other natural number. This is how you might define the natural numbers in a mathematical logic course, and it leads naturally to the following OCaml type `nat`:

```
type nat = Zero | Succ of nat
```

```
type nat = Zero | Succ of nat
```

We have defined a new type `nat`, and `Zero` and `Succ` are constructors for values of this type. This allows us to build expressions that have an arbitrary number of nested `Succ` constructors. Such values act like natural numbers:

```
let zero = Zero
let one = Succ zero
let two = Succ one
let three = Succ two
let four = Succ three
```

```
val zero : nat = Zero
```

```
val one : nat = Succ Zero
```

```
val two : nat = Succ (Succ Zero)
```

```
val three : nat = Succ (Succ (Succ Zero))
```

```
val four : nat = Succ (Succ (Succ (Succ Zero)))
```

Now we can write functions to manipulate values of this type. We'll write a lot of type annotations in the code below to help the reader keep track of which values are `nat` versus `int`; the compiler, of course, doesn't need our help.

```
let iszero = function
  | Zero -> true
  | Succ _ -> false

let pred = function
  | Zero -> failwith "pred Zero is undefined"
  | Succ m -> m
```

```
val iszero : nat -> bool = <fun>
```

```
val pred : nat -> nat = <fun>
```

Similarly we can define a function to add two numbers:

```
let rec add n1 n2 =
  match n1 with
  | Zero -> n2
  | Succ pred_n -> add pred_n (Succ n2)
```

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

We can convert `nat` values to type `int` and vice-versa:

```
let rec int_of_nat = function
  | Zero -> 0
  | Succ m -> 1 + int_of_nat m

let rec nat_of_int = function
  | i when i = 0 -> Zero
  | i when i > 0 -> Succ (nat_of_int (i - 1))
  | _ -> failwith "nat_of_int is undefined on negative ints"
```

```
val int_of_nat : nat -> int = <fun>
```

```
val nat_of_int : int -> nat = <fun>
```

To determine whether a natural number is even or odd, we can write a pair of mutually recursive functions:

```
let rec even = function Zero -> true | Succ m -> odd m
and odd = function Zero -> false | Succ m -> even m
```

```
val even : nat -> bool = <fun>
val odd : nat -> bool = <fun>
```

3.13 Summary

Lists are a highly useful built-in data structure in OCaml. The language provides a lightweight syntax for building them, rather than requiring you to use a library. Accessing parts of a list makes use of pattern matching, a very powerful feature (as you might expect from its rather lengthy semantics). We'll see more uses for pattern matching as the course proceeds.

These built-in lists are implemented as singly-linked lists. That's important to keep in mind when your needs go beyond small to medium sized lists. Recursive functions on long lists will take up a lot of stack space, so tail recursion becomes important. And if you're attempting to process really huge lists, you probably don't want linked lists at all, but instead a data structure that will do a better job of exploiting memory locality.

OCaml provides data types for variants (one-of types), tuples and products (each-of types), and options (maybe types). Pattern matching can be used to access values of each of those data types. And pattern matching can be used in let expressions and functions.

Association lists combine lists and tuples to create a lightweight implementation of dictionaries.

Variants are a powerful language feature. They are the workhorse of representing data in a functional language. OCaml variants actually combine several theoretically independent language features into one: sum types, product types, recursive types, and parameterized (polymorphic) types. The result is an ability to express many kinds of data, including lists, options, trees, and even exceptions.

3.13.1 Terms and Concepts

- algebraic data type
- append
- association list
- binary trees as variants
- binding
- branch
- carried data
- catch-all cases
- cons
- constant constructor
- constructor
- copying
- desugaring
- each-of type
- exception
- exception as variants
- exception packet
- exception pattern
- exception value
- exhaustiveness
- field
- head
- induction
- leaf
- list
- lists as variants
- maybe type
- mutually recursive functions
- natural numbers as variants
- nil
- node
- non-constant constructor
- one-of type
- options
- options as variants

- order of evaluation
- pair
- parameterized variant
- parametric polymorphism
- pattern matching
- prepend
- product type
- record
- recursion
- recursive variant
- sharing
- stack frame
- sum type
- syntactic sugar
- tag
- tail
- tail call
- tail recursion
- test-driven development (TDD)
- triple
- tuple
- type constructor
- type constructor
- type synonym
- variant
- wildcard

3.13.2 Further Reading

- *Introduction to Objective Caml*, chapters 4, 5.2, 5.3, 5.4, 6, 7, 8.1
- *OCaml from the Very Beginning*, chapters 3, 4, 5, 7, 8, 10, 11
- *Real World OCaml*, chapter 3, 5, 6, 7

3.14 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: list expressions [★]

- Construct a list that has the integers 1 through 5 in it. Use the square bracket notation for lists.
- Construct the same list, but do not use the square bracket notation. Instead use `::` and `[]`.
- Construct the same list again. This time, the following expression must appear in your answer: `[2; 3; 4]`. Use the `@` operator, and do not use `::`.

Exercise: product [★★]

Write a function that returns the product of all the elements in a list. The product of all the elements of an empty list is `1`.

Exercise: concat [★★]

Write a function that concatenates all the strings in a list. The concatenation of all the strings in an empty list is the empty string `""`.

Exercise: product test [★★]

Unit test the function `product` that you wrote in an exercise above.

Exercise: patterns [★★★]

Using pattern matching, write three functions, one for each of the following properties. Your functions should return `true` if the input list has the property and `false` otherwise.

- the list's first element is `"bigred"`
- the list has exactly two or four elements; do not use the `length` function
- the first two elements of the list are equal

Exercise: library [★★★]

Consult the `List` standard library to solve these exercises:

- Write a function that takes an `int list` and returns the fifth element of that list, if such an element exists. If the list has fewer than five elements, return `0`. *Hint:* `List.length` *and* `List.nth`.
- Write a function that takes an `int list` and returns the list sorted in descending order. *Hint:* `List.sort` *with* `Stdlib.compare` *as its first argument, and* `List.rev`.

Exercise: library test [★★★]

Write a couple OUnit unit tests for each of the functions you wrote in the previous exercise.

Exercise: library puzzle [★★★]

- Write a function that returns the last element of a list. Your function may assume that the list is non-empty. *Hint: Use two library functions, and do not write any pattern matching code of your own.*
- Write a function `any_zeroes : int list -> bool` that returns `true` if and only if the input list contains at least one `0`. *Hint: use one library function, and do not write any pattern matching code of your own.*

Your solutions will be only one or two lines of code each.

Exercise: take drop [★★★]

- Write a function `take : int -> 'a list -> 'a list` such that `take n lst` returns the first `n` elements of `lst`. If `lst` has fewer than `n` elements, return all of them.
- Write a function `drop : int -> 'a list -> 'a list` such that `drop n lst` returns all but the first `n` elements of `lst`. If `lst` has fewer than `n` elements, return the empty list.

Exercise: take drop tail [★★★★]

Revise your solutions for `take` and `drop` to be tail recursive, if they aren't already. Test them on long lists with large values of `n` to see whether they run out of stack space. To construct long lists, use the `--` operator from the *lists* section.

Exercise: unimodal [★★★]

Write a function `is_unimodal : int list -> bool` that takes an integer list and returns whether that list is unimodal. A *unimodal list* is a list that monotonically increases to some maximum value then monotonically decreases after that value. Either or both segments (increasing or decreasing) may be empty. A constant list is unimodal, as is the empty list.

Exercise: powerset [★★★]

Write a function `powerset : int list -> int list list` that takes a set S represented as a list and returns the set of all subsets of S. The order of subsets in the powerset and the order of elements in the subsets do not matter.

Hint: Consider the recursive structure of this problem. Suppose you already have `p`, such that `p = powerset s`. How could you use `p` to compute `powerset (x :: s)`?

Exercise: print int list rec [★★]

Write a function `print_int_list : int list -> unit` that prints its input list, one number per line. For example, `print_int_list [1; 2; 3]` should result in this output:

```
1
2
3
```

Here is some code to get you started:

```
let rec print_int_list = function
| [] -> ()
| h :: t -> (* fill in here *); print_int_list t
```

Exercise: print int list iter [★★]

Write a function `print_int_list' : int list -> unit` whose specification is the same as `print_int_list`. Do not use the keyword `rec` in your solution, but instead to use the List module function `List.iter`. Here is some code to get you started:

```
let print_int_list' lst =
  List.iter (fun x -> (* fill in here *)) lst
```

Exercise: student [★★]

Assume the following type definition:

```
type student = {first_name : string; last_name : string; gpa : float}
```

Give OCaml expressions that have the following types:

- `student`
- `student -> string * string` (a function that extracts the student's name)
- `string -> string -> float -> student` (a function that creates a student record)

Exercise: pokerecord [★★]

Here is a variant that represents a few Pokémon types:

```
type poketype = Normal | Fire | Water
```

- Define the type `pokemon` to be a record with fields `name` (a string), `hp` (an integer), and `ptype` (a `poketype`).
- Create a record named `charizard` of type `pokemon` that represents a Pokémon with 78 HP and Fire type.
- Create a record named `squirtle` of type `pokemon` that represents a Pokémon with 44 HP and Water type.

Exercise: safe hd and tl [★★]

Write a function `safe_hd : 'a list -> 'a option` that returns `Some x` if the head of the input list is `x`, and `None` if the input list is empty.

Also write a function `safe_tl : 'a list -> 'a list option` that returns the tail of the list, or `None` if the list is empty.

Exercise: pokefun [★★★]

Write a function `max_hp : pokemon list -> pokemon option` that, given a list of `pokemon`, finds the Pokémon with the highest HP.

Exercise: date before [★★]

Define a *date-like triple* to be a value of type `int * int * int`. Examples of date-like triples include `(2013, 2, 1)` and `(0, 0, 1000)`. A *date* is a date-like triple whose first part is a positive year (i.e., a year in the common era),

second part is a month between 1 and 12, and third part is a day between 1 and 31 (or 30, 29, or 28, depending on the month and year). `(2013, 2, 1)` is a date; `(0, 0, 1000)` is not.

Write a function `is_before` that takes two dates as input and evaluates to `true` or `false`. It evaluates to `true` if the first argument is a date that comes before the second argument. (If the two dates are the same, the result is `false`.)

Your function needs to work correctly only for dates, not for arbitrary date-like triples. However, you will probably find it easier to write your solution if you think about making it work for arbitrary date-like triples. For example, it's easier to forget about whether the input is truly a date, and simply write a function that claims (for example) that January 100, 2013 comes before February 34, 2013—because any date in January comes before any date in February, but a function that says that January 100, 2013 comes after February 34, 2013 is also valid. You may ignore leap years.

Exercise: earliest date [★★★]

Write a function `earliest : (int*int*int) list -> (int * int * int) option`. It evaluates to `None` if the input list is empty, and to `Some d` if date `d` is the earliest date in the list. *Hint: use `is_before`.*

As in the previous exercise, your function needs to work correctly only for dates, not for arbitrary date-like triples.

Exercise: assoc list [★]

Use the functions `insert` and `lookup` from the *section on association lists* to construct an association list that maps the integer 1 to the string "one", 2 to "two", and 3 to "three". Lookup the key 2. Lookup the key 4.

Exercise: cards [★★]

- Define a variant type `suit` that represents the four suits, ♣ ♦ ♥ ♠, in a standard 52-card deck. All the constructors of your type should be constant.
- Define a type `rank` that represents the possible ranks of a card: 2, 3, …, 10, Jack, Queen, King, or Ace. There are many possible solutions; you are free to choose whatever works for you. One is to make `rank` be a synonym of `int`, and to assume that Jack=11, Queen=12, King=13, and Ace=1 or 14. Another is to use variants.
- Define a type `card` that represents the suit and rank of a single card. Make it a record with two fields.
- Define a few values of type `card`: the Ace of Clubs, the Queen of Hearts, the Two of Diamonds, the Seven of Spades.

Exercise: matching [★]

For each pattern in the list below, give a value of type `int option list` that does *not* match the pattern and is not the empty list, or explain why that's impossible.

- `Some x :: tl`
- `[Some 3110; None]`
- `[Some x; _]`
- `h1 :: h2 :: tl`
- `h :: tl`

Exercise: quadrant [★★]

Complete the `quadrant` function below, which should return the quadrant of the given `x, y` point according to the diagram on the right (borrowed from Wikipedia). Points that lie on an axis do not belong to any quandrant. *Hints: (a) define a helper function for the sign of an integer, (b) match against a pair.*

```
type quad = I | II | III | IV
type sign = Neg | Zero | Pos

let sign (x:int) : sign =
  ...

let quadrant : int*int -> quad option = fun (x,y) ->
  match ... with
    | ... -> Some I
    | ... -> Some II
    | ... -> Some III
    | ... -> Some IV
    | ... -> None
```

Exercise: quadrant when [★★]

Rewrite the quadrant function to use the `when` syntax. You won't need your helper function from before.

```
let quadrant_when : int*int -> quad option = function
    | ... when ... -> Some I
    | ... when ... -> Some II
    | ... when ... -> Some III
    | ... when ... -> Some IV
    | ... -> None
```

Exercise: depth [★★]

Write a function `depth : 'a tree -> int` that returns the number of nodes in any longest path from the root to a leaf. For example, the depth of an empty tree (simply `Leaf`) is `0`, and the depth of tree `t` above is `3`. *Hint: there is a library function `max : 'a -> 'a -> 'a` that returns the maximum of any two values of the same type.*

Exercise: shape [★★★]

Write a function `same_shape : 'a tree -> 'b tree -> bool` that determines whether two trees have the same shape, regardless of whether the values they carry at each node are the same. *Hint: use a pattern match with three branches, where the expression being matched is a pair of trees.*

Exercise: list max exn [★★]

Write a function `list_max : int list -> int` that returns the maximum integer in a list, or raises `Failure "list_max"` if the list is empty.

Exercise: list max exn string [★★]

Write a function `list_max_string : int list -> string` that returns a string containing the maximum integer in a list, or the string `"empty"` (note, not the exception `Failure "empty"` but just the string `"empty"`) if the list is empty. *Hint: `string_of_int` in the standard library will do what its name suggests.*

Exercise: list max exn ounit [★]

Write two OUnit tests to determine whether your solution to **list max exn**, above, correctly raises an exception when its input is the empty list, and whether it correctly returns the max value of the input list when that list is nonempty.

Exercise: is_bst [★★★★]

Write a function `is_bst : ('a*'b) tree -> bool` that returns `true` if and only if the given tree satisfies the binary search tree invariant. An efficient version of this function that visits each node at most once is somewhat tricky to write. *Hint: write a recursive helper function that takes a tree and either gives you (i) the minimum and maximum value in the tree, or (ii) tells you that the tree is empty, or (iii) tells you that the tree does not satisfy the invariant. Your `is_bst` function will not be recursive, but will call your helper function and pattern match on the result. You will need to define a new variant type for the return type of your helper function.*

Exercise: quadrant poly [★★]

Modify your definition of quadrant to use polymorphic variants. The types of your functions should become these:

```
val sign : int -> [> `Neg | `Pos | `Zero ]
val quadrant : int * int -> [> `I | `II | `III | `IV ] option
```

CHAPTER

FOUR

HIGHER-ORDER PROGRAMMING

Functions are values just like any other value in OCaml. What does that mean exactly? This means that we can pass functions around as arguments to other functions, that we can store functions in data structures, that we can return functions as a result from other functions, and so forth.

Higher-order functions either take other functions as input or return other functions as output (or both). Higher-order functions are also known as *functionals*, and programming with them could therefore be called *functional programming*—indicating what the heart of programming in languages like OCaml is all about.

Higher-order functions were one of the more recent adoptions from functional languages into mainstream languages. The Java 8 Streams library and Python 2.3's `itertools` modules are examples of that; C++ has also been increasing its support since at least 2011.

Note: C wizards might object the adoption isn't so recent. After all, C has long had the ability to do higher-order programming through function pointers. But that ability also depends on the programming pattern of passing an additional *environment* parameter to provide the values of variables in the function to be called through the pointer. As we'll see in our later chapter on interpreters, the essence of (higher-order) functions in a functional langauge is that they are really something called a *closure* that obviates the need for that extra parameter. Bear in mind that the issue is not what is *possible* to compute in a language—after all everything is eventually compiled down to machine code, so we could just write in that exclusively—but what is *pleasant* to compute.

In this chapter we will see what all the fuss is about. Higher-order functions enable beautiful, general, reusable code.

4.1 Higher-Order Functions

Consider these functions `double` and `square` on integers:

```
let double x = 2 * x
let square x = x * x
```

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

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

Let's use these functions to write other functions that quadruple and raise a number to the fourth power:

```
let quad x = double (double x)
let fourth x = square (square x)
```

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

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

There is an obvious similarity between these two functions: what they do is apply a given function twice to a value. By passing in the function to another function `twice` as an argument, we can abstract this functionality:

```
let twice f x = f (f x)
```

```
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
```

The function `twice` is higher-order: its input `f` is a function. And—recalling that all OCaml functions really take only a single argument—its output is technically `fun x -> f (f x)`, so `twice` returns a function hence is also higher-order in that way.

Using `twice`, we can implement `quad` and `fourth` in a uniform way:

```
let quad x = twice double x
let fourth x = twice square x
```

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

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

4.1.1 The Abstraction Principle

Above, we have exploited the structural similarity between `quad` and `fourth` to save work. Admittedly, in this toy example it might not seem like much work. But imagine that `twice` were actually some much more complicated function. Then if someone comes up with a more efficient version of it, every function written in terms of it (like `quad` and `fourth`) could benefit from that improvement in efficiency, without needing to be recoded.

Part of being an excellent programmer is recognizing such similarities and *abstracting* them by creating functions (or other units of code) that implement them. Bruce MacLennan names this the **Abstraction Principle** in his textbook *Functional Programming: Theory and Practice* (1990). The Abstraction Principle says to avoid requiring something to be stated more than once; instead, *factor out* the recurring pattern.Higher-order functions enable such refactoring, because they allow us to factor out functions and parameterize functions on other functions.

Besides `twice`, here are some more relatively simple examples, indebted also to MacLennan:

Apply. We can write a function that applies its first input to its second input:

```
let apply f x = f x
```

```
val apply : ('a -> 'b) -> 'a -> 'b = <fun>
```

Of course, writing `apply f` is a lot more work than just writing `f`.

Pipeline. The pipeline operator, which we've previously seen, is a higher-order function:

```
let pipeline x f = f x
let (|>) = pipeline
let x = 5 |> double
```

```
val pipeline : 'a -> ('a -> 'b) -> 'b = <fun>
```

```
val ( |> ) : 'a -> ('a -> 'b) -> 'b = <fun>
```

```
val x : int = 10
```

Compose. We can write a function that composes two other functions:

```
let compose f g x = f (g x)
```

```
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

This function would let us create a new function that can be applied many times, such as the following:

```
let square_then_double = compose double square
let x = square_then_double 1
let y = square_then_double 2
```

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

```
val x : int = 2
```

```
val y : int = 8
```

Both. We can write a function that applies two functions to the same argument and returns a pair of the result:

```
let both f g x = (f x, g x)
let ds = both double square
let p = ds 3
```

```
val both : ('a -> 'b) -> ('a -> 'c) -> 'a -> 'b * 'c = <fun>
```

```
val ds : int -> int * int = <fun>
```

```
val p : int * int = (6, 9)
```

Cond. We can write a function that conditionally chooses which of two functions to apply based on a predicate:

```
let cond p f g x =
  if p x then f x else g x
```

```
val cond : ('a -> bool) -> ('a -> 'b) -> ('a -> 'b) -> 'a -> 'b = <fun>
```

4.1.2 The Meaning of "Higher Order"

The phrase "higher order" is used throughout logic and computer science, though not necessarily with a precise or consistent meaning in all cases.

In logic, *first-order quantification* refers primarily to the universal and existential ($\forall$ and $\exists$) quantifiers. These let you quantify over some *domain* of interest, such as the natural numbers. But for any given quantification, say $\forall x$, the variable being quantified represents an individual element of that domain, say the natural number 42.

Second-order quantification lets you do something strictly more powerful, which is to quantify over *properties* of the domain. Properties are assertions about individual elements, for example, that a natural number is even, or that it is prime. In some logics we can equate properties with sets of individual, for example the set of all even naturals. So second-order quantification is often thought of as quantification over *sets*. You can also think of properties as being functions that take in an element and return a Boolean indicating whether the element satisfies the property; this is called the *characteristic function* of the property.

Third-order logic would allow quantification over properties of properties, and *fourth-order* over properties of properties of properties, and so forth. *Higher-order logic* refers to all these logics that are more powerful than first-order logic; though one interesting result in this area is that all higher-order logics can be expressed in second-order logic.

In programming languages, *first-order functions* similarly refer to functions that operate on individual data elements (e.g., strings, ints, records, variants, etc.). Whereas *higher-order function* can operate on functions, much like higher-order logics can quantify over over properties (which are like functions).

4.1.3 Famous Higher-order Functions

In the next few sections we'll dive into three of the most famous higher-order functions: map, filter, and fold. These are functions that can be defined for many data structures, including lists and trees. The basic idea of each is that:

- *map* transforms elements,
- *filter* eliminates elements, and
- *fold* combines elements.

4.2 Map

Here are two functions we might want to write:

```
(** [add1 lst] adds 1 to each element of [lst] *)
let rec add1 = function
  | [] -> []
  | h :: t -> (h + 1) :: add1 t

let lst1 = add1 [1; 2; 3]
```

```
val add1 : int list -> int list = <fun>
```

```
val lst1 : int list = [2; 3; 4]
```

```
(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let rec concat_bang = function
  | [] -> []
  | h :: t -> (h ^ "!") :: concat_bang t
```

(continues on next page)

(continued from previous page)

```
let lst2 = concat_bang ["sweet"; "salty"]
```

```
val concat_bang : string list -> string list = <fun>
```

```
val lst2 : string list = ["sweet!"; "salty!"]
```

There's a lot of similarity between those two functions:

- They both pattern match against a list.
- They both return the same value for the base case of the empty list.
- They both recurse on the tail in the case of a non-empty list.

In fact the only difference (other than their names) is what they do for the head element: add versus concatenate. Let's rewrite the two functions to make that difference even more explicit:

```
(** [add1 lst] adds 1 to each element of [lst] *)
let rec add1 = function
  | [] -> []
  | h :: t ->
    let f = fun x -> x + 1 in
    f h :: add1 t

(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let rec concat_bang = function
  | [] -> []
  | h :: t ->
    let f = fun x -> x ^ "!" in
    f h :: concat_bang t
```

```
val add1 : int list -> int list = <fun>
```

```
val concat_bang : string list -> string list = <fun>
```

Now the only difference between the two functions (again, other than their names) is the body of helper function `f`. Why repeat all that code when there's such a small difference between the functions? We might as well *abstract* that one helper function out from each main function and make it an argument:

```
let rec add1' f = function
  | [] -> []
  | h :: t -> f h :: add1' f t

(** [add1 lst] adds 1 to each element of [lst] *)
let add1 = add1' (fun x -> x + 1)

let rec concat_bang' f = function
  | [] -> []
  | h :: t -> f h :: concat_bang' f t

(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let concat_bang = concat_bang' (fun x -> x ^ "!")
```

```
val add1' : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
val add1 : int list -> int list = <fun>
```

```
val concat_bang' : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
val concat_bang : string list -> string list = <fun>
```

But now there really is no difference at all between `add1'` and `concat_bang'` except for their names. They are totally duplicated code. Even their types are now the same, because nothing about them mentions integers or strings. We might as well just keep only one of them and come up with a good new name for it. One possibility could be `transform`, because they tranform a list by applying a function to each element of the list:

```
let rec transform f = function
  | [] -> []
  | h :: t -> f h :: transform f t

(** [add1 lst] adds 1 to each element of [lst] *)
let add1 = transform (fun x -> x + 1)

(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let concat_bang = transform (fun x -> x ^ "!")
```

```
val transform : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
val add1 : int list -> int list = <fun>
```

```
val concat_bang : string list -> string list = <fun>
```

Note: Instead of

```
let add1 lst = transform (fun x -> x + 1) lst
```

above we wrote

```
let add1 = transform (fun x -> x + 1)
```

This is another way of being higher order, but it's one we already learned about under the guise of partial application. The latter way of writing the function partially applies `transform` to just one of its two arguments, thus returning a function. That function is bound to the name `add1`.

Indeed, the C++ library does call the equivalent function `transform`. But OCaml and many other languages (including Java and Python) use the shorter word *map*, in the mathematical sense of how a function maps an input to an output. So let's make one final change to that name:

```
let rec map f = function
  | [] -> []
  | h :: t -> f h :: map f t

(** [add1 lst] adds 1 to each element of [lst] *)
let add1 = map (fun x -> x + 1)

(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let concat_bang = map (fun x -> x ^ "!")
```

```
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
val add1 : int list -> int list = <fun>
```

```
val concat_bang : string list -> string list = <fun>
```

We have now successfully applied the Abstraction Principle: the common structure has been factored out. What's left clearly expresses the computation, at least to the reader who is familiar with `map`, in a way that the original versions do not as quickly make apparent.

4.2.1 Side Effects

The `map` function exists already in OCaml's standard library as `List.map`, but with one small difference from the implementation we discovered above. First, let's see what's potentially wrong with our own implementation, then we'll look at the standard library's implementation.

We've seen before in our discussion of *exceptions* that the OCaml language specification does not generally specify evaluation order of subexpressions, and that the current language implementation generally evaluates right-to-left. Because of that, the following (rather contrived) code actually causes the list elements to be printed in what might seem like reverse order:

```
let p x = print_int x; print_newline(); x + 1

let lst = map p [1; 2]
```

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

```
2
```

```
1
```

```
val lst : int list = [2; 3]
```

Here's why:

- Expression `map p [1; 2]` evaluates to `p 1 :: map p [2]`.
- The right-hand side of that expression is then evaluated to `p 1 :: (p 2 :: map p [])`. The application of `p` to `1` has not yet occurred.
- The right-hand side of `::` is again evaluated next, yielding `p 1 :: (p 2 :: [])`.
- Then `p` is applied to `2`, and finally to `1`.

That is likely surprising to anyone who is predisposed to thinking that evaluation would occur left-to-right. The solution is to use a `let` expression to cause the evaluation of the function application to occur before the recursive call:

```
let rec map f = function
  | [] -> []
  | h :: t -> let h' = f h in h' :: map f t

let lst2 = map p [1; 2]
```

```
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
1
```

```
2
```

```
val lst2 : int list = [2; 3]
```

Here's why that works:

- Expression `map p [1; 2]` evaluates to `let h' = p 1 in h'' :: map p [2]`.
- The binding expression `p 1` is evaluated, causing `1` to be printed and `h'` to be bound to `2`.
- The body expression `h'' :: map p [2]` is then evaluated, which leads to `2` being printed next.

So that's how the standard library defines `List.map`. We should use it instead of re-defining the function ourselves from now on. But it's good that we have discovered the function "from scratch" as it were, and that if needed we could quickly re-code it.

The bigger lesson to take away from this discussion is that when evaluation order matters, we need to use `let` to ensure it. When does it matter? Only when there are side effects. Printing and exceptions are the two we've seen so far. Later we'll add mutability.

4.2.2 Map and Tail Recursion

Astute readers will have noticed that the implementation of `map` is not tail recursive. That is to some extent unavoidable. Here's a tempting but awful way to create a tail-recursive version of it:

```
let rec map_tr_aux f acc = function
  | [] -> acc
  | h :: t -> map_tr_aux f (acc @ [f h]) t

let map_tr f = map_tr_aux f []

let lst = map_tr (fun x -> x + 1) [1; 2; 3]
```

```
val map_tr_aux : ('a -> 'b) -> 'b list -> 'a list -> 'b list = <fun>
```

```
val map_tr : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
val lst : int list = [2; 3; 4]
```

To some extent that works: the output is correct, and `map_tr_aux` is tail recursive. The subtle flaw is the subexpression `acc @ [f h]`. Recall that append is a linear-time operation on singly-linked lists. That is, if there are n list elements then append takes time $O(n)$. So at each recursive call we perform a $O(n)$ operation. And there will be n recursive calls, one for each element of the list. That's a total of $n \cdot O(n)$ work, which is $O(n^2)$. So we achieved tail recursion, but at a high cost: what ought to be a linear-time operation became quadratic time.

In an attempt to fix that, we could use the constant-time cons operation instead of the linear-time append operation:

```
let rec map_tr_aux f acc = function
  | [] -> acc
  | h :: t -> map_tr_aux f (f h :: acc) t

let map_tr f = map_tr_aux f []

let lst = map_tr (fun x -> x + 1) [1; 2; 3]
```

```
val map_tr_aux : ('a -> 'b) -> 'b list -> 'a list -> 'b list = <fun>
```

```
val map_tr : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
val lst : int list = [4; 3; 2]
```

And to some extent that works: it's tail recursive and linear time. The not-so-subtle flaw this time is that the output is backwards. As we take each element off the front of the input list, we put it on the front of the output list, but that reverses their order.

Note: To understand why the reversal occurs, it might help to think of the input and output lists as people standing in a queue:

- Input: Alice, Bob.
- Output: empty.

Then we remove Alice from the input and add her to the output:

- Input: Bob.
- Output: Alice.

Then we remove Bob from the input and add him to the output:

- Input: empty.
- Output: Bob, Alice.

The point is that with singly-linked lists, we can only operate on the head of the list and still be constant time. We can't move Bob to the back of the output without making him walk past Alice—and anyone else who might be standing in the output.

For that reason, the standard library calls this function `List.rev_map`, that is, a (tail-recursive) map function that returns its output in reverse order.

```
let rec rev_map_aux f acc = function
  | [] -> acc
  | h :: t -> rev_map_aux f (f h :: acc) t

let rev_map f = rev_map_aux f []

let lst = rev_map (fun x -> x + 1) [1; 2; 3]
```

```
val rev_map_aux : ('a -> 'b) -> 'b list -> 'a list -> 'b list = <fun>
```

```
val rev_map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
val lst : int list = [4; 3; 2]
```

If you want the output in the "right" order, that's easy: just apply `List.rev` to it:

```
let lst = List.rev (List.rev_map (fun x -> x + 1) [1; 2; 3])
```

```
val lst : int list = [2; 3; 4]
```

Since `List.rev` is both linear time and tail recursive, that yields a complete solution. We get a linear-time and tail-recursive map computation. The expense is that it requires two passes through the list: one to transform, the other to reverse. We're not going to do better than this efficiency with a singly-linked list. Of course, there are other data structures that implement lists, and we'll come to those eventually. Meanwhile, recall that we generally don't have to worry about tail recursion (which is to say, about stack space) until lists have 10,000 or more elements.

Why doesn't the standard library provide this all-in-one function? Maybe it will someday if there's good enough reason. But you might discover in your own programming there's not a lot of need for it. In many cases, we can either do without the tail recursion, or be content with a reversed list.

The bigger lesson to take away from this discussion is that there can be a tradeoff between time and space efficiency for recursive functions. By attempting to make a function more space efficient (i.e., tail recursive), we can accidentally make it asympotically less time efficient (i.e., quadratic instead of linear), or if we're clever keep the asymptotic time efficiency the same (i.e., linear) at the cost of a constant factor (i.e., processing twice).

4.2.3 Map in Other Languages

We mentioned above that the idea of map exists in many programming languages. Here's an example from Python:

```
>>> print(list(map(lambda x: x + 1, [1, 2, 3])))
[2, 3, 4]
```

We have to use the `list` function to convert the result of the `map` back to a list, because Python for sake of efficiency produces each element of the `map` output as needed. Here again we see the theme of "when does it get evaluated?" returning.

In Java, map is part of the `Stream` abstraction that was added in Java 8. Since there isn't a built-in Java syntax for lists or streams, it's a little more verbose to give an example. Here we use a factory method `Stream.of` to create a stream:

```
jshell> Stream.of(1, 2, 3).map(x -> x + 1).collect(Collectors.toList())
$1 ==> [2, 3, 4]
```

Like in the Python example, we have to use something to convert the stream back into a list. In this case it's the `collect` method.

4.3 Filter

Suppose we wanted to filter out only the even numbers from a list, or the odd numbers. Here are some functions to do that:

```
(** [even n] is whether [n] is even. *)
let even n =
  n mod 2 = 0

(** [evens lst] is the sublist of [lst] containing only even numbers. *)
let rec evens = function
  | [] -> []
  | h :: t -> if even h then h :: evens t else evens t

let lst1 = evens [1; 2; 3; 4]
```

```
val even : int -> bool = <fun>
```

```
val evens : int list -> int list = <fun>
```

```
val lst1 : int list = [2; 4]
```

```
(** [odd n] is whether [n] is odd. *)
let odd n =
  n mod 2 <> 0

(** [odds lst] is the sublist of [lst] containing only odd numbers. *)
let rec odds = function
  | [] -> []
  | h :: t -> if odd h then h :: odds t else odds t

let lst2 = odds [1; 2; 3; 4]
```

```
val odd : int -> bool = <fun>
```

```
val odds : int list -> int list = <fun>
```

```
val lst2 : int list = [1; 3]
```

Functions `evens` and `odds` are nearly the same code: the only essential difference is the test they apply to the head element. So as we did with `map` in the previous section, let's factor out that test as a function. Let's name the function `p` as short for "predicate", which is a fancy way of saying that it tests whether something is true or false:

```
let rec filter p = function
  | [] -> []
  | h :: t -> if p h then h :: filter p t else filter p t
```

```
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

And now we can reimplement our original two functions:

```
let evens = filter even
let odds = filter odd
```

```
val evens : int list -> int list = <fun>
```

```
val odds : int list -> int list = <fun>
```

How simple these are! How clear! (At least to the reader who is familiar with `filter`.)

4.3.1 Filter and Tail Recursion

As we did with `map`, we can create a tail-recursive version of `filter`:

```
let rec filter_aux p acc = function
  | [] -> acc
  | h :: t -> if p h then filter_aux p (h :: acc) t else filter_aux p acc t

let filter p = filter_aux p []

let lst = filter even [1; 2; 3; 4]
```

```
val filter_aux : ('a -> bool) -> 'a list -> 'a list -> 'a list = <fun>
```

```
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

```
val lst : int list = [4; 2]
```

And again we discover the output is backwards. Here, the standard library makes a different choice than it did with `map`. It builds in the reversal to `List.filter`, which is implemented like this:

```
let rec filter_aux p acc = function
  | [] -> List.rev acc (* note the built-in reversal *)
  | h :: t -> if p h then filter_aux p (h :: acc) t else filter_aux p acc t

let filter p = filter_aux p []
```

```
val filter_aux : ('a -> bool) -> 'a list -> 'a list -> 'a list = <fun>
```

```
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Why does the standard library treat `map` and `filter` differently on this point? Good question. Perhaps there has simply never been a demand for a `filter` function whose time efficiency is a constant factor better. Or perhaps it is just historical accident.

4.3.2 Filter in Other Languages

Again, the idea of filter exists in many programming languages. Here it is in Python:

```
>>> print(list(filter(lambda x: x % 2 == 0, [1, 2, 3, 4])))
[2, 4]
```

And in Java:

```
jshell> Stream.of(1, 2, 3, 4).filter(x -> x % 2 == 0).collect(Collectors.toList())
$1 ==> [2, 4]
```

4.4 Fold

The map functional gives us a way to individually transform each element of a list. The filter functional gives us a way to individually decide whether to keep or throw away each element of a list. But both of those are really just looking at a single element at a time. What if we wanted to somehow combine all the elements of a list? That's what the *fold* functional is for. It turns out that there are two versions of it, which we'll study in this section. But to start, let's look at a related function—not actually in the standard library—that we call *combine*.

4.4.1 Combine

Once more, let's write two functions:

```
(** [sum lst] is the sum of all the elements of [lst]. *)
let rec sum = function
  | [] -> 0
  | h :: t -> h + sum t

let s = sum [1; 2; 3]
```

```
val sum : int list -> int = <fun>
```

```
val s : int = 6
```

```
(** [concat lst] is the concatenation of all the elements of [lst]. *)
let rec concat = function
  | [] -> ""
  | h :: t -> h ^ concat t

let c = concat ["a"; "b"; "c"]
```

```
val concat : string list -> string = <fun>
```

```
val c : string = "abc"
```

As when we went through similar exercises with map and filter, the functions share a great deal of common structure. The differences here are:

- the case for the empty list returns a different initial value, `0` vs `""`
- the case of a non-empty list uses a different operator to combine the head element with the result of the recursive call, `+` vs `^`.

So can we apply the Abstraction Principle again? Sure! But this time we need to factor out *two* arguments: one for each of those two differences.

To start, let's factor out only the initial value:

```
let rec sum' init = function
  | [] -> init
  | h :: t -> h + sum' init t

let sum = sum' 0

let rec concat' init = function
  | [] -> init
  | h :: t -> h ^ concat' init t

let concat = concat' ""
```

```
val sum' : int -> int list -> int = <fun>
```

```
val sum : int list -> int = <fun>
```

```
val concat' : string -> string list -> string = <fun>
```

```
val concat : string list -> string = <fun>
```

Now the only real difference left between `sum'` and `concat'` is the operator used to combine the head with the recursive call on the tail. That operator can also become an argument to a unified function we call `combine`:

```
let rec combine op init = function
  | [] -> init
  | h :: t -> op h (combine op init t)

let sum = combine ( + ) 0
let concat = combine ( ^ ) ""
```

```
val combine : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

```
val sum : int list -> int = <fun>
```

```
val concat : string list -> string = <fun>
```

One way to think of `combine` would be that:

- the `[]` value in the list gets replaced by `init`, and
- each `::` constructor gets replaced by `op`.

For example, `[a; b; c]` is just syntactic sugar for `a :: (b :: (c :: []))`. So if we replace `[]` with `0` and `::` with `(+)`, we get `a + (b + (c + 0))`. And that would be the sum of the list.

Once more, the Abstraction Principle has led us to an amazingly simple and succinct expression of the computation.

4.4.2 Fold Right

The `combine` function is the idea underlying an actual OCaml library function. To get there, we need to make a couple of changes to the implementation we have so far.

First, let's rename some of the arguments: we'll change `op` to `f` to emphasize that really we could pass in any function, not just a built-in operator like +. And we'll change `init` to `acc`, which as usual stands for "accumulator". That yields:

```
let rec combine f acc = function
  | [] -> acc
  | h :: t -> f h (combine f acc t)
```

```
val combine : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

Second, let's make an admittedly less well-motivated change. We'll swap the implicit list argument to `combine` with the `init` argument:

```
let rec combine' f lst acc = match lst with
  | [] -> acc
  | h :: t -> f h (combine' f t acc)

let sum lst = combine' ( + ) lst 0
let concat lst = combine' ( ^ ) lst ""
```

```
val combine' : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

```
val sum : int list -> int = <fun>
```

```
val concat : string list -> string = <fun>
```

It's a little less convenient to code the function this way, because we no longer get to take advantage of the `function` keyword, nor of partial application in defining `sum` and `concat`. But there's no algorithmic change.

What we now have is the actual implementation of the standard library function `List.fold_right`. All we have left to do is change the function name:

```
let rec fold_right f lst acc = match lst with
  | [] -> acc
  | h :: t -> f h (fold_right f t acc)
```

```
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Why is this function called "fold right"? The intuition is that the way it works is to "fold in" elements of the list from the right to the left, combining each new element using the operator. For example, `fold_right ( + ) [a; b; c] 0` results in evaluation of the expression `a + (b + (c + 0))`. The parentheses associate from the right-most subexpression to the left.

4.4.3 Tail Recursion and Combine

Neither `fold_right` nor `combine` are tail recursive: after the recursive call returns, there is still work to be done in applying the function argument `f` or `op`. Let's go back to `combine` and rewrite it to be tail recursive. All that requires is to change the cons branch:

```
let rec combine_tr f acc = function
  | [] -> acc
  | h :: t -> combine_tr f (f acc h) t   (* only real change *)
```

```
val combine_tr : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

(Careful readers will notice that the type of `combine_tr` is different than the type of `combine`. We will address that soon.)

Now the function `f` is applied to the head element `h` and the accumulator `acc` *before* the recursive call is made, thus ensuring there's no work remaining to be done after the call returns. If that seems a little mysterious, here's a rewriting of the two functions that might help:

```
let rec combine f acc = function
  | [] -> acc
  | h :: t ->
    let acc' = combine f acc t in
    f h acc'

let rec combine_tr f acc = function
  | [] -> acc
  | h :: t ->
    let acc' = f acc h in
    combine_tr f acc' t
```

```
val combine : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

```
val combine_tr : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Pay close attention to the definition of `acc'`, the new accumulator, in each of those version:

- In the original version, we procrastinate using the head element `h`. First, we combine all the remaining tail elements to get `acc'`. Only then do we use `f` to fold in the head. So the value passed as the initial value of `acc` turns out to be the same for every recursive invocation of `combine`: it's passed all the way down to where it's needed, at the right-most element of the list, then used there exactly once.
- But in the tail recursive version, we "pre-crastinate" by immediately folding `h` in with the old accumulator `acc`. Then we fold that in with all the tail elements. So at each recursive invocation, the value passed as the argument `acc` can be different.

The tail recursive version of combine works just fine for summation (and concatenation, which we elide):

```
let sum = combine_tr ( + ) 0
let s = sum [1; 2; 3]
```

```
val sum : int list -> int = <fun>
```

```
val s : int = 6
```

But something possibly surprising happens with subtraction:

```
let sub = combine ( - ) 0
let s = sub [3; 2; 1]

let sub_tr = combine_tr ( - ) 0
let s' = sub_tr [3; 2; 1]
```

```
val sub : int list -> int = <fun>
```

```
val s : int = 2
```

```
val sub_tr : int list -> int = <fun>
```

```
val s' : int = -6
```

The two results are different!

- With `combine` we compute `3 - (2 - (1 - 0))`. First we fold in `1`, then `2`, then `3`. We are processing the list from right to left, putting the initial accumulator at the far right.
- But with `combine_tr` we compute `(((0 - 3) - 2) - 1)`. We are processing the list from left to right, putting the initial accumulator at the far left.

With addition it didn't matter which order we processed the list, because addition is associative and commutative. But subtraction is not, so the two directions result in different answers.

Actually this shouldn't be too surprising if we think back to when we made `map` be tail recursive. Then, we discovered that tail recursion can cause us to process the list in reverse order from the non-tail recursive version of the same function. That's what happened here.

4.4.4 Fold Left

Our `combine_tr` function is also in the standard library under the name `List.fold_left`:

```
let rec fold_left f acc = function
  | [] -> acc
  | h :: t -> fold_left f (f acc h) t

let sum = fold_left ( + ) 0
let concat = fold_left ( ^ ) ""
```

```
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

```
val sum : int list -> int = <fun>
```

```
val concat : string list -> string = <fun>
```

We have once more succeeded in applying the Abstraction Principle.

4.4.5 Fold Left vs. Fold Right

Let's review the differences between `fold_right` and `fold_left`:

- They combine list elements in opposite orders, as indicated by their names. Function `fold_right` combines from the right to the left, whereas `fold_left` proceeds from the left to the right.
- Function `fold_left` is tail recursive whereas `fold_right` is not.
- The types of the functions are different.

Regarding that final point, it can be hard to remember what those types are! Luckily we can always ask the toplevel:

```
List.fold_left;;
List.fold_right;;
```

```
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

```
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

To understand those types, look for the list argument in each one of them. That tells you the type of the values in the list. Then look for the type of the return value; that tells you the type of the accumulator. From there you can work out everything else.

- In `fold_left`, the list argument is of type `'b list`, so the list contains values of type `'b`. The return type is `'a`, so the accumulator has type `'a`. Knowing that, we can figure out that the second argument is the initial value of the accumulator (because it has type `'a`). And we can figure out that the first argument, the combining operator, takes as its own first argument an accumulator value (because it has type `'a`), as its own second argument a list element (because it has type `'b`), and returns a new accumulator value.
- In `fold_right`, the list argument is of type `'a list`, so the list contains values of type `'a`. The return type is `'b`, so the accumulator has type `'b`. Knowing that, we can figure out that the third argument is the initial value of the accumulator (because it has type `'b`). And we can figure out that the first argument, the combining operator, takes as its own second argument an accumulator value (because it has type `'b`), as its own first argument a list element (because it has type `'a`), and returns a new accumulator value.

Tip: You might wonder why the argument orders are different between the two `fold` functions. Good question. Other libraries do in fact use different argument orders. One way to remember it for OCaml is that in `fold_X` the accumulator argument goes to the `X` of the list argument.

If you find it hard to keep track of all these argument orders, the `ListLabels` module in the standard library can help. It uses labeled arguments to give names to the combining operator (which it calls `f`) and the initial accumulator value (which it calls `init`). Internally, the implementation is actually identical to the `List` module.

```
ListLabels.fold_left;;
ListLabels.fold_left ~f:(fun x y -> x - y) ~init:0 [1;2;3];;
```

```
- : f:('a -> 'b -> 'a) -> init:'a -> 'b list -> 'a = <fun>
```

```
- : int = -6
```

```
ListLabels.fold_right;;
ListLabels.fold_right ~f:(fun y x -> x - y) ~init:0 [1;2;3];;
```

```
- : f:('a -> 'b -> 'b) -> 'a list -> init:'b -> 'b = <fun>
```

```
- : int = -6
```

Notice how in the two applications of fold above, we are able to write the arguments in a uniform order thanks to their labels. However, we still have to be careful about which argument to the combining operator is the list element vs. the accumulator value.

4.4.6 A Digression on Labeled Arguments and Fold

It's possible to write our own version of the fold functions that would label the arguments to the combining operator, so we don't even have to remember their order:

```
let rec fold_left ~op:(f: acc:'a -> elt:'b -> 'a) ~init:acc lst =
  match lst with
  | [] -> acc
  | h :: t -> fold_left ~op:f ~init:(f ~acc:acc ~elt:h) t

let rec fold_right ~op:(f: elt:'a -> acc:'b -> 'b) lst ~init:acc =
  match lst with
  | [] -> acc
  | h :: t -> f ~elt:h ~acc:(fold_right ~op:f t ~init:acc)
```

```
val fold_left : op:(acc:'a -> elt:'b -> 'a) -> init:'a -> 'b list -> 'a =
  <fun>
```

```
val fold_right : op:(elt:'a -> acc:'b -> 'b) -> 'a list -> init:'b -> 'b =
  <fun>
```

But those functions aren't as useful as they might seem:

```
let s = fold_left ~op:( + ) ~init:0 [1;2;3]
```

```
File "[17]", line 1, characters 22-27:
1 | let s = fold_left ~op:( + ) ~init:0 [1;2;3]
                          ^^^^^
Error: This expression has type int -> int -> int
       but an expression was expected of type acc:'a -> elt:'b -> 'a
```

The problem is that the built-in + operator doesn't have labeled arguments, so we can't pass it in as the combining operator to our labeled functions. We'd have to define our own labeled version of it:

```
let add ~acc ~elt = acc + elt
let s = fold_left ~op:add ~init:0 [1; 2; 3]
```

But now we have to remember that the `~acc` parameter to `add` will become the left-hand argument to `( + )`. That's not really much of an improvement over what we had to remember to begin with.

4.4.7 Using Fold to Implement Other Functions

Folding is so powerful that we can write many other list functions in terms of `fold_left` or `fold_right`. For example,

```
let length lst =
  List.fold_left (fun acc _ -> acc + 1) 0 lst

let rev lst =
  List.fold_left (fun acc x -> x :: acc) [] lst

let map f lst =
  List.fold_right (fun x acc -> f x :: acc) lst []

let filter f lst =
  List.fold_right (fun x acc -> if f x then x :: acc else acc) lst []
```

```
val length : 'a list -> int = <fun>
```

```
val rev : 'a list -> 'a list = <fun>
```

```
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

At this point it begins to become debatable whether it's better to express the computations above using folding or using the ways we have already seen. Even for an experienced functional programmer, understanding what a fold does can take longer than reading the naive recursive implementation. If you peruse the source code of the standard library, you'll see that none of the `List` module internally is implemented in terms of folding, which is perhaps one comment on the readability of fold. On the other hand, using fold ensures that the programmer doesn't accidentally program the recursive traversal incorrectly. And for a data structure that's more complicated than lists, that robustness might be a win.

4.4.8 Fold vs. Recursive vs. Library

We've now seen three different ways for writing functions that manipulate lists:

- directly as a recursive function that pattern matches against the empty list and against cons,
- using `fold` functions, and
- using other library functions.

Let's try using each of those ways to solve a problem, so that we can appreciate them better.

Consider writing a function `lst_and: bool list -> bool`, such that `lst_and [a1; ...; an]` returns whether all elements of the list are `true`. That is, it evaluates the same as `a1 && a2 && ... && an`. When applied to an empty list, it evaluates to `true`.

Here are three possible ways of writing such a function. We give each way a slightly different function name for clarity.

```
let rec lst_and_rec = function
  | [] -> true
  | h :: t -> h && lst_and_rec t

let lst_and_fold =
        List.fold_left (fun acc elt -> acc && elt) true

let lst_and_lib =
        List.for_all (fun x -> x)
```

```
val lst_and_rec : bool list -> bool = <fun>
```

```
val lst_and_fold : bool list -> bool = <fun>
```

```
val lst_and_lib : bool list -> bool = <fun>
```

The worst-case running time of all three functions is linear in the length of the list. But:

- The first function, `lst_and_rec` has the advantage that it need not process the entire list. It will immediately return `false` the first time they discover a `false` element in the list.
- The second function, `lst_and_fold`, will always process every element of the list.
- As for the third function `lst_and_lib`, according to the documentation of `List.for_all`, it returns `(p a1) && (p a2) && ... && (p an)`. So like `lst_and_rec` it need not process every element.

4.5 Beyond Lists

Functionals like map and fold are not restricted to lists. They make sense for nearly any kind of data collection. For example, recall this tree representation:

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree
```

4.5.1 Map on Trees

This one is easy. All we have to do is apply the function `f` to the value `v` at each node:

```
let rec map_tree f = function
  | Leaf -> Leaf
  | Node (v, l, r) -> Node (f v, map_tree f l, map_tree f r)
```

```
val map_tree : ('a -> 'b) -> 'a tree -> 'b tree = <fun>
```

4.5.2 Fold on Trees

This one is only a little harder. Let's develop a fold functional for `'a tree` similar to our `fold_right` over `'a list`. One way to think of `List.fold_right` would be that the `[]` value in the list gets replaced by the `acc` argument, and each `::` constructor gets replaced by an application of the `f` argument. For example, `[a; b; c]` is syntactic sugar for `a :: (b :: (c :: []))`. So if we replace `[]` with `0` and `::` with `( + )`, we get `a + (b + (c + 0))`. Along those lines, here's a way we could rewrite `fold_right` that will help us think a little more clearly:

```
type 'a mylist =
  | Nil
  | Cons of 'a * 'a mylist

let rec fold_mylist f acc = function
  | Nil -> acc
  | Cons (h, t) -> f h (fold_mylist f acc t)
```

```
type 'a mylist = Nil | Cons of 'a * 'a mylist
```

```
val fold_mylist : ('a -> 'b -> 'b) -> 'b -> 'a mylist -> 'b = <fun>
```

The algorithm is the same. All we've done is to change the definition of lists to use constructors written with alphabetic characters instead of punctuation, and to change the argument order of the fold function.

For trees, we'll want the initial value of `acc` to replace each `Leaf` constructor, just like it replaced `[]` in lists. And we'll want each `Node` constructor to be replaced by the operator. But now the operator will need to be *ternary* instead of *binary*—that is, it will need to take three arguments instead of two—because a tree node has a value, a left child, and a right child, whereas a list cons had only a head and a tail.

Inspired by those observations, here is the fold function on trees:

```
let rec fold_tree f acc = function
  | Leaf -> acc
  | Node (v, l, r) -> f v (fold_tree f acc l) (fold_tree f acc r)
```

```
val fold_tree : ('a -> 'b -> 'b -> 'b) -> 'b -> 'a tree -> 'b = <fun>
```

If you compare that function to `fold_mylist`, you'll note it very nearly identical. There's just one more recursive call in the second pattern-matching branch, corresponding to the one more occurrence of `'a tree` in the definition of that type.

We can then use `fold_tree` to implement some of the tree functions we've previously seen:

```
let size t = fold_tree (fun _ l r -> 1 + l + r) 0 t
let depth t = fold_tree (fun _ l r -> 1 + max l r) 0 t
let preorder t = fold_tree (fun x l r -> [x] @ l @ r) [] t
```

```
val size : 'a tree -> int = <fun>
```

```
val depth : 'a tree -> int = <fun>
```

```
val preorder : 'a tree -> 'a list = <fun>
```

Why did we pick `fold_right` and not `fold_left` for this development? Because `fold_left` is tail recursive, which is something we're never going to achieve on binary trees. Suppose we process the left branch first; then we still have to process the right branch before we can return. So there will always be work left to do after a recursive call on one branch. Thus on trees an equivalent to `fold_right` is the best which we can hope for.

The technique we used to derive `fold_tree` works for any OCaml variant type `t`:

- Write a recursive `fold` function that takes in one argument for each constructor of `t`.
- That `fold` function matches against the constructors, calling itself recursively on any value of type `t` that it encounters.
- Use the appropriate argument of `fold` to combine the results of all recursive calls as well as all data not of type `t` at each constructor.

This technique constructs something called a *catamorphism*, aka a *generalized fold operation*. To learn more about catamorphisms, take a course on category theory.

4.5.3 Filter on Trees

This one is perhaps the hardest to design. The problem is: if we decide to filter a node, what should we do with its children?

- We could recurse on the children. If after filtering them only one child remains, we could promote it in place of its parent. But what if both children remain, or neither? Then we'd somehow have to reshape the tree. Without knowing more about how the tree is intended to be used—that is, what kind of data it represents—we are stuck.
- Instead, we could just eliminate the children entirely. So the decision to filter a node means pruning the entire subtree rooted at that node.

The latter is easy to implement:

```
let rec filter_tree p = function
  | Leaf -> Leaf
  | Node (v, l, r) ->
    if p v then Node (v, filter_tree p l, filter_tree p r) else Leaf
```

```
val filter_tree : ('a -> bool) -> 'a tree -> 'a tree = <fun>
```

4.6 Pipelining

Suppose we wanted to compute the sum of squares of the numbers from 0 up to n. How might we go about it? Of course (math being the best form of optimization), the most efficient way would be a closed-form formula:

$$\frac{n(n+1)(2n+1)}{6}$$

But let's imagine you've forgotten that formula. In an imperative language you might use a for loop:

```
# Python
def sum_sq(n):
        sum = 0
        for i in range(0, n):
                sum += i * i
        return sum
```

The equivalent (tail) recursive code in OCaml would be:

```
let sum_sq n =
  let rec loop i sum =
    if i > n then sum
    else loop (i + 1) (sum + i * i)
  in loop 0 0
```

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

Another, clearer way of producing the same result in OCaml uses higher-order functions and the pipeline operator:

```
let rec ( -- ) i j = if i > j then [] else i :: i + 1 -- j
let square x = x * x
let sum = List.fold_left ( + ) 0

let sum_sq n =
  0 -- n               (* [0;1;2;...;n]   *)
  |> List.map square   (* [0;1;4;...;n*n] *)
  |> sum               (*  0+1+4+...+n*n  *)
```

```
val ( -- ) : int -> int -> int list = <fun>
```

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

```
val sum : int list -> int = <fun>
```

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

The function `sum_sq` first constructs a list containing all the numbers `0..n`. Then it uses the pipeline operator `|>` to pass that list through `List.map square`, which squares every element. Then the resulting list is pipelined through `sum`, which adds all the elements together.

The other alternatives that you might consider are somewhat uglier:

```
(* Maybe worse: a lot of extra [let..in] syntax and unnecessary names to
   for intermediate values we don't care about. *)
let sum_sq n =
  let l = 0 -- n in
  let sq_l = List.map square l in
  sum sq_l

(* Maybe worse: have to read the function applications from right to left
   rather than top to bottom, and extra parentheses. *)
let sum_sq n =
  sum (List.map square (0--n))
```

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


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


The downside of all of these compared to the original tail recursive version is that they are wasteful of space—linear instead of constant—and take a constant factor more time. So as is so often the case in programming, there is a tradeoff between clarity and efficiency of code.

Note that the inefficiency is *not* from the pipeline operator itself, but from having to construct all those unnecessary intermediate lists. So don't get the idea that pipelining is intrinsically bad. In fact it can be quite useful. When we get to the chapter on modules, we'll use it quite often with some of the data structures we study there.

4.7 Currying

We've already seen that an OCaml function that takes two arguments of types `t1` and `t2` and returns a value of type `t3` has the type `t1 -> t2 -> t3`. We use two variables after the function name in the let expression:

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

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

Another way to define a function that takes two arguments is to write a function that takes a tuple:

```
let add' t = fst t + snd t
```

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

Instead of using `fst` and `snd`, we could use a tuple pattern in the definition of the function, leading to a third implementation:

```
let add'' (x, y) = x + y
```

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

Functions written using the first style (with type `t1 -> t2 -> t3`) are called *curried* functions, and functions using the second style (with type `t1 * t2 -> t3`) are called *uncurried*. Metaphorically, curried functions are "spicier" because you can partially apply them (something you can't do with uncurried functions: you can't pass in half of a pair). Actually, the term curry does not refer to spices, but to a logician named Haskell Curry (one of a very small set of people with programming languages named after both their first and last names).

Sometimes you will come across libraries that offer an uncurried version of a function, but you want a curried version of it to use in your own code; or vice versa. So it is useful to know how to convert between the two kinds of functions, as we did with `add` above.

You could even write a couple of higher-order functions to do the conversion for you:

```
let curry f x y = f (x, y)
let uncurry f (x, y) = f x y
```

```
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

```
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

```
let uncurried_add = uncurry add
let curried_add = curry add''
```

```
val uncurried_add : int * int -> int = <fun>
```

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

4.8 Summary

This chapter is one of the most important in the book. It didn't cover any new language features. Instead, we learned how to use some of the existing features in ways that might be new, surprising, or challenging. Higher-order programming and the Abstraction Principle are two ideas that will help make you a better programmer in any language, not just OCaml. Of course, languages do vary in the extent to which they support these ideas, with some providing significantly less assistance in writing higher-order code—which is one reason we use OCaml in this course.

Map, filter, fold and other functionals are becoming widely recognized as excellent ways to structure computation. Part of the reason for that is they factor out the *iteration* over a data structure from the *computation* done at each element. Languages such as Python, Ruby, and Java 8 now have support for this kind of iteration.

4.8.1 Terms and concepts

- Abstraction Principle
- accumulator
- apply
- associative
- compose
- factor
- filter
- first-order function
- fold
- functional
- generalized fold operation
- higher-order function
- map
- pipeline
- pipelining

4.8.2 Further reading

- *Introduction to Objective Caml*, chapters 3.1.3, 5.3
- *OCaml from the Very Beginning*, chapter 6
- *More OCaml: Algorithms, Methods, and Diversions*, chapter 1, by John Whitington. This book is a sequel to *OCaml from the Very Beginning*.
- *Real World OCaml*, chapter 3 (beware that this book's `Core` library has a different `List` module than the standard library's `List` module, with different types for `map` and `fold` than those we saw here)
- "Higher Order Functions", chapter 6 of *Functional Programming: Practice and Theory*. Bruce J. MacLennan, Addison-Wesley, 1990. Our discussion of higher-order functions and the Abstraction Principle is indebted to this chapter.
- "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." John Backus' 1977 Turing Award lecture in its elaborated form as a published article.
- "Second-order and Higher-order Logic" in *The Stanford Encyclopedia of Philosophy*.

4.9 Exercises

Exercise: twice, no arguments [★]

Consider the following definitions:

```
let double x = 2*x
let square x = x*x
let twice f x = f (f x)
let quad = twice double
let fourth = twice square
```

Use the toplevel to determine what the types of `quad` and `fourth` are. Explain how it can be that `quad` is not syntactically written as a function that takes an argument, and yet its type shows that it is in fact a function.

Exercise: mystery operator 1 [★★]

What does the following operator do?

```
let ( $ ) f x = f x
```

Hint: investigate `square $ 2 + 2` vs. `square 2 + 2`.

Exercise: mystery operator 2 [★★]

What does the following operator do?

```
let ( @@ ) f g x = x |> g |> f
```

Hint: investigate `String.length @@ string_of_int` applied to `1`, `10`, `100`, etc.

Exercise: repeat [★★]

Generalize `twice` to a function `repeat`, such that `repeat f n x` applies `f` to `x` a total of `n` times. That is,

- `repeat f 0 x` yields `x`
- `repeat f 1 x` yields `f x`
- `repeat f 2 x` yields `f (f x)` (which is the same as `twice f x`)
- `repeat f 3 x` yields `f (f (f x))`
- …

Exercise: product [★]

Use `fold_left` to write a function `product_left` that computes the product of a list of floats. The product of the empty list is `1.0`. *Hint: recall how we implemented `sum` in just one line of code in lecture.*

Use `fold_right` to write a function `product_right` that computes the product of a list of floats. *Same hint applies.*

Exercise: terse product [★★]

How terse can you make your solutions to the **product** exercise? *Hints: you need only one line of code for each, and you do not need the `fun` keyword. For `fold_left`, your function definition does not even need to explicitly take a list argument. If you use `ListLabels`, the same is true for `fold_right`.*

Exercise: sum_cube_odd [★★]

Write a function `sum_cube_odd n` that computes the sum of the cubes of all the odd numbers between `0` and `n` inclusive. Do not write any new recursive functions. Instead, use the functionals map, fold, and filter, and the `( -- )` operator (defined in the discussion of pipelining).

Exercise: sum_cube_odd pipeline [★★]

Rewrite the function `sum_cube_odd` to use the pipeline operator `|>`.

Exercise: exists [★★]

Consider writing a function `exists: ('a -> bool) -> 'a list -> bool`, such that `exists p [a1; ...; an]` returns whether at least one element of the list satisfies the predicate `p`. That is, it evaluates the same as `(p a1) || (p a2) || ... || (p an)`. When applied to an empty list, it evaluates to `false`.

Write three solutions to this problem, as we did above:

- `exists_rec`, which must be a recursive function that does not use the `List` module,
- `exists_fold`, which uses either `List.fold_left` or `List.fold_right`, but not any other `List` module functions nor the `rec` keyword, and
- `exists_lib`, which uses any combination of `List` module functions other than `fold_left` or `fold_right`, and does not use the `rec` keyword.

Exercise: account balance [★★★]

Write a function which, given a list of numbers representing debits, deducts them from an account balance, and finally returns the remaining amount in the balance. Write three versions: `fold_left`, `fold_right`, and a direct recursive implementation.

Exercise: library uncurried [★★]

Here is an uncurried version of `List.nth`:

```
let uncurried_nth (lst, n) = List.nth lst n
```

In a similar way, write uncurried versions of these library functions:

- `List.append`
- `Char.compare`
- `Stdlib.max`

Exercise: map composition [★★★]

Show how to replace any expression of the form `List.map f (List.map g lst)` with an equivalent expression that calls `List.map` only once.

Exercise: more list fun [★★★]

Write functions that perform the following computations. Each function that you write should use one of `List.fold`, `List.map` or `List.filter`. To choose which of those to use, think about what the computation is doing: combining, transforming, or filtering elements.

- Find those elements of a list of strings whose length is strictly greater than 3.
- Add `1.0` to every element of a list of floats.
- Given a list of strings `strs` and another string `sep`, produce the string that contains every element of `strs` separated by `sep`. For example, given inputs `["hi";"bye"]` and `","`, produce `"hi,bye"`, being sure not to produce an extra comma either at the beginning or end of the result string.

Exercise: association list keys [★★★]

Recall that an association list is an implementation of a dictionary in terms of a list of pairs, in which we treat the first component of each pair as a key and the second component as a value.

Write a function `keys: ('a * 'b) list -> 'a list` that returns a list of the unique keys in an association list. Since they must be unique, no value should appear more than once in the output list. The order of values output does not matter. How compact and efficient can you make your solution? Can you do it in one line and linearithmic space and time? *Hint:* *`List.sort_uniq`*.

Exercise: valid matrix [★★★]

A mathematical *matrix* can be represented with lists. In *row-major* representation, this matrix

$$\begin{bmatrix} 1 & 1 & 1 \\ 9 & 8 & 7 \end{bmatrix}$$

would be represented as the list `[[1; 1; 1]; [9; 8; 7]]`. Let's represent a *row vector* as an `int list`. For example, `[9; 8; 7]` is a row vector.

A *valid* matrix is an `int list list` that has at least one row, at least one column, and in which every column has the same number of rows. There are many values of type `int list list` that are invalid, for example,

- `[]`
- `[[1; 2]; [3]]`

Implement a function `is_valid_matrix: int list list -> bool` that returns whether the input matrix is valid. Unit test the function.

Exercise: row vector add [★★★]

Implement a function `add_row_vectors: int list -> int list -> int list` for the element-wise addition of two row vectors. For example, the addition of `[1; 1; 1]` and `[9; 8; 7]` is `[10; 9; 8]`. If the two vectors do not have the same number of entries, the behavior of your function is *unspecified*—that is, it may do whatever you like. *Hint: there is an elegant one-line solution using `List.map2`.* Unit test the function.

Exercise: matrix add [★★★]

Implement a function `add_matrices: int list list -> int list list -> int list list` for matrix addition. If the two input matrices are not the same size, the behavior is unspecified. *Hint: there is an elegant one-line solution using `List.map2` and `add_row_vectors`.* Unit test the function.

Exercise: matrix multiply [★★★★]

Implement a function `multiply_matrices: int list list -> int list list -> int list list` for matrix multiplication. If the two input matrices are not of sizes that can be multiplied together, the behavior is unspecified. Unit test the function. *Hint: define functions for matrix transposition and row vector dot product.*