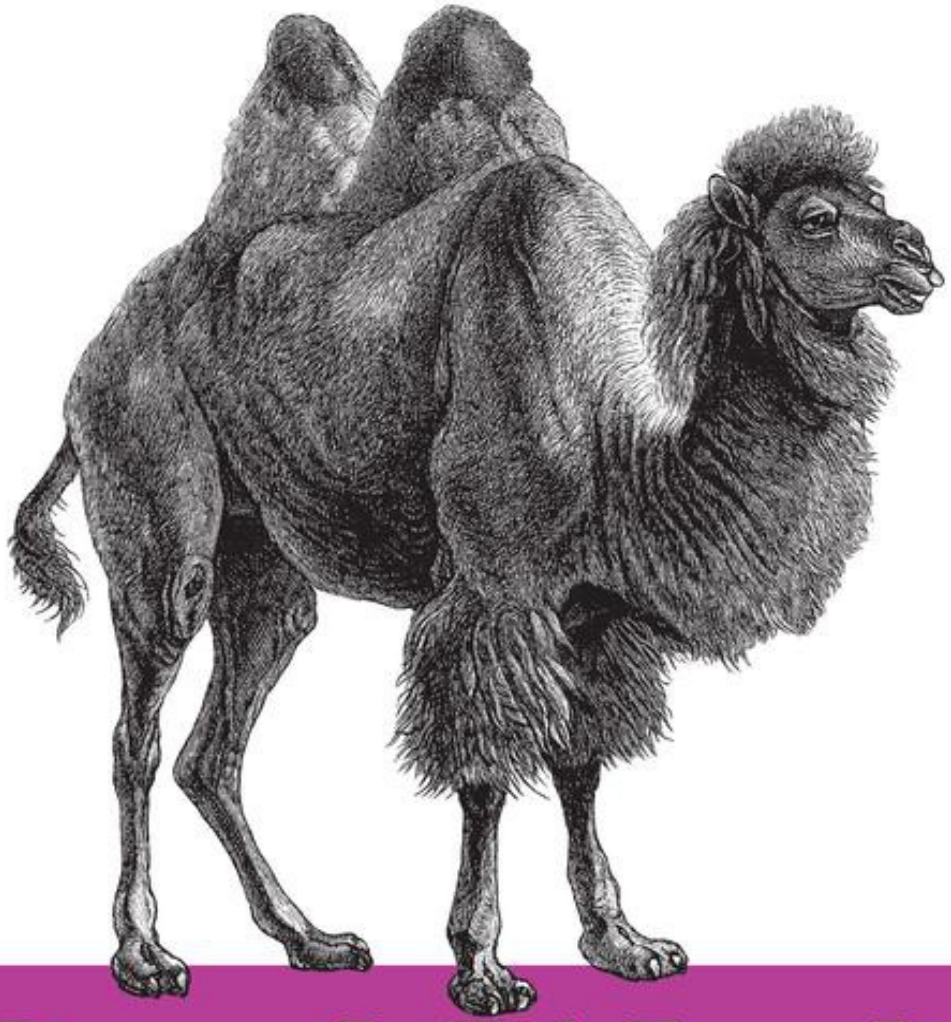
O'REILLY®



# Real World OCaml

FUNCTIONAL PROGRAMMING FOR THE MASSES

Yaron Minsky, Anil Madhavapeddy
& Jason Hickey

# Real World OCaml


## Yaron Minsky

## Anil Madhavapeddy

## Jason Hickey




Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

# Dedication

For Lisa, a believer in the power of words, who helps me find mine. — Yaron

For Mum and Dad, who took me to the library and unlocked my imagination. — Anil

For Nobu, who takes me on a new journey every day. — Jason

# Special Upgrade Offer

If you purchased this ebook directly from oreilly.com, you have the following benefits:

- DRM-free ebooks — use your ebooks across devices without restrictions or limitations
- Multiple formats — use on your laptop, tablet, or phone
- Lifetime access, with free updates
- Dropbox syncing — your files, anywhere

If you purchased this ebook from another retailer, you can upgrade your ebook to take advantage of all these benefits for just $4.99. Click here to access your ebook upgrade.

*Please note that upgrade offers are not available from sample content.*

# Prologue

# Why OCaml?

Programming languages matter. They affect the reliability, security, and efficiency of the code you write, as well as how easy it is to read, refactor, and extend. The languages you know can also change how you think, influencing the way you design software even when you're not using them.

We wrote this book because we believe in the importance of programming languages, and that OCaml in particular is an important language to learn. The three of us have been using OCaml in our academic and professional lives for over 15 years, and in that time we've come to see it as a secret weapon for building complex software systems. This book aims to make this secret weapon available to a wider audience, by providing a clear guide to what you need to know to use OCaml effectively in the real world.

What makes OCaml special is that it occupies a sweet spot in the space of programming language designs. It provides a combination of efficiency, expressiveness and practicality that is matched by no other language. That is in large part because OCaml is an elegant combination of a few key language features that have been developed over the last 40 years. These include:

- *Garbage collection* for automatic memory management, now a feature of almost every modern, high-level language.
- *First-class functions* that can be passed around like ordinary values, as seen in JavaScript, Common Lisp, and C#.
- *Static type-checking* to increase performance and reduce the number of runtime errors, as found in Java and C#.
- *Parametric polymorphism*, which enables the construction of abstractions that work across different data types, similar to generics in Java and C# and templates in C++.
- Good support for *immutable programming*, i.e., programming without making destructive updates to data structures. This is present in traditional functional languages like Scheme, and is also found in distributed, big-data frameworks like Hadoop.
- *Automatic type inference* to avoid having to laboriously define the type of every single variable in a program and instead have them inferred based on how a value is used. Available in a limited form in C# with implicitly typed local variables, and in C++11 with its `auto` keyword.
- *Algebraic data types* and *pattern matching* to define and manipulate complex data structures. Available in Scala and F#.

Some of you will know and love all of these features, and for others they will be largely new, but most of you will have seen *some* of them in other languages that you've used. As we'll demonstrate over the course of this book, there is something transformative about having them all together and able to interact in a single language. Despite their importance, these ideas have made only limited inroads into mainstream languages, and when they do arrive there, like first-class functions in C# or parametric polymorphism in Java, it's typically in a limited and awkward form. The only languages that completely embody these ideas are *statically typed, functional programming languages* like OCaml, F#, Haskell, Scala, and Standard ML.

Among this worthy set of languages, OCaml stands apart because it manages to provide a great deal of power while remaining highly pragmatic. The compiler has a straightforward compilation strategy that produces performant code without requiring heavy optimization and without the complexities of dynamic just-in-time (JIT) compilation. This, along with OCaml's strict evaluation model, makes

runtime behavior easy to predict. The garbage collector is *incremental*, letting you avoid large garbage collection (GC)-related pauses, and *precise*, meaning it will collect all unreferenced data (unlike many reference-counting collectors), and the runtime is simple and highly portable.

All of this makes OCaml a great choice for programmers who want to step up to a better programming language, and at the same time get practical work done.

## A Brief History

OCaml was written in 1996 by Xavier Leroy, Jérôme Vouillon, Damien Doligez, and Didier Rémy at INRIA in France. It was inspired by a long line of research into ML starting in the 1960s, and continues to have deep links to the academic community.

ML was originally the *meta language* of the LCF (Logic for Computable Functions) proof assistant released by Robin Milner in 1972 (at Stanford, and later at Cambridge). ML was turned into a compiler in order to make it easier to use LCF on different machines, and it was gradually turned into a full-fledged system of its own by the 1980s.

The first implementation of Caml appeared in 1987. It was created by Ascánder Suárez and later continued by Pierre Weis and Michel Mauny. In 1990, Xavier Leroy and Damien Doligez built a new implementation called Caml Light that was based on a bytecode interpreter with a fast, sequential garbage collector. Over the next few years useful libraries appeared, such as Michel Mauny's syntax manipulation tools, and this helped promote the use of Caml in education and research teams.

Xavier Leroy continued extending Caml Light with new features, which resulted in the 1995 release of Caml Special Light. This improved the executable efficiency significantly by adding a fast native code compiler that made Caml's performance competitive with mainstream languages such as C++. A module system inspired by Standard ML also provided powerful facilities for abstraction and made larger-scale programs easier to construct.

The modern OCaml emerged in 1996, when a powerful and elegant object system was implemented by Didier Rémy and Jérôme Vouillon. This object system was notable for supporting many common object-oriented idioms in a statically type-safe way, whereas the same idioms required runtime checks in languages such as C++ or Java. In 2000, Jacques Garrigue extended OCaml with several new features such as polymorphic methods, variants, and labeled and optional arguments.

The last decade has seen OCaml attract a significant user base, and language improvements have been steadily added to support the growing commercial and academic codebases. First-class modules, Generalized Algebraic Data Types (GADTs), and dynamic linking have improved the flexibility of the language. There is also fast native code support for x86_64, ARM, PowerPC, and Sparc, making OCaml a good choice for systems where resource usage, predictability, and performance all matter.

## The Core Standard Library

A language on its own isn't enough. You also need a rich set of libraries to base your applications on. A common source of frustration for those learning OCaml is that the standard library that ships with the compiler is limited, covering only a small subset of the functionality you would expect from a general-purpose standard library. That's because the standard library isn't a general-purpose tool; it was developed for use in bootstrapping the compiler and is purposefully kept small and simple.

Happily, in the world of open source software, nothing stops alternative libraries from being written to supplement the compiler-supplied standard library, and this is exactly what the Core distribution

is.

Jane Street, a company that has been using OCaml for more than a decade, developed Core for its own internal use, but designed it from the start with an eye toward being a general-purpose standard library. Like the OCaml language itself, Core is engineered with correctness, reliability, and performance in mind.

Core is distributed with syntax extensions that provide useful new functionality to OCaml, and there are additional libraries such as the Async network communications library that extend the reach of Core into building complex distributed systems. All of these libraries are distributed under a liberal Apache 2 license to permit free use in hobby, academic, and commercial settings.

## The OCaml Platform

Core is a comprehensive and effective standard library, but there's much more OCaml software out there. A large community of programmers has been using OCaml since its first release in 1996, and has generated many useful libraries and tools. We'll introduce some of these libraries in the course of the examples presented in the book.

The installation and management of these third-party libraries is made much easier via a package management tool known as OPAM. We'll explain more about OPAM as the book unfolds, but it forms the basis of the Platform, which is a set of tools and libraries that, along with the OCaml compiler, lets you build real-world applications quickly and effectively.

We'll also use OPAM for installing the *utop* command-line interface. This is a modern interactive tool that supports command history, macro expansion, module completion, and other niceties that make it much more pleasant to work with the language. We'll be using *utop* throughout the book to let you step through the examples interactively.

# About This Book

*Real World OCaml* is aimed at programmers who have some experience with conventional programming languages, but not specifically with statically typed functional programming. Depending on your background, many of the concepts we cover will be new, including traditional functional-programming techniques like higher-order functions and immutable data types, as well as aspects of OCaml's powerful type and module systems.

If you already know OCaml, this book may surprise you. Core redefines most of the standard namespace to make better use of the OCaml module system and expose a number of powerful, reusable data structures by default. Older OCaml code will still interoperate with Core, but you may need to adapt it for maximal benefit. All the new code that we write uses Core, and we believe the Core model is worth learning; it's been successfully used on large, multimillion-line codebases and removes a big barrier to building sophisticated applications in OCaml.

Code that uses only the traditional compiler standard library will always exist, but there are other online resources for learning how that works. *Real World OCaml* focuses on the techniques the authors have used in their personal experience to construct scalable, robust software systems.

## What to Expect

*Real World OCaml* is split into three parts:

- Part I covers the language itself, opening with a guided tour designed to provide a quick sketch of the language. Don't expect to understand everything in the tour; it's meant to give you a taste of many different aspects of the language, but the ideas covered there will be explained in more depth in the chapters that follow.
  After covering the core language, Part I then moves onto more advanced features like modules, functors, and objects, which may take some time to digest. Understanding these concepts is important, though. These ideas will put you in good stead even beyond OCaml when switching to other modern languages, many of which have drawn inspiration from ML.
- Part II builds on the basics by working through useful tools and techniques for addressing common practical applications, from command-line parsing to asynchronous network programming. Along the way, you'll see how some of the concepts from Part I are glued together into real libraries and tools that combine different features of the language to good effect.
- Part III discusses OCaml's runtime system and compiler toolchain. It is remarkably simple when compared to some other language implementations (such as Java's or .NET's CLR). Reading this part will enable you to build very-high-performance systems, or to interface with C libraries. This is also where we talk about profiling and debugging techniques using tools such as GNU *gdb*.

## Installation Instructions

*Real World OCaml* uses some tools that we've developed while writing this book. Some of these resulted in improvements to the OCaml compiler, which means that you will need to ensure that you have an up-to-date development environment (using the 4.01 version of the compiler). The installation process is largely automated through the OPAM package manager. Instructions on how to it set up and what packages to install can be found at this Real World OCaml page.

As of publication time, the Windows operating system is unsupported by Core, and so only Mac OS

X, Linux, FreeBSD, and OpenBSD can be expected to work reliably. Please check the online installation instructions for updates regarding Windows, or install a Linux virtual machine to work through the book as it stands.

This book is not intended as a reference manual. We aim to teach you about the language and about libraries tools and techniques that will help you be a more effective OCaml programmer. But it's no replacement for API documentation or the OCaml manual and man pages. You can find documentation for all of the libraries and tools referenced in the book online.

## Code Examples

All of the code examples in this book are available freely online under a public-domain-like license. You are most welcome to copy and use any of the snippets as you see fit in your own code, without any attribution or other restrictions on their use.

The code repository is available online at https://github.com/realworldocaml/examples. Every code snippet in the book has a clickable header that tells you the filename in that repository to find the source code, shell script, or ancillary data file that the snippet was sourced from.

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

# How to Contact Us

Please address comments and questions concerning this book to the publisher:


O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)


We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreil.ly/realworldOCaml

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at http://www.oreilly.com.

Find us on Facebook: http://facebook.com/oreilly

Follow us on Twitter: http://twitter.com/oreillymedia

Watch us on YouTube: http://www.youtube.com/oreillymedia

# Contributors

We would especially like to thank the following individuals for improving *Real World OCaml*:

- Leo White contributed greatly to the content and examples in Chapter 11 and Chapter 12.
- Jeremy Yallop authored and documented the Ctypes library described in Chapter 19.
- Stephen Weeks is responsible for much of the modular architecture behind Core, and his extensive notes formed the basis of Chapter 20 and Chapter 21.
- Jeremie Dimino, the author of *utop*, the interactive command-line interface that is used throughout this book. We're particularly grateful for the changes that he pushed through to make *utop* work better in the context of the book.
- The many people who collectively submitted over 2400 comments to online drafts of this book, through whose efforts countless errors were found and fixed.

# Part I. Language Concepts

Part I covers the basic language concepts you'll need to know when building OCaml programs. It opens up with a guided tour to give you a quick overview of the language using an interactive command-line interface. The subsequent chapters cover the material that is touched upon by the tour in much more detail, including detailed coverage of OCaml's approach to imperative programming.

The last few chapters introduce OCaml's powerful abstraction facilities. We start by using functors to build a library for programming with intervals, and then use first-class modules to build a type-safe plugin system. OCaml also supports object-oriented programming, and we close Part I with two chapters that cover the object system; the first showing how to use OCaml's objects directly, and the second showing how to use the class system to add more advanced features like inheritance. This description comes together in the design of a simple object-oriented graphics library.

# Chapter 1. A Guided Tour

This chapter gives an overview of OCaml by walking through a series of small examples that cover most of the major features of the language. This should provide a sense of what OCaml can do, without getting too deep into any one topic.

Throughout the book we're going to use Core, a more full-featured and capable replacement for OCaml's standard library. We'll also use *utop*, a shell that lets you type in expressions and evaluate them interactively. *utop* is an easier-to-use version of OCaml's standard toplevel (which you can start by typing *ocaml* at the command line). These instructions will assume you're using *utop* specifically.

Before getting started, make sure you have a working OCaml installation so you can try out the examples as you read through the chapter.

# OCaml as a Calculator

The first thing you need to do when using Core is to open `Core.Std`:

OCaml utop

```
$ utop

# open Core.Std;;
```

This makes the definitions in Core available and is required for many of the examples in the tour and in the remainder of the book.

Now let's try a few simple numerical calculations:

OCaml utop (part 1)

```
# 3 + 4;;
- : int = 7
# 8 / 3;;
- : int = 2
# 3.5 +. 6.;;
- : float = 9.5
# 30_000_000 / 300_000;;
- : int = 100
# sqrt 9.;;
- : float = 3.
```

By and large, this is pretty similar to what you'd find in any programming language, but a few things jump right out at you:

- We needed to type `;;` in order to tell the toplevel that it should evaluate an expression. This is a peculiarity of the toplevel that is not required in standalone programs (though it is sometimes helpful to include `;;` to improve OCaml's error reporting, by making it more explicit where a given top-level declaration was intended to end).
- After evaluating an expression, the toplevel first prints the type of the result, and then prints the result itself.
- Function arguments are separated by spaces instead of by parentheses and commas, which is more like the UNIX shell than it is like traditional programming languages such as C or Java.
- OCaml allows you to place underscores in the middle of numeric literals to improve readability. Note that underscores can be placed anywhere within a number, not just every three digits.
- OCaml carefully distinguishes between `float`, the type for floating-point numbers, and `int`, the type for integers. The types have different literals (`6.` instead of `6`) and different infix operators (`+.` instead of `+`), and OCaml doesn't automatically cast between these types. This can be a bit of a nuisance, but it has its benefits, since it prevents some kinds of bugs that arise in other languages due to unexpected differences between the behavior of `int` and `float`. For example, in many languages, `1 / 3` is zero, but `1 / 3.0` is a third. OCaml requires you to be explicit about which operation you're doing.

We can also create a variable to name the value of a given expression, using the `let` keyword. This is known as a *let binding*:

OCaml utop (part 2)

```
# let x = 3 + 4;;
val x : int = 7
# let y = x + x;;
```

```
val y : int = 14
```

After a new variable is created, the toplevel tells us the name of the variable (`x` or `y`), in addition to its type (`int`) and value (`7` or `14`).

Note that there are some constraints on what identifiers can be used for variable names. Punctuation is excluded, except for _ and ', and variables must start with a lowercase letter or an underscore. Thus, these are legal:

OCaml utop (part 3)

```
# let x7 = 3 + 4;;
val x7 : int = 7
# let x_plus_y = x + y;;
val x_plus_y : int = 21
# let x' = x + 1;;
val x' : int = 8
# let _x' = x' + x';;
# _x';;
- : int = 16
```

Note that by default, *utop* doesn't bother to print out variables starting with an underscore.

The following examples, however, are not legal:

OCaml utop (part 4)

```
# let Seven = 3 + 4;;
Characters 4-9:
Error: Unbound constructor Seven
# let 7x = 7;;
Characters 5-10:
Error: This expression should not be a function, the expected type is
int
# let x-plus-y = x + y;;
Characters 4-5:
Error: Parse error: [fun_binding] expected after [ipatt] (in [let_binding])
```

The error messages here are a little confusing, but they'll make more sense as you learn more about the language.

# Functions and Type Inference

The `let` syntax can also be used to define a function:

OCaml utop (part 5)

```
# let square x = x * x ;;
val square : int -> int = <fun>
# square 2;;
- : int = 4
# square (square 2);;
- : int = 16
```

Functions in OCaml are values like any other, which is why we use the `let` keyword to bind a function to a variable name, just as we use `let` to bind a simple value like an integer to a variable name. When using `let` to define a function, the first identifier after the `let` is the function name, and each subsequent identifier is a different argument to the function. Thus, `square` is a function with a single argument.

Now that we're creating more interesting values like functions, the types have gotten more interesting too. `int -> int` is a function type, in this case indicating a function that takes an `int` and returns an `int`. We can also write functions that take multiple arguments. (Note that the following example will not work if you haven't opened `Core.Std` as was suggested earlier.)

OCaml utop (part 6)

```
# let ratio x y =
    Float.of_int x /. Float.of_int y
  ;;
val ratio : int -> int -> float = <fun>
# ratio 4 7;;
- : float = 0.571428571429
```

The preceding example also happens to be our first use of modules. Here, `Float.of_int` refers to the `of_int` function contained in the `Float` module. This is different from what you might expect from an object-oriented language, where dot-notation is typically used for accessing a method of an object. Note that module names always start with a capital letter.

The notation for the type-signature of a multiargument function may be a little surprising at first, but we'll explain where it comes from when we get to function currying in Multiargument functions. For the moment, think of the arrows as separating different arguments of the function, with the type after the final arrow being the return value. Thus, `int -> int -> float` describes a function that takes two `int` arguments and returns a `float`.

We can also write functions that take other functions as arguments. Here's an example of a function that takes three arguments: a test function and two integer arguments. The function returns the sum of the integers that pass the test:

OCaml utop (part 7)

```
# let sum_if_true test first second =
    (if test first then first else 0)
    + (if test second then second else 0)
  ;;
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

If we look at the inferred type signature in detail, we see that the first argument is a function that takes an integer and returns a boolean, and that the remaining two arguments are integers. Here's an

example of this function in action:

OCaml utop (part 8)

```
# let even x =
    x mod 2 = 0 ;;
val even : int -> bool = <fun>
# sum_if_true even 3 4;;
- : int = 4
# sum_if_true even 2 4;;
- : int = 6
```

Note that in the definition of `even`, we used `=` in two different ways: once as the part of the `let` binding that separates the thing being defined from its definition; and once as an equality test, when comparing `x mod 2` to `0`. These are very different operations despite the fact that they share some syntax.

## Type Inference

As the types we encounter get more complicated, you might ask yourself how OCaml is able to figure them out, given that we didn't write down any explicit type information.

OCaml determines the type of an expression using a technique called *type inference*, by which the type of an expression is inferred from the available type information about the components of that expression.

As an example, let's walk through the process of inferring the type of `sum_if_true`:

1. OCaml requires that both branches of an `if` statement have the same type, so the expression `if test first then first else 0` requires that `first` must be the same type as `0`, and so `first` must be of type `int`. Similarly, from `if test second then second else 0` we can infer that `second` has type `int`.
2. `test` is passed `first` as an argument. Since `first` has type `int`, the input type of `test` must be `int`.
3. `test first` is used as the condition in an `if` statement, so the return type of `test` must be `bool`.
4. The fact that `+` returns `int` implies that the return value of `sum_if_true` must be int.

Together, that nails down the types of all the variables, which determines the overall type of `sum_if_true`.

Over time, you'll build a rough intuition for how the OCaml inference engine works, which makes it easier to reason through your programs. You can make it easier to understand the types of a given expression by adding explicit type annotations. These annotations don't change the behavior of an OCaml program, but they can serve as useful documentation, as well as catch unintended type changes. They can also be helpful in figuring out why a given piece of code fails to compile.

Here's an annotated version of `sum_if_true`:

OCaml utop (part 9)

```
# let sum_if_true (test : int -> bool) (x : int) (y : int) : int =
    (if test x then x else 0)
    + (if test y then y else 0)
  ;;
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

In the above, we've marked every argument to the function with its type, with the final annotation

indicating the type of the return value. Such type annotations can be placed on any expression in an OCaml program:

## Inferring Generic Types

Sometimes, there isn't enough information to fully determine the concrete type of a given value. Consider this function.

OCaml utop (part 10)

```
# let first_if_true test x y =
    if test x then x else y
  ;;
val first_if_true : ('a -> bool) -> 'a -> 'a -> 'a = <fun>
```

`first_if_true` takes as its arguments a function `test`, and two values, `x` and `y`, where `x` is to be returned if `test x` evaluates to `true`, and `y` otherwise. So what's the type of `first_if_true`? There are no obvious clues such as arithmetic operators or literals to tell you what the type of `x` and `y` are. That makes it seem like one could use `first_if_true` on values of any type.

Indeed, if we look at the type returned by the toplevel, we see that rather than choose a single concrete type, OCaml has introduced a *type variable* `'a` to express that the type is generic. (You can tell it's a type variable by the leading single quote mark.) In particular, the type of the `test` argument is `('a -> bool)`, which means that `test` is a one-argument function whose return value is `bool` and whose argument could be of any type `'a`. But, whatever type `'a` is, it has to be the same as the type of the other two arguments, `x` and `y`, and of the return value of `first_if_true`. This kind of genericity is called *parametric polymorphism* because it works by parameterizing the type in question with a type variable. It is very similar to generics in C# and Java.

The generic type of `first_if_true` allows us to write this:

OCaml utop (part 11)

```
# let long_string s = String.length s > 6;;
val long_string : string -> bool = <fun>
# first_if_true long_string "short" "looooooong";;
- : string = "looooooong"
```

As well as this:

OCaml utop (part 12)

```
# let big_number x = x > 3;;
val big_number : int -> bool = <fun>
# first_if_true big_number 4 3;;
- : int = 4
```

Both `long_string` and `big_number` are functions, and each is passed to `first_if_true` with two other arguments of the appropriate type (strings in the first example, and integers in the second). But we can't mix and match two different concrete types for `'a` in the same use of `first_if_true`:

OCaml utop (part 13)

```
# first_if_true big_number "short" "looooooong";;
Characters 25-32:
Error: This expression has type string but an expression was expected of type
         int
```

In this example, `big_number` requires that `'a` be instantiated as `int`, whereas `"short"` and `"looooooong"` require that `'a` be instantiated as `string`, and they can't both be right at the same time.

### TYPE ERRORS VERSUS EXCEPTIONS

There's a big difference in OCaml (and really in any compiled language) between errors that are caught at compile time and those that are caught at runtime. It's better to catch errors as early as possible in the development process, and compilation time is best of all.

Working in the toplevel somewhat obscures the difference between runtime and compile-time errors, but that difference is still there. Generally, type errors like this one:

OCaml utop (part 14)

```
# let add_potato x =
    x + "potato";;
Characters 28-36:
Error: This expression has type string but an expression was expected of type
         int
```

are compile-time errors (because `+` requires that both its arguments be of type `int`), whereas errors that can't be caught by the type system, like division by zero, lead to runtime exceptions:

OCaml utop (part 15)

```
# let is_a_multiple x y =
    x mod y = 0 ;;
val is_a_multiple : int -> int -> bool = <fun>
# is_a_multiple 8 2;;
- : bool = true
# is_a_multiple 8 0;;
Exception: Division_by_zero.
```

The distinction here is that type errors will stop you whether or not the offending code is ever actually executed. Merely defining `add_potato` is an error, whereas `is_a_multiple` only fails when it's called, and then, only when it's called with an input that triggers the exception.

# Tuples, Lists, Options, and Pattern Matching

## Tuples

So far we've encountered a handful of basic types like `int`, `float`, and `string`, as well as function types like `string -> int`. But we haven't yet talked about any data structures. We'll start by looking at a particularly simple data structure, the tuple. A tuple is an ordered collection of values that can each be of a different type. You can create a tuple by joining values together with a comma:

OCaml utop (part 16)

```
# let a_tuple = (3,"three");;
val a_tuple : int * string = (3, "three")
# let another_tuple = (3,"four",5.);;
val another_tuple : int * string * float = (3, "four", 5.)
```

(For the mathematically inclined, the `*` character is used because the set of all pairs of type `t * s` corresponds to the Cartesian product of the set of elements of type `t` and the set of elements of type `s`.)

You can extract the components of a tuple using OCaml's pattern-matching syntax, as shown below:

OCaml utop (part 17)

```
# let (x,y) = a_tuple;;
val x : int = 3
val y : string = "three"
```

Here, the `(x,y)` on the lefthand side of the `let` binding is the pattern. This pattern lets us mint the new variables `x` and `y`, each bound to different components of the value being matched. These can now be used in subsequent expressions:

OCaml utop (part 18)

```
# x + String.length y;;
- : int = 8
```

Note that the same syntax is used both for constructing and for pattern matching on tuples.

Pattern matching can also show up in function arguments. Here's a function for computing the distance between two points on the plane, where each point is represented as a pair of `float`s. The pattern-matching syntax lets us get at the values we need with a minimum of fuss:

OCaml utop (part 19)

```
# let distance (x1,y1) (x2,y2) =
    sqrt ((x1 -. x2) ** 2. +. (y1 -. y2) ** 2.)
  ;;
val distance : float * float -> float * float -> float = <fun>
```

The `**` operator used above is for raising a floating-point number to a power.

This is just a first taste of pattern matching. Pattern matching is a pervasive tool in OCaml, and as you'll see, it has surprising power.

## Lists

Where tuples let you combine a fixed number of items, potentially of different types, lists let you hold any number of items of the same type. Consider the following example:

OCaml utop (part 20)

```
# let languages = ["OCaml";"Perl";"C"];;
val languages : string list = ["OCaml"; "Perl"; "C"]
```

Note that you can't mix elements of different types in the same list, unlike tuples:

OCaml utop (part 21)

```
# let numbers = [3;"four";5];;
Characters 17-23:
Error: This expression has type string but an expression was expected of type
         int
```

### The List module

Core comes with a `List` module that has a rich collection of functions for working with lists. We can access values from within a module by using dot notation. For example, this is how we compute the length of a list:

OCaml utop (part 22)

```
# List.length languages;;
- : int = 3
```

Here's something a little more complicated. We can compute the list of the lengths of each language as follows:

OCaml utop (part 23)

```
# List.map languages ~f:String.length;;
- : int list = [5; 4; 1]
```

`List.map` takes two arguments: a list and a function for transforming the elements of that list. It returns a new list with the transformed elements and does not modify the original list.

Notably, the function passed to `List.map` is passed under a *labeled argument* `~f`. Labeled arguments are specified by name rather than by position, and thus allow you to change the order in which arguments are presented to a function without changing its behavior, as you can see here:

OCaml utop (part 24)

```
# List.map ~f:String.length languages;;
- : int list = [5; 4; 1]
```

We'll learn more about labeled arguments and why they're important in Chapter 2.

### Constructing lists with ::

In addition to constructing lists using brackets, we can use the operator `::` for adding elements to the front of a list:

OCaml utop (part 25)

```
# "French" :: "Spanish" :: languages;;
- : string list = ["French"; "Spanish"; "OCaml"; "Perl"; "C"]
```

Here, we're creating a new and extended list, not changing the list we started with, as you can see below:

OCaml utop (part 26)

```
# languages;;
- : string list = ["OCaml"; "Perl"; "C"]
```

**SEMICOLONS VERSUS COMMAS**

Unlike many other languages, OCaml uses semicolons to separate list elements in lists rather than commas. Commas, instead, are used for separating elements in a tuple. If you try to use commas in a list, you'll see that your code compiles but doesn't do quite what you might expect:

OCaml utop (part 27)

```
# ["OCaml", "Perl", "C"];;
- : (string * string * string) list = [("OCaml", "Perl", "C")]
```

In particular, rather than a list of three strings, what we have is a singleton list containing a three-tuple of strings.

This example uncovers the fact that commas create a tuple, even if there are no surrounding parens. So, we can write:

OCaml utop (part 28)

```
# 1,2,3;;
- : int * int * int = (1, 2, 3)
```

to allocate a tuple of integers. This is generally considered poor style and should be avoided.

The bracket notation for lists is really just syntactic sugar for `::`. Thus, the following declarations are all equivalent. Note that `[]` is used to represent the empty list and that `::` is right-associative:

OCaml utop (part 29)

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

The `::` operator can only be used for adding one element to the front of the list, with the list terminating at `[]`, the empty list. There's also a list concatenation operator, `@`, which can concatenate two lists:

OCaml utop (part 30)

```
# [1;2;3] @ [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

It's important to remember that, unlike `::`, this is not a constant-time operation. Concatenating two lists takes time proportional to the length of the first list.

### List patterns using match

The elements of a list can be accessed through pattern matching. List patterns are based on the two list constructors, `[]` and `::`. Here's a simple example:

OCaml utop (part 31)

```
# let my_favorite_language (my_favorite :: the_rest) =
    my_favorite
  ;;
Characters 25-69:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val my_favorite_language : 'a list -> 'a = <fun>
```

By pattern matching using `::`, we've isolated and named the first element of the list (`my_favorite`) and the remainder of the list (`the_rest`). If you know Lisp or Scheme, what we've done is the

equivalent of using the functions `car` and `cdr` to isolate the first element of a list and the remainder of that list.

As you can see, however, the toplevel did not like this definition and spit out a warning indicating that the pattern is not exhaustive. This means that there are values of the type in question that won't be captured by the pattern. The warning even gives an example of a value that doesn't match the provided pattern, in particular, `[]`, the empty list. If we try to run `my_favorite_language`, we'll see that it works on nonempty list and fails on empty ones:

OCaml utop (part 32)

```
# my_favorite_language ["English";"Spanish";"French"];;
- : string = "English"
# my_favorite_language [];;
Exception: (Match_failure //toplevel// 0 25).
```

You can avoid these warnings, and more importantly make sure that your code actually handles all of the possible cases, by using a `match` statement instead.

A `match` statement is a kind of juiced-up version of the `switch` statement found in C and Java. It essentially lets you list a sequence of patterns, separated by pipe characters (|). (The one before the first case is optional.) The compiler then dispatches to the code following the first matching pattern. As we've already seen, the pattern can mint new variables that correspond to substructures of the value being matched.

Here's a new version of `my_favorite_language` that uses `match` and doesn't trigger a compiler warning:

OCaml utop (part 33)

```
# let my_favorite_language languages =
    match languages with
    | first :: the_rest -> first
    | [] -> "OCaml" (* A good default! *)
 ;;
val my_favorite_language : string list -> string = <fun>
# my_favorite_language ["English";"Spanish";"French"];;
- : string = "English"
# my_favorite_language [];;
- : string = "OCaml"
```

The preceding code also includes our first comment. OCaml comments are bounded by `(*` and `*)` and can be nested arbitrarily and cover multiple lines. There's no equivalent of C++-style single-line comments that are prefixed by `//`.

The first pattern, `first :: the_rest`, covers the case where `languages` has at least one element, since every list except for the empty list can be written down with one or more `::`'s. The second pattern, `[]`, matches only the empty list. These cases are exhaustive, since every list is either empty or has at least one element, a fact that is verified by the compiler.

### Recursive list functions

Recursive functions, or functions that call themselves, are an important technique in OCaml and in any functional language. The typical approach to designing a recursive function is to separate the logic into a set of *base cases* that can be solved directly and a set of *inductive cases*, where the function breaks the problem down into smaller pieces and then calls itself to solve those smaller problems.

When writing recursive list functions, this separation between the base cases and the inductive cases is often done using pattern matching. Here's a simple example of a function that sums the elements of a list:

OCaml utop (part 34)

```
# let rec sum l =
    match l with
    | [] -> 0                    (* base case *)
    | hd :: tl -> hd + sum tl    (* inductive case *)
  ;;
val sum : int list -> int = <fun>
# sum [1;2;3];;
- : int = 6
```

Following the common OCaml idiom, we use `hd` to refer to the head of the list and `tl` to refer to the tail. Note that we had to use the `rec` keyword to allow `sum` to refer to itself. As you might imagine, the base case and inductive case are different arms of the match.

Logically, you can think of the evaluation of a simple recursive function like `sum` almost as if it were a mathematical equation whose meaning you were unfolding step by step:

OCaml: guided-tour/recursion.ml

```
sum [1;2;3]
= 1 + sum [2;3]
= 1 + (2 + sum [3])
= 1 + (2 + (3 + sum []))
= 1 + (2 + (3 + 0))
= 1 + (2 + 3)
= 1 + 5
= 6
```

This suggests a reasonable mental model for what OCaml is actually doing to evaluate a recursive function.

We can introduce more complicated list patterns as well. Here's a function for removing sequential duplicates:

OCaml utop (part 35)

```
# let rec destutter list =
    match list with
    | [] -> []
    | hd1 :: hd2 :: tl ->
      if hd1 = hd2 then destutter (hd2 :: tl)
      else hd1 :: destutter (hd2 :: tl)
  ;;
Characters 29-171:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
_::[]
val destutter : 'a list -> 'a list = <fun>
```

Again, the first arm of the match is the base case, and the second is the inductive. Unfortunately, this code has a problem, as is indicated by the warning message. In particular, we don't handle one-element lists. We can fix this warning by adding another case to the match:

OCaml utop (part 36)

```
# let rec destutter list =
    match list with
    | [] -> []
```

```
    | [hd] -> [hd]
    | hd1 :: hd2 :: tl ->
      if hd1 = hd2 then destutter (hd2 :: tl)
      else hd1 :: destutter (hd2 :: tl)
  ;;
val destutter : 'a list -> 'a list = <fun>
# destutter ["hey";"hey";"hey";"man!"];;
- : string list = ["hey"; "man!"]
```

Note that this code used another variant of the list pattern, `[hd]`, to match a list with a single element. We can do this to match a list with any fixed number of elements; for example, `[x;y;z]` will match any list with exactly three elements and will bind those elements to the variables `x`, `y`, and `z`.

In the last few examples, our list processing code involved a lot of recursive functions. In practice, this isn't usually necessary. Most of the time, you'll find yourself happy to use the iteration functions found in the `List` module. But it's good to know how to use recursion when you need to do something new.

## Options

Another common data structure in OCaml is the option. An option is used to express that a value might or might not be present. For example:

OCaml utop (part 37)

```
# let divide x y =
    if y = 0 then None else Some (x/y) ;;
val divide : int -> int -> int option = <fun>
```

The function `divide` either returns `None` if the divisor is zero, or `Some` of the result of the division otherwise. `Some` and `None` are constructors that let you build optional values, just as `::` and `[]` let you build lists. You can think of an option as a specialized list that can only have zero or one elements.

To examine the contents of an option, we use pattern matching, as we did with tuples and lists. Consider the following function for creating a log entry string given an optional time and a message. If no time is provided (i.e., if the time is `None`), the current time is computed and used in its place:

OCaml utop (part 38)

```
# let log_entry maybe_time message =
    let time =
      match maybe_time with
      | Some x -> x
      | None -> Time.now ()
    in
    Time.to_sec_string time ^ " -- " ^ message
  ;;
val log_entry : Time.t option -> string -> string = <fun>
# log_entry (Some Time.epoch) "A long long time ago";;
- : string = "1970-01-01 01:00:00 -- A long long time ago"
# log_entry None "Up to the minute";;
- : string = "2013-08-18 14:48:08 -- Up to the minute"
```

This example uses Core's `Time` module for dealing with time, as well as the `^` operator for concatenating strings. The concatenation operator is provided as part of the `Pervasives` module, which is automatically opened in every OCaml program.

### NESTING LETS WITH LET AND IN

`log_entry` was our first use of `let` to define a new variable within the body of a function. A `let` paired with an `in` can be used to

introduce a new binding within any local scope, including a function body. The `in` marks the beginning of the scope within which the new variable can be used. Thus, we could write:

OCaml utop

```
# let x = 7 in
  x + x
  ;;
- : int = 14
```

Note that the scope of the `let` binding is terminated by the double-semicolon, so the value of `x` is no longer available:

OCaml utop (part 1)

```
# x;;
Characters -1-1:
Error: Unbound value x
```

We can also have multiple `let` statements in a row, each one adding a new variable binding to what came before:

OCaml utop (part 2)

```
# let x = 7 in
  let y = x * x in
  x + y
  ;;
- : int = 56
```

This kind of nested `let` binding is a common way of building up a complex expression, with each `let` naming some component, before combining them in one final expression.

Options are important because they are the standard way in OCaml to encode a value that might not be there; there's no such thing as a `NullPointerException` in OCaml. This is different from most other languages, including Java and C#, where most if not all data types are *nullable*, meaning that, whatever their type is, any given value also contains the possibility of being a null value. In such languages, null is lurking everywhere.

In OCaml, however, missing values are explicit. A value of type `string * string` always contains two well-defined values of type `string`. If you want to allow, say, the first of those to be absent, then you need to change the type to `string option * string`. As we'll see in Chapter 7, this explicitness allows the compiler to provide a great deal of help in making sure you're correctly handling the possibility of missing data.

# Records and Variants

So far, we've only looked at data structures that were predefined in the language, like lists and tuples. But OCaml also allows us to define new data types. Here's a toy example of a data type representing a point in two-dimensional space:

OCaml utop (part 41)

```
# type point2d = { x : float; y : float };;
type point2d = { x : float; y : float; }
```

`point2d` is a *record* type, which you can think of as a tuple where the individual fields are named, rather than being defined positionally. Record types are easy enough to construct:

OCaml utop (part 42)

```
# let p = { x = 3.; y = -4. };;
val p : point2d = {x = 3.; y = -4.}
```

And we can get access to the contents of these types using pattern matching:

OCaml utop (part 43)

```
# let magnitude { x = x_pos; y = y_pos } =
    sqrt (x_pos ** 2. +. y_pos ** 2.);;
val magnitude : point2d -> float = <fun>
```

The pattern match here binds the variable `x_pos` to the value contained in the `x` field, and the variable `y_pos` to the value in the `y` field.

We can write this more tersely using what's called *field punning*. In particular, when the name of the field and the name of the variable it is bound to coincide, we don't have to write them both down. Using this, our magnitude function can be rewritten as follows:

OCaml utop (part 44)

```
# let magnitude { x; y } = sqrt (x ** 2. +. y ** 2.);;
val magnitude : point2d -> float = <fun>
```

Alternatively, we can use dot notation for accessing record fields:

OCaml utop (part 45)

```
# let distance v1 v2 =
    magnitude { x = v1.x -. v2.x; y = v1.y -. v2.y };;
val distance : point2d -> point2d -> float = <fun>
```

And we can of course include our newly defined types as components in larger types. Here, for example, are some types for modeling different geometric objects that contain values of type `point2d`:

OCaml utop (part 46)

```
# type circle_desc  = { center: point2d; radius: float }
  type rect_desc    = { lower_left: point2d; width: float; height: float }
  type segment_desc = { endpoint1: point2d; endpoint2: point2d } ;;
type circle_desc = { center : point2d; radius : float; }
type rect_desc = { lower_left : point2d; width : float; height : float; }
type segment_desc = { endpoint1 : point2d; endpoint2 : point2d; }
```

Now, imagine that you want to combine multiple objects of these types together as a description of a multiobject scene. You need some unified way of representing these objects together in a single type.

One way of doing this is using a *variant* type:

OCaml utop (part 47)

```
# type scene_element =
    | Circle  of circle_desc
    | Rect    of rect_desc
    | Segment of segment_desc
  ;;
type scene_element =
    Circle of circle_desc
  | Rect of rect_desc
  | Segment of segment_desc
```

The `|` character separates the different cases of the variant (the first `|` is optional), and each case has a capitalized tag, like `Circle`, `Rect` or `Segment`, to distinguish that case from the others.

Here's how we might write a function for testing whether a point is in the interior of some element of a list of `scene_element`s:

OCaml utop (part 48)

```
# let is_inside_scene_element point scene_element =
    match scene_element with
    | Circle { center; radius } ->
      distance center point < radius
    | Rect { lower_left; width; height } ->
      point.x    > lower_left.x && point.x < lower_left.x +. width
      && point.y > lower_left.y && point.y < lower_left.y +. height
    | Segment { endpoint1; endpoint2 } -> false
  ;;
val is_inside_scene_element : point2d -> scene_element -> bool = <fun>
# let is_inside_scene point scene =
    List.exists scene
      ~f:(fun el -> is_inside_scene_element point el)
  ;;
val is_inside_scene : point2d -> scene_element list -> bool = <fun>
# is_inside_scene {x=3.;y=7.}
    [ Circle {center = {x=4.;y= 4.}; radius = 0.5 } ];;
- : bool = false
# is_inside_scene {x=3.;y=7.}
    [ Circle {center = {x=4.;y= 4.}; radius = 5.0 } ];;
- : bool = true
```

You might at this point notice that the use of `match` here is reminiscent of how we used `match` with `option` and `list`. This is no accident: `option` and `list` are really just examples of variant types that happen to be important enough to be defined in the standard library (and in the case of lists, to have some special syntax).

We also made our first use of an *anonymous function* in the call to `List.exists`. Anonymous functions are declared using the `fun` keyword, and don't need to be explicitly named. Such functions are common in OCaml, particularly when using iteration functions like `List.exists`.

The purpose of `List.exists` is to check if there are any elements of the list in question on which the provided function evaluates to `true`. In this case, we're using `List.exists` to check if there is a scene element within which our point resides.

# Imperative Programming

The code we've written so far has been almost entirely *pure* or *functional*, which roughly speaking means that the code in question doesn't modify variables or values as part of its execution. Indeed, almost all of the data structures we've encountered are *immutable*, meaning there's no way in the language to modify them at all. This is a quite different style from *imperative* programming, where computations are structured as sequences of instructions that operate by making modifications to the state of the program.

Functional code is the default in OCaml, with variable bindings and most data structures being immutable. But OCaml also has excellent support for imperative programming, including mutable data structures like arrays and hash tables, and control-flow constructs like `for` and `while` loops.

## Arrays

Perhaps the simplest mutable data structure in OCaml is the array. Arrays in OCaml are very similar to arrays in other languages like C: indexing starts at 0, and accessing or modifying an array element is a constant-time operation. Arrays are more compact in terms of memory utilization than most other data structures in OCaml, including lists. Here's an example:

OCaml utop (part 49)

```
# let numbers = [| 1; 2; 3; 4 |];;
val numbers : int array = [|1; 2; 3; 4|]
# numbers.(2) <- 4;;
- : unit = ()
# numbers;;
- : int array = [|1; 2; 4; 4|]
```

The `.(i)` syntax is used to refer to an element of an array, and the `<-` syntax is for modification. Because the elements of the array are counted starting at zero, element `.(2)` is the third element.

The `unit` type that we see in the preceding code is interesting in that it has only one possible value, written `()`. This means that a value of type `unit` doesn't convey any information, and so is generally used as a placeholder. Thus, we use `unit` for the return value of an operation like setting a mutable field that communicates by side effect rather than by returning a value. It's also used as the argument to functions that don't require an input value. This is similar to the role that `void` plays in languages like C and Java.

## Mutable Record Fields

The array is an important mutable data structure, but it's not the only one. Records, which are immutable by default, can have some of their fields explicitly declared as mutable. Here's a small example of a data structure for storing a running statistical summary of a collection of numbers.

OCaml utop (part 50)

```
# type running_sum =
    { mutable sum: float;
      mutable sum_sq: float; (* sum of squares *)
      mutable samples: int;
    }
  ;;
type running_sum = {
  mutable sum : float;
  mutable sum_sq : float;
  mutable samples : int;
```

```
}
```

The fields in `running_sum` are designed to be easy to extend incrementally, and sufficient to compute means and standard deviations, as shown in the following example. Note that there are two `let` bindings in a row without a double semicolon between them. That's because the double semicolon is required only to tell *utop* to process the input, not to separate two declarations:

OCaml utop (part 51)

```
# let mean rsum = rsum.sum /. float rsum.samples
  let stdev rsum =
     sqrt (rsum.sum_sq /. float rsum.samples
           -. (rsum.sum /. float rsum.samples) ** 2.) ;;
val mean : running_sum -> float = <fun>
val stdev : running_sum -> float = <fun>
```

We use the function `float` above, which is a convenient equivalent of `Float.of_int` provided by the `Pervasives` library.

We also need functions to create and update `running_sum`s:

OCaml utop (part 52)

```
# let create () = { sum = 0.; sum_sq = 0.; samples = 0 }
  let update rsum x =
     rsum.samples <- rsum.samples + 1;
     rsum.sum     <- rsum.sum     +. x;
     rsum.sum_sq  <- rsum.sum_sq  +. x *. x
  ;;
val create : unit -> running_sum = <fun>
val update : running_sum -> float -> unit = <fun>
```

`create` returns a `running_sum` corresponding to the empty set, and `update rsum x` changes `rsum` to reflect the addition of `x` to its set of samples by updating the number of samples, the sum, and the sum of squares.

Note the use of single semicolons to sequence operations. When we were working purely functionally, this wasn't necessary, but you start needing it when you're writing imperative code.

Here's an example of `create` and `update` in action. Note that this code uses `List.iter`, which calls the function `~f` on each element of the provided list:

OCaml utop (part 53)

```
# let rsum = create ();;
val rsum : running_sum = {sum = 0.; sum_sq = 0.; samples = 0}
# List.iter [1.;3.;2.;-7.;4.;5.] ~f:(fun x -> update rsum x);;
- : unit = ()
# mean rsum;;
- : float = 1.33333333333
# stdev rsum;;
- : float = 3.94405318873
```

It's worth noting that the preceding algorithm is numerically naive and has poor precision in the presence of cancellation. You can look at this Wikipedia article on algorithms for calculating variance for more details, paying particular attention to the weighted incremental and parallel algorithms.

## Refs

We can create a single mutable value by using a `ref`. The `ref` type comes predefined in the standard

library, but there's nothing really special about it. It's just a record type with a single mutable field called `contents`:

OCaml utop (part 54)

```
# let x = { contents = 0 };;
val x : int ref = {contents = 0}
# x.contents <- x.contents + 1;;
- : unit = ()
# x;;
- : int ref = {contents = 1}
```

There are a handful of useful functions and operators defined for `ref`s to make them more convenient to work with:

OCaml utop (part 55)

```
# let x = ref 0  (* create a ref, i.e., { contents = 0 } *) ;;
val x : int ref = {contents = 0}
# !x             (* get the contents of a ref, i.e., x.contents *) ;;
- : int = 0
# x := !x + 1    (* assignment, i.e., x.contents <- ... *) ;;
- : unit = ()
# !x ;;
- : int = 1
```

There's nothing magical with these operators either. You can completely reimplement the `ref` type and all of these operators in just a few lines of code:

OCaml utop (part 56)

```
# type 'a ref = { mutable contents : 'a }

  let ref x = { contents = x }
  let (!) r = r.contents
  let (:=) r x = r.contents <- x
  ;;
type 'a ref = { mutable contents : 'a; }
val ref : 'a -> 'a ref = <fun>
val ( ! ) : 'a ref -> 'a = <fun>
val ( := ) : 'a ref -> 'a -> unit = <fun>
```

The `'a` before the `ref` indicates that the `ref` type is polymorphic, in the same way that lists are polymorphic, meaning it can contain values of any type. The parentheses around `!` and `:=` are needed because these are operators, rather than ordinary functions.

Even though a `ref` is just another record type, it's important because it is the standard way of simulating the traditional mutable variables you'll find in most languages. For example, we can sum over the elements of a list imperatively by calling `List.iter` to call a simple function on every element of a list, using a `ref` to accumulate the results:

OCaml utop (part 57)

```
# let sum list =
    let sum = ref 0 in
    List.iter list ~f:(fun x -> sum := !sum + x);
    !sum
  ;;
val sum : int list -> int = <fun>
```

This isn't the most idiomatic way to sum up a list, but it shows how you can use a `ref` in place of a mutable variable.

## For and While Loops

OCaml also supports traditional imperative control-flow constructs like `for and while` loops. Here, for example, is some code for permuting an array that uses a `for` loop. We use the `Random` module as our source of randomness. `Random` starts with a default seed, but you can call `Random.self_init` to choose a new seed at random:

OCaml utop (part 58)

```
# let permute array =
    let length = Array.length array in
    for i = 0 to length - 2 do
      (* pick a j to swap with *)
      let j = i + Random.int (length - i) in
      (* Swap i and j *)
      let tmp = array.(i) in
      array.(i) <- array.(j);
      array.(j) <- tmp
    done
  ;;
val permute : 'a array -> unit = <fun>
```

From a syntactic perspective, you should note the keywords that distinguish a `for` loop: `for`, `to`, `do`, and `done`.

Here's an example run of this code:

OCaml utop (part 59)

```
# let ar = Array.init 20 ~f:(fun i -> i);;
val ar : int array =
  [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19|]
# permute ar;;
- : unit = ()
# ar;;
- : int array =
[|1; 2; 4; 6; 11; 7; 14; 9; 10; 0; 13; 16; 19; 12; 17; 5; 3; 18; 8; 15|]
```

OCaml also supports `while` loops, as shown in the following function for finding the position of the first negative entry in an array. Note that `while` (like `for`) is also a keyword:

OCaml utop (part 60)

```
# let find_first_negative_entry array =
    let pos = ref 0 in
    while !pos < Array.length array && array.(!pos) >= 0 do
      pos := !pos + 1
    done;
    if !pos = Array.length array then None else Some !pos
  ;;
val find_first_negative_entry : int array -> int option = <fun>
# find_first_negative_entry [|1;2;0;3|];;
- : int option = None
# find_first_negative_entry [|1;-2;0;3|];;
- : int option = Some 1
```

As a side note, the preceding code takes advantage of the fact that `&&`, OCaml's And operator, short-circuits. In particular, in an expression of the form *`expr1 && expr2`*, *`expr2`* will only be evaluated if *`expr1`* evaluated to true. Were it not for that, then the preceding function would result in an out-of-bounds error. Indeed, we can trigger that out-of-bounds error by rewriting the function to avoid the short-circuiting:

# OCaml utop (part 61)

```
# let find_first_negative_entry array =
    let pos = ref 0 in
    while
      let pos_is_good = !pos < Array.length array in
      let element_is_non_negative = array.(!pos) >= 0 in
      pos_is_good && element_is_non_negative
    do
      pos := !pos + 1
    done;
    if !pos = Array.length array then None else Some !pos
  ;;
val find_first_negative_entry : int array -> int option = <fun>
# find_first_negative_entry [|1;2;0;3|];;
Exception: (Invalid_argument "index out of bounds").
```

The Or operator, `||`, short-circuits in a similar way to `&&`.

# A Complete Program

So far, we've played with the basic features of the language via *utop*. Now we'll show how to create a simple standalone program. In particular, we'll create a program that sums up a list of numbers read in from the standard input.

Here's the code, which you can save in a file called *sum.ml*. Note that we don't terminate expressions with `;;` here, since it's not required outside the toplevel:

OCaml

```
open Core.Std

let rec read_and_accumulate accum =
  let line = In_channel.input_line In_channel.stdin in
  match line with
  | None -> accum
  | Some x -> read_and_accumulate (accum +. Float.of_string x)

let () =
  printf "Total: %F\n" (read_and_accumulate 0.)
```

This is our first use of OCaml's input and output routines. The function `read_and_accumulate` is a recursive function that uses `In_channel.input_line` to read in lines one by one from the standard input, invoking itself at each iteration with its updated accumulated sum. Note that `input_line` returns an optional value, with `None` indicating the end of the input stream.

After `read_and_accumulate` returns, the total needs to be printed. This is done using the `printf` command, which provides support for type-safe format strings, similar to what you'll find in a variety of languages. The format string is parsed by the compiler and used to determine the number and type of the remaining arguments that are required. In this case, there is a single formatting directive, `%F`, so `printf` expects one additional argument of type `float`.

## Compiling and Running

We'll compile our program using *corebuild*, a small wrapper on top of *ocamlbuild*, a build tool that ships with the OCaml compiler. The *corebuild* script is installed along with Core, and its purpose is to pass in the flags required for building a program with Core.

Terminal

```
$ corebuild sum.native
```

The `.native` suffix indicates that we're building a native-code executable, which we'll discuss more in Chapter 4. Once the build completes, we can use the resulting program like any command-line utility. We can feed input to `sum.native` by typing in a sequence of numbers, one per line, hitting **`Ctrl-D`** when we're done:

Terminal

```
$ ./sum.native
1
2
3
94.5
Total: 100.5
```

More work is needed to make a really usable command-line program, including a proper command-

line parsing interface and better error handling, all of which is covered in Chapter 14.

# Where to Go from Here

That’s it for the guided tour! There are plenty of features left and lots of details to explain, but we hope that you now have a sense of what to expect from OCaml, and that you’ll be more comfortable reading the rest of the book as a result.

# Chapter 2. Variables and Functions

Variables and functions are fundamental ideas that show up in virtually all programming languages. OCaml has a different take on these concepts than most languages you're likely to have encountered, so this chapter will cover OCaml's approach to variables and functions in some detail, starting with the basics of how to define a variable, and ending with the intricacies of functions with labeled and optional arguments.

Don't be discouraged if you find yourself overwhelmed by some of the details, especially toward the end of the chapter. The concepts here are important, but if they don't connect for you on your first read, you should return to this chapter after you've gotten a better sense for the rest of the language.

# Variables

At its simplest, a variable is an identifier whose meaning is bound to a particular value. In OCaml these bindings are often introduced using the `let` keyword. We can type a so-called *top-level* `let` binding with the following syntax. Note that variable names must start with a lowercase letter or an underscore:

### Syntax

```
let <variable> = <expr>
```

As we'll see when we get to the module system in Chapter 4, this same syntax is used for `let` bindings at the top level of a module.

Every variable binding has a *scope*, which is the portion of the code that can refer to that binding. When using *utop*, the scope of a top-level `let` binding is everything that follows it in the session. When it shows up in a module, the scope is the remainder of that module.

Here's a simple example:

### OCaml utop

```
# let x = 3;;
val x : int = 3
# let y = 4;;
val y : int = 4
# let z = x + y;;
val z : int = 7
```

`let` can also be used to create a variable binding whose scope is limited to a particular expression, using the following syntax:

### Syntax

```
let <variable> = <expr1> in <expr2>
```

This first evaluates *`expr1`* and then evaluates *`expr2`* with *`variable`* bound to whatever value was produced by the evaluation of *`expr1`*. Here's how it looks in practice:

### OCaml utop (part 1)

```
# let languages = "OCaml,Perl,C++,C";;
val languages : string = "OCaml,Perl,C++,C"
# let dashed_languages =
    let language_list = String.split languages ~on:',' in
    String.concat ~sep:"-" language_list
  ;;
val dashed_languages : string = "OCaml-Perl-C++-C"
```

Note that the scope of `language_list` is just the expression `String.concat ~sep:"-" language_list` and is not available at the toplevel, as we can see if we try to access it now:

### OCaml utop (part 2)

```
# language_list;;
Characters -1-13:
Error: Unbound value language_list
```

A `let` binding in an inner scope can *shadow*, or hide, the definition from an outer scope. So, for example, we could have written the `dashed_languages` example as follows:

### OCaml utop (part 3)

```
# let languages = "OCaml,Perl,C++,C";;
val languages : string = "OCaml,Perl,C++,C"
# let dashed_languages =
    let languages = String.split languages ~on:',' in
    String.concat ~sep:"-" languages
  ;;
val dashed_languages : string = "OCaml-Perl-C++-C"
```

This time, in the inner scope we called the list of strings `languages` instead of `language_list`, thus hiding the original definition of `languages`. But once the definition of `dashed_languages` is complete, the inner scope has closed and the original definition of languages reappears:

OCaml utop (part 4)

```
# languages;;
- : string = "OCaml,Perl,C++,C"
```

One common idiom is to use a series of nested `let`/`in` expressions to build up the components of a larger computation. Thus, we might write:

OCaml utop (part 5)

```
# let area_of_ring inner_radius outer_radius =
    let pi = acos (-1.) in
    let area_of_circle r = pi *. r *. r in
    area_of_circle outer_radius -. area_of_circle inner_radius
  ;;
val area_of_ring : float -> float -> float = <fun>
# area_of_ring 1. 3.;;
- : float = 25.1327412287
```

It's important not to confuse a sequence of `let` bindings with the modification of a mutable variable. For example, consider how `area_of_ring` would work if we had instead written this purposefully confusing bit of code:

OCaml utop (part 6)

```
# let area_of_ring inner_radius outer_radius =
    let pi = acos (-1.) in
    let area_of_circle r = pi *. r *. r in
    let pi = 0. in
    area_of_circle outer_radius -. area_of_circle inner_radius
  ;;
Characters 126-128:
Warning 26: unused variable pi.
val area_of_ring : float -> float -> float = <fun>
```

Here, we redefined `pi` to be zero after the definition of `area_of_circle`. You might think that this would mean that the result of the computation would now be zero, but in fact, the behavior of the function is unchanged. That's because the original definition of `pi` wasn't changed; it was just shadowed, which means that any subsequent reference to `pi` would see the new definition of `pi` as `0`, but earlier references would be unchanged. But there is no later use of `pi`, so the binding of `pi` to `0.` made no difference. This explains the warning produced by the toplevel telling us that there is an unused definition of `pi`.

In OCaml, `let` bindings are immutable. There are many kinds of mutable values in OCaml, which we'll discuss in Chapter 8, but there are no mutable variables.

**WHY DON'T VARIABLES VARY?**

One source of confusion for people new to OCaml is the fact that variables are immutable. This seems pretty surprising even on

linguistic terms. Isn't the whole point of a variable that it can vary?

The answer to this is that variables in OCaml (and generally in functional languages) are really more like variables in an equation than a variable in an imperative language. If you think about the mathematical identity `x(y + z) = xy + xz`, there's no notion of mutating the variables `x`, `y`, and `z`. They vary in the sense that you can instantiate this equation with different numbers for those variables, and it still holds.

The same is true in a functional language. A function can be applied to different inputs, and thus its variables will take on different values, even without mutation.

## Pattern Matching and let

Another useful feature of `let` bindings is that they support the use of *patterns* on the lefthand side. Consider the following code, which uses `List.unzip`, a function for converting a list of pairs into a pair of lists:

OCaml utop (part 7)

```
# let (ints,strings) = List.unzip [(1,"one"); (2,"two"); (3,"three")];;
val ints : int list = [1; 2; 3]
val strings : string list = ["one"; "two"; "three"]
```

Here, `(ints,strings)` is a pattern, and the `let` binding assigns values to both of the identifiers that show up in that pattern. A pattern is essentially a description of the shape of a data structure, where some components are identifiers to be bound. As we saw in Tuples, Lists, Options, and Pattern Matching, OCaml has patterns for a variety of different data types.

Using a pattern in a `let` binding makes the most sense for a pattern that is *irrefutable*, *i.e.*, where any value of the type in question is guaranteed to match the pattern. Tuple and record patterns are irrefutable, but list patterns are not. Consider the following code that implements a function for upper casing the first element of a comma-separated list:

OCaml utop (part 8)

```
# let upcase_first_entry line =
    let (first :: rest) = String.split ~on:',' line in
    String.concat ~sep:"," (String.uppercase first :: rest)
  ;;
Characters 40-53:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val upcase_first_entry : string -> string = <fun>
```

This case can't really come up in practice, because `String.split` always returns a list with at least one element. But the compiler doesn't know this, and so it emits the warning. It's generally better to use a `match` statement to handle such cases explicitly:

OCaml utop (part 9)

```
# let upcase_first_entry line =
    match String.split ~on:',' line with
    | [] -> assert false (* String.split returns at least one element *)
    | first :: rest -> String.concat ~sep:"," (String.uppercase first :: rest)
  ;;
val upcase_first_entry : string -> string = <fun>
```

Note that this is our first use of `assert`, which is useful for marking cases that should be impossible. We'll discuss `assert` in more detail in Chapter 7.

# Functions

Given that OCaml is a functional language, it's no surprise that functions are important and pervasive. Indeed, functions have come up in almost every example we've done so far. This section will go into more depth, explaining the details of how OCaml's functions work. As you'll see, functions in OCaml differ in a variety of ways from what you'll find in most mainstream languages.

## Anonymous Functions

We'll start by looking at the most basic style of function declaration in OCaml: the *anonymous function*. An anonymous function is a function that is declared without being named. These can be declared using the `fun` keyword, as shown here:

OCaml utop (part 10)

```
# (fun x -> x + 1);;
- : int -> int = <fun>
```

Anonymous functions operate in much the same way as named functions. For example, we can apply an anonymous function to an argument:

OCaml utop (part 11)

```
# (fun x -> x + 1) 7;;
- : int = 8
```

Or pass it to another function. Passing functions to iteration functions like `List.map` is probably the most common use case for anonymous functions:

OCaml utop (part 12)

```
# List.map ~f:(fun x -> x + 1) [1;2;3];;
- : int list = [2; 3; 4]
```

You can even stuff them into a data structure:

OCaml utop (part 13)

```
# let increments = [ (fun x -> x + 1); (fun x -> x + 2) ] ;;
val increments : (int -> int) list = [<fun>; <fun>]
# List.map ~f:(fun g -> g 5) increments;;
- : int list = [6; 7]
```

It's worth stopping for a moment to puzzle this example out, since this kind of higher-order use of functions can be a bit obscure at first. Notice that `(fun g -> g 5)` is a function that takes a function as an argument, and then applies that function to the number `5`. The invocation of `List.map` applies `(fun g -> g 5)` to the elements of the `increments` list (which are themselves functions) and returns the list containing the results of these function applications.

The key thing to understand is that functions are ordinary values in OCaml, and you can do everything with them that you'd do with an ordinary value, including passing them to and returning them from other functions and storing them in data structures. We even name functions in the same way that we name other values, by using a `let` binding:

OCaml utop (part 14)

```
# let plusone = (fun x -> x + 1);;
val plusone : int -> int = <fun>
# plusone 3;;
- : int = 4
```

Defining named functions is so common that there is some syntactic sugar for it. Thus, the following definition of `plusone` is equivalent to the previous definition:

OCaml utop (part 15)

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

This is the most common and convenient way to declare a function, but syntactic niceties aside, the two styles of function definition are equivalent.

**LET AND FUN**

Functions and `let` bindings have a lot to do with each other. In some sense, you can think of the parameter of a function as a variable being bound to the value passed by the caller. Indeed, the following two expressions are nearly equivalent:

OCaml utop (part 16)

```
# (fun x -> x + 1) 7;;
- : int = 8
# let x = 7 in x + 1;;
- : int = 8
```

This connection is important, and will come up more when programming in a monadic style, as we'll see in Chapter 18.

## Multiargument functions

OCaml of course also supports multiargument functions, such as:

OCaml utop (part 17)

```
# let abs_diff x y = abs (x - y);;
val abs_diff : int -> int -> int = <fun>
# abs_diff 3 4;;
- : int = 1
```

You may find the type signature of `abs_diff` with all of its arrows a little hard to parse. To understand what's going on, let's rewrite `abs_diff` in an equivalent form, using the `fun` keyword:

OCaml utop (part 18)

```
# let abs_diff =
    (fun x -> (fun y -> abs (x - y)));;
val abs_diff : int -> int -> int = <fun>
```

This rewrite makes it explicit that `abs_diff` is actually a function of one argument that returns another function of one argument, which itself returns the final result. Because the functions are nested, the inner expression `abs (x - y)` has access to both `x`, which was bound by the outer function application, and `y`, which was bound by the inner one.

This style of function is called a *curried* function. (Currying is named after Haskell Curry, a logician who had a significant impact on the design and theory of programming languages.) The key to interpreting the type signature of a curried function is the observation that `->` is right-associative. The type signature of `abs_diff` can therefore be parenthesized as follows:

OCaml: variables-and-functions/abs_diff.mli

```
val abs_diff : int -> (int -> int)
```

The parentheses don't change the meaning of the signature, but they make it easier to see the currying.

Currying is more than just a theoretical curiosity. You can make use of currying to specialize a function by feeding in some of the arguments. Here's an example where we create a specialized version of `abs_diff` that measures the distance of a given number from `3`:

OCaml utop (part 19)

```
# let dist_from_3 = abs_diff 3;;
val dist_from_3 : int -> int = <fun>
# dist_from_3 8;;
- : int = 5
# dist_from_3 (-1);;
- : int = 4
```

The practice of applying some of the arguments of a curried function to get a new function is called *partial application*.

Note that the `fun` keyword supports its own syntax for currying, so the following definition of `abs_diff` is equivalent to the previous one.

OCaml utop (part 20)

```
# let abs_diff = (fun x y -> abs (x - y));;
val abs_diff : int -> int -> int = <fun>
```

You might worry that curried functions are terribly expensive, but this is not the case. In OCaml, there is no penalty for calling a curried function with all of its arguments. (Partial application, unsurprisingly, does have a small extra cost.)

Currying is not the only way of writing a multiargument function in OCaml. It's also possible to use the different parts of a tuple as different arguments. So, we could write:

OCaml utop (part 21)

```
# let abs_diff (x,y) = abs (x - y);;
val abs_diff : int * int -> int = <fun>
# abs_diff (3,4);;
- : int = 1
```

OCaml handles this calling convention efficiently as well. In particular it does not generally have to allocate a tuple just for the purpose of sending arguments to a tuple-style function. You can't, however, use partial application for this style of function.

There are small trade-offs between these two approaches, but most of the time, one should stick to currying, since it's the default style in the OCaml world.

## Recursive Functions

A function is *recursive* if it refers to itself in its definition. Recursion is important in any programming language, but is particularly important in functional languages, because it is the way that you build looping constructs. (As will be discussed in more detail in Chapter 8, OCaml also supports imperative looping constructs like `for` and `while`, but these are only useful when using OCaml's imperative features.)

In order to define a recursive function, you need to mark the `let` binding as recursive with the `rec` keyword, as shown in this function for finding the first sequentially repeated element in a list:

OCaml utop (part 22)

```
# let rec find_first_stutter list =
    match list with
```

```
    | [] | [_] ->
      (* only zero or one elements, so no repeats *)
      None
    | x :: y :: tl ->
      if x = y then Some x else find_first_stutter (y::tl)
  ;;
val find_first_stutter : 'a list -> 'a option = <fun>
```

Note that in the code, the pattern `| [] | [_]` is what's called an *or-pattern*, which is a disjunction of two patterns, meaning that it will be considered a match if either pattern matches. In this case, `[]` matches the empty list, and `[_]` matches any single element list. The _ is there so we don't have to put an explicit name on that single element.

We can also define multiple mutually recursive values by using `let rec` combined with the `and` keyword. Here's a (gratuitously inefficient) example:

OCaml utop (part 23)

```
# let rec is_even x =
    if x = 0 then true else is_odd (x - 1)
  and is_odd x =
    if x = 0 then false else is_even (x - 1)
 ;;
val is_even : int -> bool = <fun>
val is_odd : int -> bool = <fun>
# List.map ~f:is_even [0;1;2;3;4;5];;
- : bool list = [true; false; true; false; true; false]
# List.map ~f:is_odd [0;1;2;3;4;5];;
- : bool list = [false; true; false; true; false; true]
```

OCaml distinguishes between nonrecursive definitions (using `let`) and recursive definitions (using `let rec`) largely for technical reasons: the type-inference algorithm needs to know when a set of function definitions are mutually recursive, and for reasons that don't apply to a pure language like Haskell, these have to be marked explicitly by the programmer.

But this decision has some good effects. For one thing, recursive (and especially mutually recursive) definitions are harder to reason about than nonrecursive ones. It's therefore useful that, in the absence of an explicit `rec`, you can assume that a `let` binding is nonrecursive, and so can only build upon previous bindings.

In addition, having a nonrecursive form makes it easier to create a new definition that extends and supersedes an existing one by shadowing it.

## Prefix and Infix Operators

So far, we've seen examples of functions used in both prefix and infix style:

OCaml utop (part 24)

```
# Int.max 3 4   (* prefix *);;
- : int = 4
# 3 + 4         (* infix  *);;
- : int = 7
```

You might not have thought of the second example as an ordinary function, but it very much is. Infix operators like + really only differ syntactically from other functions. In fact, if we put parentheses around an infix operator, you can use it as an ordinary prefix function:

OCaml utop (part 25)

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

```
- : int = 7
# List.map ~f:((+) 3) [4;5;6];;
- : int list = [7; 8; 9]
```

In the second expression, we've partially applied `(+)` to create a function that increments its single argument by `3`.

A function is treated syntactically as an operator if the name of that function is chosen from one of a specialized set of identifiers. This set includes identifiers that are sequences of characters from the following set:

Syntax

```
! $ % & * + - . / : < = > ? @ ^ | ~
```

`or` is one of a handful of predetermined strings, including `mod`, the modulus operator, and `lsl`, for "logical shift left," a bit-shifting operation.

We can define (or redefine) the meaning of an operator. Here's an example of a simple vector-addition operator on `int` pairs:

OCaml utop (part 26)

```
# let (+!) (x1,y1) (x2,y2) = (x1 + x2, y1 + y2);;
val ( +! ) : int * int -> int * int -> int * int = <fun>
# (3,2) +! (-2,4);;
- : int * int = (1, 6)
```

Note that you have to be careful when dealing with operators containing `*`. Consider the following example:

OCaml utop (part 27)

```
# let (***) x y = (x ** y) ** y;;
Characters 17-18:
Error: This expression has type int but an expression was expected of type
         float
```

What's going on is that `(***)` isn't interpreted as an operator at all; it's read as a comment! To get this to work properly, we need to put spaces around any operator that begins or ends with `*`:

OCaml utop (part 28)

```
# let ( *** ) x y = (x ** y) ** y;;
val ( *** ) : float -> float -> float = <fun>
```

The syntactic role of an operator is typically determined by its first character or two, though there are a few exceptions. Table 2-1 breaks the different operators and other syntactic forms into groups from highest to lowest precedence, explaining how each behaves syntactically. We write `!...` to indicate the class of operators beginning with `!`.

*Table 2-1. Precedence and associativity*

| Operator prefix | Associativity |
|---|---|
| `!...`, `?...`, `~...` | Prefix |
| `.`, `.(`, `.[` | - |
| function application, constructor, `assert`, `lazy` | Left associative |
| `-`, `-.` | Prefix |
| | |

| | |
|---|---|
| **..., lsl, lsr, asr<br>*..., /..., %..., mod, land, lor, lxor | Right associative<br>Left associative |
| +..., -... | Left associative |
| :: | Right associative |
| @..., ^... | Right associative |
| =..., <..., >..., \|..., &..., $... | Left associative |
| &, && | Right associative |
| or, \|\| | Right associative |
| , | - |
| <-, := | Right associative |
| if | - |
| ; | Right associative |

There's one important special case: `-` and `-.`, which are the integer and floating-point subtraction operators, and can act as both prefix operators (for negation) and infix operators (for subtraction). So, both `-x` and `x - y` are meaningful expressions. Another thing to remember about negation is that it has lower precedence than function application, which means that if you want to pass a negative value, you need to wrap it in parentheses, as you can see in this code:

OCaml utop (part 29)

```
# Int.max 3 (-4);;
- : int = 3
# Int.max 3 -4;;
Characters -1-9:
Error: This expression has type int -> int
       but an expression was expected of type int
```

Here, OCaml is interpreting the second expression as equivalent to:

OCaml utop (part 30)

```
# (Int.max 3) - 4;;
Characters 1-10:
Error: This expression has type int -> int
       but an expression was expected of type int
```

which obviously doesn't make sense.

Here's an example of a very useful operator from the standard library whose behavior depends critically on the precedence rules described previously:

OCaml utop (part 31)

```
# let (|>) x f = f x ;;
val ( |> ) : 'a -> ('a -> 'b) -> 'b = <fun>
```

It's not quite obvious at first what the purpose of this operator is: it just takes a value and a function and applies the function to the value. Despite that bland-sounding description, it has the useful role of a sequencing operator, similar in spirit to using the pipe character in the UNIX shell. Consider, for example, the following code for printing out the unique elements of your `PATH`. Note that `List.dedup` that follows removes duplicates from a list by sorting the list using the provided comparison function:

OCaml utop (part 32)

```
# let path = "/usr/bin:/usr/local/bin:/bin:/sbin";;
val path : string = "/usr/bin:/usr/local/bin:/bin:/sbin"
# String.split ~on:':' path
  |> List.dedup ~compare:String.compare
  |> List.iter ~f:print_endline
  ;;
/bin
/sbin
/usr/bin
/usr/local/bin
- : unit = ()
```

Note that we can do this without `|>`, but the result is a bit more verbose:

OCaml utop (part 33)

```
# let split_path = String.split ~on:':' path in
  let deduped_path = List.dedup ~compare:String.compare split_path in
  List.iter ~f:print_endline deduped_path
  ;;
/bin
/sbin
/usr/bin
/usr/local/bin
- : unit = ()
```

An important part of what's happening here is partial application. For example, `List.iter` normally takes two arguments: a function to be called on each element of the list, and the list to iterate over. We can call `List.iter` with all its arguments:

OCaml utop (part 34)

```
# List.iter ~f:print_endline ["Two"; "lines"];;
Two
lines
- : unit = ()
```

Or, we can pass it just the function argument, leaving us with a function for printing out a list of strings:

OCaml utop (part 35)

```
# List.iter ~f:print_endline;;
- : string list -> unit = <fun>
```

It is this later form that we're using in the preceding `|>` pipeline.

But `|>` only works in the intended way because it is left-associative. Let's see what happens if we try using a right-associative operator, like (^>):

OCaml utop (part 36)

```
# let (^>) x f = f x;;
val ( ^> ) : 'a -> ('a -> 'b) -> 'b = <fun>
# Sys.getenv_exn "PATH"
  ^> String.split ~on:':' path
  ^> List.dedup ~compare:String.compare
  ^> List.iter ~f:print_endline
  ;;
Characters 98-124:
Error: This expression has type string list -> unit
       but an expression was expected of type
         (string list -> string list) -> 'a
       Type string list is not compatible with type
         string list -> string list
```

The type error is a little bewildering at first glance. What's going on is that, because `^>` is right associative, the operator is trying to feed the value `List.dedup ~compare:String.compare` to the function `List.iter ~f:print_endline`. But `List.iter ~f:print_endline` expects a list of strings as its input, not a function.

The type error aside, this example highlights the importance of choosing the operator you use with care, particularly with respect to associativity.

## Declaring Functions with Function

Another way to define a function is using the `function` keyword. Instead of having syntactic support for declaring multiargument (curried) functions, `function` has built-in pattern matching. Here's an example:

OCaml utop (part 37)

```
# let some_or_zero = function
    | Some x -> x
    | None -> 0
  ;;
val some_or_zero : int option -> int = <fun>
# List.map ~f:some_or_zero [Some 3; None; Some 4];;
- : int list = [3; 0; 4]
```

This is equivalent to combining an ordinary function definition with a `match`:

OCaml utop (part 38)

```
# let some_or_zero num_opt =
    match num_opt with
    | Some x -> x
    | None -> 0
  ;;
val some_or_zero : int option -> int = <fun>
```

We can also combine the different styles of function declaration together, as in the following example, where we declare a two-argument (curried) function with a pattern match on the second argument:

OCaml utop (part 39)

```
# let some_or_default default = function
    | Some x -> x
    | None -> default
  ;;
val some_or_default : 'a -> 'a option -> 'a = <fun>
# some_or_default 3 (Some 5);;
- : int = 5
# List.map ~f:(some_or_default 100) [Some 3; None; Some 4];;
- : int list = [3; 100; 4]
```

Also, note the use of partial application to generate the function passed to `List.map`. In other words, `some_or_default 100` is a function that was created by feeding just the first argument to `some_or_default`.

## Labeled Arguments

Up until now, the functions we've defined have specified their arguments positionally, *i.e.*, by the order in which the arguments are passed to the function. OCaml also supports labeled arguments, which let you identify a function argument by name. Indeed, we've already encountered functions from Core like `List.map` that use labeled arguments. Labeled arguments are marked by a leading tilde, and

a label (followed by a colon) is put in front of the variable to be labeled. Here's an example:

OCaml utop (part 40)

```
# let ratio ~num ~denom = float num /. float denom;;
val ratio : num:int -> denom:int -> float = <fun>
```

We can then provide a labeled argument using a similar convention. As you can see, the arguments can be provided in any order:

OCaml utop (part 41)

```
# ratio ~num:3 ~denom:10;;
- : float = 0.3
# ratio ~denom:10 ~num:3;;
- : float = 0.3
```

OCaml also supports *label punning*, meaning that you get to drop the text after the `:` if the name of the label and the name of the variable being used are the same. We were actually already using label punning when defining `ratio`. The following shows how punning can be used when invoking a function:

OCaml utop (part 42)

```
# let num = 3 in
  let denom = 4 in
  ratio ~num ~denom;;
- : float = 0.75
```

Labeled arguments are useful in a few different cases:

- When defining a function with lots of arguments. Beyond a certain number, arguments are easier to remember by name than by position.
- When the meaning of a particular argument is unclear from the type alone. Consider a function for creating a hash table whose first argument is the initial size of the array backing the hash table, and the second is a Boolean flag, which indicates whether that array will ever shrink when elements are removed:

  OCaml

  ```
  val create_hashtable : int -> bool -> ('a,'b) Hashtable.t
  ```

  The signature makes it hard to divine the meaning of those two arguments. but with labeled arguments, we can make the intent immediately clear:

  OCaml

  ```
  val create_hashtable :
    init_size:int -> allow_shrinking:bool -> ('a,'b) Hashtable.t
  ```

  Choosing label names well is especially important for Boolean values, since it's often easy to get confused about whether a value being true is meant to enable or disable a given feature.
- When defining functions that have multiple arguments that might get confused with each other. This is most at issue when the arguments are of the same type. For example, consider this signature for a function that extracts a substring:

  OCaml

  ```
  val substring: string -> int -> int -> string
  ```

Here, the two `int`s are the starting position and length of the substring to extract, respectively. We can make this fact more obvious from the signature by adding labeled:

OCaml

```
val substring: string -> pos:int -> len:int -> string
```

This improves the readability of both the signature and of client code that makes use of `substring` and makes it harder to accidentally swap the position and the length.

- When you want flexibility on the order in which arguments are passed. Consider a function like `List.iter`, which takes two arguments: a function and a list of elements to call that function on. A common pattern is to partially apply `List.iter` by giving it just the function, as in the following example from earlier in the chapter:

  OCaml utop (part 43)

  ```
  # String.split ~on:':' path
    |> List.dedup ~compare:String.compare
    |> List.iter ~f:print_endline
    ;;
  /bin
  /sbin
  /usr/bin
  /usr/local/bin
  - : unit = ()
  ```

  This requires that we put the function argument first. In other cases, you want to put the function argument second. One common reason is readability. In particular, a multiline function passed as an argument to another function is easiest to read when it is the final argument to that function.

### Higher-order functions and labels

One surprising gotcha with labeled arguments is that while order doesn't matter when calling a function with labeled arguments, it does matter in a higher-order context, *e.g.*, when passing a function with labeled arguments to another function. Here's an example:

OCaml utop (part 44)

```
# let apply_to_tuple f (first,second) = f ~first ~second;;
val apply_to_tuple : (first:'a -> second:'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

Here, the definition of `apply_to_tuple` sets up the expectation that its first argument is a function with two labeled arguments, `first` and `second`, listed in that order. We could have defined `apply_to_tuple` differently to change the order in which the labeled arguments were listed:

OCaml utop (part 45)

```
# let apply_to_tuple_2 f (first,second) = f ~second ~first;;
val apply_to_tuple_2 : (second:'a -> first:'b -> 'c) -> 'b * 'a -> 'c = <fun>
```

It turns out this order matters. In particular, if we define a function that has a different order

OCaml utop (part 46)

```
# let divide ~first ~second = first / second;;
val divide : first:int -> second:int -> int = <fun>
```

we'll find that it can't be passed in to `apply_to_tuple_2`.

OCaml utop (part 47)

```
# apply_to_tuple_2 divide (3,4);;
Characters 17-23:
Error: This expression has type first:int -> second:int -> int
       but an expression was expected of type second:'a -> first:'b -> 'c
```

But, it works smoothly with the original `apply_to_tuple`:

OCaml utop (part 48)

```
# let apply_to_tuple f (first,second) = f ~first ~second;;
val apply_to_tuple : (first:'a -> second:'b -> 'c) -> 'a * 'b -> 'c = <fun>
# apply_to_tuple divide (3,4);;
- : int = 0
```

As a result, when passing labeled functions as arguments, you need to take care to be consistent in your ordering of labeled arguments.

## Optional Arguments

An optional argument is like a labeled argument that the caller can choose whether or not to provide. Optional arguments are passed in using the same syntax as labeled arguments, and, like labeled arguments, can be provided in any order.

Here's an example of a string concatenation function with an optional separator. This function uses the `^` operator for pairwise string concatenation:

OCaml utop (part 49)

```
# let concat ?sep x y =
    let sep = match sep with None -> "" | Some x -> x in
    x ^ sep ^ y
  ;;
val concat : ?sep:string -> string -> string -> string = <fun>
# concat "foo" "bar"              (* without the optional argument *);;
- : string = "foobar"
# concat ~sep:":" "foo" "bar"     (* with the optional argument    *);;
- : string = "foo:bar"
```

Here, `?` is used in the definition of the function to mark `sep` as optional. And while the caller can pass a value of type `string` for `sep`, internally to the function, `sep` is seen as a `string option`, with `None` appearing when `sep` is not provided by the caller.

The preceding example needed a bit of boilerplate to choose a default separator when none was provided. This is a common enough pattern that there's an explicit syntax for providing a default value, which allows us to write `concat` more concisely:

OCaml utop (part 50)

```
# let concat ?(sep="") x y = x ^ sep ^ y ;;
val concat : ?sep:string -> string -> string -> string = <fun>
```

Optional arguments are very useful, but they're also easy to abuse. The key advantage of optional arguments is that they let you write functions with multiple arguments that users can ignore most of the time, only worrying about them when they specifically want to invoke those options. They also allow you to extend an API with new functionality without changing existing code.

The downside is that the caller may be unaware that there is a choice to be made, and so may

unknowingly (and wrongly) pick the default behavior. Optional arguments really only make sense when the extra concision of omitting the argument outweighs the corresponding loss of explicitness.

This means that rarely used functions should not have optional arguments. A good rule of thumb is to avoid optional arguments for functions internal to a module, *i.e.*, functions that are not included in the module's interface, or `mli` file. We'll learn more about `mli`s in Chapter 4.

### Explicit passing of an optional argument

Under the covers, a function with an optional argument receives `None` when the caller doesn't provide the argument, and `Some` when it does. But the `Some` and `None` are normally not explicitly passed in by the caller.

But sometimes, passing in `Some` or `None` explicitly is exactly what you want. OCaml lets you do this by using `?` instead of `~` to mark the argument. Thus, the following two lines are equivalent ways of specifying the `sep` argument to `concat`:

OCaml utop (part 51)

```
# concat ~sep:":" "foo" "bar" (* provide the optional argument *);;
- : string = "foo:bar"
# concat ?sep:(Some ":") "foo" "bar" (* pass an explicit [Some] *);;
- : string = "foo:bar"
```

And the following two lines are equivalent ways of calling `concat` without specifying `sep`:

OCaml utop (part 52)

```
# concat "foo" "bar" (* don't provide the optional argument *);;
- : string = "foobar"
# concat ?sep:None "foo" "bar" (* explicitly pass `None` *);;
- : string = "foobar"
```

One use case for this is when you want to define a wrapper function that mimics the optional arguments of the function it's wrapping. For example, imagine we wanted to create a function called `uppercase_concat`, which is the same as `concat` except that it converts the first string that it's passed to uppercase. We could write the function as follows:

OCaml utop (part 53)

```
# let uppercase_concat ?(sep="") a b = concat ~sep (String.uppercase a) b ;;
val uppercase_concat : ?sep:string -> string -> string -> string = <fun>
# uppercase_concat "foo" "bar";;
- : string = "FOObar"
# uppercase_concat "foo" "bar" ~sep:":";;
- : string = "FOO:bar"
```

In the way we've written it, we've been forced to separately make the decision as to what the default separator is. Thus, if we later change `concat`'s default behavior, we'll need to remember to change `uppercase_concat` to match it.

Instead, we can have `uppercase_concat` simply pass through the optional argument to `concat` using the `?` syntax:

OCaml utop (part 54)

```
# let uppercase_concat ?sep a b = concat ?sep (String.uppercase a) b ;;
val uppercase_concat : ?sep:string -> string -> string -> string = <fun>
```

Now, if someone calls `uppercase_concat` without an argument, an explicit `None` will be passed to

`concat`, leaving `concat` to decide what the default behavior should be.

### Inference of labeled and optional arguments

One subtle aspect of labeled and optional arguments is how they are inferred by the type system. Consider the following example for computing numerical derivatives of a function of two real variables. The function takes an argument `delta`, which determines the scale at which to compute the derivative; values `x` and `y`, which determine at which point to compute the derivative; and the function `f`, whose derivative is being computed. The function `f` itself takes two labeled arguments, `x` and `y`. Note that you can use an apostrophe as part of a variable name, so `x'` and `y'` are just ordinary variables:

OCaml utop (part 55)

```
# let numeric_deriv ~delta ~x ~y ~f =
    let x' = x +. delta in
    let y' = y +. delta in
    let base = f ~x ~y in
    let dx = (f ~x:x' ~y -. base) /. delta in
    let dy = (f ~x ~y:y' -. base) /. delta in
    (dx,dy)
  ;;
val numeric_deriv :
  delta:float ->
  x:float -> y:float -> f:(x:float -> y:float -> float) -> float * float =
  <fun>
```

In principle, it's not obvious how the order of the arguments to `f` should be chosen. Since labeled arguments can be passed in arbitrary order, it seems like it could as well be `y:float -> x:float -> float` as it is `x:float -> y:float -> float`.

Even worse, it would be perfectly consistent for `f` to take an optional argument instead of a labeled one, which could lead to this type signature for `numeric_deriv`:

OCaml

```
val numeric_deriv :
  delta:float ->
  x:float -> y:float -> f:(?x:float -> y:float -> float) -> float * float
```

Since there are multiple plausible types to choose from, OCaml needs some heuristic for choosing between them. The heuristic the compiler uses is to prefer labels to options and to choose the order of arguments that shows up in the source code.

Note that these heuristics might at different points in the source suggest different types. Here's a version of `numeric_deriv` where different invocations of `f` list the arguments in different orders:

OCaml utop (part 56)

```
# let numeric_deriv ~delta ~x ~y ~f =
    let x' = x +. delta in
    let y' = y +. delta in
    let base = f ~x ~y in
    let dx = (f ~y ~x:x' -. base) /. delta in
    let dy = (f ~x ~y:y' -. base) /. delta in
    (dx,dy)
  ;;
Characters 130-131:
Error: This function is applied to arguments
in an order different from other calls.
```

```
This is only allowed when the real type is known.
```

As suggested by the error message, we can get OCaml to accept the fact that `f` is used with different argument orders if we provide explicit type information. Thus, the following code compiles without error, due to the type annotation on `f`:

OCaml utop (part 57)

```
# let numeric_deriv ~delta ~x ~y ~(f: x:float -> y:float -> float) =
    let x' = x +. delta in
    let y' = y +. delta in
    let base = f ~x ~y in
    let dx = (f ~y ~x:x' -. base) /. delta in
    let dy = (f ~x ~y:y' -. base) /. delta in
    (dx,dy)
  ;;
val numeric_deriv :
  delta:float ->
  x:float -> y:float -> f:(x:float -> y:float -> float) -> float * float =
  <fun>
```

### Optional arguments and partial application

Optional arguments can be tricky to think about in the presence of partial application. We can of course partially apply the optional argument itself:

OCaml utop (part 58)

```
# let colon_concat = concat ~sep:":";;
val colon_concat : string -> string -> string = <fun>
# colon_concat "a" "b";;
- : string = "a:b"
```

But what happens if we partially apply just the first argument?

OCaml utop (part 59)

```
# let prepend_pound = concat "# ";;
val prepend_pound : string -> string = <fun>
# prepend_pound "a BASH comment";;
- : string = "# a BASH comment"
```

The optional argument `?sep` has now disappeared, or been *erased*. Indeed, if we try to pass in that optional argument now, it will be rejected:

OCaml utop (part 60)

```
# prepend_pound "a BASH comment" ~sep:":";;
Characters -1-13:
Error: This function has type string -> string
       It is applied to too many arguments; maybe you forgot a `;'.
```

So when does OCaml decide to erase an optional argument?

The rule is: an optional argument is erased as soon as the first positional (i.e., neither labeled nor optional) argument defined *after* the optional argument is passed in. That explains the behavior of `prepend_pound`. But if we had instead defined `concat` with the optional argument in the second position:

OCaml utop (part 61)

```
# let concat x ?(sep="") y = x ^ sep ^ y ;;
val concat : string -> ?sep:string -> string -> string = <fun>
```

then application of the first argument would not cause the optional argument to be erased.

OCaml utop (part 62)

```
# let prepend_pound = concat "# ";;
val prepend_pound : ?sep:string -> string -> string = <fun>
# prepend_pound "a BASH comment";;
- : string = "# a BASH comment"
# prepend_pound "a BASH comment" ~sep:"--- ";;
- : string = "# --- a BASH comment"
```

However, if all arguments to a function are presented at once, then erasure of optional arguments isn't applied until all of the arguments are passed in. This preserves our ability to pass in optional arguments anywhere on the argument list. Thus, we can write:

OCaml utop (part 63)

```
# concat "a" "b" ~sep:"=";;
- : string = "a=b"
```

An optional argument that doesn't have any following positional arguments can't be erased at all, which leads to a compiler warning:

OCaml utop (part 64)

```
# let concat x y ?(sep="") = x ^ sep ^ y ;;
Characters 15-38:
Warning 16: this optional argument cannot be erased.
val concat : string -> string -> ?sep:string -> string = <fun>
```

And indeed, when we provide the two positional arguments, the `sep` argument is not erased, instead returning a function that expects the `sep` argument to be provided:

OCaml utop (part 65)

```
# concat "a" "b";;
- : ?sep:string -> string = <fun>
```

As you can see, OCaml's support for labeled and optional arguments is not without its complexities. But don't let these complexities obscure the usefulness of these features. Labels and optional arguments are very effective tools for making your APIs both more convenient and safer, and it's worth the effort of learning how to use them effectively.

# Chapter 3. Lists and Patterns

This chapter will focus on two common elements of programming in OCaml: lists and pattern matching. Both of these were discussed in Chapter 1, but we'll go into more depth here, presenting the two topics together and using one to help illustrate the other.

# List Basics

An OCaml list is an immutable, finite sequence of elements of the same type. As we've seen, OCaml lists can be generated using a bracket-and-semicolon notation:

OCaml utop

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

And they can also be generated using the equivalent `::` notation:

OCaml utop (part 1)

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

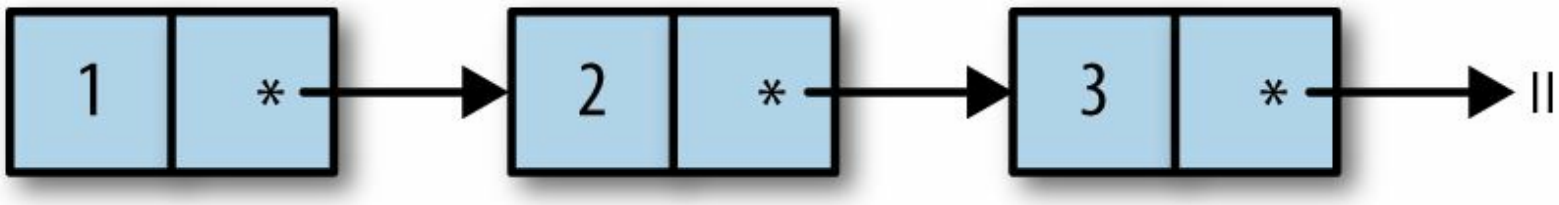
As you can see, the `::` operator is right-associative, which means that we can build up lists without parentheses. The empty list `[]` is used to terminate a list. Note that the empty list is polymorphic, meaning it can be used with elements of any type, as you can see here:

OCaml utop (part 2)

```
# let empty = [];;
val empty : 'a list = []
# 3 :: empty;;
- : int list = [3]
# "three" :: empty;;
- : string list = ["three"]
```

The way in which the `::` operator attaches elements to the front of a list reflects the fact that OCaml's lists are in fact singly linked lists. The figure below is a rough graphical representation of how the list `1 :: 2 :: 3 :: []` is laid out as a data structure. The final arrow (from the box containing `3`) points to the empty list.

Diagram




Each `::` essentially adds a new block to the proceding picture. Such a block contains two things: a reference to the data in that list element, and a reference to the remainder of the list. This is why `::` can extend a list without modifying it; extension allocates a new list element but change any of the existing ones, as you can see:

OCaml utop (part 3)

```
# let l = 1 :: 2 :: 3 :: [];;
val l : int list = [1; 2; 3]
# let m = 0 :: l;;
val m : int list = [0; 1; 2; 3]
# l;;
- : int list = [1; 2; 3]
```

# Using Patterns to Extract Data from a List

We can read data out of a list using a `match` statement. Here's a simple example of a recursive function that computes the sum of all elements of a list:

OCaml utop (part 4)

```
# let rec sum l =
    match l with
    | [] -> 0
    | hd :: tl -> hd + sum tl
  ;;
val sum : int list -> int = <fun>
# sum [1;2;3];;
- : int = 6
# sum [];;
- : int = 0
```

This code follows the convention of using `hd` to represent the first element (or head) of the list, and `tl` to represent the remainder (or tail).

The `match` statement in `sum` is really doing two things: first, it's acting as a case-analysis tool, breaking down the possibilities into a pattern-indexed list of cases. Second, it lets you name substructures within the data structure being matched. In this case, the variables `hd` and `tl` are bound by the pattern that defines the second case of the match statement. Variables that are bound in this way can be used in the expression to the right of the arrow for the pattern in question.

The fact that `match` statements can be used to bind new variables can be a source of confusion. To see how, imagine we wanted to write a function that filtered out from a list all elements equal to a particular value. You might be tempted to write that code as follows, but when you do, the compiler will immediately warn you that something is wrong:

OCaml utop (part 5)

```
# let rec drop_value l to_drop =
    match l with
    | [] -> []
    | to_drop :: tl -> drop_value tl to_drop
    | hd :: tl -> hd :: drop_value tl to_drop
  ;;
Characters 114-122:
Warning 11: this match case is unused.
val drop_value : 'a list -> 'a -> 'a list = <fun>
```

Moreover, the function clearly does the wrong thing, filtering out all elements of the list rather than just those equal to the provided value, as you can see here:

OCaml utop (part 6)

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

So, what's going on?

The key observation is that the appearance of `to_drop` in the second case doesn't imply a check that the first element is equal to the value `to_drop` passed in as an argument to `drop_value`. Instead, it just causes a new variable `to_drop` to be bound to whatever happens to be in the first element of the list, shadowing the earlier definition of `to_drop`. The third case is unused because it is essentially the same pattern as we had in the second case.

A better way to write this code is not to use pattern matching for determining whether the first element is equal to `to_drop`, but to instead use an ordinary `if` statement:

OCaml utop (part 7)

```
# let rec drop_value l to_drop =
    match l with
    | [] -> []
    | hd :: tl ->
      let new_tl = drop_value tl to_drop in
      if hd = to_drop then new_tl else hd :: new_tl
  ;;
val drop_value : 'a list -> 'a -> 'a list = <fun>
# drop_value [1;2;3] 2;;
- : int list = [1; 3]
```

Note that if we wanted to drop a particular literal value (rather than a value that was passed in), we could do this using something like our original implementation of `drop_value`:

OCaml utop (part 8)

```
# let rec drop_zero l =
    match l with
    | [] -> []
    | 0  :: tl -> drop_zero tl
    | hd :: tl -> hd :: drop_zero tl
  ;;
val drop_zero : int list -> int list = <fun>
# drop_zero [1;2;0;3];;
- : int list = [1; 2; 3]
```

# Limitations (and Blessings) of Pattern Matching

The preceding example highlights an important fact about patterns, which is that they can't be used to express arbitrary conditions. Patterns can characterize the layout of a data structure and can even include literals, as in the `drop_zero` example, but that's where they stop. A pattern can check if a list has two elements, but it can't check if the first two elements are equal to each other.

You can think of patterns as a specialized sublanguage that can express a limited (though still quite rich) set of conditions. The fact that the pattern language is limited turns out to be a very good thing, making it possible to build better support for patterns in the compiler. In particular, both the efficiency of `match` statements and the ability of the compiler to detect errors in matches depend on the constrained nature of patterns.

## Performance

Naively, you might think that it would be necessary to check each case in a `match` in sequence to figure out which one fires. If the cases of a match were guarded by arbitrary code, that would be the case. But OCaml is often able to generate machine code that jumps directly to the matched case based on an efficiently chosen set of runtime checks.

As an example, consider the following rather silly functions for incrementing an integer by one. The first is implemented with a `match` statement, and the second with a sequence of `if` statements:

OCaml utop (part 9)

```
# let plus_one_match x =
    match x with
    | 0 -> 1
    | 1 -> 2
    | 2 -> 3
    | _ -> x + 1

  let plus_one_if x =
    if      x = 0 then 1
    else if x = 1 then 2
    else if x = 2 then 3
    else x + 1
  ;;
val plus_one_match : int -> int = <fun>
val plus_one_if : int -> int = <fun>
```

Note the use of _ in the above match. This is a wildcard pattern that matches any value, but without binding a variable name to the value in question.

If you benchmark these functions, you'll see that `plus_one_if` is considerably slower than `plus_one_match`, and the advantage gets larger as the number of cases increases. Here, we'll benchmark these functions using the `core_bench` library, which can be installed by running `opam install core_bench` from the command line:

OCaml utop (part 10)

```
# #require "core_bench";;
# open Core_bench.Std;;
# let run_bench tests =
  Bench.bench
    ~ascii_table:true
    ~display:Textutils.Ascii_table.Display.column_titles
    tests
```

```
;;
val run_bench : Bench.Test.t list -> unit = <fun>
# [ Bench.Test.create ~name:"plus_one_match" (fun () ->
      ignore (plus_one_match 10))
  ; Bench.Test.create ~name:"plus_one_if" (fun () ->
      ignore (plus_one_if 10)) ]
  |> run_bench
  ;;
Estimated testing time 20s (change using -quota SECS).

  Name             Time (ns)   % of max
 --------------- ----------- ----------
  plus_one_match       46.81      68.21
  plus_one_if          68.63     100.00

- : unit = ()
```

Here's another, less artificial example. We can rewrite the `sum` function we described earlier in the chapter using an `if` statement rather than a match. We can then use the functions `is_empty`, `hd_exn`, and `tl_exn` from the `List` module to deconstruct the list, allowing us to implement the entire function without pattern matching:

OCaml utop (part 11)

```
# let rec sum_if l =
    if List.is_empty l then 0
    else List.hd_exn l + sum_if (List.tl_exn l)
  ;;
val sum_if : int list -> int = <fun>
```

Again, we can benchmark these to see the difference:

OCaml utop (part 12)

```
# let numbers = List.range 0 1000 in
  [ Bench.Test.create ~name:"sum_if" (fun () -> ignore (sum_if numbers))
  ; Bench.Test.create ~name:"sum"    (fun () -> ignore (sum numbers)) ]
  |> run_bench
  ;;
Estimated testing time 20s (change using -quota SECS).

  Name     Time (ns)   % of max
 -------- ----------- ----------
  sum_if     110_535     100.00
  sum         22_361      20.23

- : unit = ()
```

In this case, the `match`-based implementation is many times faster than the `if`-based implementation. The difference comes because we need to effectively do the same work multiple times, since each function we call has to reexamine the first element of the list to determine whether or not it's the empty cell. With a `match` statement, this work happens exactly once per list element.

Generally, pattern matching is more efficient than the alternatives you might code by hand. One notable exception is matches over strings, which are in fact tested sequentially, so matches containing a long sequence of strings can be outperformed by a hash table. But most of the time, pattern matching is a clear performance win.

## Detecting Errors

The error-detecting capabilities of `match` statements are if anything more important than their performance. We've already seen one example of OCaml's ability to find problems in a pattern

match: in our broken implementation of `drop_value`, OCaml warned us that the final case was redundant. There are no algorithms for determining if a predicate written in a general-purpose language is redundant, but it can be solved reliably in the context of patterns.

OCaml also checks `match` statements for exhaustiveness. Consider what happens if we modify `drop_zero` by deleting the handler for one of the cases. As you can see, the compiler will produce a warning that we've missed a case, along with an example of an unmatched pattern:

OCaml utop (part 13)

```
# let rec drop_zero l =
    match l with
    | [] -> []
    | 0  :: tl -> drop_zero tl
  ;;
Characters 26-84:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1::_
val drop_zero : int list -> 'a list = <fun>
```

Even for simple examples like this, exhaustiveness checks are pretty useful. But as we'll see in Chapter 6, they become yet more valuable as you get to more complicated examples, especially those involving user-defined types. In addition to catching outright errors, they act as a sort of refactoring tool, guiding you to the locations where you need to adapt your code to deal with changing types.

# Using the List Module Effectively

We've so far written a fair amount of list-munging code using pattern matching and recursive functions. But in real life, you're usually better off using the `List` module, which is full of reusable functions that abstract out common patterns for computing with lists.

Let's work through a concrete example to see this in action. We'll write a function `render_table` that, given a list of column headers and a list of rows, prints them out in a well-formatted text table, as follows:

OCaml utop (part 69)

```
# printf "%s\n"
   (render_table
     ["language";"architect";"first release"]
     [ ["Lisp" ;"John McCarthy" ;"1958"] ;
       ["C"    ;"Dennis Ritchie";"1969"] ;
       ["ML"   ;"Robin Milner"  ;"1973"] ;
       ["OCaml";"Xavier Leroy"  ;"1996"] ;
     ]);;
| language | architect      | first release |
|----------+----------------+---------------|
| Lisp     | John McCarthy  | 1958          |
| C        | Dennis Ritchie | 1969          |
| ML       | Robin Milner   | 1973          |
| OCaml    | Xavier Leroy   | 1996          |
- : unit = ()
```

The first step is to write a function to compute the maximum width of each column of data. We can do this by converting the header and each row into a list of integer lengths, and then taking the element-wise max of those lists of lengths. Writing the code for all of this directly would be a bit of a chore, but we can do it quite concisely by making use of three functions from the `List` module: `map`, `map2_exn`, and `fold`.

`List.map` is the simplest to explain. It takes a list and a function for transforming elements of that list, and returns a new list with the transformed elements. Thus, we can write:

OCaml utop (part 14)

```
# List.map ~f:String.length ["Hello"; "World!"];;
- : int list = [5; 6]
```

`List.map2_exn` is similar to `List.map`, except that it takes two lists and a function for combining them. Thus, we might write:

OCaml utop (part 15)

```
# List.map2_exn ~f:Int.max [1;2;3] [3;2;1];;
- : int list = [3; 2; 3]
```

The `_exn` is there because the function throws an exception if the lists are of mismatched length:

OCaml utop (part 16)

```
# List.map2_exn ~f:Int.max [1;2;3] [3;2;1;0];;
Exception: (Invalid_argument "length mismatch in rev_map2_exn: 3 <> 4 ").
```

`List.fold` is the most complicated of the three, taking three arguments: a list to process, an initial accumulator value, and a function for updating the accumulator. `List.fold` walks over the list from left to right, updating the accumulator at each step and returning the final value of the accumulator

when it's done. You can see some of this by looking at the type-signature for `fold`:

OCaml utop (part 17)

```
# List.fold;;
- : 'a list -> init:'accum -> f:('accum -> 'a -> 'accum) -> 'accum = <fun>
```

We can use `List.fold` for something as simple as summing up a list:

OCaml utop (part 18)

```
# List.fold ~init:0 ~f:(+) [1;2;3;4];;
- : int = 10
```

This example is particularly simple because the accumulator and the list elements are of the same type. But `fold` is not limited to such cases. We can for example use `fold` to reverse a list, in which case the accumulator is itself a list:

OCaml utop (part 19)

```
# List.fold ~init:[] ~f:(fun list x -> x :: list) [1;2;3;4];;
- : int list = [4; 3; 2; 1]
```

Let's bring our three functions together to compute the maximum column widths:

OCaml utop (part 20)

```
# let max_widths header rows =
    let lengths l = List.map ~f:String.length l in
    List.fold rows
      ~init:(lengths header)
      ~f:(fun acc row ->
        List.map2_exn ~f:Int.max acc (lengths row))
  ;;
val max_widths : string list -> string list list -> int list = <fun>
```

Using `List.map` we define the function `lengths`, which converts a list of strings to a list of integer lengths. `List.fold` is then used to iterate over the rows, using `map2_exn` to take the max of the accumulator with the lengths of the strings in each row of the table, with the accumulator initialized to the lengths of the header row.

Now that we know how to compute column widths, we can write the code to generate the line that separates the header from the rest of the text table. We'll do this in part by mapping `String.make` over the lengths of the columns to generate a string of dashes of the appropriate length. We'll then join these sequences of dashes together using `String.concat`, which concatenates a list of strings with an optional separator string, and `^`, which is a pairwise string concatenation function, to add the delimiters on the outside:

OCaml utop (part 21)

```
# let render_separator widths =
    let pieces = List.map widths
      ~f:(fun w -> String.make (w + 2) '-')
    in
    "|" ^ String.concat ~sep:"+" pieces ^ "|"
  ;;
val render_separator : int list -> string = <fun>
# render_separator [3;6;2];;
- : string = "|-----+--------+----|"
```

Note that we make the line of dashes two larger than the provided width to provide some whitespace around each entry in the table.

**PERFORMANCE OF STRING.CONCAT AND ^**

In the preceding code we've concatenated strings two different ways: `String.concat`, which operates on lists of strings; and `^`, which is a pairwise operator. You should avoid `^` for joining long numbers of strings, since it allocates a new string every time it runs. Thus, the following code

OCaml utop (part 22)

```
# let s = "." ^ "." ^ "." ^ "." ^ "." ^ "." ^ ".";;
val s : string = "......."
```

will allocate strings of length 2, 3, 4, 5, 6 and 7, whereas this code

OCaml utop (part 23)

```
# let s = String.concat [".";".";".";".";".";".";"."];;
val s : string = "......."
```

allocates one string of size 7, as well as a list of length 7. At these small sizes, the differences don't amount to much, but for assembling large strings, it can be a serious performance issue.

Now we need code for rendering a row with data in it. We'll first write a function called `pad`, for padding out a string to a specified length plus one blank space on both sides:

OCaml utop (part 24)

```
# let pad s length =
    " " ^ s ^ String.make (length - String.length s + 1) ' '
  ;;
val pad : string -> int -> string = <fun>
# pad "hello" 10;;
- : string = " hello      "
```

We can render a row of data by merging together the padded strings. Again, we'll use `List.map2_exn` for combining the list of data in the row with the list of widths:

OCaml utop (part 25)

```
# let render_row row widths =
    let padded = List.map2_exn row widths ~f:pad in
    "|" ^ String.concat ~sep:"|" padded ^ "|"
  ;;
val render_row : string list -> int list -> string = <fun>
# render_row ["Hello";"World"] [10;15];;
- : string = "| Hello      | World           |"
```

Now we can bring this all together in a single function that renders the table:

OCaml utop (part 26)

```
# let render_table header rows =
    let widths = max_widths header rows in
    String.concat ~sep:"\n"
      (render_row header widths
       :: render_separator widths
       :: List.map rows ~f:(fun row -> render_row row widths)
      )
  ;;
val render_table : string list -> string list list -> string = <fun>
```

## More Useful List Functions

The previous example we worked through touched on only three of the functions in `List`. We won't cover the entire interface (for that you should look at the online docs), but a few more functions are

useful enough to mention here.

### Combining list elements with List.reduce

`List.fold`, which we described earlier, is a very general and powerful function. Sometimes, however, you want something simpler and easier to use. One such function is `List.reduce`, which is essentially a specialized version of `List.fold` that doesn't require an explicit starting value, and whose accumulator has to consume and produce values of the same type as the elements of the list it applies to.

Here's the type signature:

OCaml utop (part 27)

```
# List.reduce;;
- : 'a list -> f:('a -> 'a -> 'a) -> 'a option = <fun>
```

`reduce` returns an optional result, returning `None` when the input list is empty.

Now we can see `reduce` in action:

OCaml utop (part 28)

```
# List.reduce ~f:(+) [1;2;3;4;5];;
- : int option = Some 15
# List.reduce ~f:(+) [];;
- : int option = None
```

### Filtering with List.filter and List.filter_map

Very often when processing lists, you wants to restrict your attention to a subset of the values on your list. The `List.filter` function is one way of doing that:

OCaml utop (part 29)

```
# List.filter ~f:(fun x -> x mod 2 = 0) [1;2;3;4;5];;
- : int list = [2; 4]
```

Note that the `mod` used above is an infix operator, as described in Chapter 2.

Sometimes, you want to both transform and filter as part of the same computation. In that case, `List.filter_map` is what you need. The function passed to `List.filter_map` returns an optional value, and `List.filter_map` drops all elements for which `None` is returned.

Here's an example. The following expression computes the list of file extensions in the current directory, piping the results through `List.dedup` to remove duplicates. Note that this example also uses some functions from other modules, including `Sys.ls_dir` to get a directory listing, and `String.rsplit2` to split a string on the rightmost appearance of a given character:

OCaml utop (part 30)

```
# List.filter_map (Sys.ls_dir ".") ~f:(fun fname ->
    match String.rsplit2 ~on:'.' fname with
    | None  | Some ("",_) -> None
    | Some (_,ext) ->
      Some ext)
  |> List.dedup
  ;;
- : string list = ["ascii"; "ml"; "mli"; "topscript"]
```

The preceding code is also an example of an Or pattern, which allows you to have multiple

subpatterns within a larger pattern. In this case, `None | Some ("",_)` is an Or pattern. As we'll see later, Or patterns can be nested anywhere within larger patterns.

### Partitioning with List.partition_tf

Another useful operation that's closely related to filtering is partitioning. The function `List.partition_tf` takes a list and a function for computing a Boolean condition on the list elements, and returns two lists. The `tf` in the name is a mnemonic to remind the user that `true` elements go to the first list and `false` ones go to the second. Here's an example:

OCaml utop (part 31)

```
# let is_ocaml_source s =
    match String.rsplit2 s ~on:'.' with
    | Some (_,("ml"|"mli")) -> true
    | _ -> false
  ;;
val is_ocaml_source : string -> bool = <fun>
# let (ml_files,other_files) =
    List.partition_tf (Sys.ls_dir ".")  ~f:is_ocaml_source;;
val ml_files : string list = ["example.mli"; "example.ml"]
val other_files : string list = ["main.topscript"; "lists_layout.ascii"]
```

### Combining lists

Another very common operation on lists is concatenation. The list module actually comes with a few different ways of doing this. First, there's `List.append`, for concatenating a pair of lists:

OCaml utop (part 32)

```
# List.append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

There's also `@`, an operator equivalent of `List.append`:

OCaml utop (part 33)

```
# [1;2;3] @ [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

In addition, there is `List.concat`, for concatenating a list of lists:

OCaml utop (part 34)

```
# List.concat [[1;2];[3;4;5];[6];[]];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Here's an example of using `List.concat` along with `List.map` to compute a recursive listing of a directory tree:

OCaml utop (part 35)

```
# let rec ls_rec s =
    if Sys.is_file_exn ~follow_symlinks:true s
    then [s]
    else
      Sys.ls_dir s
      |> List.map ~f:(fun sub -> ls_rec (s ^/ sub))
      |> List.concat
  ;;
val ls_rec : string -> string list = <fun>
```

Note that `^/` is an infix operator provided by Core for adding a new element to a string representing a

file path. It is equivalent to Core's `Filename.concat`.

The preceding combination of `List.map` and `List.concat` is common enough that there is a function `List.concat_map` that combines these into one, more efficient operation:

OCaml utop (part 36)

```
# let rec ls_rec s =
    if Sys.is_file_exn ~follow_symlinks:true s
    then [s]
    else
      Sys.ls_dir s
      |> List.concat_map ~f:(fun sub -> ls_rec (s ^/ sub))
  ;;
val ls_rec : string -> string list = <fun>
```

# Tail Recursion

The only way to compute the length of an OCaml list is to walk the list from beginning to end. As a result, computing the length of a list takes time linear in the size of the list. Here's a simple function for doing so:

OCaml utop (part 37)

```
# let rec length = function
    | [] -> 0
    | _ :: tl -> 1 + length tl
  ;;
val length : 'a list -> int = <fun>
# length [1;2;3];;
- : int = 3
```

This looks simple enough, but you'll discover that this implementation runs into problems on very large lists, as we'll show in the following code:

OCaml utop (part 38)

```
# let make_list n = List.init n ~f:(fun x -> x);;
val make_list : int -> int list = <fun>
# length (make_list 10);;
- : int = 10
# length (make_list 10_000_000);;
Stack overflow during evaluation (looping recursion?).
```

The preceding example creates lists using `List.init`, which takes an integer `n` and a function `f` and creates a list of length `n`, where the data for each element is created by calling `f` on the index of that element.

To understand where the error in the above example comes from, you need to learn a bit more about how function calls work. Typically, a function call needs some space to keep track of information associated with the call, such as the arguments passed to the function, or the location of the code that needs to start executing when the function call is complete. To allow for nested function calls, this information is typically organized in a stack, where a new *stack frame* is allocated for each nested function call, and then deallocated when the function call is complete.

And that's the problem with our call to `length`: it tried to allocate 10 million stack frames, which exhausted the available stack space. Happily, there's a way around this problem. Consider the following alternative implementation:

OCaml utop (part 39)

```
# let rec length_plus_n l n =
    match l with
    | [] -> n
    | _ :: tl -> length_plus_n tl (n + 1)
  ;;
val length_plus_n : 'a list -> int -> int = <fun>
# let length l = length_plus_n l 0 ;;
val length : 'a list -> int = <fun>
# length [1;2;3;4];;
- : int = 4
```

This implementation depends on a helper function, `length_plus_n`, that computes the length of a given list plus a given `n`. In practice, `n` acts as an accumulator in which the answer is built up, step by step. As a result, we can do the additions along the way rather than doing them as we unwind the

nested sequence of function calls, as we did in our first implementation of `length`.

The advantage of this approach is that the recursive call in `length_plus_n` is a *tail call*. We'll explain more precisely what it means to be a tail call shortly, but the reason it's important is that tail calls don't require the allocation of a new stack frame, due to what is called the *tail-call optimization*. A recursive function is said to be *tail recursive* if all of its recursive calls are tail calls. `length_plus_n` is indeed tail recursive, and as a result, `length` can take a long list as input without blowing the stack:

OCaml utop (part 40)

```
# length (make_list 10_000_000);;
- : int = 10000000
```

So when is a call a tail call? Let's think about the situation where one function (the *caller*) invokes another (the *callee*). The invocation is considered a tail call when the caller doesn't do anything with the value returned by the callee except to return it. The tail-call optimization makes sense because, when a caller makes a tail call, the caller's stack frame need never be used again, and so you don't need to keep it around. Thus, instead of allocating a new stack frame for the callee, the compiler is free to reuse the caller's stack frame.

Tail recursion is important for more than just lists. Ordinary nontail recursive calls are reasonable when dealing with data structures like binary trees, where the depth of the tree is logarithmic in the size of your data. But when dealing with situations where the depth of the sequence of nested calls is on the order of the size of your data, tail recursion is usually the right approach.

# Terser and Faster Patterns

Now that we know more about how lists and patterns work, let's consider how we can improve on an example from Recursive list functions: the function `destutter`, which removes sequential duplicates from a list. Here's the implementation that was described earlier:

OCaml utop (part 41)

```
# let rec destutter list =
    match list with
    | [] -> []
    | [hd] -> [hd]
    | hd :: hd' :: tl ->
      if hd = hd' then destutter (hd' :: tl)
      else hd :: destutter (hd' :: tl)
  ;;
val destutter : 'a list -> 'a list = <fun>
```

We'll consider some ways of making this code more concise and more efficient.

First, let's consider efficiency. One problem with the `destutter` code above is that it in some cases re-creates on the righthand side of the arrow a value that already existed on the lefthand side. Thus, the pattern `[hd] -> [hd]` actually allocates a new list element, when really, it should be able to just return the list being matched. We can reduce allocation here by using an `as` pattern, which allows us to declare a name for the thing matched by a pattern or subpattern. While we're at it, we'll use the `function` keyword to eliminate the need for an explicit match:

OCaml utop (part 42)

```
# let rec destutter = function
    | [] as l -> l
    | [_] as l -> l
    | hd :: (hd' :: _ as tl) ->
      if hd = hd' then destutter tl
      else hd :: destutter tl
  ;;
val destutter : 'a list -> 'a list = <fun>
```

We can further collapse this by combining the first two cases into one, using an Or pattern:

OCaml utop (part 43)

```
# let rec destutter = function
    | [] | [_] as l -> l
    | hd :: (hd' :: _ as tl) ->
      if hd = hd' then destutter tl
      else hd :: destutter tl
  ;;
val destutter : 'a list -> 'a list = <fun>
```

We can make the code slightly terser now by using a `when` clause. A `when` clause allows us to add an extra precondition to a pattern in the form of an arbitrary OCaml expression. In this case, we can use it to include the check on whether the first two elements are equal:

OCaml utop (part 44)

```
# let rec destutter = function
    | [] | [_] as l -> l
    | hd :: (hd' :: _ as tl) when hd = hd' -> destutter tl
    | hd :: tl -> hd :: destutter tl
  ;;
val destutter : 'a list -> 'a list = <fun>
```

### POLYMORPHIC COMPARE

In the preceding `destutter` example, we made use of the fact that OCaml lets us test equality between values of any type, using the `=` operator. Thus, we can write:

OCaml utop (part 45)

```
# 3 = 4;;
- : bool = false
# [3;4;5] = [3;4;5];;
- : bool = true
# [Some 3; None] = [None; Some 3];;
- : bool = false
```

Indeed, if we look at the type of the equality operator, we'll see that it is polymorphic:

OCaml utop (part 46)

```
# (=);;
- : 'a -> 'a -> bool = <fun>
```

OCaml comes with a whole family of polymorphic comparison operators, including the standard infix comparators, `<`, `>=`, etc., as well as the function `compare` that returns `-1`, `0`, or `1` to flag whether the first operand is smaller than, equal to, or greater than the second, respectively.

You might wonder how you could build functions like these yourself if OCaml didn't come with them built in. It turns out that you *can't* build these functions on your own. OCaml's polymorphic comparison functions are built into the runtime to a low level. These comparisons are polymorphic on the basis of ignoring almost everything about the types of the values that are being compared, paying attention only to the structure of the values as they're laid out in memory.

Polymorphic compare does have some limitations. For example, it will fail at runtime if it encounters a function value:

OCaml utop (part 47)

```
# (fun x -> x + 1) = (fun x -> x + 1);;
Exception: (Invalid_argument "equal: functional value").
```

Similarly, it will fail on values that come from outside the OCaml heap, like values from C bindings. But it will work in a reasonable way for other kinds of values.

For simple atomic types, polymorphic compare has the semantics you would expect: for floating-point numbers and integers, polymorphic compare corresponds to the expected numerical comparison functions. For strings, it's a lexicographic comparison.

Sometimes, however, the type-ignoring nature of polymorphic compare is a problem, particularly when you have your own notion of equality and ordering that you want to impose. We'll discuss this issue more, as well as some of the other downsides of polymorphic compare, in Chapter 13.

Note that `when` clauses have some downsides. As we noted earlier, the static checks associated with pattern matches rely on the fact that patterns are restricted in what they can express. Once we add the ability to add an arbitrary condition to a pattern, something will be lost. In particular, the ability of the compiler to determine if a match is exhaustive, or if some case is redundant, is compromised.

Consider the following function, which takes a list of optional values, and returns the number of those values that are `Some`. Because this implementation uses `when` clauses, the compiler can't tell that the code is exhaustive:

OCaml utop (part 48)

```
# let rec count_some list =
    match list with
    | [] -> 0
    | x :: tl when Option.is_none x -> count_some tl
    | x :: tl when Option.is_some x -> 1 + count_some tl
  ;;
Characters 30-169:
```

```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
_::_
(However, some guarded clause may match this value.)
val count_some : 'a option list -> int = <fun>
```

Despite the warning, the function does work fine:

OCaml utop (part 49)

```
# count_some [Some 3; None; Some 4];;
- : int = 2
```

If we add another redundant case without a `when` clause, the compiler will stop complaining about exhaustiveness and won't produce a warning about the redundancy.

OCaml utop (part 50)

```
# let rec count_some list =
    match list with
    | [] -> 0
    | x :: tl when Option.is_none x -> count_some tl
    | x :: tl when Option.is_some x -> 1 + count_some tl
    | x :: tl -> -1 (* unreachable *)
  ;;
val count_some : 'a option list -> int = <fun>
```

Probably a better approach is to simply drop the second `when` clause:

OCaml utop (part 51)

```
# let rec count_some list =
    match list with
    | [] -> 0
    | x :: tl when Option.is_none x -> count_some tl
    | _ :: tl -> 1 + count_some tl
  ;;
val count_some : 'a option list -> int = <fun>
```

This is a little less clear, however, than the direct pattern-matching solution, where the meaning of each pattern is clearer on its own:

OCaml utop (part 52)

```
# let rec count_some list =
    match list with
    | [] -> 0
    | None   :: tl -> count_some tl
    | Some _ :: tl -> 1 + count_some tl
  ;;
val count_some : 'a option list -> int = <fun>
```

The takeaway from all of this is although `when` clauses can be useful, we should prefer patterns wherever they are sufficient.

As a side note, the above implementation of `count_some` is longer than necessary; even worse, it is not tail recursive. In real life, you would probably just use the `List.count` function from `Core`:

OCaml utop (part 53)

```
# let count_some l = List.count ~f:Option.is_some l;;
val count_some : 'a option list -> int = <fun>
```

# Chapter 4. Files, Modules, and Programs

We've so far experienced OCaml largely through the toplevel. As you move from exercises to real-world programs, you'll need to leave the toplevel behind and start building programs from files. Files are more than just a convenient way to store and manage your code; in OCaml, they also correspond to modules, which act as boundaries that divide your program into conceptual units.

In this chapter, we'll show you how to build an OCaml program from a collection of files, as well as the basics of working with modules and module signatures.

# Single-File Programs

We'll start with an example: a utility that reads lines from `stdin` and computes a frequency count of the lines. At the end, the 10 lines with the highest frequency counts are written out. We'll start with a simple implementation, which we'll save as the file *freq.ml*.

This implementation will use two functions from the `List.Assoc` module, which provides utility functions for interacting with association lists, i.e., lists of key/value pairs. In particular, we use the function `List.Assoc.find`, which looks up a key in an association list; and `List.Assoc.add`, which adds a new binding to an association list, as shown here:

OCaml utop

```
# let assoc = [("one", 1); ("two",2); ("three",3)] ;;
val assoc : (string * int) list = [("one", 1); ("two", 2); ("three", 3)]
# List.Assoc.find assoc "two" ;;
- : int option = Some 2
# List.Assoc.add assoc "four" 4 (* add a new key *) ;;
- : (string, int) List.Assoc.t =
[("four", 4); ("one", 1); ("two", 2); ("three", 3)]
# List.Assoc.add assoc "two"  4 (* overwrite an existing key *) ;;
- : (string, int) List.Assoc.t = [("two", 4); ("one", 1); ("three", 3)]
```

Note that `List.Assoc.add` doesn't modify the original list, but instead allocates a new list with the requisite key/value pair added.

Now we can write `freq.ml`:

OCaml

```
open Core.Std

let build_counts () =
  In_channel.fold_lines stdin ~init:[] ~f:(fun counts line ->
    let count =
      match List.Assoc.find counts line with
      | None -> 0
      | Some x -> x
    in
    List.Assoc.add counts line (count + 1)
  )

let () =
  build_counts ()
  |> List.sort ~cmp:(fun (_,x) (_,y) -> Int.descending x y)
  |> (fun l -> List.take l 10)
  |> List.iter ~f:(fun (line,count) -> printf "%3d: %s\n" count line)
```

The function `build_counts` reads in lines from `stdin`, constructing from those lines an association list with the frequencies of each line. It does this by invoking `In_channel.fold_lines` (similar to the function `List.fold` described in Chapter 3), which reads through the lines one by one, calling the provided `fold` function for each line to update the accumulator. That accumulator is initialized to the empty list.

With `build_counts` defined, we then call the function to build the association list, sort that list by frequency in descending order, grab the first 10 elements off the list, and then iterate over those 10 elements and print them to the screen. These operations are tied together using the `|>` operator described in Chapter 2.

### WHERE IS THE MAIN FUNCTION?

Unlike C, programs in OCaml do not have a unique `main` function. When an OCaml program is evaluated, all the statements in the implementation files are evaluated in the order in which they were linked together. These implementation files can contain arbitrary expressions, not just function definitions. In this example, the declaration starting with `let () =` plays the role of the `main` function, kicking off the processing. But really the entire file is evaluated at startup, and so in some sense the full codebase is one big `main` function.

The idiom of writing `let () =` may seem a bit odd, but it has a purpose. The `let` binding here is a pattern-match to a value of type `unit`, which is there to ensure that the expression on the righthand side returns `unit`, as is common for functions that operate primarily by side effect.

If we weren't using Core or any other external libraries, we could build the executable like this:

Terminal

```
$ ocamlc freq.ml -o freq.byte
File "freq.ml", line 1, characters 0-13:
Error: Unbound module Core
```

But as you can see, it fails because it can't find Core. We need a somewhat more complex invocation to get Core linked in:

Terminal

```
$ ocamlfind ocamlc -linkpkg -thread -package core freq.ml -o freq.byte
```

This uses *ocamlfind*, a tool which itself invokes other parts of the OCaml toolchain (in this case, *ocamlc*) with the appropriate flags to link in particular libraries and packages. Here, `-package core` is asking *ocamlfind* to link in the Core library; `-linkpkg` asks ocamlfind to link in the packages as is necessary for building an executable, while `-thread` turns on threading support (which is required for Core).

While this works well enough for a one-file project, more complicated projects require a tool to orchestrate the build. One good tool for this task is *ocamlbuild*, which is shipped with the OCaml compiler. We'll talk more about *ocamlbuild* in Chapter 22, but for now, we'll just use a simple wrapper around *ocamlbuild* called *corebuild* that sets build parameters appropriately for building against Core and its related libraries:

Terminal

```
$ corebuild freq.byte
```

If we'd invoked *corebuild* with a target of `freq.native` instead of `freq.byte`, we would have gotten native code instead.

We can run the resulting executable from the command line. The following line extracts strings from the *ocamlopt* binary, reporting the most frequently occurring ones. Note that the specific results will vary from platform to platform, since the binary itself will differ between platforms:

Terminal

```
$ strings `which ocamlopt` | ./freq.byte
  6: +pci_expr =
  6: -pci_params =
  6: .pci_virt = %a
  4: #lsr
  4: #lsl
  4: $lxor
```

```
4: #lor
4: $land
4: #mod
3: 6    .section .rdata,"dr"
```

### BYTECODE VERSUS NATIVE CODE

OCaml ships with two compilers: the *ocamlc* bytecode compiler and the *ocamlopt* native-code compiler. Programs compiled with *ocamlc* are interpreted by a virtual machine, while programs compiled with *ocamlopt* are compiled to native machine code to be run on a specific operating system and processor architecture. With *ocamlbuild*, targets ending with `.byte` are build as bytecode executables, and those ending with `.native` are built as native code.

Aside from performance, executables generated by the two compilers have nearly identical behavior. There are a few things to be aware of. First, the bytecode compiler can be used on more architectures, and has some tools that are not available for native code. For example, the OCaml debugger only works with bytecode (although *gdb*, the GNU Debugger, works with OCaml native-code applications). The bytecode compiler is also quicker than the native-code compiler. In addition, in order to run a bytecode executable, you typically need to have OCaml installed on the system in question. That's not strictly required, though, since you can build a bytecode executable with an embedded runtime, using the `-custom` compiler flag.

As a general matter, production executables should usually be built using the native-code compiler, but it sometimes makes sense to use bytecode for development builds. And, of course, bytecode makes sense when targeting a platform not supported by the native-code compiler. We'll cover both compilers in more detail in Chapter 23.

# Multifile Programs and Modules

Source files in OCaml are tied into the module system, with each file compiling down into a module whose name is derived from the name of the file. We've encountered modules before, such as when we used functions like `find` and `add` from the `List.Assoc` module. At its simplest, you can think of a module as a collection of definitions that are stored within a namespace.

Let's consider how we can use modules to refactor the implementation of `freq.ml`. Remember that the variable `counts` contains an association list representing the counts of the lines seen so far. But updating an association list takes time linear in the length of the list, meaning that the time complexity of processing a file is quadratic in the number of distinct lines in the file.

We can fix this problem by replacing association lists with a more efficient data structure. To do that, we'll first factor out the key functionality into a separate module with an explicit interface. We can consider alternative (and more efficient) implementations once we have a clear interface to program against.

We'll start by creating a file, `counter.ml`, that contains the logic for maintaining the association list used to represent the frequency counts. The key function, called `touch`, bumps the frequency count of a given line by one:

OCaml

```
open Core.Std

let touch t s =
  let count =
    match List.Assoc.find t s with
    | None -> 0
    | Some x -> x
  in
  List.Assoc.add t s (count + 1)
```

The file *counter.ml* will be compiled into a module named `Counter`, where the name of the module is derived automatically from the filename. The module name is capitalized even if the file is not. Indeed, module names are always capitalized.

We can now rewrite `freq.ml` to use `Counter`. Note that the resulting code can still be built with *ocamlbuild*, which will discover dependencies and realize that `counter.ml` needs to be compiled:

OCaml

```
open Core.Std

let build_counts () =
  In_channel.fold_lines stdin ~init:[] ~f:Counter.touch

let () =
  build_counts ()
  |> List.sort ~cmp:(fun (_,x) (_,y) -> Int.descending x y)
  |> (fun l -> List.take l 10)
  |> List.iter ~f:(fun (line,count) -> printf "%3d: %s\n" count line)
```

# Signatures and Abstract Types

While we've pushed some of the logic to the `Counter` module, the code in `freq.ml` can still depend on the details of the implementation of `Counter`. Indeed, if you look at the definition of `build_counts`, you'll see that it depends on the fact that the empty set of frequency counts is represented as an empty list. We'd like to prevent this kind of dependency, so we can change the implementation of `Counter` without needing to change client code like that in `freq.ml`.

The implementation details of a module can be hidden by attaching an *interface*. (Note that in the context of OCaml, the terms *interface*, *signature*, and *module type* are all used interchangeably.) A module defined by a file `filename.ml` can be constrained by a signature placed in a file called `filename.mli`.

For `counter.mli`, we'll start by writing down an interface that describes what's currently available in `counter.ml`, without hiding anything. `val` declarations are used to specify values in a signature. The syntax of a `val` declaration is as follows:

Syntax

```
val <identifier> : <type>
```

Using this syntax, we can write the signature of `counter.ml` as follows:

OCaml

```
open Core.Std

(** Bump the frequency count for the given string. *)
val touch : (string * int) list -> string -> (string * int) list
```

Note that *ocamlbuild* will detect the presence of the `mli` file automatically and include it in the build.

**AUTOGENERATING MLI FILES**

If you don't want to construct an `mli` entirely by hand, you can ask OCaml to autogenerate one for you from the source, which you can then adjust to fit your needs. Here's how you can do that using *corebuild*:

Terminal

```
$ corebuild counter.inferred.mli
$ cat _build/counter.inferred.mli
val touch :
  ('a, int) Core.Std.List.Assoc.t -> 'a -> ('a, int) Core.Std.List.Assoc.t
```

The generated code is basically equivalent to the `mli` that we wrote by hand but is a bit uglier and more verbose and, of course, has no comments. In general, autogenerated `mli`s are only useful as a starting point. In OCaml, the `mli` is the key place where you present and document your interface, and there's no replacement for careful human editing and organization.

To hide the fact that frequency counts are represented as association lists, we'll need to make the type of frequency counts *abstract*. A type is abstract if its name is exposed in the interface, but its definition is not. Here's an abstract interface for `Counter`:

OCaml

```
open Core.Std

(** A collection of string frequency counts *)
type t

(** The empty set of frequency counts  *)
```

```
val empty : t

(** Bump the frequency count for the given string. *)
val touch : t -> string -> t

(** Converts the set of frequency counts to an association list.  A string shows
    up at most once, and the counts are >= 1. *)
val to_list : t -> (string * int) list
```

Note that we needed to add `empty` and `to_list` to `Counter`, since otherwise there would be no way to create a `Counter.t` or get data out of one.

We also used this opportunity to document the module. The `mli` file is the place where you specify your module's interface, and as such is a natural place to put documentation. We started our comments with a double asterisk to cause them to be picked up by the *ocamldoc* tool when generating API documentation. We'll discuss *ocamldoc* more in Chapter 22.

Here's a rewrite of `counter.ml` to match the new `counter.mli`:

OCaml

```
open Core.Std

type t = (string * int) list

let empty = []

let to_list x = x

let touch t s =
  let count =
    match List.Assoc.find t s with
    | None -> 0
    | Some x -> x
  in
  List.Assoc.add t s (count + 1)
```

If we now try to compile `freq.ml`, we'll get the following error:

Terminal

```
$ corebuild freq.byte
File "freq.ml", line 4, characters 42-55:
Error: This expression has type Counter.t -> string -> Counter.t
       but an expression was expected of type 'a list -> string -> 'a list
       Type Counter.t is not compatible with type 'a list
Command exited with code 2.
```

This is because `freq.ml` depends on the fact that frequency counts are represented as association lists, a fact that we've just hidden. We just need to fix `build_counts` to use `Counter.empty` instead of `[]` and `Counter.to_list` to get the association list out at the end for processing and printing. The resulting implementation is shown below:

OCaml

```
open Core.Std

let build_counts () =
  In_channel.fold_lines stdin ~init:Counter.empty ~f:Counter.touch

let () =
  build_counts ()
  |> Counter.to_list
  |> List.sort ~cmp:(fun (_,x) (_,y) -> Int.descending x y)
```

```
  |> (fun counts -> List.take counts 10)
  |> List.iter ~f:(fun (line,count) -> printf "%3d: %s\n" count line)
```

Now we can turn to optimizing the implementation of `Counter`. Here's an alternate and far more efficient implementation, based on the `Map` data structure in Core:

OCaml

```
open Core.Std

type t = int String.Map.t

let empty = String.Map.empty

let to_list t = Map.to_alist t

let touch t s =
  let count =
    match Map.find t s with
    | None -> 0
    | Some x -> x
  in
  Map.add t ~key:s ~data:(count + 1)
```

Note that in the preceding example we use `String.Map` in some places and simply `Map` in others. This has to do with the fact that for some operations, like creating a `Map.t`, you need access to type-specialized information, and for others, like looking something up in `Map.t`, you don't. This is covered in more detail in Chapter 13.

# Concrete Types in Signatures

In our frequency-count example, the module `Counter` had an abstract type `Counter.t` for representing a collection of frequency counts. Sometimes, you'll want to make a type in your interface *concrete*, by including the type definition in the interface.

For example, imagine we wanted to add a function to `Counter` for returning the line with the median frequency count. If the number of lines is even, then there is no precise median, and the function would return the lines before and after the median instead. We'll use a custom type to represent the fact that there are two possible return values. Here's a possible implementation:

OCaml (part 1)

```
type median = | Median of string
              | Before_and_after of string * string

let median t =
  let sorted_strings = List.sort (Map.to_alist t)
                         ~cmp:(fun (_,x) (_,y) -> Int.descending x y)
  in
  let len = List.length sorted_strings in
  if len = 0 then failwith "median: empty frequency count";
  let nth n = fst (List.nth_exn sorted_strings n) in
  if len mod 2 = 1
  then Median (nth (len/2))
  else Before_and_after (nth (len/2 - 1), nth (len/2));;
```

In the preceding implementation, we use `failwith` to throw an exception for the case of the empty list. We'll discuss exceptions more in Chapter 7. Note also that the function `fst` simply returns the first element of any two-tuple.

Now, to expose this usefully in the interface, we need to expose both the function and the type `median` with its definition. Note that values (of which functions are an example) and types have distinct namespaces, so there's no name clash here. Adding the following two lines added to `counter.mli` does the trick:

OCaml (part 1)

```
(** Represents the median computed from a set of strings.  In the case where
    there is an even number of choices, the one before and after the median is
    returned.  *)
type median = | Median of string
              | Before_and_after of string * string

val median : t -> median
```

The decision of whether a given type should be abstract or concrete is an important one. Abstract types give you more control over how values are created and accessed, and make it easier to enforce invariants beyond what is enforced by the type itself; concrete types let you expose more detail and structure to client code in a lightweight way. The right choice depends very much on the context.

# Nested Modules

Up until now, we've only considered modules that correspond to files, like `counter.ml`. But modules (and module signatures) can be nested inside other modules. As a simple example, consider a program that needs to deal with multiple identifiers like usernames and hostnames. If you just represent these as strings, then it becomes easy to confuse one with the other.

A better approach is to mint new abstract types for each identifier, where those types are under the covers just implemented as strings. That way, the type system will prevent you from confusing a username with a hostname, and if you do need to convert, you can do so using explicit conversions to and from the string type.

Here's how you might create such an abstract type, within a submodule:

OCaml

```
open Core.Std

module Username : sig
  type t
  val of_string : string -> t
  val to_string : t -> string
end = struct
  type t = string
  let of_string x = x
  let to_string x = x
end
```

Note that the `to_string` and `of_string` functions above are implemented simply as the identity function, which means they have no runtime effect. They are there purely as part of the discipline that they enforce on the code through the type system.

The basic structure of a module declaration like this is:

Syntax

```
module <name> : <signature> = <implementation>
```

We could have written this slightly differently, by giving the signature its own top-level `module type` declaration, making it possible to create multiple distinct types with the same underlying implementation in a lightweight way:

OCaml

```
open Core.Std

module type ID = sig
  type t
  val of_string : string -> t
  val to_string : t -> string
end

module String_id = struct
  type t = string
  let of_string x = x
  let to_string x = x
end

module Username : ID = String_id
module Hostname : ID = String_id
```

```
type session_info = { user: Username.t;
                      host: Hostname.t;
                      when_started: Time.t;
                    }

let sessions_have_same_user s1 s2 =
  s1.user = s2.host
```

The preceding code has a bug: it compares the username in one session to the host in the other session, when it should be comparing the usernames in both cases. Because of how we defined our types, however, the compiler will flag this bug for us:

Terminal

```
$ corebuild session_info.native
File "session_info.ml", line 24, characters 12-19:
Error: This expression has type Hostname.t
       but an expression was expected of type Username.t
Command exited with code 2.
```

This is a trivial example, but confusing different kinds of identifiers is a very real source of bugs, and the approach of minting abstract types for different classes of identifiers is an effective way of avoiding such issues.

# Opening Modules

Most of the time, you refer to values and types within a module by using the module name as an explicit qualifier. For example, you write `List.map` to refer to the `map` function in the `List` module. Sometimes, though, you want to be able to refer to the contents of a module without this explicit qualification. That's what the `open` statement is for.

We've encountered `open` already, specifically where we've written `open Core.Std` to get access to the standard definitions in the Core library. In general, opening a module adds the contents of that module to the environment that the compiler looks at to find the definition of various identifiers. Here's an example:

OCaml utop

```
# module M = struct let foo = 3 end;;
module M : sig val foo : int end
# foo;;
Characters -1-3:
Error: Unbound value foo
# open M;;
# foo;;
- : int = 3
```

`open` is essential when you want to modify your environment for a standard library like Core, but it's generally good style to keep the opening of modules to a minimum. Opening a module is basically a trade-off between terseness and explicitness — the more modules you open, the fewer module qualifications you need, and the harder it is to look at an identifier and figure out where it comes from.

Here's some general advice on how to deal with `open`s:

- Opening modules at the toplevel of a module should be done quite sparingly, and generally only with modules that have been specifically designed to be opened, like `Core.Std` or `Option.Monad_infix`.
- If you do need to do an open, it's better to do a *local open*. There are two syntaxes for local opens. For example, you can write:
  OCaml utop (part 1)

  ```
  # let average x y =
      let open Int64 in
      x + y / of_int 2;;
  val average : int64 -> int64 -> int64 = <fun>
  ```

  Here, `of_int` and the infix operators are the ones from the `Int64` module.
  There's another, even more lightweight syntax for local `open`s, which is particularly useful for small expressions:
  OCaml utop (part 2)

  ```
  # let average x y =
      Int64.(x + y / of_int 2);;
  val average : int64 -> int64 -> int64 = <fun>
  ```

- An alternative to local `open`s that makes your code terser without giving up on explicitness is to locally rebind the name of a module. So, when using the `Counter.median` type, instead of writing:

OCaml (part 1)

```
let print_median m =
  match m with
  | Counter.Median string -> printf "True median:\n   %s\n" string
  | Counter.Before_and_after (before, after) ->
    printf "Before and after median:\n   %s\n   %s\n" before after
```

you could write:

OCaml (part 1)

```
let print_median m =
  let module C = Counter in
  match m with
  | C.Median string -> printf "True median:\n   %s\n" string
  | C.Before_and_after (before, after) ->
    printf "Before and after median:\n   %s\n   %s\n" before after
```

Because the module name `C` only exists for a short scope, it's easy to read and remember what `C` stands for. Rebinding modules to very short names at the top level of your module is usually a mistake.

# Including Modules

While opening a module affects the environment used to search for identifiers, *including* a module is a way of actually adding new identifiers to a module proper. Consider the following simple module for representing a range of integer values:

OCaml utop (part 3)

```
# module Interval = struct
    type t = | Interval of int * int
             | Empty

    let create low high =
      if high < low then Empty else Interval (low,high)
  end;;
module Interval :
  sig type t = Interval of int * int | Empty val create : int -> int -> t end
```

We can use the `include` directive to create a new, extended version of the `Interval` module:

OCaml utop (part 4)

```
# module Extended_interval = struct
    include Interval

    let contains t x =
      match t with
      | Empty -> false
      | Interval (low,high) -> x >= low && x <= high
  end;;
module Extended_interval :
  sig
    type t = Interval.t = Interval of int * int | Empty
    val create : int -> int -> t
    val contains : t -> int -> bool
  end
# Extended_interval.contains (Extended_interval.create 3 10) 4;;
- : bool = true
```

The difference between `include` and `open` is that we've done more than change how identifiers are searched for: we've changed what's in the module. If we'd used `open`, we'd have gotten a quite different result:

OCaml utop (part 5)

```
# module Extended_interval = struct
    open Interval

    let contains t x =
      match t with
      | Empty -> false
      | Interval (low,high) -> x >= low && x <= high
  end;;
module Extended_interval :
  sig val contains : Extended_interval.t -> int -> bool end
# Extended_interval.contains (Extended_interval.create 3 10) 4;;
Characters 28-52:
Error: Unbound value Extended_interval.create
```

To consider a more realistic example, imagine you wanted to build an extended version of the `List` module, where you've added some functionality not present in the module as distributed in Core. `include` allows us to do just that:

OCaml

```
open Core.Std

(* The new function we're going to add *)
let rec intersperse list el =
  match list with
  | [] | [ _ ]   -> list
  | x :: y :: tl -> x :: el :: intersperse (y::tl) el

(* The remainder of the list module *)
include List
```

Now, how do we write an interface for this new module? It turns out that `include` works on signatures as well, so we can pull essentially the same trick to write our `mli`. The only issues is that we need to get our hands on the signature for the `List` module. This can be done using `module type of`, which computes a signature from a module:

OCaml

```
open Core.Std

(* Include the interface of the list module from Core *)
include (module type of List)

(* Signature of function we're adding *)
val intersperse : 'a list -> 'a -> 'a list
```

Note that the order of declarations in the `mli` does not need to match the order of declarations in the `ml`. The order of declarations in the `ml` mostly matters insofar as it affects which values are shadowed. If we wanted to replace a function in `List` with a new function of the same name, the declaration of that function in the `ml` would have to come after the `include List` declaration.

We can now use `Ext_list` as a replacement for `List`. If we want to use `Ext_list` in preference to `List` in our project, we can create a file of common definitions:

OCaml

```
module List = Ext_list
```

And if we then put `open Common` after `open Core.Std` at the top of each file in our project, then references to `List` will automatically go to `Ext_list` instead.

# Common Errors with Modules

When OCaml compiles a program with an `ml` and an `mli`, it will complain if it detects a mismatch between the two. Here are some of the common errors you'll run into.

## Type Mismatches

The simplest kind of error is where the type specified in the signature does not match the type in the implementation of the module. As an example, if we replace the `val` declaration in `counter.mli` by swapping the types of the first two arguments:

OCaml (part 1)

```
(** Bump the frequency count for the given string. *)
val touch : string -> t -> t
```

and we try to compile, we'll get the following error:

Terminal

```
$ corebuild freq.byte
File "freq.ml", line 4, characters 53-66:
Error: This expression has type string -> Counter.t -> Counter.t
       but an expression was expected of type
         Counter.t -> string -> Counter.t
       Type string is not compatible with type Counter.t
Command exited with code 2.
```

## Missing Definitions

We might decide that we want a new function in `Counter` for pulling out the frequency count of a given string. We can update the `mli` by adding the following line:

OCaml (part 1)

```
val count : t -> string -> int
```

Now, if we try to compile without actually adding the implementation, we'll get this error:

Terminal

```
$ corebuild freq.byte
File "counter.ml", line 1:
Error: The implementation counter.ml
       does not match the interface counter.cmi:
       The field `count' is required but not provided
Command exited with code 2.
```

A missing type definition will lead to a similar error.

## Type Definition Mismatches

Type definitions that show up in an `mli` need to match up with corresponding definitions in the `ml`. Consider again the example of the type `median`. The order of the declaration of variants matters to the OCaml compiler, so the definition of `median` in the implementation listing those options in a different order:

OCaml (part 1)

```
(** Represents the median computed from a set of strings.  In the case where
    there is an even number of choices, the one before and after the median is
    returned.  *)
type median = | Before_and_after of string * string
```

```
              | Median of string
```

will lead to a compilation error:

Terminal

```
$ corebuild freq.byte
File "counter.ml", line 1:
Error: The implementation counter.ml
       does not match the interface counter.cmi:
       Type declarations do not match:
         type median = Median of string | Before_and_after of string * string
       is not included in
         type median = Before_and_after of string * string | Median of string
       File "counter.ml", line 18, characters 5-84: Actual declaration
       Fields number 1 have different names, Median and Before_and_after.
Command exited with code 2.
```

Order is similarly important to other type declarations, including the order in which record fields are declared and the order of arguments (including labeled and optional arguments) to a function.

## Cyclic Dependencies

In most cases, OCaml doesn't allow cyclic dependencies, i.e., a collection of definitions that all refer to one another. If you want to create such definitions, you typically have to mark them specially. For example, when defining a set of mutually recursive values (like the definition of `is_even` and `is_odd` in Recursive Functions), you need to define them using `let rec` rather than ordinary `let`.

The same is true at the module level. By default, cyclic dependencies between modules are not allowed, and cyclic dependencies among files are never allowed. Recursive modules are possible but are a rare case, and we won't discuss them further here.

The simplest example of a forbidden circular reference is a module referring to its own module name. So, if we tried to add a reference to `Counter` from within `counter.ml`:

OCaml (part 1)

```
let singleton l = Counter.touch Counter.empty
```

we'll see this error when we try to build:

Terminal

```
$ corebuild freq.byte
File "counter.ml", line 18, characters 18-31:
Error: Unbound module Counter
Command exited with code 2.
```

The problem manifests in a different way if we create cyclic references between files. We could create such a situation by adding a reference to `Freq` from `counter.ml`, e.g., by adding the following line:

OCaml (part 1)

```
let _build_counts = Freq.build_counts
```

In this case, *ocamlbuild* (which is invoked by the *corebuild* script) will notice the error and complain explicitly about the cycle:

Terminal

```
$ corebuild freq.byte
Circular dependencies: "freq.cmo" already seen in
```

```
[ "counter.cmo"; "freq.cmo" ]
```

# Designing with Modules

The module system is a key part of how an OCaml program is structured. As such, we'll close this chapter with some advice on how to think about designing that structure effectively.

## Expose Concrete Types Rarely

When designing an `mli`, one choice that you need to make is whether to expose the concrete definition of your types or leave them abstract. Most of the time, abstraction is the right choice, for two reasons: it enhances the flexibility of your design, and it makes it possible to enforce invariants on the use of your module.

Abstraction enhances flexibility by restricting how users can interact with your types, thus reducing the ways in which users can depend on the details of your implementation. If you expose types explicitly, then users can depend on any and every detail of the types you choose. If they're abstract, then only the specific operations you want to expose are available. This means that you can freely change the implementation without affecting clients, as long as you preserve the semantics of those operations.

In a similar way, abstraction allows you to enforce invariants on your types. If your types are exposed, then users of the module can create new instances of that type (or if mutable, modify existing instances) in any way allowed by the underlying type. That may violate a desired invariant *i.e.*, a property about your type that is always supposed to be true. Abstract types allow you to protect invariants by making sure that you only expose functions that preserves your invariants.

Despite these benefits, there is a trade-off here. In particular, exposing types concretely makes it possible to use pattern-matching with those types, which as we saw in Chapter 3 is a powerful and important tool. You should generally only expose the concrete implementation of your types when there's significant value in the ability to pattern match, and when the invariants that you care about are already enforced by the data type itself.

## Design for the Call Site

When writing an interface, you should think not just about how easy it is to understand the interface for someone who reads your carefully documented `mli` file, but more importantly, you want the call to be as obvious as possible for someone who is reading it at the call site.

The reason for this is that most of the time, people interacting with your API will be doing so by reading and modifying code that uses the API, not by reading the interface definition. By making your API as obvious as possible from that perspective, you simplify the lives of your users.

There are many ways of improving readability at the call site. One example is labeled arguments (discussed in Labeled Arguments), which act as documentation that is available at the call site.

You can also improve readability simply by choosing good names for your functions, variant tags and record fields. Good names aren't always long, to be clear. If you wanted to write an anonymous function for doubling a number: `(fun x -> x * 2)`, a short variable name like `x` is best. A good rule of thumb is that names that have a small scope should be short, whereas names that have a large scope, like the name of a function in an a module interface, should be longer and more explicit.

There is of course a tradeoff here, in that making your APIs more explicit tends to make them more verbose as well. Another useful rule of thumb is that more rarely used names should be longer and

more explicit, since the cost of concision and the benefit of explicitness become more important the more often a name is used.

## Create Uniform Interfaces

Designing the interface of a module is a task that should not be thought of in isolation. The interfaces that appear in your codebase should play together harmoniously. Part of achieving that is standardizing aspects of those interfaces.

Core itself is a library that works hard to create uniform interfaces. Here are some of the guidelines that are used in Core.

- *A module for (almost) every type.* You should mint a module for almost every type in your program, and the primary type of a given module should be called `t`.
- *Put `t` first.* If you have a module `M` whose primary type is `M.t`, the functions in `M` that take a value of `M.t` should take it as their first argument.
- Functions that routinely throw an exception should end in `_exn`. Otherwise, errors should be signaled by returning an `option` or an `Or_error.t` (both of which are discussed in Chapter 7 ).

There are also standards in Core about what the type signature for specific functions should be. For example, the signature for `map` is always essentially the same, no matter what the underlying type it is applied to. This kind of function-by-function API uniformity is achieved through the use of *signature includes*, which allow for different modules to share components of their interface. This approach is described in Using Multiple Interfaces.

Core's standards may or may not fit your projects, but you can improve the usability of your codebase by finding some consistent set of standards to apply.

## Interfaces before implementations

OCaml's concise and flexible type language enables a type-oriented approach to software design. Such an approach involves thinking through and writing out the types you're going to use before embarking on the implementation itself.

This is a good approach both when working in the core language, where you would write your type definitions before writing the logic of your computations, as well as at the module level, where you would write a first draft of your `mli` before working on the `ml`.

Of course, the design process goes in both directions. You'll often find yourself going back and modifying your types in response to things you learn by working on the implementation. But types and signatures provide a lightweight tool for constructing a skeleton of your design in a way that helps clarify your goals and intent, before you spend a lot of time and effort fleshing it out.

# Chapter 5. Records

One of OCaml's best features is its concise and expressive system for declaring new data types, and records are a key element of that system. We discussed records briefly in Chapter 1, but this chapter will go into more depth, covering the details of how records work, as well as advice on how to use them effectively in your software designs.

A record represents a collection of values stored together as one, where each component is identified by a different field name. The basic syntax for a record type declaration is as follows:

### Syntax

```
type <record-name> =
    { <field> : <type> ;
    <field> : <type> ;
    ...
    }
```

Note that record field names must start with a lowercase letter.

Here's a simple example, a `host_info` record that summarizes information about a given computer:

### OCaml utop

```
# type host_info =
    { hostname   : string;
      os_name    : string;
      cpu_arch   : string;
      timestamp  : Time.t;
    };;
type host_info = {
  hostname : string;
  os_name : string;
  cpu_arch : string;
  timestamp : Time.t;
}
```

We can construct a `host_info` just as easily. The following code uses the `Shell` module from `Core_extended` to dispatch commands to the shell to extract the information we need about the computer we're running on. It also uses the `Time.now` call from Core's `Time` module:

### OCaml utop (part 1)

```
# #require "core_extended";;
# open Core_extended.Std;;
# let my_host =
    let sh = Shell.sh_one_exn in
    { hostname   = sh "hostname";
      os_name    = sh "uname -s";
      cpu_arch   = sh "uname -p";
      timestamp  = Time.now ();
    };;
val my_host : host_info =
  {hostname = "ocaml-www1"; os_name = "Linux"; cpu_arch = "unknown";
   timestamp = 2013-08-18 14:50:48.986085+01:00}
```

You might wonder how the compiler inferred that `my_host` is of type `host_info`. The hook that the compiler uses in this case to figure out the type is the record field name. Later in the chapter, we'll talk about what happens when there is more than one record type in scope with the same field name.

Once we have a record value in hand, we can extract elements from the record field using dot

notation:

OCaml utop (part 2)

```
# my_host.cpu_arch;;
- : string = "unknown"
```

When declaring an OCaml type, you always have the option of parameterizing it by a polymorphic type. Records are no different in this regard. So, for example, here's a type one might use to timestamp arbitrary items:

OCaml utop (part 3)

```
# type 'a timestamped = { item: 'a; time: Time.t };;
type 'a timestamped = { item : 'a; time : Time.t; }
```

We can then write polymorphic functions that operate over this parameterized type:

OCaml utop (part 4)

```
# let first_timestamped list =
    List.reduce list ~f:(fun a b -> if a.time < b.time then a else b)
  ;;
val first_timestamped : 'a timestamped list -> 'a timestamped option = <fun>
```

# Patterns and Exhaustiveness

Another way of getting information out of a record is by using a pattern match, as in the definition of `host_info_to_string`:

OCaml utop (part 5)

```
# let host_info_to_string { hostname = h; os_name = os;
                            cpu_arch = c; timestamp = ts;
                          } =
      sprintf "%s (%s / %s, on %s)" h os c (Time.to_sec_string ts);;
val host_info_to_string : host_info -> string = <fun>
# host_info_to_string my_host;;
- : string = "ocaml-www1 (Linux / unknown, on 2013-08-18 14:50:48)"
```

Note that the pattern we used had only a single case, rather than using several cases separated by `|`'s. We needed only one pattern because record patterns are *irrefutable*, meaning that a record pattern match will never fail at runtime. This makes sense, because the set of fields available in a record is always the same. In general, patterns for types with a fixed structure, like records and tuples, are irrefutable, unlike types with variable structures like lists and variants.

Another important characteristic of record patterns is that they don't need to be complete; a pattern can mention only a subset of the fields in the record. This can be convenient, but it can also be error prone. In particular, this means that when new fields are added to the record, code that should be updated to react to the presence of those new fields will not be flagged by the compiler.

As an example, imagine that we wanted to add a new field to our `host_info` record called `os_release`:

OCaml utop (part 6)

```
# type host_info =
    { hostname   : string;
      os_name    : string;
      cpu_arch   : string;
      os_release : string;
      timestamp  : Time.t;
    } ;;
type host_info = {
  hostname : string;
  os_name : string;
  cpu_arch : string;
  os_release : string;
  timestamp : Time.t;
}
```

The code for `host_info_to_string` would continue to compile without change. In this particular case, it's pretty clear that you might want to update `host_info_to_string` in order to include `os_release`, and it would be nice if the type system would give you a warning about the change.

Happily, OCaml does offer an optional warning for missing fields in a record pattern. With that warning turned on (which you can do in the toplevel by typing `#warnings "+9"`), the compiler will warn about the missing field:

OCaml utop (part 7)

```
# #warnings "+9";;
# let host_info_to_string { hostname = h; os_name = os;
                            cpu_arch = c; timestamp = ts;
                          } =
```

```
    sprintf "%s (%s / %s, on %s)" h os c (Time.to_sec_string ts);;
Characters 24-139:
Warning 9: the following labels are not bound in this record pattern:
os_release
Either bind these labels explicitly or add '; _' to the pattern.
val host_info_to_string : host_info -> string = <fun>
```

We can disable the warning for a given pattern by explicitly acknowledging that we are ignoring extra fields. This is done by adding an underscore to the pattern:

OCaml utop (part 8)

```
# let host_info_to_string { hostname = h; os_name = os;
                            cpu_arch = c; timestamp = ts; _
                          } =
    sprintf "%s (%s / %s, on %s)" h os c (Time.to_sec_string ts);;
val host_info_to_string : host_info -> string = <fun>
```

It's a good idea to enable the warning for incomplete record matches and to explicitly disable it with an _ where necessary.

**COMPILER WARNINGS**

The OCaml compiler is packed full of useful warnings that can be enabled and disabled separately. These are documented in the compiler itself, so we could have found out about warning 9 as follows:

Terminal

```
$ ocaml -warn-help | egrep '\b9\b'
  9 Missing fields in a record pattern.
  R Synonym for warning 9.
```

You should think of OCaml's warnings as a powerful set of optional static analysis tools, and you should eagerly enable them in your build environment. You don't typically enable all warnings, but the defaults that ship with the compiler are pretty good.

The warnings used for building the examples in this book are specified with the following flag: `-w @A-4-33-41-42-43-34-44`.

The syntax of this can be found by running `ocaml -help`, but this particular invocation turns on all warnings as errors, disabling only the numbers listed explicitly after the `A`.

Treating warnings as errors (i.e., making OCaml fail to compile any code that triggers a warning) is good practice, since without it, warnings are too often ignored during development. When preparing a package for distribution, however, this is a bad idea, since the list of warnings may grow from one release of the compiler to another, and so this may lead your package to fail to compile on newer compiler releases.

# Field Punning

When the name of a variable coincides with the name of a record field, OCaml provides some handy syntactic shortcuts. For example, the pattern in the following function binds all of the fields in question to variables of the same name. This is called *field punning*:

OCaml utop (part 9)

```
# let host_info_to_string { hostname; os_name; cpu_arch; timestamp; _ } =
    sprintf "%s (%s / %s) <%s>" hostname os_name cpu_arch
      (Time.to_string timestamp);;
val host_info_to_string : host_info -> string = <fun>
```

Field punning can also be used to construct a record. Consider the following code for generating a `host_info` record:

OCaml utop (part 10)

```
# let my_host =
    let sh cmd = Shell.sh_one_exn cmd in
    let hostname   = sh "hostname" in
    let os_name    = sh "uname -s" in
    let cpu_arch   = sh "uname -p" in
    let os_release = sh "uname -r" in
    let timestamp  = Time.now () in
    { hostname; os_name; cpu_arch; os_release; timestamp };;
val my_host : host_info =
  {hostname = "ocaml-www1"; os_name = "Linux"; cpu_arch = "unknown";
   os_release = "3.2.0-1-amd64";
   timestamp = 2013-08-18 14:50:55.287342+01:00}
```

In the preceding code, we defined variables corresponding to the record fields first, and then the record declaration itself simply listed the fields that needed to be included.

You can take advantage of both field punning and label punning when writing a function for constructing a record from labeled arguments:

OCaml utop (part 11)

```
# let create_host_info ~hostname ~os_name ~cpu_arch ~os_release =
    { os_name; cpu_arch; os_release;
      hostname = String.lowercase hostname;
      timestamp = Time.now () };;
val create_host_info :
  hostname:string ->
  os_name:string -> cpu_arch:string -> os_release:string -> host_info = <fun>
```

This is considerably more concise than what you would get without punning:

OCaml utop (part 12)

```
# let create_host_info
    ~hostname:hostname ~os_name:os_name
    ~cpu_arch:cpu_arch ~os_release:os_release =
    { os_name = os_name;
      cpu_arch = cpu_arch;
      os_release = os_release;
      hostname = String.lowercase hostname;
      timestamp = Time.now () };;
val create_host_info :
  hostname:string ->
  os_name:string -> cpu_arch:string -> os_release:string -> host_info = <fun>
```

Together, labeled arguments, field names, and field and label punning encourage a style where you

propagate the same names throughout your codebase. This is generally good practice, since it encourages consistent naming, which makes it easier to navigate the source.

# Reusing Field Names

Defining records with the same field names can be problematic. Let's consider a simple example: building types to represent the protocol used for a logging server.

We'll describe three message types: `log_entry`, `heartbeat`, and `logon`. The `log_entry` message is used to deliver a log entry to the server; the `logon` message is sent to initiate a connection and includes the identity of the user connecting and credentials used for authentication; and the `heartbeat` message is periodically sent by the client to demonstrate to the server that the client is alive and connected. All of these messages include a session ID and the time the message was generated:

OCaml utop (part 13)

```
# type log_entry =
    { session_id: string;
      time: Time.t;
      important: bool;
      message: string;
    }
  type heartbeat =
    { session_id: string;
      time: Time.t;
      status_message: string;
    }
  type logon =
    { session_id: string;
      time: Time.t;
      user: string;
      credentials: string;
    }
;;
type log_entry = {
  session_id : string;
  time : Time.t;
  important : bool;
  message : string;
}
type heartbeat = {
  session_id : string;
  time : Time.t;
  status_message : string;
}
type logon = {
  session_id : string;
  time : Time.t;
  user : string;
  credentials : string;
}
```

Reusing field names can lead to some ambiguity. For example, if we want to write a function to grab the `session_id` from a record, what type will it have?

OCaml utop (part 14)

```
# let get_session_id t = t.session_id;;
val get_session_id : logon -> string = <fun>
```

In this case, OCaml just picks the most recent definition of that record field. We can force OCaml to assume we're dealing with a different type (say, a `heartbeat`) using a type annotation:

OCaml utop (part 15)

```
# let get_heartbeat_session_id (t:heartbeat) = t.session_id;;
```

```
val get_heartbeat_session_id : heartbeat -> string = <fun>
```

While it's possible to resolve ambiguous field names using type annotations, the ambiguity can be a bit confusing. Consider the following functions for grabbing the session ID and status from a heartbeat:

OCaml utop (part 16)

```
# let status_and_session t = (t.status_message, t.session_id);;
val status_and_session : heartbeat -> string * string = <fun>
# let session_and_status t = (t.session_id, t.status_message);;
Characters 44-58:
Error: The record type logon has no field status_message
# let session_and_status (t:heartbeat) = (t.session_id, t.status_message);;
val session_and_status : heartbeat -> string * string = <fun>
```

Why did the first definition succeed without a type annotation and the second one fail? The difference is that in the first case, the type-checker considered the `status_message` field first and thus concluded that the record was a `heartbeat`. When the order was switched, the `session_id` field was considered first, and so that drove the type to be considered to be a `logon`, at which point `t.status_message` no longer made sense.

We can avoid this ambiguity altogether, either by using nonoverlapping field names or, more generally, by minting a module for each type. Packing types into modules is a broadly useful idiom (and one used quite extensively by Core), providing for each type a namespace within which to put related values. When using this style, it is standard practice to name the type associated with the module `t`. Using this style we would write:

OCaml utop (part 17)

```
# module Log_entry = struct
    type t =
      { session_id: string;
        time: Time.t;
        important: bool;
        message: string;
      }
  end
  module Heartbeat = struct
    type t =
      { session_id: string;
        time: Time.t;
        status_message: string;
      }
  end
  module Logon = struct
    type t =
      { session_id: string;
        time: Time.t;
        user: string;
        credentials: string;
      }
  end;;
module Log_entry :
  sig
    type t = {
      session_id : string;
      time : Time.t;
      important : bool;
      message : string;
    }
  end
```

```
module Heartbeat :
  sig
    type t = { session_id : string; time : Time.t; status_message : string; }
  end
module Logon :
  sig
    type t = {
      session_id : string;
      time : Time.t;
      user : string;
      credentials : string;
    }
  end
```

Now, our log-entry-creation function can be rendered as follows:

OCaml utop (part 18)

```
# let create_log_entry ~session_id ~important message =
    { Log_entry.time = Time.now (); Log_entry.session_id;
      Log_entry.important; Log_entry.message }
  ;;
val create_log_entry :
  session_id:string -> important:bool -> string -> Log_entry.t = <fun>
```

The module name `Log_entry` is required to qualify the fields, because this function is outside of the `Log_entry` module where the record was defined. OCaml only requires the module qualification for one record field, however, so we can write this more concisely. Note that we are allowed to insert whitespace between the module path and the field name:

OCaml utop (part 19)

```
# let create_log_entry ~session_id ~important message =
    { Log_entry.
      time = Time.now (); session_id; important; message }
  ;;
val create_log_entry :
  session_id:string -> important:bool -> string -> Log_entry.t = <fun>
```

This is not restricted to constructing a record; we can use the same trick when pattern matching:

OCaml utop (part 20)

```
# let message_to_string { Log_entry.important; message; _ } =
    if important then String.uppercase message else message
  ;;
val message_to_string : Log_entry.t -> string = <fun>
```

When using dot notation for accessing record fields, we can qualify the field by the module directly:

OCaml utop (part 21)

```
# let is_important t = t.Log_entry.important;;
val is_important : Log_entry.t -> bool = <fun>
```

The syntax here is a little surprising when you first encounter it. The thing to keep in mind is that the dot is being used in two ways: the first dot is a record field access, with everything to the right of the dot being interpreted as a field name; the second dot is accessing the contents of a module, referring to the record field `important` from within the module `Log_entry`. The fact that `Log_entry` is capitalized and so can't be a field name is what disambiguates the two uses.

For functions defined within the module where a given record is defined, the module qualification goes away entirely.

# Functional Updates

Fairly often, you will find yourself wanting to create a new record that differs from an existing record in only a subset of the fields. For example, imagine our logging server had a record type for representing the state of a given client, including when the last heartbeat was received from that client. The following defines a type for representing this information, as well as a function for updating the client information when a new heartbeat arrives:

OCaml utop (part 22)

```
# type client_info =
   { addr: Unix.Inet_addr.t;
     port: int;
     user: string;
     credentials: string;
     last_heartbeat_time: Time.t;
   };;
type client_info = {
  addr : UnixLabels.inet_addr;
  port : int;
  user : string;
  credentials : string;
  last_heartbeat_time : Time.t;
}
# let register_heartbeat t hb =
      { addr = t.addr;
        port = t.port;
        user = t.user;
        credentials = t.credentials;
        last_heartbeat_time = hb.Heartbeat.time;
      };;
val register_heartbeat : client_info -> Heartbeat.t -> client_info = <fun>
```

This is fairly verbose, given that there's only one field that we actually want to change, and all the others are just being copied over from `t`. We can use OCaml's *functional update* syntax to do this more tersely. The syntax of a functional update is as follows:

Syntax

```
{ <record> with <field> = <value>;
      <field> = <value>;
      ...
}
```

The purpose of the functional update is to create a new record based on an existing one, with a set of field changes layered on top.

Given this, we can rewrite `register_heartbeat` more concisely:

OCaml utop (part 23)

```
# let register_heartbeat t hb =
    { t with last_heartbeat_time = hb.Heartbeat.time };;
val register_heartbeat : client_info -> Heartbeat.t -> client_info = <fun>
```

Functional updates make your code independent of the identity of the fields in the record that are not changing. This is often what you want, but it has downsides as well. In particular, if you change the definition of your record to have more fields, the type system will not prompt you to reconsider whether your code needs to change to accommodate the new fields. Consider what happens if we decided to add a field for the status message received on the last heartbeat:

OCaml utop (part 24)

```
# type client_info =
   { addr: Unix.Inet_addr.t;
     port: int;
     user: string;
     credentials: string;
     last_heartbeat_time: Time.t;
     last_heartbeat_status: string;
   };;
type client_info = {
  addr : UnixLabels.inet_addr;
  port : int;
  user : string;
  credentials : string;
  last_heartbeat_time : Time.t;
  last_heartbeat_status : string;
}
```

The original implementation of `register_heartbeat` would now be invalid, and thus the compiler would effectively warn us to think about how to handle this new field. But the version using a functional update continues to compile as is, even though it incorrectly ignores the new field. The correct thing to do would be to update the code as follows:

OCaml utop (part 25)

```
# let register_heartbeat t hb =
    { t with last_heartbeat_time   = hb.Heartbeat.time;
             last_heartbeat_status = hb.Heartbeat.status_message;
    };;
val register_heartbeat : client_info -> Heartbeat.t -> client_info = <fun>
```

# Mutable Fields

Like most OCaml values, records are immutable by default. You can, however, declare individual record fields as mutable. In the following code, we've made the last two fields of `client_info` mutable:

OCaml utop (part 26)

```
# type client_info =
   { addr: Unix.Inet_addr.t;
     port: int;
     user: string;
     credentials: string;
     mutable last_heartbeat_time: Time.t;
     mutable last_heartbeat_status: string;
   };;
type client_info = {
  addr : UnixLabels.inet_addr;
  port : int;
  user : string;
  credentials : string;
  mutable last_heartbeat_time : Time.t;
  mutable last_heartbeat_status : string;
}
```

The `<-` operator is used for setting a mutable field. The side-effecting version of `register_heartbeat` would be written as follows:

OCaml utop (part 27)

```
# let register_heartbeat t hb =
    t.last_heartbeat_time   <- hb.Heartbeat.time;
    t.last_heartbeat_status <- hb.Heartbeat.status_message
  ;;
val register_heartbeat : client_info -> Heartbeat.t -> unit = <fun>
```

Note that mutable assignment, and thus the `<-` operator, is not needed for initialization because all fields of a record, including mutable ones, are specified when the record is created.

OCaml's policy of immutable-by-default is a good one, but imperative programming is an important part of programming in OCaml. We go into more depth about how (and when) to use OCaml's imperative features in Imperative Programming.

# First-Class Fields

Consider the following function for extracting the usernames from a list of `Logon` messages:

OCaml utop (part 28)

```
# let get_users logons =
    List.dedup (List.map logons ~f:(fun x -> x.Logon.user));;
val get_users : Logon.t list -> string list = <fun>
```

Here, we wrote a small function `(fun x -> x.Logon.user)` to access the `user` field. This kind of accessor function is a common enough pattern that it would be convenient to generate it automatically. The `fieldslib` syntax extension that ships with Core does just that.

The `with fields` annotation at the end of the declaration of a record type will cause the extension to be applied to a given type declaration. So, for example, we could have defined `Logon` as follows:

OCaml utop

```
# module Logon = struct
    type t =
      { session_id: string;
        time: Time.t;
        user: string;
        credentials: string;
      }
    with fields
  end;;
module Logon :
  sig
    type t = {
      session_id : string;
      time : Time.t;
      user : string;
      credentials : string;
    }
    val credentials : t -> string
    val user : t -> string
    val time : t -> Time.t
    val session_id : t -> string
    module Fields :
      sig
        val names : string list
        val credentials :
          ([< `Read | `Set_and_create ], t, string) Field.t_with_perm
        val user :
          ([< `Read | `Set_and_create ], t, string) Field.t_with_perm
        val time :
          ([< `Read | `Set_and_create ], t, Time.t) Field.t_with_perm
        val session_id :
          ([< `Read | `Set_and_create ], t, string) Field.t_with_perm

        [ ... many definitions omitted ... ]

      end
  end
```

Note that this will generate *a lot* of output because `fieldslib` generates a large collection of helper functions for working with record fields. We'll only discuss a few of these; you can learn about the remainder from the documentation that comes with `fieldslib`.

One of the functions we obtain is `Logon.user`, which we can use to extract the user field from a logon message:

OCaml utop (part 30)

```
# let get_users logons = List.dedup (List.map logons ~f:Logon.user);;
val get_users : Logon.t list -> string list = <fun>
```

In addition to generating field accessor functions, `fieldslib` also creates a submodule called `Fields` that contains a first-class representative of each field, in the form of a value of type `Field.t`. The `Field` module provides the following functions:

`Field.name`
: Returns the name of a field

`Field.get`
: Returns the content of a field

`Field.fset`
: Does a functional update of a field

`Field.setter`
: Returns `None` if the field is not mutable or `Some f` if it is, where `f` is a function for mutating that field

A `Field.t` has two type parameters: the first for the type of the record, and the second for the type of the field in question. Thus, the type of `Logon.Fields.session_id` is `(Logon.t, string) Field.t`, whereas the type of `Logon.Fields.time` is `(Logon.t, Time.t) Field.t`. Thus, if you call `Field.get` on `Logon.Fields.user`, you'll get a function for extracting the `user` field from a `Logon.t`:

OCaml utop (part 31)

```
# Field.get Logon.Fields.user;;
- : Logon.t -> string = <fun>
```

Thus, the first parameter of the `Field.t` corresponds to the record you pass to `get`, and the second parameter corresponds to the value contained in the field, which is also the return type of `get`.

The type of `Field.get` is a little more complicated than you might naively expect from the preceding one:

OCaml utop (part 32)

```
# Field.get;;
- : ('b, 'r, 'a) Field.t_with_perm -> 'r -> 'a = <fun>
```

The type is `Field.t_with_perm` rather than `Field.t` because fields have a notion of access control that comes up in some special cases where we expose the ability to read a field from a record, but not the ability to create new records, and so we can't expose functional updates.

We can use first-class fields to do things like write a generic function for displaying a record field:

OCaml utop (part 33)

```
# let show_field field to_string record =
    let name = Field.name field in
    let field_string = to_string (Field.get field record) in
    name ^ ": " ^ field_string
  ;;
val show_field :
  ('a, 'b, 'c) Field.t_with_perm -> ('c -> string) -> 'b -> string = <fun>
```

This takes three arguments: the `Field.t`, a function for converting the contents of the field in question to a string, and a record from which the field can be grabbed.

Here's an example of `show_field` in action:

OCaml utop (part 34)

```
# let logon = { Logon.
                session_id = "26685";
                time = Time.now ();
                user = "yminsky";
                credentials = "Xy2d9W"; }
  ;;
val logon : Logon.t =
  {Logon.session_id = "26685"; time = 2013-08-18 14:51:00.509463+01:00;
   user = "yminsky"; credentials = "Xy2d9W"}
# show_field Logon.Fields.user Fn.id logon;;
- : string = "user: yminsky"
# show_field Logon.Fields.time Time.to_string logon;;
- : string = "time: 2013-08-18 14:51:00.509463+01:00"
```

As a side note, the preceding example is our first use of the `Fn` module (short for "function"), which provides a collection of useful primitives for dealing with functions. `Fn.id` is the identity function.

`fieldslib` also provides higher-level operators, like `Fields.fold` and `Fields.iter`, which let you walk over the fields of a record. So, for example, in the case of `Logon.t`, the field iterator has the following type:

OCaml utop (part 35)

```
# Logon.Fields.iter;;
- : session_id:(([< `Read | `Set_and_create ], Logon.t, string)
                 Field.t_with_perm -> 'a) ->
    time:(([< `Read | `Set_and_create ], Logon.t, Time.t) Field.t_with_perm ->
          'b) ->
    user:(([< `Read | `Set_and_create ], Logon.t, string) Field.t_with_perm ->
          'c) ->
    credentials:(([< `Read | `Set_and_create ], Logon.t, string)
                  Field.t_with_perm -> 'd) ->
    'd
= <fun>
```

This is a bit daunting to look at, largely because of the access control markers, but the structure is actually pretty simple. Each labeled argument is a function that takes a first-class field of the necessary type as an argument. Note that `iter` passes each of these callbacks the `Field.t`, not the contents of the specific record field. The contents of the field, though, can be looked up using the combination of the record and the `Field.t`.

Now, let's use `Logon.Fields.iter` and `show_field` to print out all the fields of a `Logon` record:

OCaml utop (part 36)

```
# let print_logon logon =
    let print to_string field =
      printf "%s\n" (show_field field to_string logon)
    in
    Logon.Fields.iter
      ~session_id:(print Fn.id)
      ~time:(print Time.to_string)
      ~user:(print Fn.id)
      ~credentials:(print Fn.id)
  ;;
val print_logon : Logon.t -> unit = <fun>
```

```
# print_logon logon;;
session_id: 26685
time: 2013-08-18 14:51:00.509463+01:00
user: yminsky
credentials: Xy2d9W
- : unit = ()
```

One nice side effect of this approach is that it helps you adapt your code when the fields of a record change. If you were to add a field to `Logon.t`, the type of `Logon.Fields.iter` would change along with it, acquiring a new argument. Any code using `Logon.Fields.iter` won't compile until it's fixed to take this new argument into account.

Field iterators are useful for a variety of record-related tasks, from building record-validation functions to scaffolding the definition of a web form from a record type. Such applications can benefit from the guarantee that all fields of the record type in question have been considered.

# Chapter 6. Variants

Variant types are one of the most useful features of OCaml and also one of the most unusual. They let you represent data that may take on multiple different forms, where each form is marked by an explicit tag. As we'll see, when combined with pattern matching, variants give you a powerful way of representing complex data and of organizing the case-analysis on that information.

The basic syntax of a variant type declaration is as follows:

### Syntax

```
type <variant> =
  | <Tag> [ of <type> [* <type>]... ]
  | <Tag> [ of <type> [* <type>]... ]
  | ...
```

Each row essentially represents a case of the variant. Each case has an associated tag and may optionally have a sequence of fields, where each field has a specified type.

Let's consider a concrete example of how variants can be useful. Almost all terminals support a set of eight basic colors, and we can represent those colors using a variant. Each color is declared as a simple tag, with pipes used to separate the different cases. Note that variant tags must be capitalized:

### OCaml utop

```
# type basic_color =
   | Black | Red | Green | Yellow | Blue | Magenta | Cyan | White ;;
type basic_color =
    Black
  | Red
  | Green
  | Yellow
  | Blue
  | Magenta
  | Cyan
  | White
# Cyan ;;
- : basic_color = Cyan
# [Blue; Magenta; Red] ;;
- : basic_color list = [Blue; Magenta; Red]
```

The following function uses pattern matching to convert a `basic_color` to a corresponding integer. The exhaustiveness checking on pattern matches means that the compiler will warn us if we miss a color:

### OCaml utop (part 1)

```
# let basic_color_to_int = function
  | Black -> 0 | Red     -> 1 | Green -> 2 | Yellow -> 3
  | Blue  -> 4 | Magenta -> 5 | Cyan  -> 6 | White  -> 7 ;;
val basic_color_to_int : basic_color -> int = <fun>
# List.map ~f:basic_color_to_int [Blue;Red];;
- : int list = [4; 1]
```

Using the preceding function, we can generate escape codes to change the color of a given string displayed in a terminal:

### OCaml utop

```
# let color_by_number number text =
    sprintf "\027[38;5;%dm%s\027[0m" number text;;
```

```
val color_by_number : int -> string -> string = <fun>
# let blue = color_by_number (basic_color_to_int Blue) "Blue";;
val blue : string = "\027[38;5;4mBlue\027[0m"
# printf "Hello %s World!\n" blue;;
Hello Blue World!
```

On most terminals, that word “Blue” will be rendered in blue.

In this example, the cases of the variant are simple tags with no associated data. This is substantively the same as the enumerations found in languages like C and Java. But as we’ll see, variants can do considerably more than represent a simple enumeration. As it happens, an enumeration isn’t enough to effectively describe the full set of colors that a modern terminal can display. Many terminals, including the venerable `xterm`, support 256 different colors, broken up into the following groups:

- The eight basic colors, in regular and bold versions
- A 6 × 6 × 6 RGB color cube
- A 24-level grayscale ramp

We’ll also represent this more complicated color space as a variant, but this time, the different tags will have arguments that describe the data available in each case. Note that variants can have multiple arguments, which are separated by `*`s:

OCaml utop (part 3)

```
# type weight = Regular | Bold
  type color =
  | Basic of basic_color * weight (* basic colors, regular and bold *)
  | RGB   of int * int * int       (* 6x6x6 color cube *)
  | Gray  of int                   (* 24 grayscale levels *)
;;
type weight = Regular | Bold
type color =
    Basic of basic_color * weight
  | RGB of int * int * int
  | Gray of int
# [RGB (250,70,70); Basic (Green, Regular)];;
- : color list = [RGB (250, 70, 70); Basic (Green, Regular)]
```

Once again, we’ll use pattern matching to convert a color to a corresponding integer. But in this case, the pattern matching does more than separate out the different cases; it also allows us to extract the data associated with each tag:

OCaml utop (part 4)

```
# let color_to_int = function
    | Basic (basic_color,weight) ->
      let base = match weight with Bold -> 8 | Regular -> 0 in
      base + basic_color_to_int basic_color
    | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | Gray i -> 232 + i ;;
val color_to_int : color -> int = <fun>
```

Now, we can print text using the full set of available colors:

OCaml utop

```
# let color_print color s =
    printf "%s\n" (color_by_number (color_to_int color) s);;
val color_print : color -> string -> unit = <fun>
# color_print (Basic (Red,Bold)) "A bold red!";;
A bold red!
# color_print (Gray 4) "A muted gray...";;
A muted gray...
```