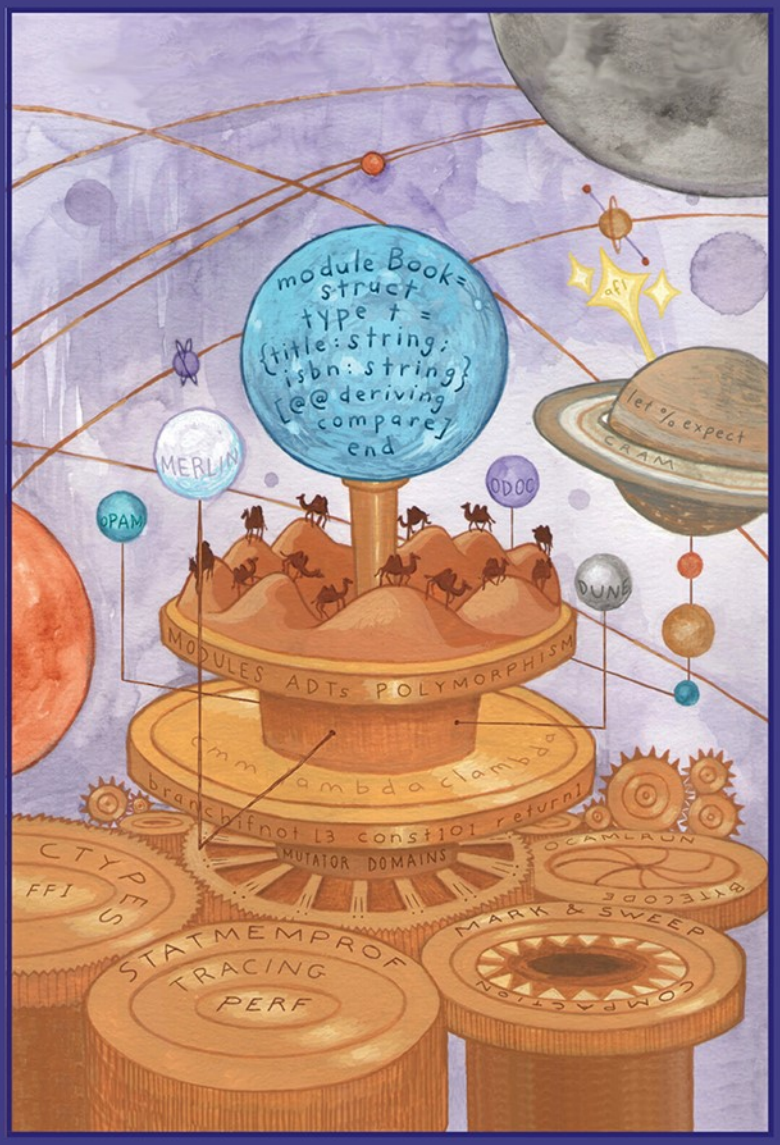
SECOND EDITION

REAL WORLD OCAML

Functional Programming for the Masses



ANIL MADHAVAPEDDY
AND YARON MINSKY

Real World OCaml: Functional Programming for the Masses

This fast-moving tutorial introduces you to OCaml, an industrial-strength programming language designed for expressiveness, safety, and speed. Through the book's many examples, you'll quickly learn how OCaml stands out as a tool for writing fast, succinct, and readable systems code using functional programming.

Real World OCaml takes you through the concepts of the language at a brisk pace, and then helps you explore the tools and techniques that make OCaml an effective and practical tool. You'll also delve deep into the details of the compiler toolchain and OCaml's simple and efficient runtime system.

This second edition brings the book up to date with almost a decade of improvements in the OCaml language and ecosystem, with new chapters covering testing, GADTs, and platform tooling. All of the example code is available online at realworldocaml.org.

This title is also available as open access on Cambridge Core, thanks to the support of Tarides. Their generous contribution will bring more people to OCaml.

Anil Madhavapeddy is an associate professor in the Department of Computer Science and Technology at the University of Cambridge. He has used OCaml professionally for over two decades in numerous ventures, such as XenSource/Citrix and Unikernel Systems/Docker, and co-founded the MirageOS unikernel project. He is a member of the OCaml development team.

Yaron Minsky is Co-head of Technology at Jane Street, a major quantitative trading firm, where he introduced OCaml, and helped it become the firm's primary development platform. He is also the host of Jane Street's tech podcast, *Signals & Threads*, and has worked on everything from developer tools to trading strategies.

Real World OCaml: Functional Programming for the Masses

ANIL MADHAVAPEDDY
University of Cambridge

YARON MINSKY
Jane Street Group

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre,
New Delhi – 110025, India

103 Penang Road, #05–06/07, Visioncrest Commercial, Singapore 238467

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of
education, learning, and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781009125802
DOI: 10.1017/9781009129220

First edition published in 2014 by O'Reilly Media, Inc.
Second edition 2022

Printed in the United Kingdom by TJ Books Limited, Padstow Cornwall

A catalogue record for this publication is available from the British Library.

ISBN 978-1-009-12580-2 Paperback

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

Contents

1 Prologue

1.1 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. Both of us have been using OCaml in our academic and professional lives for over 20 years, and in that time we've come to see it as a powerful tool for building complex software systems. This book aims to make this tool 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 set of language features that have been developed over the last 60 years. These include:

- *Garbage collection* for automatic memory management, now a common feature of modern, high-level languages.
- *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, Rust, 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 commonly found in everything from distributed, big-data frameworks to user-interface toolkits.
- *Type inference*, so you don't need to annotate every variable in your program with its type. Instead, types are 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, Rust, and F#.

Some of you will know and love all of these features, and for others they'll 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 all these features 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, Rust, 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.

1.1.1 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 (as part of the Formel project at INRIA headed by Gérard Huet) 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 two decades has seen OCaml attract a significant user base, and language improvements have been steadily added to support the growing commercial and academic codebases. By 2012, the OCaml 4.0 release had added Generalized Algebraic Data Types (GADTs) and first-class modules to increase the flexibility of the language. Since then, OCaml has had a steady yearly release cadence, and OCaml 5.0 with multicore support is around the corner in 2022. There is also fast native code support for the latest CPU architectures such as x86_64, ARM, RISC-V and PowerPC, making OCaml a good choice for systems where resource usage, predictability, and performance all matter.

1.1.2 The `Base` Standard Library

However good it is, a language on its own isn't enough. You also need a set of libraries to build 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 subset of the functionality you would expect from a general-purpose standard library. That's because the standard library isn't really a general-purpose tool; its fundamental role is in bootstrapping the compiler, and has been purposefully kept small and portable.

Happily, in the world of open source software, nothing stops alternative libraries from being written to supplement the compiler-supplied standard library. `Base` is an example of such a library, and it's the standard library we'll use through most of this book.

Jane Street, a company that has been using OCaml for more than 20 years, developed the code in `Base` for its own internal use, but from the start designed it with an eye toward being a general-purpose standard library. Like the OCaml language itself, `Base` is engineered with correctness, reliability, and performance in mind. It's also designed to be easy to install and highly portable. As such, it works on every platform OCaml does, including UNIX, macOS, Windows, and JavaScript.

`Base` is distributed with a set of syntax extensions that provide useful new functionality to OCaml, and there are additional libraries that are designed to work well with it, including `Core`, an extension to `Base` that includes a wealth of new data structures and tools; and `Async`, a library for concurrent programming of the kind that often comes up when building user interfaces or networked applications. All of these libraries are

distributed under a liberal Apache 2 license to permit free use in hobby, academic, and commercial settings.

1.1.3 The OCaml Platform

`Base` 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[1]. We'll explain more about opam as the book unfolds, but it forms the basis of the OCaml Platform, which is a set of tools and libraries that, along with the OCaml compiler, lets you build real-world applications quickly and effectively. Constituent tools of the OCaml Platform include the dune[2] build system and a language server to integrate with popular editors such as Visual Studio Code (or Emacs or Vim).

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.

1.2 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. Base 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 Base, but you may need to adapt it for maximal benefit. All the new code that we write uses Base, and we believe the Base 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.

[1] https://opam.ocaml.org/
[2] https://dune.build

1.2.1 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`.

1.2.2 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 4.13.1 version of the compiler). The installation process is largely automated through the opam package manager. Instructions on how to set it up and what packages to install can be found at the installation page[3].

Some of the libraries we use, notably `Base`, work anywhere OCaml does, and in particular work on Windows and JavaScript. The examples in Part I of the book will for the most part stick to such highly portable libraries. Some of the libraries used, however, require a UNIX based operating system, and so only work on systems like macOS, Linux, FreeBSD, OpenBSD, and the Windows Subsystem for Linux (WSL). `Core` and `Async` are notable examples here.

This book is not intended as a reference manual. We aim to teach you about the language as well as the 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[4].

[3] `http://dev.realworldocaml.org/install.html`
[4] `https://v3.ocaml.org/packages`

1.2.3 Code Examples

All of the code examples in this book are available freely online under a public-domain-like license. You are 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 full text of the book, along with all of the example code is available online at https://github.com/realworldocaml/book[5].

1.3 Contributors

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

- Jason Hickey was our co-author on the first edition of this book, and is instrumental to the structure and content that formed the basis of this revised edition.
- Leo White and Jason Hickey contributed greatly to the content and examples in Chapter 13 (Objects) and Chapter 14 (Classes).
- Jeremy Yallop authored and documented the Ctypes library described in Chapter 23 (Foreign Function Interface).
- Stephen Weeks is responsible for much of the modular architecture behind Base and Core, and his extensive notes formed the basis of Chapter 24 (Memory Representation of Values) and Chapter 25 (Understanding the Garbage Collector). Sadiq Jaffer subsequently refreshed the garbage collector chapter to reflect the latest changes in OCaml 4.13.
- Jérémie 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.
- Thomas Gazagnaire, Thibaut Mattio, David Allsopp and Jonathan Ludlam contributed to the OCaml Platform chapter, including fixes to core tools to better aid new user installation.
- Ashish Agarwal, Christoph Troestler, Thomas Gazagnaire, Etienne Millon, Nathan Rebours, Charles-Edouard Lecat, Jules Aguillon, Rudi Grinberg, Sonja Heinze and Frederic Bour worked on improving the book's toolchain. This allowed us to update the book to track changes to OCaml and various libraries and tools. Ashish also developed a new and improved version of the book's website.
- David Tranah, Clare Dennison, Anna Scriven and Suresh Kumar from Cambridge University Press for input into the layout of the print edition. Airlie Anderson drew the cover art, and Christy Nyberg advised on the design and layout.
- The many people who collectively submitted over 4000 comments to online drafts of this book, through whose efforts countless errors were found and fixed.

[5] `https://github.com/realworldocaml/book`

Part I

Language Concepts

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.

2 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 `Base`, 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`, but the ordinary toplevel should mostly work fine.

Before going any further, make sure you've followed the steps in the installation page[1].

Base and Core

`Base` comes along with another, yet more extensive standard library replacement, called `Core`. We're going to mostly stick to `Base`, but it's worth understanding the differences between these libraries.

- *`Base`* is designed to be lightweight, portable, and stable, while providing all of the fundamentals you need from a standard library. It comes with a minimum of external dependencies, so `Base` just takes seconds to build and install.
- *`Core`* extends `Base` in a number of ways: it adds new data structures, like heaps, hash-sets, and functional queues; it provides types to represent times and time-zones; well-integrated support for efficient binary serializers; and much more. At the same time, it has many more dependencies, and so takes longer to build, and will add more to the size of your executables.

As of the version of `Base` and `Core` used in this book (version `v0.14`), `Core` is less portable than `Base`, running only on UNIX-like systems. For that reason, there is another package, `Core_kernel`, which is the portable subset of `Core`. That said, in the latest stable release, `v0.15` (which was released too late to be adopted for this edition of the book) `Core` is portable, and `Core_kernel` has been deprecated. Given that, we don't use `Core_kernel` in this text.

[1] `http://dev.realworldocaml.org/install.html`

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

2.1 OCaml as a Calculator

Our first step is to open `Base`:

```
# open Base;;
```

By opening `Base`, we make the definitions it contains available without having to reference `Base` explicitly. This 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:

```
# 3 + 4;;
- : int = 7
# 8 / 3;;
- : int = 2
# 3.5 +. 6.;;
- : float = 9.5
# 30_000_000 / 300_000;;
- : int = 100
# 3 * 5 > 14;;
- : bool = true
```

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.
- 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 `1.0 /. 3.0` is a third. OCaml requires you to be explicit about which operation you're using.

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*:

```
# 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:

```
# 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
```

The following examples, however, are not legal:

```
# let Seven = 3 + 4;;
Line 1, characters 5-10:
Error: Unbound constructor Seven
# let 7x = 7;;
Line 1, characters 5-7:
Error: Unknown modifier 'x' for literal 7x
# let x-plus-y = x + y;;
Line 1, characters 7-11:
Error: Syntax error
```

This highlights that variables can't be capitalized, can't begin with numbers, and can't contain dashes.

2.2 Functions and Type Inference

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

```
# 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. (Reminder: Don't forget `open Base`, or these examples won't work!)

```
# let ratio x y =
    Float.of_int x /. Float.of_int y;;
val ratio : int -> int -> float = <fun>
# ratio 4 7;;
- : float = 0.571428571428571397
```

Note that in OCaml, 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 Python or Java.

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.

Modules can also be opened to make their contents available without explicitly qualifying by the module name. We did that once already, when we opened `Base` earlier. We can use that to make this code a little easier to read, both avoiding the repetition of `Float` above, and avoiding use of the slightly awkward `/.` operator. In the following example, we open the `Float.O` module, which has a bunch of useful operators and functions that are designed to be used in this kind of context. Note that this causes the standard int-only arithmetic operators to be shadowed locally.

```
# let ratio x y =
    let open Float.O in
    of_int x / of_int y;;
val ratio : int -> int -> float = <fun>
```

We used a slightly different syntax for opening the module, since we were only opening it in the local scope inside the definition of `ratio`. There's also a more concise syntax for local opens, as you can see here.

```
# let ratio x y =
    Float.O.(of_int x / of_int y);;
val ratio : int -> int -> float = <fun>
```

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 Chapter 3.2.2 (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:

```
# 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:

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

2.2.1 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` expression 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` expression, 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 also 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`:

```
# 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.

2.2.2 Inferring Generic Types

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

```
# 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 the `x` argument to `first_if_true`? There are no obvious clues such as arithmetic operators or literals to narrow it down. That makes it seem like `first_if_true` would work 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.

Because the type of `first_if_true` is generic, we can write this:

```
# 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:

```
# 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`:

```
# first_if_true big_number "short" "looooooong";;
Line 1, characters 26-33:
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 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:

```
# let add_potato x =
    x + "potato";;
Line 2, characters 9-17:
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:

```
# let is_a_multiple x y =
    x % y = 0;;
val is_a_multiple : int -> int -> bool = <fun>
# is_a_multiple 8 2;;
- : bool = true
# is_a_multiple 8 0;;
Exception:
(Invalid_argument "8 % 0 in core_int.ml: modulus should be positive")
```

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.

2.3 Tuples, Lists, Options, and Pattern Matching

2.3.1 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.

```
# 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, `*` is used in the type `t * s` because that type corresponds to the set of all pairs containing one value of type `t` and one of type `s`. In other words, it's the *Cartesian product* of the two types, which is why we use `*`, the symbol for product.

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

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

Here, the `(x,y)` on the left-hand 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:

```
# 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 `floats`. The pattern-matching syntax lets us get at the values we need with a minimum of fuss:

```
# let distance (x1,y1) (x2,y2) =
    Float.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.

Operators in `Base` and the Stdlib

OCaml's standard library and `Base` mostly use the same operators for the same things, but there are some differences. For example, in `Base`, `**.` is float exponentiation, and `**` is integer exponentiation, whereas in the standard library, `**` is float exponentiation, and integer exponentiation isn't exposed as an operator.

`Base` does what it does to be consistent with other numerical operators like `*.` and `*`, where the period at the end is used to mark the floating-point versions.

In general, `Base` is not shy about presenting different APIs than OCaml's standard library when it's done in the service of consistency and clarity.

2.3.2 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:

```
# 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:

```
# let numbers = [3;"four";5];;
Line 1, characters 18-24:
Error: This expression has type string but an expression was expected
    of type
         int
```

The List Module

`Base` 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:

```
# 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:

```
# 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:

```
# 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 3 (Variables and Functions).

Constructing Lists with ::

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

```
# "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:

```
# 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", "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:

```
# 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:

```
# [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 `::` constructor 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:

```
# [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:

```
# let my_favorite_language (my_favorite :: the_rest) =
    my_favorite;;
Lines 1-2, characters 26-16:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case 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 lists and fails on empty ones:

```
# my_favorite_language ["English";"Spanish";"French"];;
- : string = "English"
# my_favorite_language [];;
Exception: "Match_failure //toplevel//:1:26"
```

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

A `match` expression 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 parts of the value being matched.

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

```
# 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 part of working in OCaml or really 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:

```
# 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:

```
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 if not entirely accurate 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:

```
# let rec remove_sequential_duplicates list =
    match list with
    | [] -> []
    | first :: second :: tl ->
      if first = second then
        remove_sequential_duplicates (second :: tl)
      else
        first :: remove_sequential_duplicates (second :: tl);;
Lines 2-8, characters 5-61:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
_::[]
val remove_sequential_duplicates : int list -> int list = <fun>
```

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

```
# let rec remove_sequential_duplicates list =
    match list with
    | [] -> []
    | [x] -> [x]
    | first :: second :: tl ->
      if first = second then
        remove_sequential_duplicates (second :: tl)
      else
        first :: remove_sequential_duplicates (second :: tl);;
val remove_sequential_duplicates : int list -> int list = <fun>
# remove_sequential_duplicates [1;1;2;3;3;4;4;1;1;1];;
- : int list = [1; 2; 3; 4; 1]
```

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 for when you need to iterate in a new way.

2.3.3 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:

```
# 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. Let's see how this plays out in a small example. We'll write a function that takes a filename, and returns a version of that filename with the file extension (the part after the dot) downcased. We'll base this on the function `String.rsplit2` to split the string based on the rightmost period found in the string. Note that `String.rsplit2` has return type `(string * string) option`, returning `None` when no character was found to split on.

```
# let downcase_extension filename =
    match String.rsplit2 filename ~on:'.' with
    | None -> filename
    | Some (base,ext) ->
      base ^ "." ^ String.lowercase ext;;
val downcase_extension : string -> string = <fun>
# List.map ~f:downcase_extension
    [ "Hello_World.TXT"; "Hello_World.txt"; "Hello_World" ];;
```

```
- : string list = ["Hello_World.txt"; "Hello_World.txt";
    "Hello_World"]
```

Note that we used the `^` operator for concatenating strings. The concatenation operator is provided as part of the `Stdlib` module, which is automatically opened in every OCaml program.

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 8 (Error Handling), this explicitness allows the compiler to provide a great deal of help in making sure you're correctly handling the possibility of missing data.

2.4 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:

```
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:

```
# 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:

```
# let magnitude { x = x_pos; y = y_pos } =
    Float.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:

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

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

```
# 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`:

```
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 multi-object scene. You need some unified way of representing these objects together in a single type. *Variant* types let you do just that:

```
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_elements`. 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

```
# let is_inside_scene_element point scene_element =
    let open Float.O in
    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 _ -> false

  let is_inside_scene point scene =
    List.exists scene
      ~f:(fun el -> is_inside_scene_element point el);;
val is_inside_scene_element : point2d -> scene_element -> bool = <fun>
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 just

examples of variant types that are 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 for 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.

> **`Base` and Polymorphic Comparison**
> One other thing to notice was the fact that we opened `Float.O` in the definition of `is_inside_scene_element`. That allowed us to use the simple, un-dotted infix operators, but more importantly it brought the float comparison operators into scope. When using `Base`, the default comparison operators work only on integers, and you need to explicitly choose other comparison operators when you want them. OCaml also offers a special set of *polymorphic comparison operators* that can work on almost any type, but those are considered to be problematic, and so are hidden by default by `Base`. We'll learn more about polymorphic compare in Chapter 4.6 (Terser and Faster Patterns).

2.5 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.

2.5.1 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:

```
# 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 `numbers.(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.

2.5.2 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 an example of a mutable data structure for storing a running statistical summary of a collection of numbers.

```
type running_sum =
  { mutable sum: float;
    mutable sum_sq: float; (* sum of squares *)
    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.

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

We also need functions to create and update `running_sums`:

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

```
# 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 = 1.33333333333333326
# stdev rsum;;
- : float = 3.94405318873307698
```

Warning: the preceding algorithm is numerically naive and has poor precision in the presence of many values that cancel each other out. This Wikipedia article on algorithms for calculating variance[2] provides more details.

2.5.3 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`:

```
# 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:

```
# let x = ref 0  (* create a ref, i.e., { contents = 0 } *);;
val x : int ref = {Base.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:

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

[2] http://en.wikipedia.org/wiki/Algorithms_for_calculating_variance

```
val ref : 'a -> 'a ref = <fun>
# let (!) r = r.contents;;
val ( ! ) : 'a ref -> 'a = <fun>
# let (:=) r x = r.contents <- x;;
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:

```
# 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.

Nesting lets with `let` and `in`

The definition of `sum` in the above examples 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:

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

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

```
# z;;
Line 1, characters 1-2:
Error: Unbound value z
```

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

```
# 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.

2.5.4 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:

```
# 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>
```

This is our first use of the `Random` module. Note that `Random` starts with a fixed seed, but you can call `Random.self_init` to choose a new seed at random.

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:

```
# 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 =
[|12; 16; 5; 13; 1; 6; 0; 7; 15; 19; 14; 4; 2; 11; 3; 8; 17; 9; 10;
    18|]
```

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:

```
# 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:

```
# 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 `&&`.

2.6 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.

```
open Base
open Stdio

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, and we needed to open another library, `Stdio`, to get access to them. 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. 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`.

2.6.1 Compiling and Running

We'll compile our program using `dune`, a build system that's designed for use with OCaml projects. First, we need to write a `dune-project` file to specify the project's root directory.

```
(lang dune 2.9)
(name rwo-example)
```

Then, we need to write a `dune` file to specify the specific thing being built. Note that a single project will have just one `dune-project` file, but potentially many sub-directories with different `dune` files.

In this case, however, we just have one:

```
(executable
 (name      sum)
 (libraries base stdio))
```

All we need to specify is the fact that we're building an executable (rather than a library), the name of the executable, and the name of the libraries we depend on.

We can now invoke dune to build the executable.

```
$ dune build sum.exe
```

The `.exe` suffix indicates that we're building a native-code executable, which we'll discuss more in Chapter 5 (Files, Modules, and Programs). Once the build completes, we can use the resulting program like any command-line utility. We can feed input to `sum.exe` by typing in a sequence of numbers, one per line, hitting **`Ctrl-D`** when we're done:

```
$ ./_build/default/sum.exe
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 16 (Command-Line Parsing).

2.7 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.

3 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 of the rest of the language.

3.1 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.

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

As we'll see when we get to the module system in Chapter 5 (Files, Modules, and Programs), 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.

```
# open Base;;
# 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.

```
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.

```
# 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. [let bindings/local]

```
# language_list;;
Line 1, characters 1-14:
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.

```
# 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 is still available.

```
# 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.

```
# let area_of_ring inner_radius outer_radius =
    let pi = Float.pi 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 = 25.132741228718345
```

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:

```
# let area_of_ring inner_radius outer_radius =
    let pi = Float.pi in
    let area_of_circle r = pi *. r *. r in
```

```
    let pi = 0. in
    area_of_circle outer_radius -. area_of_circle inner_radius;;
Line 4, characters 9-11:
Warning 26 [unused-var]: 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 still see the old one. But there is no later use of `pi`, so the binding of `pi` to `0.` made no difference at all. This explains the warning produced by the toplevel telling us that there is an unused variable.

In OCaml, `let` bindings are immutable. There are many kinds of mutable values in OCaml, which we'll discuss in Chapter 9 (Imperative Programming), 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.

3.1.1 Pattern Matching and Let

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

```
# 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 names to be bound to values. As we saw in Chapter 2.3 (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.

```
# let upcase_first_entry line =
    let (first :: rest) = String.split ~on:',' line in
    String.concat ~sep:"," (String.uppercase first :: rest);;
Lines 2-3, characters 5-60:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case 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, even when given the empty string.

```
# upcase_first_entry "one,two,three";;
- : string = "ONE,two,three"
# upcase_first_entry "";;
- : string = ""
```

But the compiler doesn't know this, and so it emits the warning. It's generally better to use a `match` expression to handle such cases explicitly:

```
# 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 8 (Error Handling).

3.2 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 looked at 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.

3.2.1 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.

```
# (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:

```
# (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.

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

You can even stuff a function into a data structure, like a list:

```
# let transforms = [ String.uppercase; String.lowercase ];;
val transforms : (string -> string) list = [<fun>; <fun>]
# List.map ~f:(fun g -> g "Hello World") transforms;;
- : string list = ["HELLO WORLD"; "hello world"]
```

It's worth stopping for a moment to puzzle this example out. Notice that `(fun g -> g "Hello World")` is a function that takes a function as an argument, and then applies that function to the string `"Hello World"`. The invocation of `List.map` applies `(fun g -> g "Hello World")` to the elements of `transforms`, which are themselves functions. The returned list contains 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.

```
# 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.

```
# 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.

```
# (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 17 (Concurrent Programming with Async).

3.2.2 Multiargument Functions

OCaml of course also supports multiargument functions, such as:

```
# 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.

```
# 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.

```
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`.

```
# 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.

```
# 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.

```
# 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.

3.2.3 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 9 (Imperative Programming), 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.

```
# let rec find_first_repeat 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_repeat (y::tl);;
val find_first_repeat : int list -> int option = <fun>
```

The pattern `[] | [_]` is itself a disjunction of multiple patterns, otherwise known as an *or-pattern*. An or-pattern matches if any of the sub-patterns match. 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.

```
# 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 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 definitions.

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

3.2.4 Prefix and Infix Operators

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

```
# 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.

```
# (+) 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:

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

as long as the first character is not `~`, `!`, or `$`.

There are also a handful of predetermined strings that count as infix operators, 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.

```
# 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)
```

You have to be careful when dealing with operators containing `*`. Consider the following example.

```
# let (***) x y = (x **. y) **. y;;
Line 1, characters 18-19:
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 `*`.

```
# 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. The OCaml manual has an explicit table of each class of operator[1] and its associated precedence.

We won't go through the full list here, but there's one important special case worth mentioning: `-` 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.

```
# Int.max 3 (-4);;
- : int = 3
# Int.max 3 -4;;
Line 1, characters 1-10:
Warning 5 [ignored-partial-application]: this function application is
    partial,
maybe some arguments are missing.
Line 1, characters 1-10:
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.

```
# (Int.max 3) - 4;;
Line 1, characters 1-12:
Warning 5 [ignored-partial-application]: this function application is
    partial,
maybe some arguments are missing.
```

[1] https://ocaml.org/manual/expr.html#ss:precedence-and-associativity

```
Line 1, characters 1-12:
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.

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

This is called the *reverse application operator*, and it's not quite obvious at first what its purpose 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 sequencing operations, 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`.

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

We can do this without `|>` by naming the intermediate values, but the result is a bit more verbose.

```
# let split_path = String.split ~on:':' path in
  let deduped_path = List.dedup_and_sort ~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` 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:

```
# 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.

```
# 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 (^>).

```
# let (^>) x f = f x;;
val ( ^> ) : 'a -> ('a -> 'b) -> 'b = <fun>
# String.split ~on:':' path
  ^> List.dedup_and_sort ~compare:String.compare
  ^> List.iter ~f:print_endline;;
Line 3, characters 6-32:
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_and_sort ~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.

The Application Operator

|> is known as the *reverse application operator*. You might be unsurprised to learn that there's also an *application operator*:

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

This one is useful for cases where you want to avoid many layers of parentheses when applying functions to complex expressions. In particular, you can replace f (g (h x)) with f @@ g @@ h x. Note that, just as we needed |> to be left associative, we need @@ to be right associative.

3.2.5 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.

```
# 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.

```
# 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.

```
# 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`.

3.2.6 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 `Base` 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.

```
# let ratio ~num ~denom = Float.of_int num /. Float.of_int 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.

```
# 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 colon 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.

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

Where Are Labels Useful?

Labeled arguments are a surprisingly useful feature, and it's worth walking through some of the cases where they come up.

Explicating Long Argument Lists

Beyond a certain number, arguments are easier to remember by name than by position. Letting the names be used at the call-site (and used in any order) makes client code easier to read and to write.

Adding Information to Uninformative Argument Types

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.

```
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.

```
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.

Disambiguating Similar Arguments

This issue comes up most often when a function has multiple arguments of the same type. Consider this signature for a function that extracts a substring.

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

Here, the two `int`s are the starting position and length of the substring to extract, respectively, but you wouldn't know that from the type signature. We can make the signature more informative by adding labels.

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

This improves the readability of both the signature and of client code, and makes it harder to accidentally swap the position and the length.

Flexible Argument Ordering and Partial Application

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.

```
# String.split ~on:':' path
  |> List.dedup_and_sort ~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.

Other orderings can be useful either for partial application, or for simple reasons of readability. For example, when using `List.iter` with a complex, multi-line iteration function, it's generally easier to read if the function comes second, after the statement of what list is being iterated over. On the other hand, when calling `List.iter` with a small function, but a large, explicitly written list of values, it's generally easier if the values come last.

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.

```
# 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.

```
# 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:

```
# 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`.

```
# apply_to_tuple_2 divide (3,4);;
Line 1, characters 18-24:
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`.

```
# 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.

3.2.7 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.

```
# let concat ?sep x y =
    let sep = match sep with None -> "" | Some s -> s 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.

```
# 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 `mlis` in Chapter 5 (Files, Modules, and Programs).

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`:

```
# 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`.

```
# 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.

```
# 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.

```
# 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.

```
# 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`.

```
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.

```
# 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);;
Line 5, characters 15-16:
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`.

```
# 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.

```
# 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?

```
# 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.

```
# prepend_pound "a BASH comment" ~sep:":";;
Line 1, characters 1-14:
Error: This function has type Base.String.t -> Base.String.t
       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:

```
# 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.

```
# 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.

```
# 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.

```
# let concat x y ?(sep="") = x ^ sep ^ y;;
Line 1, characters 18-24:
```

```
Warning 16 [unerasable-optional-argument]: 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.

```
# 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.

4 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 2 (A Guided Tour), but we'll go into more depth here, presenting the two topics together and using one to help illustrate the other.

4.1 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:

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

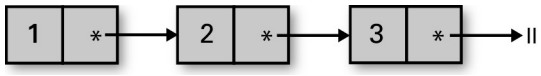
And they can also be generated using the equivalent `::` notation:

```
# 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:

```
# 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.



Each :: essentially adds a new block to the preceding 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 does not change any of the existing ones, as you can see:

```
# 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]
```

4.2 Using Patterns to Extract Data from a List

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

```
# 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 expression 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 expression. 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 expressions 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:

```
# 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;;
```

```
Line 5, characters 7-15:
Warning 11 [redundant-case]: 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:

```
# 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` that was 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` expression:

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

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`:

```
# 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]
```

4.3 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 good thing, making it possible to build better support for patterns in the compiler. In particular, both the efficiency of `match` expressions and the ability of the compiler to detect errors in matches depend on the constrained nature of patterns.

4.3.1 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` expression, and the second with a sequence of `if` expressions:

```
# let plus_one_match x =
    match x with
    | 0 -> 1
    | 1 -> 2
    | 2 -> 3
    | 3 -> 4
    | 4 -> 5
    | 5 -> 6
    | _ -> x + 1;;
val plus_one_match : int -> int = <fun>
# let plus_one_if x =
    if      x = 0 then 1
    else if x = 1 then 2
    else if x = 2 then 3
    else if x = 3 then 4
    else if x = 4 then 5
    else if x = 5 then 6
    else x + 1;;
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.

```
# #require "core_bench";;
# open Core_bench;;
# [ Bench.Test.create ~name:"plus_one_match" (fun () ->
        plus_one_match 10)
  ; Bench.Test.create ~name:"plus_one_if" (fun () ->
        plus_one_if 10) ]
  |> Bench.bench;;
Estimated testing time 20s (2 benchmarks x 10s). Change using -quota
    SECS.

 Name            Time/Run
```

```
 plus_one_match   34.86ns
 plus_one_if      54.89ns

- : unit = ()
```

Here's another, less artificial example. We can rewrite the `sum` function we described earlier in the chapter using an `if` expression 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:

```
# 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:

```
# let numbers = List.range 0 1000 in
  [ Bench.Test.create ~name:"sum_if" (fun () -> sum_if numbers)
  ; Bench.Test.create ~name:"sum"    (fun () -> sum numbers) ]
  |> Bench.bench;;
Estimated testing time 20s (2 benchmarks x 10s). Change using -quota
    SECS.

 Name    Time/Run

 sum_if   62.00us
 sum      17.99us

- : 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` expression, this work happens exactly once per list element.

This is a more general phenomenon: pattern matching is very efficient, and is usually faster than what you might write yourself.

4.3.2 Detecting Errors

The error-detecting capabilities of `match` expressions 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` expressions 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:

```
# let rec drop_zero l =
    match l with
    | [] -> []
    | 0  :: tl -> drop_zero tl;;
Lines 2-4, characters 5-31:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case 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 7 (Variants), 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.

4.4 Using the List Module Effectively

We've so far written a fair amount of list-munging code using pattern matching and recursive functions. 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. 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. When we're done, here's how the resulting function should work:

```
# Stdio.print_endline
    (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:

```
# 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:

```
# 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:

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

```
# 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:

```
# 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:

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

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

```
# 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:

```
# let render_separator widths =
    let pieces = List.map widths
        ~f:(fun w -> String.make w '-')
    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 large numbers of strings, since it allocates a new string every time it runs. Thus, the following code

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

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

```
# 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:

```
# let pad s length =
    s ^ String.make (length - String.length s) ' ';;
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:

```
# 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           |"
```

Finally, we can bring this all together to build the `render_table` function we wanted at the start!

```
# 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>
```

4.4.1 More Useful List Functions

The previous example 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[1]), 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:

```
# 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:

```
# 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 want to restrict your attention to a subset of the values on your list. The `List.filter` function is one way of doing that:

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

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 function computes a list of file extensions from a list of files, piping the results through `List.dedup_and_sort` to return the list with duplicates removed and in sorted order. Note that this example uses `String.rsplit2` from the String module to split a string on the rightmost appearance of a given character:

[1] https://v3.ocaml.org/p/base/v0.14.3/doc/Base/List/index.html

```
# let extensions filenames =
    List.filter_map filenames ~f:(fun fname ->
        match String.rsplit2 ~on:'.' fname with
        | None  | Some ("",_) -> None
        | Some (_,ext) ->
          Some ext)
    |> List.dedup_and_sort ~compare:String.compare;;
val extensions : string list -> string list = <fun>
# extensions ["foo.c"; "foo.ml"; "bar.ml"; "bar.mli"];;
- : string list = ["c"; "ml"; "mli"]
```

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:

```
# 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 ["foo.c"; "foo.ml"; "bar.ml"; "bar.mli"]
    ~f:is_ocaml_source;;
val ml_files : string list = ["foo.ml"; "bar.ml"; "bar.mli"]
val other_files : string list = ["foo.c"]
```

Combining lists

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

```
# 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`.

```
# [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:

```
# 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.

```
# module Sys = Core.Sys
  module Filename = Core.Filename;;
module Sys = Core.Sys
module Filename = Core.Filename
# 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 (Filename.concat s sub))
      |> List.concat;;
val ls_rec : string -> string list = <fun>
```

Note that this example uses some functions from the `Sys` and `Filename` modules from `Core` for accessing the filesystem and dealing with filenames.

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:

```
# 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 (Filename.concat s
    sub));;
val ls_rec : string -> string list = <fun>
```

4.5 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:

```
# 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:

```
# 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:

```
# 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:

```
# 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 non-tail 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.

4.6 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 Chapter 2.3.2 (Recursive List Functions): the function `remove_sequential_duplicates`. Here's the implementation that was described earlier:

```
# let rec remove_sequential_duplicates list =
    match list with
    | [] -> []
    | [x] -> [x]
    | first :: second :: tl ->
      if first = second then
        remove_sequential_duplicates (second :: tl)
      else
        first :: remove_sequential_duplicates (second :: tl);;
val remove_sequential_duplicates : int list -> int 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 above code is that it in some cases re-creates on the right-hand side of the arrow a value that already existed on the left-hand 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:

```
# let rec remove_sequential_duplicates = function
    | [] as l -> l
    | [_] as l -> l
    | first :: (second :: _ as tl) ->
      if first = second then
        remove_sequential_duplicates tl
      else
        first :: remove_sequential_duplicates tl;;
val remove_sequential_duplicates : int list -> int list = <fun>
```

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

```
# let rec remove_sequential_duplicates list =
    match list with
    | [] | [_] as l -> l
    | first :: (second :: _ as tl) ->
      if first = second then
        remove_sequential_duplicates tl
      else
        first :: remove_sequential_duplicates tl;;
val remove_sequential_duplicates : int list -> int 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:

```
# let rec remove_sequential_duplicates list =
    match list with
    | [] | [_] as l -> l
    | first :: (second :: _ as tl) when first = second ->
      remove_sequential_duplicates tl
    | first :: tl -> first :: remove_sequential_duplicates tl;;
val remove_sequential_duplicates : int list -> int list = <fun>
```

Polymorphic Compare

You might have noticed that `remove_sequential_duplicates` is specialized to lists of integers. That's because `Base`'s default equality operator is specialized to integers, as you can see if you try to apply it to values of a different type.

```
# open Base;;
# "foo" = "bar";;
Line 1, characters 1-6:
Error: This expression has type string but an expression was expected
    of type
         int
```

OCaml also has a collection of polymorphic equality and comparison operators, which we can make available by opening the module `Base.Poly`.

```
# open Base.Poly;;
# "foo" = "bar";;
- : bool = false
# 3 = 4;;
- : bool = false
# [1;2;3] = [1;2;3];;
- : bool = true
```

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

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

If we rewrite `remove_sequential_duplicates` with `Base.Poly` open, we'll see that it gets a polymorphic type, and can now be used on inputs of different types.

```
# let rec remove_sequential_duplicates list =
    match list with
    | [] | [_] as l -> l
    | first :: (second :: _ as tl) when first = second ->
      remove_sequential_duplicates tl
    | first :: tl -> first :: remove_sequential_duplicates tl;;
val remove_sequential_duplicates : 'a list -> 'a list = <fun>
# remove_sequential_duplicates [1;2;2;3;4;3;3];;
- : int list = [1; 2; 3; 4; 3]
# remove_sequential_duplicates ["one";"two";"two";"two";"three"];;
- : string list = ["one"; "two"; "three"]
```

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. (You can learn more about this structure in Chapter 24 (Memory Representation of Values).)

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

```
# (fun x -> x + 1) = (fun x -> x + 1);;
Exception: (Invalid_argument "compare: 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 most 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.

That said, experienced OCaml developers typically avoid polymorphic comparison. That's surprising, given how obviously useful it is, but there's a good reason. While it's very convenient, in some cases, the type oblivious nature of polymorphic compare means that it does something that doesn't make sense for the particular type of values you're dealing with. This can lead to surprising and hard to resolve bugs in your code. It's for this reason that `Base` discourages the use of polymorphic compare by hiding it by default.

We'll discuss the downsides of polymorphic compare in more detail in Chapter 15 (Maps and Hash Tables).

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 is 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:

```
# 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;;
Lines 2-5, characters 5-57:
```

```
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case 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:

```
# 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.

```
# 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:

```
# 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:

```
# 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:

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

5 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.

5.1 Single-File Programs

We'll start with an example: a utility that reads lines from `stdin`, computes a frequency count of the lines, and prints out the ten most frequent lines. 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:

```
# open Base;;
# let assoc = [("one", 1); ("two",2); ("three",3)];;
val assoc : (string * int) list = [("one", 1); ("two", 2); ("three",
    3)]
# List.Assoc.find ~equal:String.equal assoc "two";;
- : int option = Some 2
# List.Assoc.add ~equal:String.equal assoc "four" 4;;
- : (string, int) Base.List.Assoc.t =
[("four", 4); ("one", 1); ("two", 2); ("three", 3)]
# List.Assoc.add ~equal:String.equal assoc "two"  4;;
- : (string, int) Base.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`.

```
open Base
open Stdio
```

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

let () =
  build_counts ()
  |> List.sort ~compare:(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 4 (Lists and Patterns)), 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 3.2.4 (Prefix and Infix Operators).

Where Is `main`?

Unlike programs in C, Java or C#, programs in OCaml don't 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 right-hand side returns `unit`, as is common for functions that operate primarily by side effect.

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

```
$ ocamlopt freq.ml -o freq
File "freq.ml", line 1, characters 5-9:
1 | open Base
         ^^^^
Error: Unbound module Base
```

```
[2]
```

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

```
$ ocamlfind ocamlopt -linkpkg -package base -package stdio freq.ml -o
    freq
```

This uses `ocamlfind`, a tool which itself invokes other parts of the OCaml toolchain (in this case, `ocamlopt`) with the appropriate flags to link in particular libraries and packages. Here, `-package base` is asking `ocamlfind` to link in the `Base` library; `-linkpkg` asks ocamlfind to link in the packages as is necessary for building an executable.

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 `dune`. To invoke `dune`, you need to have two files: a `dune-project` file for the overall project, and a `dune` file that configures the particular directory. This is a single-directory project, so we'll just have one of each, but more realistic projects will have one `dune-project` and many `dune` files.

At its simplest, the `dune-project` just specifies the version of the `dune` configuration-language in use.

```
(lang dune 3.0)
```

We also need a `dune` file to declare the executable we want to build, along with the libraries it depends on.

```
(executable
  (name      freq)
  (libraries base stdio))
```

With that in place, we can invoke `dune` as follows.

```
$ dune build freq.exe
```

We can run the resulting executable, `freq.exe`, from the command line. Executables built with `dune` will be left in the `_build/default` directory, from which they can be invoked. The specific invocation below will count the words that come up in the file `freq.ml` itself.

```
$ grep -Eo '[[:alpha:]]+' freq.ml | ./_build/default/freq.exe
  5: line
  5: List
  5: counts
  4: count
  4: fun
  4: x
  4: equal
  3: let
  2: f
  2: l
```

Conveniently, `dune` allows us to combine the building and running an executable into a single operation, which we can do using `dune exec`.

```
$ grep -Eo '[[:alpha:]]+' freq.ml | dune exec ./freq.exe
  5: line
  5: List
  5: counts
  4: count
  4: fun
  4: x
  4: equal
  3: let
  2: f
  2: l
```

We've really just scratched the surface of what can be done with `dune`. We'll discuss `dune` in more detail in Chapter 1.1.3 (The OCaml Platform).

Bytecode Versus Native Code

OCaml ships with two compilers: the `ocamlopt` native code compiler and the `ocamlc` bytecode compiler. Programs compiled with `ocamlc` are interpreted by a virtual machine, while programs compiled with `ocamlopt` are compiled to machine code to be run on a specific operating system and processor architecture. With `dune`, targets ending with `.bc` are built as bytecode executables, and those ending with `.exe` 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 some limitations on 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 27 (The Compiler Backend: Bytecode and Native code).

5.2 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.

```
open Base

let touch counts line =
  let count =
    match List.Assoc.find ~equal:String.equal counts line with
    | None -> 0
    | Some x -> x
  in
  List.Assoc.add ~equal:String.equal counts line (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`.

```
open Base
open Stdio

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

let () =
  build_counts ()
  |> List.sort ~compare:(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 resulting code can still be built with `dune`, which will discover dependencies and realize that `counter.ml` needs to be compiled.

```
$ dune build freq.exe
```

5.3 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:

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

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

```
open Base

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

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

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`:

```
open Base

(** 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
```

We added `empty` and `to_list` to `Counter`, since without them 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 `odoc` tool when generating API documentation. We'll discuss `odoc` more in Chapter 22.2.2 (Browsing Interface Documentation).

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

```
open Base

type t = (string * int) list
```

```
let empty = []
let to_list x = x

let touch counts line =
  let count =
    match List.Assoc.find ~equal:String.equal counts line with
    | None -> 0
    | Some x -> x
  in
  List.Assoc.add ~equal:String.equal counts line (count + 1)
```

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

```
$ dune build freq.exe
File "freq.ml", line 5, characters 53-66:
5 |   In_channel.fold_lines In_channel.stdin ~init:[] ~f:Counter.touch
                                                         ^^^^^^^^^^^^^
Error: This expression has type Counter.t -> Export.string ->
    Counter.t
       but an expression was expected of type
         'a list -> Export.string -> 'a list
       Type Counter.t is not compatible with type 'a list
[1]
```

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 to use `Counter.to_list` to convert the completed counts to an association list. The resulting implementation is shown below.

```
open Base
open Stdio

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

let () =
  build_counts ()
  |> Counter.to_list
  |> List.sort ~compare:(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)
```

With this implementation, the build now succeeds!

```
$ dune build freq.exe
```

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

```
open Base

type t = int Map.M(String).t

let empty = Map.empty (module String)
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.set t ~key:s ~data:(count + 1)
```

There's some unfamiliar syntax in the above example, in particular the use of `int Map.M(String).t` to indicate the type of a map, and `Map.empty (module String)` to generate an empty map. Here, we're making use of a more advanced feature of the language (specifically, functors and first-class modules, which we'll get to in later chapters). The use of these features for the Map data-structure in particular is covered in Chapter 15 (Maps and Hash Tables).

5.4 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 single 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:

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

let median t =
  let sorted_strings =
    List.sort (Map.to_alist t) ~compare:(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 % 2 = 1
  then Median (nth (len / 2))
  else Before_and_after (nth ((len / 2) - 1), nth (len / 2))
```

In the above, we use `failwith` to throw an exception for the case of the empty list. We'll discuss exceptions more in Chapter 8 (Error Handling). 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 to `counter.mli` does the trick.

```
(** 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.

5.5 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:

```
open Base

module Username : sig
  type t

  val of_string : string -> t
  val to_string : t -> string
  val ( = ) : t -> t -> bool
end = struct
  type t = string

  let of_string x = x
  let to_string x = x
  let ( = ) = String.( = )
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. We also chose to put in an equality function, so you can check if two usernames match. In a real application, we might want more functionality, like the ability to hash and compare usernames, but we've kept this example purposefully simple.

The basic structure of a module declaration like this is:

```
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:

```
open Base
module Time = Core.Time

module type ID = sig
  type t

  val of_string : string -> t
  val to_string : t -> string
  val ( = ) : t -> t -> bool
end

module String_id = struct
  type t = string

  let of_string x = x
  let to_string x = x
  let ( = ) = String.( = )
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 = Username.( = ) 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.

```
$ dune build session_info.exe
File "session_info.ml", line 29, characters 59-66:
29 | let sessions_have_same_user s1 s2 = Username.( = ) s1.user
     s2.host

     ^^^^^^^
Error: This expression has type Hostname.t
       but an expression was expected of type Username.t
[1]
```

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.

5.6 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 Base` to get access to the standard definitions in the `Base` 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:

```
# module M = struct let foo = 3 end;;
module M : sig val foo : int end
# foo;;
Line 1, characters 1-4:
Error: Unbound value foo
# open M;;
# foo;;
- : int = 3
```

Here's some general advice on how to use `open` effectively.

5.6.1 Open Modules Rarely

`open` is essential when you're using an alternative standard library like `Base`, 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.

When you do use `open`, it should mostly be with modules that were designed to be opened, like `Base` itself, or `Option.Monad_infix` or `Float.O` within `Base`..

5.6.2 Prefer Local Opens

It's generally better to keep down the amount of code affected by an `open`. One great tool for this is *local open*s, which let you restrict the scope of an open to an arbitrary expression. There are two syntaxes for local opens. The following example shows the `let open` syntax;

```
# 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.

The following shows off a more lightweight syntax which is particularly useful for small expressions.

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

5.6.3 Using Module Shortcuts Instead

An alternative to local opens 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:

```
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:

```
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.

5.7 Including Modules

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

```
# 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:

```
# 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:

```
# 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;;
Line 1, characters 29-53:
Error: Unbound value Extended_interval.create
```

To consider a more realistic example, imagine you wanted to build an extended version of the `Option` module, where you've added some functionality not present in the module as distributed in `Base`. That's a job for `include`.

```
open Base

(* The new function we're going to add *)
let apply f_opt x =
  match f_opt with
  | None -> None
  | Some f -> Some (f x)

(* The remainder of the option module *)
include Option
```

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 issue is that we need to get our hands on the signature for the `Option` module. This can be done using `module type of`, which computes a signature from a module:

```
open Base

(* Include the interface of the option module from Base *)
include module type of Option
```

```
(* Signature of function we're adding *)
val apply : ('a -> 'b) t -> 'a -> 'b t
```

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 `Option` with a new function of the same name, the declaration of that function in the `ml` would have to come after the `include Option` declaration.

We can now use `Ext_option` as a replacement for `Option`. If we want to use `Ext_option` in preference to `Option` in our project, we can create a file of common definitions, which in this case we'll call `import.ml`.

```
module Option = Ext_option
```

Then, by opening `Import`, we can shadow `Base`'s `Option` module with our extension.

```
open Base
open Import

let lookup_and_apply map key x = Option.apply (Map.find map key) x
```

5.8 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.

5.8.1 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:

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

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

```
$ dune build freq.exe
File "counter.ml", line 1:
Error: The implementation counter.ml
       does not match the interface
    .freq.eobjs/byte/dune__exe__Counter.cmi:
       Values do not match:
         val touch :
           ('a, int, 'b) Base.Map.t -> 'a -> ('a, int, 'b) Base.Map.t
       is not included in
         val touch : string -> t -> t
       File "counter.mli", line 16, characters 0-28: Expected
    declaration
       File "counter.ml", line 8, characters 4-9: Actual declaration
[1]
```

5.8.2 Missing Definitions

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

```
(** Returns the frequency count for the given string *)
val count : t -> string -> int
```

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

```
$ dune build freq.exe
File "counter.ml", line 1:
Error: The implementation counter.ml
       does not match the interface
    .freq.eobjs/byte/dune__exe__Counter.cmi:
       The value `count' is required but not provided
       File "counter.mli", line 15, characters 0-30: Expected
    declaration
[1]
```

A missing type definition will lead to a similar error.

5.8.3 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:

```
(** 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

val median : t -> median
```

will lead to a compilation error.

```
$ dune build freq.exe
File "counter.ml", line 1:
Error: The implementation counter.ml
       does not match the interface
    .freq.eobjs/byte/dune__exe__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
       Constructors number 1 have different names, Median and
    Before_and_after.
       File "counter.mli", lines 21-23, characters 0-20: Expected
    declaration
```

```
       File "counter.ml", lines 17-19, characters 0-39: Actual
    declaration
[1]
```

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.

5.8.4 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 Chapter 3.2.3 (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`.

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

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

```
$ dune build freq.exe
File "counter.ml", line 18, characters 18-31:
18 | let singleton l = Counter.touch Counter.empty
                       ^^^^^^^^^^^^^
Error: The module Counter is an alias for module Dune__exe__Counter,
    which is the current compilation unit
[1]
```

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.

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

In this case, `dune` will notice the error and complain explicitly about the cycle:

```
$ dune build freq.exe
Error: Dependency cycle between the following files:
   _build/default/.freq.eobjs/freq.impl.all-deps
-> _build/default/.freq.eobjs/counter.impl.all-deps
-> _build/default/.freq.eobjs/freq.impl.all-deps
[1]
```

5.9 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.

5.9.1 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 preserve 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 4 (Lists and Patterns) 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.

5.9.2 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 of client code. One example is labeled arguments (discussed in Chapter 3.2.6 (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 a module interface, should be longer and more descriptive.

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 verbosity goes down and the benefit of explicitness goes up the less often a name is used.

5.9.3 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.

`Base`, `Core` and related libraries have been designed with a uniform set of standards in mind around the design of module interfaces. Here are some of the guidelines that they use.

- *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 type `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 8 (Error Handling)).

There are also standards in Base 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 Chapter 11.2.4 (Using Multiple Interfaces).

Base'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.

5.9.4 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.

6 Records

One of OCaml's best features is its concise and expressive system for declaring new data types. *Records* are a key element of that system. We discussed records briefly in Chapter 2 (A Guided Tour), but this chapter will go into more depth, covering more of the technical details, as well as providing advice on how to use records 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:

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

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

Here's a simple example: a `service_info` record that represents an entry from the `/etc/services` file on a typical Unix system. That file is used for keeping track of the well-known port and protocol name for protocols such as FTP or SSH. Note that we're going to open `Core` in this example rather than `Base`, since we're using the Unix API, which you need `Core` for.

```
open Core
type service_info =
  { service_name : string;
    port         : int;
    protocol     : string;
  }
```

We can construct a `service_info` just as easily as we declared its type. The following function tries to construct such a record given as input a line from `/etc/services` file. To do this, we'll use `Re`, a regular expression engine for OCaml. If you don't know how regular expressions work, you can just think of them as a simple pattern language you can use for parsing a string. (You may need to install it first by running `opam install re`.)

```
# #require "re";;
# let service_info_of_string line =
    let matches =
      let pat = "([a-zA-Z]+)[ \t]+([0-9]+)/([a-zA-Z]+)" in
```

```
      Re.exec (Re.Posix.compile_pat pat) line
    in
    { service_name = Re.Group.get matches 1;
      port = Int.of_string (Re.Group.get matches 2);
      protocol = Re.Group.get matches 3;
    }
  ;;
val service_info_of_string : string -> service_info = <fun>
```

We can now construct a concrete record by calling the function on a line from the file.

```
# let ssh = service_info_of_string "ssh 22/udp # SSH Remote Login
    Protocol";;
val ssh : service_info = {service_name = "ssh"; port = 22; protocol =
    "udp"}
```

You might wonder how the compiler inferred that our function returns a value of type `service_info`. In this case, the compiler bases its inference on the field names used in constructing the record. That inference is most straightforward when each field name belongs to only one record type. We'll discuss later in the chapter what happens when field names are shared across different record types.

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

```
# ssh.port;;
- : int = 22
```

When declaring an OCaml type, you always have the option of parameterizing it by a polymorphic type. Records are no different in this regard. As an example, here's a type that represents an arbitrary item tagged with a line number.

```
type 'a with_line_num = { item: 'a; line_num: int }
```

We can then write polymorphic functions that operate over this parameterized type. For example, this function takes a file and parses it as a series of lines, using the provided function for parsing each individual line.

```
# let parse_lines parse file_contents =
    let lines = String.split ~on:'\n' file_contents in
    List.mapi lines ~f:(fun line_num line ->
      { item = parse line;
        line_num = line_num + 1;
      })
  ;;
val parse_lines : (string -> 'a) -> string -> 'a with_line_num list =
    <fun>
```

We can then use this function for parsing a snippet of a real `/etc/services` file.

```
# parse_lines service_info_of_string
    "rtmp              1/ddp     # Routing Table Maintenance Protocol
     tcpmux            1/udp     # TCP Port Service Multiplexer
     tcpmux            1/tcp     # TCP Port Service Multiplexer";;
- : service_info with_line_num list =
[{item = {service_name = "rtmp"; port = 1; protocol = "ddp"};
    line_num = 1};
```

```
 {item = {service_name = "tcpmux"; port = 1; protocol = "udp"};
    line_num = 2};
 {item = {service_name = "tcpmux"; port = 1; protocol = "tcp"};
    line_num = 3}]
```

The polymorphism lets us use the same function when parsing a different format, like this function for parsing a file containing an integer on every line.

```
# parse_lines Int.of_string "1\n10\n100\n1000";;
- : int with_line_num list =
[{item = 1; line_num = 1}; {item = 10; line_num = 2};
 {item = 100; line_num = 3}; {item = 1000; line_num = 4}]
```

6.1 Patterns and Exhaustiveness

Another way of getting information out of a record is by using a pattern match, as shown in the following function.

```
# let service_info_to_string
    { service_name = name; port = port; protocol = prot  }
    =
    sprintf "%s %i/%s" name port prot
  ;;
val service_info_to_string : service_info -> string = <fun>
# service_info_to_string ssh;;
- : string = "ssh 22/udp"
```

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. That's 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 change our `service_info` record so that it preserves comments. We can do this by providing a new definition of `service_info` that includes a `comment` field:

```
type service_info =
  { service_name : string;
    port         : int;
    protocol     : string;
    comment      : string option;
  }
```

The code for `service_info_to_string` would continue to compile without change. But in this case, we should probably update the code so that the generated string

includes the comment if it's there. It would be nice if the type system would warn us that we should consider updating the function.

Happily, OCaml offers an optional warning for missing fields in record patterns. With that warning turned on (which you can do in the toplevel by typing `#warnings "+9"`), the compiler will indeed warn us.

```
# #warnings "+9";;
# let service_info_to_string
    { service_name = name; port = port; protocol = prot  }
    =
    sprintf "%s %i/%s" name port prot
  ;;
Line 2, characters 5-59:
Warning 9 [missing-record-field-pattern]: the following labels are
    not bound in this record pattern:
comment
Either bind these labels explicitly or add '; _' to the pattern.
val service_info_to_string : service_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:

```
# let service_info_to_string
    { service_name = name; port = port; protocol = prot; _ }
    =
    sprintf "%s %i/%s" name port prot
  ;;
val service_info_to_string : service_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:

```
$ ocaml -warn-help | egrep '\b9\b'
  9 [missing-record-field-pattern] Missing fields in a record pattern.
  R Alias for warning 9.
```

You can think of OCaml's warnings as a powerful set of optional static analysis tools. They're enormously helpful in catching all sorts of bugs, and you should 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-4-33-40-41-42-43-34-44`.

The syntax of `-w` 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.

6.2 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*:

```
# let service_info_to_string { service_name; port; protocol; comment
    } =
    let base = sprintf "%s %i/%s" service_name port protocol in
    match comment with
    | None -> base
    | Some text -> base ^ " #" ^ text;;
val service_info_to_string : service_info -> string = <fun>
```

Field punning can also be used to construct a record. Consider the following updated version of `service_info_of_string`.

```
# let service_info_of_string line =
    (* first, split off any comment *)
    let (line,comment) =
      match String.rsplit2 line ~on:'#' with
      | None -> (line,None)
      | Some (ordinary,comment) -> (ordinary, Some comment)
    in
    (* now, use a regular expression to break up the
       service definition *)
    let matches =
      Re.exec
        (Re.Posix.compile_pat
           "([a-zA-Z]+)[ \t]+([0-9]+)/([a-zA-Z]+)")
        line
    in
    let service_name = Re.Group.get matches 1 in
    let port = Int.of_string (Re.Group.get matches 2) in
    let protocol = Re.Group.get matches 3 in
    { service_name; port; protocol; comment };;
val service_info_of_string : string -> service_info = <fun>
```

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:

```
# let create_service_info ~service_name ~port ~protocol ~comment =
    { service_name; port; protocol; comment };;
val create_service_info :
  service_name:string ->
  port:int -> protocol:string -> comment:string option ->
    service_info =
  <fun>
```

This is considerably more concise than what you would get without punning:

```
# let create_service_info
        ~service_name:service_name ~port:port
        ~protocol:protocol ~comment:comment =
    { service_name = service_name;
      port = port;
      protocol = protocol;
      comment = comment;
    };;
val create_service_info :
  service_name:string ->
  port:int -> protocol:string -> comment:string option ->
    service_info =
  <fun>
```

Together, 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.

6.3 Reusing Field Names

Defining records with the same field names can be problematic. As a simple example, let's consider a collection of types representing the protocol of 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 when initiating 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.

```
type log_entry =
  { session_id: string;
    time: Time_ns.t;
    important: bool;
    message: string;
  }
type heartbeat =
  { session_id: string;
    time: Time_ns.t;
    status_message: string;
  }
type logon =
  { session_id: string;
    time: Time_ns.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?

```
# 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:

```
# 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:

```
# 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);;
Line 1, characters 45-59:
Error: This expression has type logon
       There is no field status_message within type logon
```

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.

Adding a type annotation resolves the ambiguity, no matter what order the fields are considered in.

```
# let session_and_status (t:heartbeat) = (t.session_id,
    t.status_message);;
val session_and_status : heartbeat -> string * string = <fun>
```

We can avoid the ambiguity altogether, either by using nonoverlapping field names or by putting different record types in different modules. Indeed, packing types into modules is a broadly useful idiom (and one used quite extensively by `Base`), 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`. So, we would write:

```
module Log_entry = struct
  type t =
    { session_id: string;
      time: Time_ns.t;
      important: bool;
      message: string;
    }
end
module Heartbeat = struct
  type t =
    { session_id: string;
      time: Time_ns.t;
      status_message: string;
    }
```

```
end
module Logon = struct
  type t =
    { session_id: string;
      time: Time_ns.t;
      user: string;
      credentials: string;
    }
end
```

Now, our log-entry-creation function can be rendered as follows:

```
# let create_log_entry ~session_id ~important message =
    { Log_entry.time = Time_ns.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:

```
# let create_log_entry ~session_id ~important message =
    { Log_entry.
      time = Time_ns.now (); session_id; important; message };;
val create_log_entry :
  session_id:string -> important:bool -> string -> Log_entry.t = <fun>
```

Earlier, we saw that you could help OCaml understand which record field was intended by adding a type annotation. We can use that here to make the example even more concise.

```
# let create_log_entry ~session_id ~important message : Log_entry.t =
    { time = Time_ns.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 approaches when pattern matching:

```
# 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 as well.

```
# 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.

Qualifying a record field by the module it comes from can be awkward. Happily, OCaml doesn't require that the record field be qualified if it can otherwise infer the type of the record in question. In particular, we can rewrite the above declarations by adding type annotations and removing the module qualifications.

```
# let message_to_string ({ important; message; _ } : Log_entry.t) =
    if important then String.uppercase message else message;;
val message_to_string : Log_entry.t -> string = <fun>
# let is_important (t:Log_entry.t) = t.important;;
val is_important : Log_entry.t -> bool = <fun>
```

This feature of the language, known by the somewhat imposing name of *type-directed constructor disambiguation*, applies to variant tags as well as record fields, as we'll see in Chapter 7 (Variants).

6.4 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.

```
type client_info =
  { addr: Unix.Inet_addr.t;
    port: int;
    user: string;
    credentials: string;
    last_heartbeat_time: Time_ns.t;
}
```

We could define a function for updating the client information when a new heartbeat arrives as follows.

```
# 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 following shows how we can use functional updates to rewrite `register_heartbeat` more concisely.

```
# let register_heartbeat t hb =
    { t with last_heartbeat_time = hb.Heartbeat.time };;
val register_heartbeat : client_info -> Heartbeat.t -> client_info =
    <fun>
```

The `with` keyword marks that this is a functional update, and the value assignments on the right-hand side indicate the changes to be made to the record on the left-hand side of the `with`.

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:

```
type client_info =
  { addr: Unix.Inet_addr.t;
    port: int;
    user: string;
    credentials: string;
    last_heartbeat_time: Time_ns.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:

```
# 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>
```

These downsides notwithstanding, functional updates are very useful, and a good choice for cases where it's not important that you consider every field of the record when making a change.

6.5 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:

```
type client_info =
  { addr: Unix.Inet_addr.t;
    port: int;
    user: string;
```

```
    credentials: string;
    mutable last_heartbeat_time: Time_ns.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:

```
# let register_heartbeat t (hb:Heartbeat.t) =
    t.last_heartbeat_time   <- hb.time;
    t.last_heartbeat_status <- hb.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 Chapter 9 (Imperative Programming).

6.6 First-Class Fields

Consider the following function for extracting the usernames from a list of `Logon` messages:

```
# let get_users logons =
    List.dedup_and_sort ~compare:String.compare
      (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 `ppx_fields_conv` syntax extension that ships with `Core` does just that.

The `[@@deriving fields]` annotation at the end of the declaration of a record type will cause the extension to be applied to a given type declaration. We need to enable the extension explicitly,

```
# #require "ppx_jane";;
```

at which point, we can define `Logon` as follows:

```
# module Logon = struct
    type t =
      { session_id: string;
        time: Time_ns.t;
        user: string;
        credentials: string;
      }
    [@@deriving fields]
  end;;
module Logon :
```

```
  sig
    type t = {
      session_id : string;
      time : Time_ns.t;
      user : string;
      credentials : string;
    }
    val credentials : t -> string
    val user : t -> string
    val time : t -> Time_ns.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_ns.t)
    Field.t_with_perm
...
      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:

```
# let get_users logons =
    List.dedup_and_sort ~compare:String.compare
  (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`:

```
# 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:

```
# 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:

```
# 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:

```
# let logon = { Logon.
                session_id = "26685";
                time = Time_ns.of_string "2017-07-21 10:11:45 EST";
                user = "yminsky";
                credentials = "Xy2d9W"; };;
val logon : Logon.t =
  {Logon.session_id = "26685"; time = 2017-07-21 15:11:45.000000000Z;
   user = "yminsky"; credentials = "Xy2d9W"}
# show_field Logon.Fields.user Fn.id logon;;
- : string = "user: yminsky"
# show_field Logon.Fields.time Time_ns.to_string logon;;
- : string = "time: 2017-07-21 15:11:45.000000000Z"
```

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:

```
# Logon.Fields.iter;;
- : session_id:(([< `Read | `Set_and_create ], Logon.t, string)
                Field.t_with_perm -> unit) ->
```

```
    time:(([< `Read | `Set_and_create ], Logon.t, Time_ns.t)
          Field.t_with_perm -> unit) ->
    user:(([< `Read | `Set_and_create ], Logon.t, string)
    Field.t_with_perm ->
          unit) ->
    credentials:(([< `Read | `Set_and_create ], Logon.t, string)
                 Field.t_with_perm -> unit) ->
    unit
= <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:

```
# 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_ns.to_string)
      ~user:(print Fn.id)
      ~credentials:(print Fn.id);;
val print_logon : Logon.t -> unit = <fun>
# print_logon logon;;
session_id: 26685
time: 2017-07-21 15:11:45.000000000Z
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.

7 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:

```
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 (also called a *constructor*, since you use it to construct a value) 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. Most UNIX-like operating systems support terminals as a fundamental, text-based user interface. Almost all of these terminals support a set of eight basic colors.

Those colors can be naturally represented as 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.

```
open Base
open Stdio
type basic_color =
  | Black | Red | Green | Yellow | Blue | Magenta | Cyan | White
```

As we show below, the variant tags introduced by the definition of `basic_color` can be used for constructing values of that type.

```
# Cyan;;
- : basic_color = Cyan
# [Blue; Magenta; Red];;
- : basic_color list = [Blue; Magenta; Red]
```

The following function uses pattern matching to convert each of these to the corresponding integer code that is used for communicating these colors to the terminal.

```
# 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]
```

We know that the above function handles every color in `basic_color` because the compiler would have warned us if we'd missed one:

```
# let incomplete_color_to_int = function
    | Black -> 0 | Red -> 1 | White -> 7;;
Lines 1-2, characters 31-41:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Green|Yellow|Blue|Magenta|Cyan)
val incomplete_color_to_int : basic_color -> int = <fun>
```

In any case, using the correct function, we can generate escape codes to change the color of a given string displayed in a terminal.

```
# let color_by_number number text =
  Printf.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!
- : unit = ()
```

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 simple enumerations.

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 $6 \times 6 \times 6$ 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.

```
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 *)
```

As before, we can use these introduced tags to construct values of our newly defined type.

```
# [RGB (250,70,70); Basic (Green, Regular)];;
- : color list = [RGB (250, 70, 70); Basic (Green, Regular)]
```

And again, we'll use pattern matching to convert a color to a corresponding integer. 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:

```
# 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:

```
# 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!
- : unit = ()
# color_print (Gray 4) "A muted gray...";;
A muted gray...
- : unit = ()
```

Variants, Tuples and Parens

Variants with multiple arguments look an awful lot like tuples. Consider the following example of a value of the type `color` we defined earlier.

```
# RGB (200,0,200);;
- : color = RGB (200, 0, 200)
```

It really looks like we've created a 3-tuple and wrapped it with the `RGB` constructor. But that's not what's really going on, as you can see if we create a tuple first and then place it inside the `RGB` constructor.

```
# let purple = (200,0,200);;
val purple : int * int * int = (200, 0, 200)
# RGB purple;;
Line 1, characters 1-11:
Error: The constructor RGB expects 3 argument(s),
       but is applied here to 1 argument(s)
```

We can also create variants that explicitly contain tuples, like this one.

```
# type tupled = Tupled of (int * int);;
type tupled = Tupled of (int * int)
```

The syntactic difference is unfortunately quite subtle, coming down to the extra set of parens around the arguments. But having defined it this way, we can now take the tuple in and out freely.

```
# let of_tuple x = Tupled x;;
val of_tuple : int * int -> tupled = <fun>
# let to_tuple (Tupled x) = x;;
val to_tuple : tupled -> int * int = <fun>
```

If, on the other hand, we define a variant without the parens, then we get the same behavior we got with the `RGB` constructor.

```
# type untupled = Untupled of int * int;;
type untupled = Untupled of int * int
# let of_tuple x = Untupled x;;
Line 1, characters 18-28:
Error: The constructor Untupled expects 2 argument(s),
       but is applied here to 1 argument(s)
# let to_tuple (Untupled x) = x;;
Line 1, characters 14-26:
Error: The constructor Untupled expects 2 argument(s),
       but is applied here to 1 argument(s)
```

Note that, while we can't just grab the tuple as a whole from this type, we can achieve more or less the same ends by explicitly deconstructing and reconstructing the data we need.

```
# let of_tuple (x,y) = Untupled (x,y);;
val of_tuple : int * int -> untupled = <fun>
# let to_tuple (Untupled (x,y)) = (x,y);;
val to_tuple : untupled -> int * int = <fun>
```

The differences between a multi-argument variant and a variant containing a tuple are mostly about performance. A multi-argument variant is a single allocated block in memory, while a variant containing a tuple requires an extra heap-allocated block for the tuple. You can learn more about OCaml's memory representation in Chapter 24 (Memory Representation of Values).

7.1 Catch-All Cases and Refactoring

OCaml's type system can act as a refactoring tool, warning you of places where your code needs to be updated to match an interface change. This is particularly valuable in the context of variants.

Consider what would happen if we were to change the definition of `color` to the following:

```
type color =
  | Basic of basic_color     (* basic colors *)
  | Bold  of basic_color     (* bold basic colors *)
  | RGB   of int * int * int (* 6x6x6 color cube *)
  | Gray  of int             (* 24 grayscale levels *)
```

We've essentially broken out the `Basic` case into two cases, `Basic` and `Bold`, and `Basic` has changed from having two arguments to one. `color_to_int` as we wrote it still expects the old structure of the variant, and if we try to compile that same code again, the compiler will notice the discrepancy:

```
# 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;;
Line 2, characters 13-33:
Error: This pattern matches values of type 'a * 'b
       but a pattern was expected which matches values of type
    basic_color
```

Here, the compiler is complaining that the `Basic` tag is used with the wrong number of arguments. If we fix that, however, the compiler will flag a second problem, which is that we haven't handled the new `Bold` tag:

```
# let color_to_int = function
    | Basic basic_color -> basic_color_to_int basic_color
    | RGB (r,g,b) -> 16 + b + g * 6 + r * 36
    | Gray i -> 232 + i;;
Lines 1-4, characters 20-24:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Bold _
val color_to_int : color -> int = <fun>
```

Fixing this now leads us to the correct implementation:

```
# let color_to_int = function
    | Basic basic_color -> basic_color_to_int basic_color
    | Bold  basic_color -> 8 + 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>
```

As we've seen, the type errors identified the things that needed to be fixed to complete the refactoring of the code. This is fantastically useful, but for it to work well and reliably, you need to write your code in a way that maximizes the compiler's chances of helping you find the bugs. To this end, a useful rule of thumb is to avoid catch-all cases in pattern matches.

Here's an example that illustrates how catch-all cases interact with exhaustion checks. Imagine we wanted a version of `color_to_int` that works on older terminals by rendering the first 16 colors (the eight `basic_colors` in regular and bold) in the normal way, but renders everything else as white. We might have written the function as follows.

```
# let oldschool_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
    | _ -> basic_color_to_int White;;
val oldschool_color_to_int : color -> int = <fun>
```

If we then applied the same fix we did above, we would have ended up with this.

```
# let oldschool_color_to_int = function
    | Basic basic_color -> basic_color_to_int basic_color
    | _ -> basic_color_to_int White;;
val oldschool_color_to_int : color -> int = <fun>
```

Because of the catch-all case, we'll no longer be warned about missing the `Bold` case. That's why you should beware of catch-all cases: they suppress exhaustiveness checking.

7.2 Combining Records and Variants

The term *algebraic data types* is often used to describe a collection of types that includes variants, records, and tuples. Algebraic data types act as a peculiarly useful and powerful language for describing data. At the heart of their utility is the fact that they combine two different kinds of types: *product types*, like tuples and records, which combine multiple different types together and are mathematically similar to Cartesian products; and *sum types*, like variants, which let you combine multiple different possibilities into one type, and are mathematically similar to disjoint unions.

Algebraic data types gain much of their power from the ability to construct layered combinations of sums and products. Let's see what we can achieve with this by reiterating the `Log_entry` message type that was described in Chapter 6 (Records).

```
module Time_ns = Core.Time_ns
module Log_entry = struct
  type t =
    { session_id: string;
      time: Time_ns.t;
      important: bool;
      message: string;
    }
end
```

This record type combines multiple pieces of data into a single value. In particular, a single `Log_entry.t` has a `session_id` *and* a `time` *and* an `important` flag *and* a `message`. More generally, you can think of record types as conjunctions. Variants, on the other hand, are disjunctions, letting you represent multiple possibilities. To construct an example of where this is useful, we'll first write out the other message types that came along-side `Log_entry`.

```
module Heartbeat = struct
  type t =
    { session_id: string;
      time: Time_ns.t;
      status_message: string;
    }
end
module Logon = struct
  type t =
    { session_id: string;
      time: Time_ns.t;
      user: string;
      credentials: string;
    }
end
```

A variant comes in handy when we want to represent values that could be any of these three types. The `client_message` type below lets you do just that.

```
type client_message = | Logon of Logon.t
                      | Heartbeat of Heartbeat.t
                      | Log_entry of Log_entry.t
```

In particular, a `client_message` is a `Logon` *or* a `Heartbeat` *or* a `Log_entry`. If we want to write code that processes messages generically, rather than code specialized to a fixed message type, we need something like `client_message` to act as one overarching type for the different possible messages. We can then match on the `client_message` to determine the type of the particular message being handled.

You can increase the precision of your types by using variants to represent differences between different cases, and records to represent shared structure. Consider the following function that takes a list of `client_messages` and returns all messages generated by a given user. The code in question is implemented by folding over the list of messages, where the accumulator is a pair of:

- The set of session identifiers for the user that have been seen thus far
- The set of messages so far that are associated with the user

Here's the concrete code:

```
# let messages_for_user user messages =
    let (user_messages,_) =
      List.fold messages ~init:([], Set.empty (module String))
        ~f:(fun ((messages,user_sessions) as acc) message ->
          match message with
          | Logon m ->
            if String.(m.user = user) then
              (message::messages, Set.add user_sessions m.session_id)
            else acc
          | Heartbeat _ | Log_entry _ ->
            let session_id = match message with
              | Logon     m -> m.session_id
              | Heartbeat m -> m.session_id
              | Log_entry m -> m.session_id
            in
            if Set.mem user_sessions session_id then
              (message::messages,user_sessions)
            else acc
        )
    in
    List.rev user_messages;;
val messages_for_user : string -> client_message list ->
    client_message list =
  <fun>
```

We take advantage of the fact that the type of the record `m` is known in the above code, so we don't have to qualify the record fields by the module they come from. *e.g.*, we write `m.user` instead of `m.Logon.user`.

One annoyance of the above code is that the logic for determining the session ID is somewhat repetitive, contemplating each of the possible message types (including the

Logon case, which isn't actually possible at that point in the code) and extracting the session ID in each case. This per-message-type handling seems unnecessary, since the session ID works the same way for all message types.

We can improve the code by refactoring our types to explicitly reflect the information that's shared between the different messages. The first step is to cut down the definitions of each per-message record to contain just the information unique to that record:

```
module Log_entry = struct
  type t = { important: bool;
             message: string;
           }
end
module Heartbeat = struct
  type t = { status_message: string; }
end
module Logon = struct
  type t = { user: string;
             credentials: string;
           }
end
```

We can then define a variant type that combines these types:

```
type details =
  | Logon of Logon.t
  | Heartbeat of Heartbeat.t
  | Log_entry of Log_entry.t
```

Separately, we need a record that contains the fields that are common across all messages:

```
module Common = struct
  type t = { session_id: string;
             time: Time_ns.t;
           }
end
```

A full message can then be represented as a pair of a `Common.t` and a `details`. Using this, we can rewrite our preceding example as follows. Note that we add extra type annotations so that OCaml recognizes the record fields correctly. Otherwise, we'd need to qualify them explicitly.

```
# let messages_for_user user (messages : (Common.t * details) list) =
    let (user_messages,_) =
      List.fold messages ~init:([],Set.empty (module String))
        ~f:(fun ((messages,user_sessions) as acc) ((common,details)
    as message) ->
          match details with
          | Logon m ->
            if String.(=) m.user user then
              (message::messages, Set.add user_sessions
    common.session_id)
            else acc
          | Heartbeat _ | Log_entry _ ->
            if Set.mem user_sessions common.session_id then
              (message::messages, user_sessions)
```

```
            else acc
        )
    in
    List.rev user_messages;;
val messages_for_user :
  string -> (Common.t * details) list -> (Common.t * details) list =
    <fun>
```

As you can see, the code for extracting the session ID has been replaced with the simple expression `common.session_id`.

In addition, this design allows us to grab the specific message and dispatch code to handle just that message type. In particular, while we use the type `Common.t * details` to represent an arbitrary message, we can use `Common.t * Logon.t` to represent a logon message. Thus, if we had functions for handling individual message types, we could write a dispatch function as follows:

```
# let handle_message server_state ((common:Common.t), details) =
    match details with
    | Log_entry m -> handle_log_entry server_state (common,m)
    | Logon     m -> handle_logon     server_state (common,m)
    | Heartbeat m -> handle_heartbeat server_state (common,m);;
val handle_message : server_state -> Common.t * details -> unit =
    <fun>
```

And it's explicit at the type level that `handle_log_entry` sees only `Log_entry` messages, `handle_logon` sees only `Logon` messages, etc.

7.2.1 Embedded Records

If we don't need to be able to pass the record types separately from the variant, then OCaml allows us to embed the records directly into the variant.

```
type details =
  | Logon     of { user: string; credentials: string; }
  | Heartbeat of { status_message: string; }
  | Log_entry of { important: bool; message: string; }
```

Even though the type is different, we can write `messages_for_user` in essentially the same way we did before.

```
# let messages_for_user user (messages : (Common.t * details) list) =
    let (user_messages,_) =
      List.fold messages ~init:([],Set.empty (module String))
        ~f:(fun ((messages,user_sessions) as acc) ((common,details)
    as message) ->
          match details with
          | Logon m ->
            if String.(=) m.user user then
              (message::messages, Set.add user_sessions
    common.session_id)
            else acc
          | Heartbeat _ | Log_entry _ ->
            if Set.mem user_sessions common.session_id then
              (message::messages, user_sessions)
```

```
            else acc
        )
    in
    List.rev user_messages;;
val messages_for_user :
  string -> (Common.t * details) list -> (Common.t * details) list =
    <fun>
```

Variants with inline records are both more concise and more efficient than having variants containing references to free-standing record types, because they don't require a separate allocated object for the contents of the variant.

The main downside is the obvious one, which is that an inline record can't be treated as its own free-standing object. And, as you can see below, OCaml will reject code that tries to do so.

```
# let get_logon_contents = function
    | Logon m -> Some m
    | _ -> None;;
Line 2, characters 23-24:
Error: This form is not allowed as the type of the inlined record
    could escape.
```

7.3 Variants and Recursive Data Structures

Another common application of variants is to represent tree-like recursive data structures. We'll show how this can be done by walking through the design of a simple Boolean expression language. Such a language can be useful anywhere you need to specify filters, which are used in everything from packet analyzers to mail clients.

An expression in this language will be defined by the variant `expr`, with one tag for each kind of expression we want to support:

```
type 'a expr =
  | Base  of 'a
  | Const of bool
  | And   of 'a expr list
  | Or    of 'a expr list
  | Not   of 'a expr
```

Note that the definition of the type `expr` is recursive, meaning that an `expr` may contain other `expr`s. Also, `expr` is parameterized by a polymorphic type `'a` which is used for specifying the type of the value that goes under the `Base` tag.

The purpose of each tag is pretty straightforward. `And`, `Or`, and `Not` are the basic operators for building up Boolean expressions, and `Const` lets you enter the constants `true` and `false`.

The `Base` tag is what allows you to tie the `expr` to your application, by letting you specify an element of some base predicate type, whose truth or falsehood is determined by your application. If you were writing a filter language for an email processor, your base predicates might specify the tests you would run against an email, as in the following example:

```
type mail_field = To | From | CC | Date | Subject
type mail_predicate = { field: mail_field;
                        contains: string }
```

Using the preceding code, we can construct a simple expression with `mail_predicate` as its base predicate:

```
# let test field contains = Base { field; contains };;
val test : mail_field -> string -> mail_predicate expr = <fun>
# And [ Or [ test To "doligez"; test CC "doligez" ];
        test Subject "runtime";
      ];;
- : mail_predicate expr =
And
 [Or
   [Base {field = To; contains = "doligez"};
    Base {field = CC; contains = "doligez"}];
  Base {field = Subject; contains = "runtime"}]
```

Being able to construct such expressions isn't enough; we also need to be able to evaluate them. Here's a function for doing just that:

```
# let rec eval expr base_eval =
    (* a shortcut, so we don't need to repeatedly pass [base_eval]
       explicitly to [eval] *)
    let eval' expr = eval expr base_eval in
    match expr with
    | Base  base  -> base_eval base
    | Const bool  -> bool
    | And   exprs -> List.for_all exprs ~f:eval'
    | Or    exprs -> List.exists  exprs ~f:eval'
    | Not   expr  -> not (eval' expr);;
val eval : 'a expr -> ('a -> bool) -> bool = <fun>
```

The structure of the code is pretty straightforward—we're just pattern matching over the structure of the data, doing the appropriate calculation based on which tag we see. To use this evaluator on a concrete example, we just need to write the `base_eval` function, which is capable of evaluating a base predicate.

Another useful operation on expressions is *simplification*, which is the process of taking a boolean expression and reducing it to an equivalent one that is smaller. First, we'll build a few simplifying construction functions that mirror the tags of an `expr`.

The `and_` function below does a few things:

- Reduces the entire expression to the constant `false` if any of the arms of the `And` are themselves `false`.
- Drops any arms of the `And` that there the constant `true`.
- Drops the `And` if it only has one arm.
- If the `And` has no arms, then reduces it to `Const true`.

The code is below.

```
# let and_ l =
    if List.exists l ~f:(function Const false -> true | _ -> false)
    then Const false
```

```
    else
      match List.filter l ~f:(function Const true -> false | _ ->
    true) with
      | [] -> Const true
      | [ x ] -> x
      | l -> And l;;
val and_ : 'a expr list -> 'a expr = <fun>
```

`Or` is the dual of `And`, and as you can see, the code for `or_` follows a similar pattern as that for `and_`, mostly reversing the role of `true` and `false`.

```
# let or_ l =
    if List.exists l ~f:(function Const true -> true | _ -> false)
    then Const true
    else
      match List.filter l ~f:(function Const false -> false | _ ->
    true) with
      | [] -> Const false
      | [x] -> x
      | l -> Or l;;
val or_ : 'a expr list -> 'a expr = <fun>
```

Finally, `not_` just has special handling for constants, applying the ordinary boolean negation function to them.

```
# let not_ = function
    | Const b -> Const (not b)
    | e -> Not e;;
val not_ : 'a expr -> 'a expr = <fun>
```

We can now write a simplification routine that is based on the preceding functions. Note that this function is recursive, in that it applies all of these simplifications in a bottom-up way across the entire expression.

```
# let rec simplify = function
    | Base _ | Const _ as x -> x
    | And l -> and_ (List.map ~f:simplify l)
    | Or l  -> or_  (List.map ~f:simplify l)
    | Not e -> not_ (simplify e);;
val simplify : 'a expr -> 'a expr = <fun>
```

We can now apply this to a Boolean expression and see how good a job it does at simplifying it.

```
# simplify (Not (And [ Or [Base "it's snowing"; Const true];
  Base "it's raining"]));;
- : string expr = Not (Base "it's raining")
```

Here, it correctly converted the `Or` branch to `Const true` and then eliminated the `And` entirely, since the `And` then had only one nontrivial component.

There are some simplifications it misses, however. In particular, see what happens if we add a double negation in.

```
# simplify (Not (And [ Or [Base "it's snowing"; Const true];
  Not (Not (Base "it's raining"))]));;
- : string expr = Not (Not (Not (Base "it's raining")))
```

It fails to remove the double negation, and it's easy to see why. The `not_` function has a catch-all case, so it ignores everything but the one case it explicitly considers, that of the negation of a constant. Catch-all cases are generally a bad idea, and if we make the code more explicit, we see that the missing of the double negation is more obvious:

```
# let not_ = function
    | Const b -> Const (not b)
    | (Base _ | And _ | Or _ | Not _) as e -> Not e;;
val not_ : 'a expr -> 'a expr = <fun>
```

We can of course fix this by simply adding an explicit case for double negation:

```
# let not_ = function
    | Const b -> Const (not b)
    | Not e -> e
    | (Base _ | And _ | Or _ ) as e -> Not e;;
val not_ : 'a expr -> 'a expr = <fun>
```

The example of a Boolean expression language is more than a toy. There's a module very much in this spirit in `Core` called `Blang` (short for "Boolean language"), and it gets a lot of practical use in a variety of applications. The simplification algorithm in particular is useful when you want to use it to specialize the evaluation of expressions for which the evaluation of some of the base predicates is already known.

More generally, using variants to build recursive data structures is a common technique, and shows up everywhere from designing little languages to building complex data structures.

7.4 Polymorphic Variants

In addition to the ordinary variants we've seen so far, OCaml also supports so-called *polymorphic variants*. As we'll see, polymorphic variants are more flexible and syntactically more lightweight than ordinary variants, but that extra power comes at a cost.

Syntactically, polymorphic variants are distinguished from ordinary variants by the leading backtick. And unlike ordinary variants, polymorphic variants can be used without an explicit type declaration:

```
# let three = `Int 3;;
val three : [> `Int of int ] = `Int 3
# let four = `Float 4.;;
val four : [> `Float of float ] = `Float 4.
# let nan = `Not_a_number;;
val nan : [> `Not_a_number ] = `Not_a_number
# [three; four; nan];;
- : [> `Float of float | `Int of int | `Not_a_number ] list =
[`Int 3; `Float 4.; `Not_a_number]
```

As you can see, polymorphic variant types are inferred automatically, and when we combine variants with different tags, the compiler infers a new type that knows about

all of those tags. Note that in the preceding example, the tag name (e.g., `Int) matches the type name (int). This is a common convention in OCaml.

The type system will complain if it sees incompatible uses of the same tag:

```
# let five = `Int "five";;
val five : [> `Int of string ] = `Int "five"
# [three; four; five];;
Line 1, characters 15-19:
Error: This expression has type [> `Int of string ]
       but an expression was expected of type
         [> `Float of float | `Int of int ]
       Types for tag `Int are incompatible
```

The > at the beginning of the variant types above is critical because it marks the types as being open to combination with other variant types. We can read the type [> `Float of float | `Int of int] as describing a variant whose tags include `Float of float and `Int of int, but may include more tags as well. In other words, you can roughly translate > to mean: "these tags or more."

OCaml will in some cases infer a variant type with <, to indicate "these tags or less," as in the following example:

```
# let is_positive = function
    | `Int   x -> x > 0
    | `Float x -> Float.(x > 0.);;
val is_positive : [< `Float of float | `Int of int ] -> bool = <fun>
```

The < is there because is_positive has no way of dealing with values that have tags other than `Float of float or `Int of int, but can handle types that have either or both of those two tags.

We can think of these < and > markers as indications of upper and lower bounds on the tags involved. If the same set of tags are both an upper and a lower bound, we end up with an *exact* polymorphic variant type, which has neither marker. For example:

```
# let exact = List.filter ~f:is_positive [three;four];;
val exact : [ `Float of float | `Int of int ] list = [`Int 3; `Float
    4.]
```

Perhaps surprisingly, we can also create polymorphic variant types that have distinct upper and lower bounds. Note that Ok and Error in the following example come from the Result.t type from Base.

```
# let is_positive = function
    | `Int   x -> Ok (x > 0)
    | `Float x -> Ok Float.O.(x > 0.)
    | `Not_a_number -> Error "not a number";;
val is_positive :
  [< `Float of float | `Int of int | `Not_a_number ] -> (bool, string)
    result =
  <fun>
# List.filter [three; four] ~f:(fun x ->
  match is_positive x with Error _ -> false | Ok b -> b);;
- : [< `Float of float | `Int of int | `Not_a_number > `Float `Int ]
    list =
[`Int 3; `Float 4.]
```

Here, the inferred type states that the tags can be no more than `` `Float ``, `` `Int ``, and `` `Not_a_number ``, and must contain at least `` `Float `` and `` `Int ``. As you can already start to see, polymorphic variants can lead to fairly complex inferred types.

Polymorphic Variants and Catch-All Cases

As we saw with the definition of `is_positive`, a `match` expression can lead to the inference of an upper bound on a variant type, limiting the possible tags to those that can be handled by the match. If we add a catch-all case to our `match` expression, we end up with a type with a lower bound.

```
# let is_positive_permissive = function
    | `Int   x -> Ok Int.(x > 0)
    | `Float x -> Ok Float.(x > 0.)
    | _ -> Error "Unknown number type";;
val is_positive_permissive :
  [> `Float of float | `Int of int ] -> (bool, string) result = <fun>
# is_positive_permissive (`Int 0);;
- : (bool, string) result = Ok false
# is_positive_permissive (`Ratio (3,4));;
- : (bool, string) result = Error "Unknown number type"
```

Catch-all cases are error-prone even with ordinary variants, but they are especially so with polymorphic variants. That's because you have no way of bounding what tags your function might have to deal with. Such code is particularly vulnerable to typos. For instance, if code that uses `is_positive_permissive` passes in `Float` misspelled as `Floot`, the erroneous code will compile without complaint.

```
# is_positive_permissive (`Floot 3.5);;
- : (bool, string) result = Error "Unknown number type"
```

With ordinary variants, such a typo would have been caught as an unknown tag. As a general matter, one should be wary about mixing catch-all cases and polymorphic variants.

7.4.1 Example: Terminal Colors Redux

To see how to use polymorphic variants in practice, we'll return to terminal colors. Imagine that we have a new terminal type that adds yet more colors, say, by adding an alpha channel so you can specify translucent colors. We could model this extended set of colors as follows, using an ordinary variant:

```
type extended_color =
  | Basic of basic_color * weight  (* basic colors, regular and bold
    *)
  | RGB   of int * int * int       (* 6x6x6 color space *)
  | Gray  of int                   (* 24 grayscale levels *)
  | RGBA  of int * int * int * int (* 6x6x6x6 color space *)
```

We want to write a function `extended_color_to_int` that works like `color_to_int` for all of the old kinds of colors, with new logic only for handling colors that include an alpha channel. One might try to write such a function as follows.

```
# let extended_color_to_int = function
    | RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
    | (Basic _ | RGB _ | Gray _) as color -> color_to_int color;;
Line 3, characters 59-64:
Error: This expression has type extended_color
       but an expression was expected of type color
```

The code looks reasonable enough, but it leads to a type error because `extended_color` and `color` are in the compiler's view distinct and unrelated types. The compiler doesn't, for example, recognize any equality between the `Basic` tag in the two types.

What we want to do is to share tags between two different variant types, and polymorphic variants let us do this in a natural way. First, let's rewrite `basic_color_to_int` and `color_to_int` using polymorphic variants. The translation here is pretty straightforward:

```
# 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 :
  [< `Black | `Blue | `Cyan | `Green | `Magenta | `Red | `White |
    `Yellow ] ->
  int = <fun>
# 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 :
  [< `Basic of
       [< `Black
        | `Blue
        | `Cyan
        | `Green
        | `Magenta
        | `Red
        | `White
        | `Yellow ] *
       [< `Bold | `Regular ]
   | `Gray of int
   | `RGB of int * int * int ] ->
  int = <fun>
```

Now we can try writing `extended_color_to_int`. The key issue with this code is that `extended_color_to_int` needs to invoke `color_to_int` with a narrower type, i.e., one that includes fewer tags. Written properly, this narrowing can be done via a pattern match. In particular, in the following code, the type of the variable `color` includes only the tags `` `Basic ``, `` `RGB ``, and `` `Gray ``, and not `` `RGBA ``:

```
# let extended_color_to_int = function
    | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
    | (`Basic _ | `RGB _ | `Gray _) as color -> color_to_int color;;
val extended_color_to_int :
  [< `Basic of
```

```
        [< `Black
         | `Blue
         | `Cyan
         | `Green
         | `Magenta
         | `Red
         | `White
         | `Yellow ] *
        [< `Bold | `Regular ]
    | `Gray of int
    | `RGB of int * int * int
    | `RGBA of int * int * int * int ] ->
  int = <fun>
```

The preceding code is more delicately balanced than one might imagine. In particular, if we use a catch-all case instead of an explicit enumeration of the cases, the type is no longer narrowed, and so compilation fails:

```
# let extended_color_to_int = function
    | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
    | color -> color_to_int color;;
Line 3, characters 29-34:
Error: This expression has type [> `RGBA of int * int * int * int ]
       but an expression was expected of type
         [< `Basic of
              [< `Black
               | `Blue
               | `Cyan
               | `Green
               | `Magenta
               | `Red
               | `White
               | `Yellow ] *
              [< `Bold | `Regular ]
          | `Gray of int
          | `RGB of int * int * int ]
       The second variant type does not allow tag(s) `RGBA
```

Let's consider how we might turn our code into a proper library with an implementation in an `ml` file and an interface in a separate `mli`, as we saw in Chapter 5 (Files, Modules, and Programs). Let's start with the `mli`.

```
open Base

type basic_color =
  [ `Black   | `Blue | `Cyan  | `Green
  | `Magenta | `Red  | `White | `Yellow ]

type color =
  [ `Basic of basic_color * [ `Bold | `Regular ]
  | `Gray of int
  | `RGB  of int * int * int ]

type extended_color =
  [ color
  | `RGBA of int * int * int * int ]
```

```
val color_to_int          : color -> int
val extended_color_to_int : extended_color -> int
```

Here, `extended_color` is defined as an explicit extension of `color`. Also, notice that we defined all of these types as exact variants. We can implement this library as follows.

```
open Base

type basic_color =
  [ `Black   | `Blue | `Cyan  | `Green
  | `Magenta | `Red  | `White | `Yellow ]

type color =
  [ `Basic of basic_color * [ `Bold | `Regular ]
  | `Gray of int
  | `RGB  of int * int * int ]

type extended_color =
  [ color
  | `RGBA of int * int * int * int ]

let basic_color_to_int = function
  | `Black -> 0 | `Red     -> 1 | `Green -> 2 | `Yellow -> 3
  | `Blue  -> 4 | `Magenta -> 5 | `Cyan  -> 6 | `White  -> 7

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

let extended_color_to_int = function
  | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
  | `Grey x -> 2000 + x
  | (`Basic _ | `RGB _ | `Gray _) as color -> color_to_int color
```

In the preceding code, we did something funny to the definition of `extended_color_to_int` that highlights some of the downsides of polymorphic variants. In particular, we added some special-case handling for the color gray, rather than using `color_to_int`. Unfortunately, we misspelled `Gray` as `Grey`. This is exactly the kind of error that the compiler would catch with ordinary variants, but with polymorphic variants, this compiles without issue. All that happened was that the compiler inferred a wider type for `extended_color_to_int`, which happens to be compatible with the narrower type that was listed in the `mli`. As a result, this library builds without error.

```
$ dune build @all
```

If we add an explicit type annotation to the code itself (rather than just in the `mli`), then the compiler has enough information to warn us:

```
let extended_color_to_int : extended_color -> int = function
  | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
```

```
  | `Grey x -> 2000 + x
  | (`Basic _ | `RGB _ | `Gray _) as color -> color_to_int color
```

In particular, the compiler will complain that the `Grey case is unused:

```
$ dune build @all
File "terminal_color.ml", line 30, characters 4-11:
30 |   | `Grey x -> 2000 + x
         ^^^^^^^
Error: This pattern matches values of type [? `Grey of 'a ]
       but a pattern was expected which matches values of type
     extended_color
       The second variant type does not allow tag(s) `Grey
[1]
```

Once we have type definitions at our disposal, we can revisit the question of how we write the pattern match that narrows the type. In particular, we can explicitly use the type name as part of the pattern match, by prefixing it with a #:

```
let extended_color_to_int : extended_color -> int = function
  | `RGBA (r,g,b,a) -> 256 + a + b * 6 + g * 36 + r * 216
  | #color as color -> color_to_int color
```

This is useful when you want to narrow down to a type whose definition is long, and you don't want the verbosity of writing the tags down explicitly in the match.

7.4.2 When to Use Polymorphic Variants

At first glance, polymorphic variants look like a strict improvement over ordinary variants. You can do everything that ordinary variants can do, plus it's more flexible and more concise. What's not to like?

In reality, regular variants are the more pragmatic choice most of the time. That's because the flexibility of polymorphic variants comes at a price. Here are some of the downsides:

Complexity The typing rules for polymorphic variants are a lot more complicated than they are for regular variants. This means that heavy use of polymorphic variants can leave you scratching your head trying to figure out why a given piece of code did or didn't compile. It can also lead to absurdly long and hard to decode error messages. Indeed, concision at the value level is often balanced out by more verbosity at the type level.

Error-finding Polymorphic variants are type-safe, but the typing discipline that they impose is, by dint of its flexibility, less likely to catch bugs in your program.

Efficiency This isn't a huge effect, but polymorphic variants are somewhat heavier than regular variants, and OCaml can't generate code for matching on polymorphic variants that is quite as efficient as what it generated for regular variants.

All that said, polymorphic variants are still a useful and powerful feature, but it's worth understanding their limitations and how to use them sensibly and modestly.

Probably the safest and most common use case for polymorphic variants is where

ordinary variants would be sufficient but are syntactically too heavyweight. For example, you often want to create a variant type for encoding the inputs or outputs to a function, where it's not worth declaring a separate type for it. Polymorphic variants are very useful here, and as long as there are type annotations that constrain these to have explicit, exact types, this tends to work well.

Variants are most problematic exactly where you take full advantage of their power; in particular, when you take advantage of the ability of polymorphic variant types to overlap in the tags they support. This ties into OCaml's support for subtyping. As we'll discuss further when we cover objects in Chapter 13 (Objects), subtyping brings in a lot of complexity, and most of the time, that's complexity you want to avoid.

8 Error Handling

Nobody likes dealing with errors. It's tedious, it's easy to get wrong, and it's usually just not as fun as thinking about how your program is going to succeed. But error handling is important, and however much you don't like thinking about it, having your software fail due to poor error handling is worse.

Thankfully, OCaml has powerful tools for handling errors reliably and with a minimum of pain. In this chapter we'll discuss some of the different approaches in OCaml to handling errors, and give some advice on how to design interfaces that make error handling easier.

We'll start by describing the two basic approaches for reporting errors in OCaml: error-aware return types and exceptions.

8.1 Error-Aware Return Types

The best way in OCaml to signal an error is to include that error in your return value. Consider the type of the `find` function in the `List` module:

```
# open Base;;
# List.find;;
- : 'a list -> f:('a -> bool) -> 'a option = <fun>
```

The option in the return type indicates that the function may not succeed in finding a suitable element:

```
# List.find [1;2;3] ~f:(fun x -> x >= 2);;
- : int option = Some 2
# List.find [1;2;3] ~f:(fun x -> x >= 10);;
- : int option = None
```

Including errors in the return values of your functions requires the caller to handle the error explicitly, allowing the caller to make the choice of whether to recover from the error or propagate it onward.

Consider the `compute_bounds` function below, which takes a list and a comparison function and returns upper and lower bounds for the list by finding the smallest and largest element on the list. `List.hd` and `List.last`, which return `None` when they encounter an empty list, are used to extract the largest and smallest element of the list:

```
# let compute_bounds ~compare list =
    let sorted = List.sort ~compare list in
```

```
    match List.hd sorted, List.last sorted with
    | None,_ | _, None -> None
    | Some x, Some y -> Some (x,y);;
val compute_bounds : compare:('a -> 'a -> int) -> 'a list -> ('a * 'a)
    option =
  <fun>
```

The `match` expression is used to handle the error cases, propagating a `None` in `hd` or `last` into the return value of `compute_bounds`.

On the other hand, in the `find_mismatches` that follows, errors encountered during the computation do not propagate to the return value of the function. `find_mismatches` takes two hash tables as arguments and searches for keys that have different data in one table than in the other. As such, the failure to find a key in one table isn't a failure of any sort:

```
# let find_mismatches table1 table2 =
    Hashtbl.fold table1 ~init:[] ~f:(fun ~key ~data mismatches ->
      match Hashtbl.find table2 key with
      | Some data' when data' <> data -> key :: mismatches
      | _ -> mismatches
    );;
val find_mismatches :
  ('a, int) Hashtbl.Poly.t -> ('a, int) Hashtbl.Poly.t -> 'a list =
    <fun>
```

The use of options to encode errors underlines the fact that it's not clear whether a particular outcome, like not finding something on a list, is an error or is just another valid outcome. This depends on the larger context of your program, and thus is not something that a general-purpose library can know in advance. One of the advantages of error-aware return types is that they work well in both situations.

8.1.1 Encoding Errors with Result

Options aren't always a sufficiently expressive way to report errors. Specifically, when you encode an error as `None`, there's nowhere to say anything about the nature of the error.

`Result.t` is meant to address this deficiency. The type is defined as follows:

```
module Result : sig
   type ('a,'b) t = | Ok of 'a
                    | Error of 'b
end
```

A `Result.t` is essentially an option augmented with the ability to store other information in the error case. Like `Some` and `None` for options, the constructors `Ok` and `Error` are available at the toplevel. As such, we can write:

```
# [ Ok 3; Error "abject failure"; Ok 4 ];;
- : (int, string) result list = [Ok 3; Error "abject failure"; Ok 4]
```

without first opening the `Result` module.

8.1.2 Error and Or_error

`Result.t` gives you complete freedom to choose the type of value you use to represent errors, but it's often useful to standardize on an error type. Among other things, this makes it easier to write utility functions to automate common error handling patterns.

But which type to choose? Is it better to represent errors as strings? Some more structured representation like XML? Or something else entirely?

Base's answer to this question is the `Error.t` type. You can, for example, construct one from a string.

```
# Error.of_string "something went wrong";;
- : Error.t = something went wrong
```

An `Or_error.t` is simply a `Result.t` with the error case specialized to the `Error.t` type. Here's an example.

```
# Error (Error.of_string "failed!");;
- : ('a, Error.t) result = Error failed!
```

The `Or_error` module provides a bunch of useful operators for constructing errors. For example, `Or_error.try_with` can be used for catching exceptions from a computation.

```
# let float_of_string s =
    Or_error.try_with (fun () -> Float.of_string s);;
val float_of_string : string -> float Or_error.t = <fun>
# float_of_string "3.34";;
- : float Or_error.t = Base__.Result.Ok 3.34
# float_of_string "a.bc";;
- : float Or_error.t =
Base__.Result.Error (Invalid_argument "Float.of_string a.bc")
```

Perhaps the most common way to create `Error.t`s is using *s-expressions*. An s-expression is a balanced parenthetical expression where the leaves of the expressions are strings. Here's a simple example:

```
(This (is an) (s expression))
```

S-expressions are supported by the Sexplib package that is distributed with Base and is the most common serialization format used in Base. Indeed, most types in Base come with built-in s-expression converters.

```
# Error.create "Unexpected character" 'c' Char.sexp_of_t;;
- : Error.t = ("Unexpected character" c)
```

We're not restricted to doing this kind of error reporting with built-in types. As we'll discuss in more detail in Chapter 21 (Data Serialization with S-Expressions), Sexplib comes with a syntax extension that can autogenerate sexp converters for specific types. We can enable it in the toplevel with a `#require` statement enabling `ppx_jane`, which is a package that pulls in multiple different syntax extensions, including `ppx_sexp_value`, the one we need here. (Because of technical issues with the toplevel, we can't easily enable these syntax extensions individually.)

```
# #require "ppx_jane";;
```

```
# Error.t_of_sexp
    [%sexp ("List is too long",[1;2;3] : string * int list)];;
- : Error.t = ("List is too long" (1 2 3))
```

`Error` also supports operations for transforming errors. For example, it's often useful to augment an error with information about the context of the error or to combine multiple errors together. `Error.tag` and `Error.of_list` fulfill these roles:

```
# Error.tag
    (Error.of_list [ Error.of_string "Your tires were slashed";
                     Error.of_string "Your windshield was smashed" ])
    ~tag:"over the weekend";;
- : Error.t =
("over the weekend" "Your tires were slashed" "Your windshield was
    smashed")
```

A very common way of generating errors is the `%message` syntax extension, which provides a compact syntax for providing a string describing the error, along with further values represented as s-expressions. Here's an example.

```
# let a = "foo" and b = ("foo",[3;4]);;
val a : string = "foo"
val b : string * int list = ("foo", [3; 4])
# Or_error.error_s
    [%message "Something went wrong" (a:string) (b: string * int
    list)];;
- : 'a Or_error.t =
Base__.Result.Error ("Something went wrong" (a foo) (b (foo (3 4))))
```

This is the most common idiom for generating `Error.t`'s.

8.1.3 `bind` and Other Error Handling Idioms

As you write more error handling code in OCaml, you'll discover that certain patterns start to emerge. A number of these common patterns have been codified by functions in modules like `Option` and `Result`. One particularly useful pattern is built around the function `bind`, which is both an ordinary function and an infix operator `>>=`. Here's the definition of `bind` for options:

```
# let bind option ~f =
    match option with
    | None -> None
    | Some x -> f x;;
val bind : 'a option -> f:('a -> 'b option) -> 'b option = <fun>
```

As you can see, `bind None f` returns `None` without calling `f`, and `bind (Some x) ~f` returns `f x`. `bind` can be used as a way of sequencing together error-producing functions so that the first one to produce an error terminates the computation. Here's a rewrite of `compute_bounds` to use a nested series of `bind`s:

```
# let compute_bounds ~compare list =
    let sorted = List.sort ~compare list in
    Option.bind (List.hd sorted) ~f:(fun first ->
      Option.bind (List.last sorted) ~f:(fun last ->
```

```
        Some (first,last)));;
val compute_bounds : compare:('a -> 'a -> int) -> 'a list -> ('a * 'a)
    option =
  <fun>
```

The preceding code is a little bit hard to swallow, however, on a syntactic level. We can make it easier to read and drop some of the parentheses, by using the infix operator form of `bind`, which we get access to by locally opening `Option.Monad_infix`. The module is called `Monad_infix` because the `bind` operator is part of a subinterface called `Monad`, which we'll see again in Chapter 17 (Concurrent Programming with Async).

```
# let compute_bounds ~compare list =
    let open Option.Monad_infix in
    let sorted = List.sort ~compare list in
    List.hd sorted   >>= fun first ->
    List.last sorted >>= fun last  ->
    Some (first,last);;
val compute_bounds : compare:('a -> 'a -> int) -> 'a list -> ('a * 'a)
    option =
  <fun>
```

This use of `bind` isn't really materially better than the one we started with, and indeed, for small examples like this, direct matching of options is generally better than using `bind`. But for large, complex examples with many stages of error handling, the `bind` idiom becomes clearer and easier to manage.

Monads and `Let_syntax`

We can make this look a little bit more ordinary by using a syntax extension that's designed specifically for monadic binds, called `Let_syntax`. Here's what the above example looks like using this extension.

```
# #require "ppx_let";;
# let compute_bounds ~compare list =
    let open Option.Let_syntax in
    let sorted = List.sort ~compare list in
    let%bind first = List.hd sorted in
    let%bind last  = List.last sorted in
    Some (first,last);;
val compute_bounds : compare:('a -> 'a -> int) -> 'a list -> ('a * 'a)
    option =
  <fun>
```

Note that we needed a `#require` statement to enable the extension.

To understand what's going on here, you need to know that `let%bind x = some_expr in some_other_expr` is rewritten into `some_expr >>= fun x -> some_other_expr`.

The advantage of `Let_syntax` is that it makes monadic bind look more like a regular let-binding. This works nicely because you can think of the monadic bind in this case as a special form of let binding that has some built-in error handling semantics.

There are other useful idioms encoded in the functions in `Option`. One example is `Option.both`, which takes two optional values and produces a new optional pair

that is `None` if either of its arguments are `None`. Using `Option.both`, we can make `compute_bounds` even shorter:

```
# let compute_bounds ~compare list =
    let sorted = List.sort ~compare list in
    Option.both (List.hd sorted) (List.last sorted);;
val compute_bounds : compare:('a -> 'a -> int) -> 'a list -> ('a * 'a)
    option =
  <fun>
```

These error-handling functions are valuable because they let you express your error handling both explicitly and concisely. We've only discussed these functions in the context of the `Option` module, but more functionality of this kind can be found in the `Result` and `Or_error` modules.

8.2 Exceptions

Exceptions in OCaml are not that different from exceptions in many other languages, like Java, C#, and Python. Exceptions are a way to terminate a computation and report an error, while providing a mechanism to catch and handle (and possibly recover from) exceptions that are triggered by subcomputations.

You can trigger an exception by, for example, dividing an integer by zero:

```
# 3 / 0;;
Exception: Division_by_zero.
```

And an exception can terminate a computation even if it happens nested somewhere deep within it:

```
# List.map ~f:(fun x -> 100 / x) [1;3;0;4];;
Exception: Division_by_zero.
```

If we put a `printf` in the middle of the computation, we can see that `List.map` is interrupted partway through its execution, never getting to the end of the list:

```
# List.map ~f:(fun x -> Stdio.printf "%d\n%!" x; 100 / x) [1;3;0;4];;
1
3
0
Exception: Division_by_zero.
```

In addition to built-in exceptions like `Divide_by_zero`, OCaml lets you define your own:

```
# exception Key_not_found of string;;
exception Key_not_found of string
# raise (Key_not_found "a");;
Exception: Key_not_found("a").
```

Exceptions are ordinary values and can be manipulated just like other OCaml values:

```
# let exceptions = [ Division_by_zero; Key_not_found "b" ];;
val exceptions : exn list = [Division_by_zero; Key_not_found("b")]
# List.filter exceptions  ~f:(function
```

```
    | Key_not_found _ -> true
    | _ -> false);;
- : exn list = [Key_not_found("b")]
```

Exceptions are all of the same type, `exn`, which is itself something of a special case in the OCaml type system. It is similar to the variant types we encountered in Chapter 7 (Variants), except that it is *open*, meaning that it's not fully defined in any one place. In particular, new tags (specifically, new exceptions) can be added to it by different parts of the program. This is in contrast to ordinary variants, which are defined with a closed universe of available tags. One result of this is that you can never have an exhaustive match on an `exn`, since the full set of possible exceptions is not known.

The following function uses the `Key_not_found` exception we defined above to signal an error:

```
# let rec find_exn alist key = match alist with
    | [] -> raise (Key_not_found key)
    | (key',data) :: tl -> if String.(=) key key' then data else
    find_exn tl key;;
val find_exn : (string * 'a) list -> string -> 'a = <fun>
# let alist = [("a",1); ("b",2)];;
val alist : (string * int) list = [("a", 1); ("b", 2)]
# find_exn alist "a";;
- : int = 1
# find_exn alist "c";;
Exception: Key_not_found("c").
```

Note that we named the function `find_exn` to warn the user that the function routinely throws exceptions, a convention that is used heavily in Base.

In the preceding example, `raise` throws the exception, thus terminating the computation. The type of raise is a bit surprising when you first see it: [raise}{.idx}

```
# raise;;
- : exn -> 'a = <fun>
```

The return type of `'a` makes it look like `raise` manufactures a value to return that is completely unconstrained in its type. That seems impossible, and it is. Really, `raise` has a return type of `'a` because it never returns at all. This behavior isn't restricted to functions like `raise` that terminate by throwing exceptions. Here's another example of a function that doesn't return a value:

```
# let rec forever () = forever ();;
val forever : unit -> 'a = <fun>
```

`forever` doesn't return a value for a different reason: it's an infinite loop.

This all matters because it means that the return type of `raise` can be whatever it needs to be to fit into the context it is called in. Thus, the type system will let us throw an exception anywhere in a program.

Declaring Exceptions Using `[@@deriving sexp]`

OCaml can't always generate a useful textual representation of an exception. For example:

```
# type 'a bounds = { lower: 'a; upper: 'a };;
```

```
type 'a bounds = { lower : 'a; upper : 'a; }
# exception Crossed_bounds of int bounds;;
exception Crossed_bounds of int bounds
# Crossed_bounds { lower=10; upper=0 };;
- : exn = Crossed_bounds(_)
```

But if we declare the exception (and the types it depends on) using `[@@deriving sexp]`, we'll get something with more information:

```
# type 'a bounds = { lower: 'a; upper: 'a } [@@deriving sexp];;
type 'a bounds = { lower : 'a; upper : 'a; }
val bounds_of_sexp : (Sexp.t -> 'a) -> Sexp.t -> 'a bounds = <fun>
val sexp_of_bounds : ('a -> Sexp.t) -> 'a bounds -> Sexp.t = <fun>
# exception Crossed_bounds of int bounds [@@deriving sexp];;
exception Crossed_bounds of int bounds
# Crossed_bounds { lower=10; upper=0 };;
- : exn = (//toplevel//.Crossed_bounds ((lower 10) (upper 0)))
```

The period in front of `Crossed_bounds` is there because the representation generated by `[@@deriving sexp]` includes the full module path of the module where the exception in question is defined. In this case, the string `//toplevel//` is used to indicate that this was declared at the utop prompt, rather than in a module.

This is all part of the support for s-expressions provided by the Sexplib library and syntax extension, which is described in more detail in Chapter 21 (Data Serialization with S-Expressions).

8.2.1 Helper Functions for Throwing Exceptions

Base provides a number of helper functions to simplify the task of throwing exceptions. The simplest one is `failwith`, which could be defined as follows:

```
# let failwith msg = raise (Failure msg);;
val failwith : string -> 'a = <fun>
```

There are several other useful functions for raising exceptions, which can be found in the API documentation for the `Common` and `Exn` modules in Base.

Another important way of throwing an exception is the `assert` directive. `assert` is used for situations where a violation of the condition in question indicates a bug. Consider the following piece of code for zipping together two lists:

```
# let merge_lists xs ys ~f =
    if List.length xs <> List.length ys then None
    else
      let rec loop xs ys =
        match xs,ys with
        | [],[] -> []
        | x::xs, y::ys -> f x y :: loop xs ys
        | _ -> assert false
      in
      Some (loop xs ys);;
val merge_lists : 'a list -> 'b list -> f:('a -> 'b -> 'c) -> 'c list
    option =
```

```
  <fun>
# merge_lists [1;2;3] [-1;1;2] ~f:(+);;
- : int list option = Some [0; 3; 5]
# merge_lists [1;2;3] [-1;1] ~f:(+);;
- : int list option = None
```

Here we use `assert false`, which means that the `assert`, once reached, is guaranteed to trigger. In general, one can put an arbitrary condition in the assertion.

In this case, the `assert` can never be triggered because we have a check that makes sure that the lists are of the same length before we call `loop`. If we change the code so that we drop this test, then we can trigger the `assert`:

```
# let merge_lists xs ys ~f =
    let rec loop xs ys =
      match xs,ys with
      | [],[] -> []
      | x::xs, y::ys -> f x y :: loop xs ys
      | _ -> assert false
    in
    loop xs ys;;
val merge_lists : 'a list -> 'b list -> f:('a -> 'b -> 'c) -> 'c list =
    <fun>
# merge_lists [1;2;3] [-1] ~f:(+);;
Exception: "Assert_failure //toplevel//:6:14"
```

This shows what's special about `assert`: it captures the line number and character offset of the source location from which the assertion was made.

8.2.2 Exception Handlers

So far, we've only seen exceptions fully terminate the execution of a computation. But sometimes, we want a program to be able to respond to and recover from an exception. This is achieved through the use of *exception handlers*.

In OCaml, an exception handler is declared using a `try/with` expression. Here's the basic syntax.

```
try <expr> with
| <pat1> -> <expr1>
| <pat2> -> <expr2>
...
```

A `try/with` clause first evaluates its body, *`expr`*. If no exception is thrown, then the result of evaluating the body is what the entire `try/with` clause evaluates to.

But if the evaluation of the body throws an exception, then the exception will be fed to the pattern-match clauses following the `with`. If the exception matches a pattern, then we consider the exception caught, and the `try/with` clause evaluates to the expression on the right-hand side of the matching pattern.

Otherwise, the original exception continues up the stack of function calls, to be handled by the next outer exception handler. If the exception is never caught, it terminates the program.

8.2.3 Cleaning Up in the Presence of Exceptions

One headache with exceptions is that they can terminate your execution at unexpected places, leaving your program in an awkward state. Consider the following function for loading a file full of numerical data. This code parses data that matches a simple comma-separated file format, where each field is a floating point number. In this example we open `Stdio`, to get access to routines for reading from files.

```
# open Stdio;;
# let parse_line line =
    String.split_on_chars ~on:[','] line
    |> List.map ~f:Float.of_string;;
val parse_line : string -> float list = <fun>
# let load filename =
    let inc = In_channel.create filename in
    let data =
      In_channel.input_lines inc
      |> List.map ~f:parse_line
    in
    In_channel.close inc;
    data;;
val load : string -> float list list = <fun>
```

One problem with this code is that the parsing function can throw an exception if the file in question is malformed. Unfortunately, that means that the `In_channel.t` that was opened will never be closed, leading to a file-descriptor leak.

We can fix this using Base's `Exn.protect` function, which takes two arguments: a thunk `f`, which is the main body of the computation to be run; and a thunk `finally`, which is to be called when `f` exits, whether it exits normally or with an exception. This is similar to the `try/finally` construct available in many programming languages, but it is implemented in a library, rather than being a built-in primitive. Here's how it could be used to fix our `load` function:

```
# let load filename =
    let inc = In_channel.create filename in
    Exn.protect
      ~f:(fun () -> In_channel.input_lines inc |> List.map
    ~f:parse_line)
      ~finally:(fun () -> In_channel.close inc);;
val load : string -> float list list = <fun>
```

This is a common enough problem that `In_channel` has a function called `with_file` that automates this pattern:

```
# let load filename =
    In_channel.with_file filename ~f:(fun inc ->
      In_channel.input_lines inc |> List.map ~f:parse_line);;
val load : string -> float list list = <fun>
```

`In_channel.with_file` is built on top of `protect` so that it can clean up after itself in the presence of exceptions.

8.2.4 Catching Specific Exceptions

OCaml's exception-handling system allows you to tune your error-recovery logic to the particular error that was thrown. For example, `find_exn`, which we defined earlier in the chapter, throws `Key_not_found` when the element in question can't be found. Let's look at an example of how you could take advantage of this. In particular, consider the following function:

```
# let lookup_weight ~compute_weight alist key =
    try
      let data = find_exn alist key in
      compute_weight data
    with
    Key_not_found _ -> 0.;;
val lookup_weight :
  compute_weight:('a -> float) -> (string * 'a) list -> string ->
    float =
  <fun>
```

As you can see from the type, `lookup_weight` takes an association list, a key for looking up a corresponding value in that list, and a function for computing a floating-point weight from the looked-up value. If no value is found, then a weight of `0.` should be returned.

The use of exceptions in this code, however, presents some problems. In particular, what happens if `compute_weight` throws an exception? Ideally, `lookup_weight` should propagate that exception on, but if the exception happens to be `Key_not_found`, then that's not what will happen:

```
# lookup_weight ~compute_weight:(fun _ -> raise (Key_not_found "foo"))
  ["a",3; "b",4] "a";;
- : float = 0.
```

This kind of problem is hard to detect in advance because the type system doesn't tell you what exceptions a given function might throw. For this reason, it's usually best to avoid relying on the identity of the exception to determine the nature of a failure. A better approach is to narrow the scope of the exception handler, so that when it fires it's very clear what part of the code failed:

```
# let lookup_weight ~compute_weight alist key =
    match
      try Some (find_exn alist key)
      with _ -> None
    with
    | None -> 0.
    | Some data -> compute_weight data;;
val lookup_weight :
  compute_weight:('a -> float) -> (string * 'a) list -> string ->
    float =
  <fun>
```

This nesting of a `try` within a `match` expression is both awkward and involves some unnecessary computation (in particular, the allocation of the option). Happily, OCaml allows for exceptions to be caught by match expressions directly, which lets you write this more concisely as follows.

```
# let lookup_weight ~compute_weight alist key =
    match find_exn alist key with
    | exception _ -> 0.
    | data -> compute_weight data;;
val lookup_weight :
  compute_weight:('a -> float) -> (string * 'a) list -> string ->
    float =
  <fun>
```

Note that the `exception` keyword is used to mark the exception-handling cases.

Best of all is to avoid exceptions entirely, which we could do by using the exception-free function from Base, `List.Assoc.find`, instead:

```
# let lookup_weight ~compute_weight alist key =
    match List.Assoc.find ~equal:String.equal alist key with
    | None -> 0.
    | Some data -> compute_weight data;;
val lookup_weight :
  compute_weight:('a -> float) ->
  (string, 'a) Base.List.Assoc.t -> string -> float = <fun>
```

8.2.5 Backtraces

A big part of the value of exceptions is that they provide useful debugging information in the form of a stack backtrace. Consider the following simple program:

```
open Base
open Stdio
exception Empty_list

let list_max = function
  | [] -> raise Empty_list
  | hd :: tl -> List.fold tl ~init:hd ~f:(Int.max)

let () =
  printf "%d\n" (list_max [1;2;3]);
  printf "%d\n" (list_max [])
```

If we build and run this program, we'll get a stack backtrace that will provide some information about where the error occurred and the stack of function calls that were in place at the time of the error:

```
$ dune exec -- ./blow_up.exe
3
Fatal error: exception Dune__exe__Blow_up.Empty_list
Raised at Dune__exe__Blow_up.list_max in file "blow_up.ml", line 6,
    characters 10-26
Called from Dune__exe__Blow_up in file "blow_up.ml", line 11,
    characters 16-29
[2]
```

You can also capture a backtrace within your program by calling `Backtrace.Exn.most_recent`, which returns the backtrace of the most recently thrown exception. This is useful for reporting detailed information on errors that did not cause your program to fail.

This works well if you have backtraces enabled, but that isn't always the case. In fact, by default, OCaml has backtraces turned off, and even if you have them turned on at runtime, you can't get backtraces unless you have compiled with debugging symbols. Base reverses the default, so if you're linking in Base, you will have backtraces enabled by default.

Even using Base and compiling with debugging symbols, you can turn backtraces off via the `OCAMLRUNPARAM` environment variable, as shown below.

```
$ OCAMLRUNPARAM=b=0 dune exec -- ./blow_up.exe
3
Fatal error: exception Dune__exe__Blow_up.Empty_list
[2]
```

The resulting error message is considerably less informative. You can also turn backtraces off in your code by calling `Backtrace.Exn.set_recording false`.

There is a legitimate reason to run without backtraces: speed. OCaml's exceptions are fairly fast, but they're faster still if you disable backtraces. Here's a simple benchmark that shows the effect, using the `core_bench` package:

```
open Core
open Core_bench

exception Exit

let x = 0

type how_to_end = Ordinary | Raise | Raise_no_backtrace

let computation how_to_end =
  let x = 10 in
  let y = 40 in
  let _z = x + (y * y) in
  match how_to_end with
  | Ordinary -> ()
  | Raise -> raise Exit
  | Raise_no_backtrace -> raise_notrace Exit

let computation_with_handler how = try computation how with Exit -> ()

let () =
  [
    Bench.Test.create ~name:"simple computation" (fun () ->
        computation Ordinary);
    Bench.Test.create ~name:"computation w/handler" (fun () ->
        computation_with_handler Ordinary);
    Bench.Test.create ~name:"end with exn" (fun () ->
        computation_with_handler Raise);
    Bench.Test.create ~name:"end with exn notrace" (fun () ->
        computation_with_handler Raise_no_backtrace);
  ]
  |> Bench.make_command |> Command.run
```

We're testing four cases here:

- a simple computation with no exception,

- the same, but with an exception handler but no exception thrown,
- the same, but where an exception is thrown,
- and finally, the same, but where we throw an exception using `raise_notrace`, which is a version of `raise` which locally avoids the costs of keeping track of the backtrace.

Here are the results.

```
$ dune exec -- \
> ./exn_cost.exe -ascii -quota 1 -clear-columns time cycles
Estimated testing time 4s (4 benchmarks x 1s). Change using '-quota'.

  Name                    Time/Run   Cycls/Run
 ----------------------- ---------- -----------
  simple computation        1.84ns       3.66c
  computation w/handler     3.13ns       6.23c
  end with exn             27.96ns      55.69c
  end with exn notrace     11.69ns      23.28c
```

Note that we lose just a small number of cycles to setting up an exception handler, which means that an unused exception handler is quite cheap indeed. We lose a much bigger chunk, around 55 cycles, to actually raising an exception. If we explicitly raise an exception with no backtrace, it costs us about 25 cycles.

We can also disable backtraces, as we discussed, using `OCAMLRUNPARAM`. That changes the results a bit.

```
$ OCAMLRUNPARAM=b=0 dune exec -- \
> ./exn_cost.exe -ascii -quota 1 -clear-columns time cycles
Estimated testing time 4s (4 benchmarks x 1s). Change using '-quota'.

  Name                    Time/Run   Cycls/Run
 ----------------------- ---------- -----------
  simple computation        1.71ns       3.41c
  computation w/handler     3.04ns       6.05c
  end with exn             19.36ns      38.57c
  end with exn notrace     11.48ns      22.86c
```

The only significant change here is that raising an exception in the ordinary way becomes just a bit cheaper: 20 cycles instead of 55 cycles. But it's still not as fast as using `raise_notrace` explicitly.

Differences on this scale should only matter if you're using exceptions routinely as part of your flow control. That's not a common pattern, and when you do need it, it's better from a performance perspective to use `raise_notrace`. All of which is to say, you should almost always leave stack-traces on.

8.2.6 From Exceptions to Error-Aware Types and Back Again

Both exceptions and error-aware types are necessary parts of programming in OCaml. As such, you often need to move between these two worlds. Happily, Base comes with some useful helper functions to help you do just that. For example, given a piece of code that can throw an exception, you can capture that exception into an option as follows:

```
# let find alist key =
  Option.try_with (fun () -> find_exn alist key);;
val find : (string * 'a) list -> string -> 'a option = <fun>
# find ["a",1; "b",2] "c";;
- : int option = Base.Option.None
# find ["a",1; "b",2] "b";;
- : int option = Base.Option.Some 2
```

`Result` and `Or_error` have similar `try_with` functions. So, we could write:

```
# let find alist key =
  Or_error.try_with (fun () -> find_exn alist key);;
val find : (string * 'a) list -> string -> 'a Or_error.t = <fun>
# find ["a",1; "b",2] "c";;
- : int Or_error.t = Base__.Result.Error ("Key_not_found(\"c\")")
```

We can then reraise that exception:

```
# Or_error.ok_exn (find ["a",1; "b",2] "b");;
- : int = 2
# Or_error.ok_exn (find ["a",1; "b",2] "c");;
Exception: Key_not_found("c").
```

8.3 Choosing an Error-Handling Strategy

Given that OCaml supports both exceptions and error-aware return types, how do you choose between them? The key is to think about the trade-off between concision and explicitness.

Exceptions are more concise because they allow you to defer the job of error handling to some larger scope, and because they don't clutter up your types. But this concision comes at a cost: exceptions are all too easy to ignore. Error-aware return types, on the other hand, are fully manifest in your type definitions, making the errors that your code might generate explicit and impossible to ignore.

The right trade-off depends on your application. If you're writing a rough-and-ready program where getting it done quickly is key and failure is not that expensive, then using exceptions extensively may be the way to go. If, on the other hand, you're writing production software whose failure is costly, then you should probably lean in the direction of using error-aware return types.

To be clear, it doesn't make sense to avoid exceptions entirely. The maxim of "use exceptions for exceptional conditions" applies. If an error occurs sufficiently rarely, then throwing an exception is often the right behavior.

Also, for errors that are omnipresent, error-aware return types may be overkill. A good example is out-of-memory errors, which can occur anywhere, and so you'd need to use error-aware return types everywhere to capture those. Having every operation marked as one that might fail is no more explicit than having none of them marked.

In short, for errors that are a foreseeable and ordinary part of the execution of your production code and that are not omnipresent, error-aware return types are typically the right solution.

9 Imperative Programming

This chapter includes contributions from Jason Hickey.

Most of the code shown so far in this book, and indeed, most OCaml code in general, is *pure*. Pure code works without mutating the program's internal state, performing I/O, reading the clock, or in any other way interacting with changeable parts of the world. Thus, a pure function behaves like a mathematical function, always returning the same results when given the same inputs, and never affecting the world except insofar as it returns the value of its computation. *Imperative* code, on the other hand, operates by side effects that modify a program's internal state or interact with the outside world. An imperative function has a new effect, and potentially returns different results, every time it's called.

Pure code is the default in OCaml, and for good reason—it's generally easier to reason about, less error prone and more composable. But imperative code is of fundamental importance to any practical programming language, because real-world tasks require that you interact with the outside world, which is by its nature imperative. Imperative programming can also be important for performance. While pure code is quite efficient in OCaml, there are many algorithms that can only be implemented efficiently using imperative techniques.

OCaml offers a happy compromise here, making it easy and natural to program in a pure style, but also providing great support for imperative programming. This chapter will walk you through OCaml's imperative features, and help you use them to their fullest.

9.1 Example: Imperative Dictionaries

We'll start with the implementation of a simple imperative dictionary, i.e., a mutable mapping from keys to values. This is very much a toy implementation, and it's really not suitable for any real-world use. That's fine, since both `Base` and the standard library provide effective imperative dictionaries. There's more advice on using `Base`'s implementation in particular in Chapter 15 (Maps and Hash Tables).

The dictionary we'll describe now, like those in `Base` and the standard library, will be implemented as a hash table. In particular, we'll use an *open hashing* scheme, where

the hash table will be an array of buckets, each bucket containing a list of key/value pairs that have been hashed into that bucket.

Here's the signature we'll match, provided as an interface file, `dictionary.mli`. The type `('a, 'b) t` represents a dictionary with keys of type `'a` and data of type `'b`.

```
open Base

type ('a, 'b) t

val create
  :  hash:('a -> int)
  -> equal:('a -> 'a -> bool)
  -> ('a, 'b) t

val length : ('a, 'b) t -> int
val add : ('a, 'b) t -> key:'a -> data:'b -> unit
val find : ('a, 'b) t -> 'a -> 'b option
val iter : ('a, 'b) t -> f:(key:'a -> data:'b -> unit) -> unit
val remove : ('a, 'b) t -> 'a -> unit
```

This `mli` also includes a collection of helper functions whose purpose and behavior should be largely inferable from their names and type signatures. Note that the `create` function takes as its arguments functions for hashing keys and testing them for equality.

You might notice that some of the functions, like `add` and `iter`, return `unit`. This is unusual for functional code, but common for imperative functions whose primary purpose is to mutate some data structure, rather than to compute a value.

We'll now walk through the implementation (contained in the corresponding `ml` file, `dictionary.ml`) piece by piece, explaining different imperative constructs as they come up.

Our first step is to define the type of a dictionary as a record.

```
open Base

type ('a, 'b) t =
  { mutable length : int
  ; buckets : ('a * 'b) list array
  ; hash : 'a -> int
  ; equal : 'a -> 'a -> bool
  }
```

The first field, `length`, is declared as mutable. In OCaml, records are immutable by default, but individual fields are mutable when marked as such. The second field, `buckets`, is immutable but contains an array, which is itself a mutable data structure. The remaining fields contain the functions for hashing and equality checking.

Now we'll start putting together the basic functions for manipulating a dictionary:

```
let num_buckets = 17
let hash_bucket t key = t.hash key % num_buckets

let create ~hash ~equal =
  { length = 0
  ; buckets = Array.create ~len:num_buckets []
  ; hash
```

```
  ; equal
  }

let length t = t.length

let find t key =
  List.find_map
    t.buckets.(hash_bucket t key)
    ~f:(fun (key', data) ->
      if t.equal key' key then Some data else None)
```

Note that `num_buckets` is a constant, which means our bucket array is of fixed length. A practical implementation would need to be able to grow the array as the number of elements in the dictionary increases, but we'll omit this to simplify the presentation.

The function `hash_bucket` is used throughout the rest of the module to choose the position in the array that a given key should be stored at.

The other functions defined above are fairly straightforward:

`create` Creates an empty dictionary.

`length` Grabs the length from the corresponding record field, thus returning the number of entries stored in the dictionary.

`find` Looks for a matching key in the table and returns the corresponding value if found as an option.

Another important piece of imperative syntax shows up in `find`: we write `array.(index)` to grab a value from an array. `find` also uses `List.find_map`, which you can see the type of by typing it into the toplevel:

```
# open Base;;
# List.find_map;;
- : 'a list -> f:('a -> 'b option) -> 'b option = <fun>
```

`List.find_map` iterates over the elements of the list, calling `f` on each one until a `Some` is returned by `f`, at which point that value is returned. If `f` returns `None` on all values, then `None` is returned.

Now let's look at the implementation of `iter`:

```
let iter t ~f =
  for i = 0 to Array.length t.buckets - 1 do
    List.iter t.buckets.(i) ~f:(fun (key, data) -> f ~key ~data)
  done
```

`iter` is designed to walk over all the entries in the dictionary. In particular, `iter t ~f` will call `f` for each key/value pair in dictionary `t`. Note that `f` must return `unit`, since it is expected to work by side effect rather than by returning a value, and the overall `iter` function returns `unit` as well.

The code for `iter` uses two forms of iteration: a `for` loop to walk over the array of buckets; and within that loop a call to `List.iter` to walk over the values in a given bucket. We could have done the outer loop with a recursive function instead of a `for` loop, but `for` loops are syntactically convenient, and are more familiar and idiomatic in imperative contexts.

The following code is for adding and removing mappings from the dictionary:

```
let bucket_has_key t i key =
  List.exists t.buckets.(i) ~f:(fun (key', _) -> t.equal key' key)

let add t ~key ~data =
  let i = hash_bucket t key in
  let replace = bucket_has_key t i key in
  let filtered_bucket =
    if replace
    then
      List.filter t.buckets.(i) ~f:(fun (key', _) ->
          not (t.equal key' key))
    else t.buckets.(i)
  in
  t.buckets.(i) <- (key, data) :: filtered_bucket;
  if not replace then t.length <- t.length + 1

let remove t key =
  let i = hash_bucket t key in
  if bucket_has_key t i key
  then (
    let filtered_bucket =
      List.filter t.buckets.(i) ~f:(fun (key', _) ->
          not (t.equal key' key))
    in
    t.buckets.(i) <- filtered_bucket;
    t.length <- t.length - 1)
```

This preceding code is made more complicated by the fact that we need to detect whether we are overwriting or removing an existing binding, so we can decide whether `t.length` needs to be changed. The helper function `bucket_has_key` is used for this purpose.

Another piece of syntax shows up in both `add` and `remove`: the use of the `<-` operator to update elements of an array (`array.(i) <- expr`) and for updating a record field (`record.field <- expression`).

We also use `;`, the sequencing operator, to express a sequence of imperative actions. We could have done the same using `let` bindings:

```
let () = t.buckets.(i) <- (key, data) :: filtered_bucket in
if not replace then t.length <- t.length + 1
```

but `;` is more concise and idiomatic. More generally,

```
<expr1>;
<expr2>;
...
<exprN>
```

is equivalent to

```
let () = <expr1> in
let () = <expr2> in
...
<exprN>
```

When a sequence expression `expr1; expr2` is evaluated, `expr1` is evaluated first, and then `expr2`. The expression `expr1` should have type `unit` (though this is a warning

rather than a hard restriction. The `-strict-sequence` compiler flag makes this a hard restriction, which is generally a good idea), and the value of `expr2` is returned as the value of the entire sequence. For example, the sequence `print_string "hello world"; 1 + 2` first prints the string `"hello world"`, then returns the integer `3`.

Note also that we do all of the side-effecting operations at the very end of each function. This is good practice because it minimizes the chance that such operations will be interrupted with an exception, leaving the data structure in an inconsistent state.

9.2 Primitive Mutable Data

Now that we've looked at a complete example, let's take a more systematic look at imperative programming in OCaml. We encountered two different forms of mutable data above: records with mutable fields and arrays. We'll now discuss these in more detail, along with the other primitive forms of mutable data that are available in OCaml.

9.2.1 Array-Like Data

OCaml supports a number of array-like data structures; i.e., mutable integer-indexed containers that provide constant-time access to their elements. We'll discuss several of them in this section.

Ordinary arrays

The `array` type is used for general-purpose polymorphic arrays. The `Array` module has a variety of utility functions for interacting with arrays, including a number of mutating operations. These include `Array.set`, for setting an individual element, and `Array.blit`, for efficiently copying values from one range of indices to another.

Arrays also come with special syntax for retrieving an element from an array:

```
<array_expr>.(<index_expr>)
```

and for setting an element in an array:

```
<array_expr>.(<index_expr>) <- <value_expr>
```

Out-of-bounds accesses for arrays (and indeed for all the array-like data structures) will lead to an exception being thrown.

Array literals are written using `[|` and `|]` as delimiters. Thus, `[| 1; 2; 3 |]` is a literal integer array.

`bytes` and `strings`.

The strings we've encountered thus far are essentially byte arrays, and are most often used for textual data. You could imagine using a `char array` (a `char` represents an 8-bit character) for the same purpose, but strings are considerably more space-efficient; an `array` uses one 8-byte word on a 64-bit machine—to store a single entry, whereas strings use one byte per character.

Unlike arrays, though, strings are immutable, and sometimes, it's convenient to have a space-efficient, mutable array of bytes. Happily, OCaml has that, via the `bytes` type.

You can set individual characters using `Bytes.set`, and a value of type `bytes` can be converted to and from the `string` type.

```
# let b = Bytes.of_string "foobar";;
val b : bytes = "foobar"
# Bytes.set b 0 (Char.uppercase (Bytes.get b 0));;
- : unit = ()
# Bytes.to_string b;;
- : string = "Foobar"
```

Bigarrays

A `Bigarray.t` is a handle to a block of memory stored outside of the OCaml heap. These are mostly useful for interacting with C or Fortran libraries, and are discussed in Chapter 24 (Memory Representation of Values). Bigarrays too have their own getting and setting syntax:

```
<bigarray_expr>.{<index_expr>}
<bigarray_expr>.{<index_expr>} <- <value_expr>
```

9.2.2 Mutable Record and Object Fields and Ref Cells

As we've seen, records are immutable by default, but individual record fields can be declared as mutable. These mutable fields can be set using the `<-` operator, i.e., `record.field <- expr`.

As we'll see in Chapter 13 (Objects), fields of an object can similarly be declared as mutable, and can then be modified in much the same way as record fields.

Ref cells

Variables in OCaml are never mutable—they can refer to mutable data, but what the variable points to can't be changed. Sometimes, though, you want to do exactly what you would do with a mutable variable in another language: define a single, mutable value. In OCaml this is typically achieved using a `ref`, which is essentially a container with a single mutable polymorphic field.

The definition for the `ref` type is as follows:

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

The standard library defines the following operators for working with `ref`s.

`ref expr` Constructs a reference cell containing the value defined by the expression `expr`.

`!refcell` Returns the contents of the reference cell.

`refcell := expr` Replaces the contents of the reference cell.

You can see these in action:

```
# let x = ref 1;;
val x : int ref = {Base.Ref.contents = 1}
# !x;;
- : int = 1
# x := !x + 1;;
- : unit = ()
# !x;;
- : int = 2
```

The preceding are just ordinary OCaml functions, which could be defined as follows:

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

This reflects the fact that ref cells are really just a special case of mutable record fields.

9.2.3 Foreign Functions

Another source of imperative operations in OCaml is resources that come from interfacing with external libraries through OCaml's foreign function interface (FFI). The FFI opens OCaml up to imperative constructs that are exported by system calls or other external libraries. Many of these come built in, like access to the `write` system call or to the `clock`, while others come from user libraries. OCaml's FFI is discussed in more detail in Chapter 23 (Foreign Function Interface).

9.3 For and While Loops

OCaml provides support for traditional imperative looping constructs, in particular, `for` and `while` loops. Neither of these constructs is strictly necessary, since they can be simulated with recursive functions. Nonetheless, explicit `for` and `while` loops are both more concise and more idiomatic when programming imperatively.

The `for` loop is the simpler of the two. Indeed, we've already seen the `for` loop in action—the `iter` function in `Dictionary` is built using it. Here's a simple example of `for`. Note that we open the `Stdio` library to get access to the `printf` function.

```
# open Stdio;;
# for i = 0 to 3 do printf "i = %d\n" i done;;
i = 0
i = 1
i = 2
i = 3
- : unit = ()
```

As you can see, the upper and lower bounds are inclusive. We can also use `downto` to iterate in the other direction:

```
# for i = 3 downto 0 do printf "i = %d\n" i done;;
i = 3
i = 2
i = 1
i = 0
- : unit = ()
```

Note that the loop variable of a `for` loop, `i` in this case, is immutable in the scope of the loop and is also local to the loop, i.e., it can't be referenced outside of the loop.

OCaml also supports `while` loops, which include a condition and a body. The loop first evaluates the condition, and then, if it evaluates to true, evaluates the body and starts the loop again. Here's a simple example of a function for reversing an array in place:

```
# let rev_inplace ar =
    let i = ref 0 in
    let j = ref (Array.length ar - 1) in
    (* terminate when the upper and lower indices meet *)
    while !i < !j do
      (* swap the two elements *)
      let tmp = ar.(!i) in
      ar.(!i) <- ar.(!j);
      ar.(!j) <- tmp;
      (* bump the indices *)
      Int.incr i;
      Int.decr j
    done;;
val rev_inplace : 'a array -> unit = <fun>
# let nums = [|1;2;3;4;5|];;
val nums : int array = [|1; 2; 3; 4; 5|]
# rev_inplace nums;;
- : unit = ()
# nums;;
- : int array = [|5; 4; 3; 2; 1|]
```

In the preceding example, we used `Int.incr` and `Int.decr`, which are built-in functions for incrementing and decrementing an `int ref` by one, respectively.

9.4 Example: Doubly Linked Lists

Another common imperative data structure is the doubly linked list. Doubly linked lists can be traversed in both directions, and elements can be added and removed from the list in constant time. Core defines a doubly linked list (the module is called `Doubly_linked`), but we'll define our own linked list library as an illustration.

Here's the `mli` of the module we'll build:

```
open Base

type 'a t
type 'a element

(** Basic list operations *)
```

```
val create : unit -> 'a t
val is_empty : 'a t -> bool

(** Navigation using [element]s *)

val first : 'a t -> 'a element option
val next : 'a element -> 'a element option
val prev : 'a element -> 'a element option
val value : 'a element -> 'a

(** Whole-data-structure iteration *)

val iter : 'a t -> f:('a -> unit) -> unit
val find_el : 'a t -> f:('a -> bool) -> 'a element option

(** Mutation *)

val insert_first : 'a t -> 'a -> 'a element
val insert_after : 'a element -> 'a -> 'a element
val remove : 'a t -> 'a element -> unit
```

Note that there are two types defined here: `'a t`, the type of a list; and `'a element`, the type of an element. Elements act as pointers to the interior of a list and allow us to navigate the list and give us a point at which to apply mutating operations.

Now let's look at the implementation. We'll start by defining `'a element` and `'a t`:

```
open Base

type 'a element =
  { value : 'a
  ; mutable next : 'a element option
  ; mutable prev : 'a element option
  }

type 'a t = 'a element option ref
```

An `'a element` is a record containing the value to be stored in that node as well as optional (and mutable) fields pointing to the previous and next elements. At the beginning of the list, the `prev` field is `None`, and at the end of the list, the `next` field is `None`.

The type of the list itself, `'a t`, is a mutable reference to an optional `element`. This reference is `None` if the list is empty, and `Some` otherwise.

Now we can define a few basic functions that operate on lists and elements:

```
let create () = ref None
let is_empty t = Option.is_none !t
let value elt = elt.value
let first t = !t
let next elt = elt.next
let prev elt = elt.prev
```

These all follow relatively straightforwardly from our type definitions.

Cyclic Data Structures

Doubly linked lists are a cyclic data structure, meaning that it is possible to follow a nontrivial sequence of pointers that closes in on itself. In general, building cyclic data structures requires the use of side effects. This is done by constructing the data elements first, and then adding cycles using assignment afterward.

There is an exception to this, though: you can construct fixed-size cyclic data structures using `let rec`:

```
# let rec endless_loop = 1 :: 2 :: 3 :: endless_loop;;
val endless_loop : int list = [1; 2; 3; <cycle>]
```

This approach is quite limited, however. General-purpose cyclic data structures require mutation.

9.4.1 Modifying the List

Now, we'll start considering operations that mutate the list, starting with `insert_first`, which inserts an element at the front of the list:

```
let insert_first t value =
  let new_elt = { prev = None; next = !t; value } in
  (match !t with
  | Some old_first -> old_first.prev <- Some new_elt
  | None -> ());
  t := Some new_elt;
  new_elt
```

`insert_first` first defines a new element `new_elt`, and then links it into the list, finally setting the list itself to point to `new_elt`. Note that the precedence of a `match` expression is very low, so to separate it from the following assignment (`t := Some new_elt`), we surround the match with parentheses. We could have used `begin ... end` for the same purpose, but without some kind of bracketing, the final assignment would incorrectly become part of the `None` case.

We can use `insert_after` to insert elements later in the list. `insert_after` takes as arguments both an `element` after which to insert the new node and a value to insert:

```
let insert_after elt value =
  let new_elt = { value; prev = Some elt; next = elt.next } in
  (match elt.next with
  | Some old_next -> old_next.prev <- Some new_elt
  | None -> ());
  elt.next <- Some new_elt;
  new_elt
```

We also need a `remove` function:

```
let remove t elt =
  let { prev; next; _ } = elt in
  (match prev with
  | Some prev -> prev.next <- next
  | None -> t := next);
  (match next with
```

```
  | Some next -> next.prev <- prev
  | None -> ());
  elt.prev <- None;
  elt.next <- None
```

Note that the preceding code is careful to change the `prev` pointer of the following element and the `next` pointer of the previous element, if they exist. If there's no previous element, then the list pointer itself is updated. In any case, the next and previous pointers of the element itself are set to `None`.

These functions are more fragile than they may seem. In particular, misuse of the interface may lead to corrupted data. For example, double-removing an element will cause the main list reference to be set to `None`, thus emptying the list. Similar problems arise from removing an element from a list it doesn't belong to.

This shouldn't be a big surprise. Complex imperative data structures can be quite tricky, considerably trickier than their pure equivalents. The issues described previously can be dealt with by more careful error detection, and such error correction is taken care of in modules like Core's `Doubly_linked`. You should use imperative data structures from a well-designed library when you can. And when you can't, you should make sure to put great care into your error handling.

9.4.2 Iteration Functions

When defining containers like lists, dictionaries, and trees, you'll typically want to define a set of iteration functions like `iter`, `map`, and `fold`, which let you concisely express common iteration patterns.

`Dlist` has two such iterators: `iter`, the goal of which is to call a `unit`-producing function on every element of the list, in order; and `find_el`, which runs a provided test function on each value stored in the list, returning the first `element` that passes the test. Both `iter` and `find_el` are implemented using simple recursive loops that use `next` to walk from element to element and `value` to extract the element from a given node:

```
let iter t ~f =
  let rec loop = function
    | None -> ()
    | Some el ->
      f (value el);
      loop (next el)
  in
  loop !t

let find_el t ~f =
  let rec loop = function
    | None -> None
    | Some elt -> if f (value elt) then Some elt else loop (next elt)
  in
  loop !t
```

This completes our implementation, but there's still considerably more work to be done to make a really usable doubly linked list. As mentioned earlier, you're probably better off using something like Core `Doubly_linked` module that has a more complete

interface and has more of the tricky corner cases worked out. Nonetheless, this example should serve to demonstrate some of the techniques you can use to build nontrivial imperative data structure in OCaml, as well as some of the pitfalls.

9.5 Laziness and Other Benign Effects

There are many instances where you basically want to program in a pure style, but you want to make limited use of side effects to improve the performance of your code. Such side effects are sometimes called *benign effects*, and they are a useful way of leveraging OCaml's imperative features while still maintaining most of the benefits of pure programming.

One of the simplest benign effects is *laziness*. A lazy value is one that is not computed until it is actually needed. In OCaml, lazy values are created using the `lazy` keyword, which can be used to convert any expression of type `s` into a lazy value of type `s lazy_t`. The evaluation of that expression is delayed until forced with `Lazy.force`:

```
# let v = lazy (print_endline "performing lazy computation";
    Float.sqrt 16.);;
val v : float lazy_t = <lazy>
# Lazy.force v;;
performing lazy computation
- : float = 4.
# Lazy.force v;;
- : float = 4.
```

You can see from the `print` statement that the actual computation was performed only once, and only after `force` had been called.

To better understand how laziness works, let's walk through the implementation of our own lazy type. We'll start by declaring types to represent a lazy value:

```
# type 'a lazy_state =
    | Delayed of (unit -> 'a)
    | Value of 'a
    | Exn of exn;;
type 'a lazy_state = Delayed of (unit -> 'a) | Value of 'a | Exn of exn
# type 'a our_lazy = { mutable state : 'a lazy_state };;
type 'a our_lazy = { mutable state : 'a lazy_state; }
```

A `lazy_state` represents the possible states of a lazy value. A lazy value is `Delayed` before it has been run, where `Delayed` holds a function for computing the value in question. A lazy value is in the `Value` state when it has been forced and the computation ended normally. The `Exn` case is for when the lazy value has been forced, but the computation ended with an exception. A lazy value is simply a record with a single mutable field containing a `lazy_state`, where the mutability makes it possible to change from being in the `Delayed` state to being in the `Value` or `Exn` states.

We can create a lazy value from a thunk, i.e., a function that takes a unit argument. Wrapping an expression in a thunk is another way to suspend the computation of an expression:

```
# let our_lazy f = { state = Delayed f };;
val our_lazy : (unit -> 'a) -> 'a our_lazy = <fun>
# let v =
    our_lazy (fun () ->
      print_endline "performing lazy computation"; Float.sqrt 16.);;
val v : float our_lazy = {state = Delayed <fun>}
```

Now we just need a way to force a lazy value. The following function does just that.

```
# let our_force l =
    match l.state with
    | Value x -> x
    | Exn e -> raise e
    | Delayed f ->
      try
        let x = f () in
        l.state <- Value x;
        x
      with exn ->
        l.state <- Exn exn;
        raise exn;;
val our_force : 'a our_lazy -> 'a = <fun>
```

Which we can use in the same way we used `Lazy.force`:

```
# our_force v;;
performing lazy computation
- : float = 4.
# our_force v;;
- : float = 4.
```

The main user-visible difference between our implementation of laziness and the built-in version is syntax. Rather than writing `our_lazy (fun () -> sqrt 16.)`, we can (with the built-in `lazy`) just write `lazy (sqrt 16.)`, avoiding the necessity of declaring a function.

9.5.1 Memoization and Dynamic Programming

Another benign effect is *memoization*. A memoized function remembers the result of previous invocations of the function so that they can be returned without further computation when the same arguments are presented again.

Here's a function that takes as an argument an arbitrary single-argument function and returns a memoized version of that function. Here we'll use Base's `Hashtbl` module, rather than our toy `Dictionary`.

This implementation requires an argument of a `Hashtbl.Key.t`, which plays the role of the `hash` and `equal` from `Dictionary`. `Hashtbl.Key.t` is an example of what's called a first-class module, which we'll see more of in Chapter 12 (First-Class Modules).

```
# let memoize m f =
    let memo_table = Hashtbl.create m in
    (fun x ->
       Hashtbl.find_or_add memo_table x ~default:(fun () -> f x));;
val memoize : 'a Hashtbl.Key.t -> ('a -> 'b) -> 'a -> 'b = <fun>
```

The preceding code is a bit tricky. `memoize` takes as its argument a function `f` and then allocates a polymorphic hash table (called `memo_table`), and returns a new function which is the memoized version of `f`. When called, this new function uses `Hashtbl.find_or_add` to try to find a value in the `memo_table`, and if it fails, to call `f` and store the result. Note that `memo_table` is referred to by the function, and so won't be collected until the function returned by `memoize` is itself collected.

Memoization can be useful whenever you have a function that is expensive to recompute and you don't mind caching old values indefinitely. One important caution: a memoized function by its nature leaks memory. As long as you hold on to the memoized function, you're holding every result it has returned thus far.

Memoization is also useful for efficiently implementing some recursive algorithms. One good example is the algorithm for computing the *edit distance* (also called the Levenshtein distance) between two strings. The edit distance is the number of single-character changes (including letter switches, insertions, and deletions) required to convert one string to the other. This kind of distance metric can be useful for a variety of approximate string-matching problems, like spellcheckers.

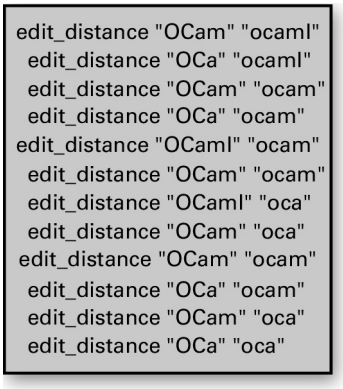
Consider the following code for computing the edit distance. Understanding the algorithm isn't important here, but you should pay attention to the structure of the recursive calls:

```
# let rec edit_distance s t =
    match String.length s, String.length t with
    | (0,x) | (x,0) -> x
    | (len_s,len_t) ->
      let s' = String.drop_suffix s 1 in
      let t' = String.drop_suffix t 1 in
      let cost_to_drop_both =
        if Char.(=) s.[len_s - 1] t.[len_t - 1] then 0 else 1
      in
      List.reduce_exn ~f:Int.min
        [ edit_distance s' t  + 1
        ; edit_distance s  t' + 1
        ; edit_distance s' t' + cost_to_drop_both
        ];;
val edit_distance : string -> string -> int = <fun>
# edit_distance "OCaml" "ocaml";;
- : int = 2
```

The thing to note is that if you call `edit_distance "OCaml" "ocaml"`, then that will in turn dispatch the following calls:

```
edit_distance "OCam" "ocaml"
edit_distance "OCaml" "ocam"
edit_distance "OCam" "ocam"
```

And these calls will in turn dispatch other calls:



As you can see, some of these calls are repeats. For example, there are two different calls to `edit_distance "OCam" "oca"`. The number of redundant calls grows exponentially with the size of the strings, meaning that our implementation of `edit_distance` is brutally slow for large strings. We can see this by writing a small timing function, using Core's `Time` module.

```
# let time f =
    let open Core in
    let start = Time.now () in
    let x = f () in
    let stop = Time.now () in
    printf "Time: %F ms\n" (Time.diff stop start |> Time.Span.to_ms);
    x;;
val time : (unit -> 'a) -> 'a = <fun>
```

And now we can use this to try out some examples:

```
# time (fun () -> edit_distance "OCaml" "ocaml");;
Time: 0.655651092529 ms
- : int = 2
# time (fun () -> edit_distance "OCaml 4.13" "ocaml 4.13");;
Time: 2541.6533947 ms
- : int = 2
```

Just those few extra characters made it thousands of times slower!

Memoization would be a huge help here, but to fix the problem, we need to memoize the calls that `edit_distance` makes to itself. Such recursive memoization is closely related to a common algorithmic technique called *dynamic programming*, except that with dynamic programming, you do the necessary sub-computations bottom-up, in anticipation of needing them. With recursive memoization, you go top-down, only doing a sub-computation when you discover that you need it.

To see how to do this, let's step away from `edit_distance` and instead consider a much simpler example: computing the *n*th element of the Fibonacci sequence. The Fibonacci sequence by definition starts out with two 1s, with every subsequent element being the sum of the previous two. The classic recursive definition of Fibonacci is as follows:

```
# let rec fib i =
```

```
  if i <= 1 then i else fib (i - 1) + fib (i - 2);;
val fib : int -> int = <fun>
```

This is, however, exponentially slow, for the same reason that `edit_distance` was slow: we end up making many redundant calls to `fib`. It shows up quite dramatically in the performance:

```
# time (fun () -> fib 20);;
Time: 1.14369392395 ms
- : int = 6765
# time (fun () -> fib 40);;
Time: 14752.7184486 ms
- : int = 102334155
```

As you can see, `fib 40` takes thousands of times longer to compute than `fib 20`.

So, how can we use memoization to make this faster? The tricky bit is that we need to insert the memoization before the recursive calls within `fib`. We can't just define `fib` in the ordinary way and memoize it after the fact and expect the first call to `fib` to be improved.

```
# let fib = memoize (module Int) fib;;
val fib : int -> int = <fun>
# time (fun () -> fib 40);;
Time: 18174.5970249 ms
- : int = 102334155
# time (fun () -> fib 40);;
Time: 0.00524520874023 ms
- : int = 102334155
```

In order to make `fib` fast, our first step will be to rewrite `fib` in a way that unwinds the recursion. The following version expects as its first argument a function (called `fib`) that will be called in lieu of the usual recursive call.

```
# let fib_norec fib i =
    if i <= 1 then i
    else fib (i - 1) + fib (i - 2);;
val fib_norec : (int -> int) -> int -> int = <fun>
```

We can now turn this back into an ordinary Fibonacci function by tying the recursive knot:

```
# let rec fib i = fib_norec fib i;;
val fib : int -> int = <fun>
# fib 20;;
- : int = 6765
```

We can even write a polymorphic function that we'll call `make_rec` that can tie the recursive knot for any function of this form:

```
# let make_rec f_norec =
    let rec f x = f_norec f x in
    f;;
val make_rec : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
# let fib = make_rec fib_norec;;
val fib : int -> int = <fun>
# fib 20;;
- : int = 6765
```

This is a pretty strange piece of code, and it may take a few moments of thought to figure out what's going on. Like `fib_norec`, the function `f_norec` passed in to `make_rec` is a function that isn't recursive but takes as an argument a function that it will call. What `make_rec` does is to essentially feed `f_norec` to itself, thus making it a true recursive function.

This is clever enough, but all we've really done is find a new way to implement the same old slow Fibonacci function. To make it faster, we need a variant of `make_rec` that inserts memoization when it ties the recursive knot. We'll call that function `memo_rec`:

```
# let memo_rec m f_norec x =
    let fref = ref (fun _ -> assert false) in
    let f = memoize m (fun x -> f_norec !fref x) in
    fref := f;
    f x;;
val memo_rec : 'a Hashtbl.Key.t -> (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b =
  <fun>
```

Note that `memo_rec` has almost the same signature as `make_rec`.

We're using the reference here as a way of tying the recursive knot without using a `let rec`, which for reasons we'll describe later wouldn't work here.

Using `memo_rec`, we can now build an efficient version of `fib`:

```
# let fib = memo_rec (module Int) fib_norec;;
val fib : int -> int = <fun>
# time (fun () -> fib 40);;
Time: 0.121355056763 ms
- : int = 102334155
```

As you can see, the exponential time complexity is now gone.

The memory behavior here is important. If you look back at the definition of `memo_rec`, you'll see that the call `memo_rec fib_norec` does not trigger a call to `memoize`. Only when `fib` is called and thereby the final argument to `memo_rec` is presented does `memoize` get called. The result of that call falls out of scope when the `fib` call returns, and so calling `memo_rec` on a function does not create a memory leak—the memoization table is collected after the computation completes.

We can use `memo_rec` as part of a single declaration that makes this look like it's little more than a special form of `let rec`:

```
# let fib = memo_rec (module Int) (fun fib i ->
  if i <= 1 then 1 else fib (i - 1) + fib (i - 2));;
val fib : int -> int = <fun>
```

Memoization is overkill for implementing Fibonacci, and indeed, the `fib` defined above is not especially efficient, allocating space linear in the number passed into `fib`. It's easy enough to write a Fibonacci function that takes a constant amount of space.

But memoization is a good approach for optimizing `edit_distance`, and we can apply the same approach we used on `fib` here. We will need to change `edit_distance` to take a pair of strings as a single argument, since `memo_rec` only works on single-argument functions. (We can always recover the original interface with a wrapper function.) With just that change and the addition of the `memo_rec` call, we can get a memoized version of `edit_distance`. The memoization key is going to be a pair of

strings, so we need to get our hands on a module with the necessary functionality for building a hash-table in `Base`.

Writing hash-functions and equality tests and the like by hand can be tedious and error prone, so instead we'll use a few different syntax extensions for deriving the necessary functionality automatically. By enabling `ppx_jane`, we pull in a collection of such derivers, three of which we use in defining `String_pair` below.

```
# #require "ppx_jane";;
# module String_pair = struct
    type t = string * string [@@deriving sexp_of, hash, compare]
  end;;
module String_pair :
  sig
    type t = string * string
    val sexp_of_t : t -> Sexp.t
    val hash_fold_t : Hash.state -> t -> Hash.state
    val hash : t -> int
    val compare : t -> t -> int
  end
```

With that in hand, we can define our optimized form of `edit_distance`.

```
# let edit_distance = memo_rec (module String_pair)
    (fun edit_distance (s,t) ->
       match String.length s, String.length t with
       | (0,x) | (x,0) -> x
       | (len_s,len_t) ->
         let s' = String.drop_suffix s 1 in
         let t' = String.drop_suffix t 1 in
         let cost_to_drop_both =
           if Char.(=) s.[len_s - 1] t.[len_t - 1] then 0 else 1
         in
         List.reduce_exn ~f:Int.min
           [ edit_distance (s',t ) + 1
           ; edit_distance (s ,t') + 1
           ; edit_distance (s',t') + cost_to_drop_both
  ]);;
val edit_distance : String_pair.t -> int = <fun>
```

This new version of `edit_distance` is much more efficient than the one we started with; the following call is many thousands of times faster than it was without memoization.

```
# time (fun () -> edit_distance ("OCaml 4.09","ocaml 4.09"));;
Time: 0.964403152466 ms
- : int = 2
```

Limitations of let rec

You might wonder why we didn't tie the recursive knot in `memo_rec` using `let rec`, as we did for `make_rec` earlier. Here's code that tries to do just that:

```
# let memo_rec m f_norec =
    let rec f = memoize m (fun x -> f_norec f x) in
    f;;
Line 2, characters 17-49:
```

```
Error: This kind of expression is not allowed as right-hand side of
    `let rec'
```

OCaml rejects the definition because OCaml, as a strict language, has limits on what it can put on the right-hand side of a `let rec`. In particular, imagine how the following code snippet would be compiled:

```
let rec x = x + 1
```

Note that `x` is an ordinary value, not a function. As such, it's not clear how this definition should be handled by the compiler. You could imagine it compiling down to an infinite loop, but `x` is of type `int`, and there's no `int` that corresponds to an infinite loop. As such, this construct is effectively impossible to compile.

To avoid such impossible cases, the compiler only allows three possible constructs to show up on the right-hand side of a `let rec`: a function definition, a constructor, or the lazy keyword. This excludes some reasonable things, like our definition of `memo_rec`, but it also blocks things that don't make sense, like our definition of `x`.

It's worth noting that these restrictions don't show up in a lazy language like Haskell. Indeed, we can make something like our definition of `x` work if we use OCaml's laziness:

```
# let rec x = lazy (force x + 1);;
val x : int lazy_t = <lazy>
```

Of course, actually trying to compute this will fail. OCaml's `lazy` throws an exception when a lazy value tries to force itself as part of its own evaluation.

```
# force x;;
Exception: Lazy.Undefined
```

But we can also create useful recursive definitions with `lazy`. In particular, we can use laziness to make our definition of `memo_rec` work without explicit mutation:

```
# let lazy_memo_rec m f_norec x =
    let rec f = lazy (memoize m (fun x -> f_norec (force f) x)) in
    (force f) x;;
val lazy_memo_rec : 'a Hashtbl.Key.t -> (('a -> 'b) -> 'a -> 'b) -> 'a ->
    'b =
  <fun>
# time (fun () -> lazy_memo_rec (module Int) fib_norec 40);;
Time: 0.181913375854 ms
- : int = 102334155
```

Laziness is more constrained than explicit mutation, and so in some cases can lead to code whose behavior is easier to think about.

9.6 Input and Output

Imperative programming is about more than modifying in-memory data structures. Any function that doesn't boil down to a deterministic transformation from its arguments to its return value is imperative in nature. That includes not only things that mutate your program's data, but also operations that interact with the world outside of your

program. An important example of this kind of interaction is I/O, i.e., operations for reading or writing data to things like files, terminal input and output, and network sockets.

There are multiple I/O libraries in OCaml. In this section we'll discuss OCaml's buffered I/O library that can be used through the `In_channel` and `Out_channel` modules in Stdio. Other I/O primitives are also available through the `Unix` module in `Core` as well as `Async`, the asynchronous I/O library that is covered in Chapter 17 (Concurrent Programming with Async). Most of the functionality in `Core`'s `In_channel` and `Out_channel` (and in `Core`'s `Unix` module) derives from the standard library, but we'll use `Core`'s interfaces here.

9.6.1 Terminal I/O

OCaml's buffered I/O library is organized around two types: `in_channel`, for channels you read from, and `out_channel`, for channels you write to. The `In_channel` and `Out_channel` modules only have direct support for channels corresponding to files and terminals; other kinds of channels can be created through the `Unix` module.

We'll start our discussion of I/O by focusing on the terminal. Following the UNIX model, communication with the terminal is organized around three channels, which correspond to the three standard file descriptors in Unix:

`In_channel.stdin` The "standard input" channel. By default, input comes from the terminal, which handles keyboard input.

`Out_channel.stdout` The "standard output" channel. By default, output written to `stdout` appears on the user terminal.

`Out_channel.stderr` The "standard error" channel. This is similar to `stdout` but is intended for error messages.

The values `stdin`, `stdout`, and `stderr` are useful enough that they are also available in the top level of `Core`'s namespace directly, without having to go through the `In_channel` and `Out_channel` modules.

Let's see this in action in a simple interactive application. The following program, `time_converter`, prompts the user for a time zone, and then prints out the current time in that time zone. Here, we use `Core`'s `Zone` module for looking up a time zone, and the `Time` module for computing the current time and printing it out in the time zone in question:

```
open Core

let () =
  Out_channel.output_string stdout "Pick a timezone: ";
  Out_channel.flush stdout;
  match In_channel.(input_line stdin) with
  | None -> failwith "No timezone provided"
  | Some zone_string ->
    let zone = Time.Zone.find_exn zone_string in
    let time_string = Time.to_string_abs (Time.now ()) ~zone in
    Out_channel.output_string stdout
```

```
      (String.concat
         ["The time in ";Time.Zone.to_string zone;" is
    ";time_string;".\n"]);
    Out_channel.flush stdout
```

We can build this program using `dune` and run it, though you'll need to add a `dune-project` and `dune` file, as described in Chapter 5 (Files, Modules, and Programs). You'll see that it prompts you for input, as follows:

```
$ dune exec ./time_converter.exe
Pick a timezone:
```

You can then type in the name of a time zone and hit Return, and it will print out the current time in the time zone in question:

```
Pick a timezone: Europe/London
The time in Europe/London is 2013-08-15 00:03:10.666220+01:00.
```

We called `Out_channel.flush` on `stdout` because `out_channels` are buffered, which is to say that OCaml doesn't immediately do a write every time you call `output_string`. Instead, writes are buffered until either enough has been written to trigger the flushing of the buffers, or until a flush is explicitly requested. This greatly increases the efficiency of the writing process by reducing the number of system calls.

Note that `In_channel.input_line` returns a `string option`, with `None` indicating that the input stream has ended (i.e., an end-of-file condition). `Out_channel.output_string` is used to print the final output, and `Out_channel.flush` is called to flush that output to the screen. The final flush is not technically required, since the program ends after that instruction, at which point all remaining output will be flushed anyway, but the explicit flush is nonetheless good practice.

9.6.2 Formatted Output with printf

Generating output with functions like `Out_channel.output_string` is simple and easy to understand, but can be a bit verbose. OCaml also supports formatted output using the `printf` function, which is modeled after `printf` in the C standard library. `printf` takes a *format string* that describes what to print and how to format it, as well as arguments to be printed, as determined by the formatting directives embedded in the format string. So, for example, we can write:

```
# printf
    "%i is an integer, %F is a float, \"%s\" is a string\n"
  3 4.5 "five";;
3 is an integer, 4.5 is a float, "five" is a string
- : unit = ()
```

Unlike C's `printf`, the `printf` in OCaml is type-safe. In particular, if we provide an argument whose type doesn't match what's presented in the format string, we'll get a type error:

```
# printf "An integer: %i\n" 4.5;;
Line 1, characters 27-30:
```

```
Error: This expression has type float but an expression was expected
    of type
         int
```

Understanding Format Strings

The format strings used by `printf` turn out to be quite different from ordinary strings. This difference ties to the fact that OCaml's `printf` facility, unlike the equivalent in C, is type-safe. In particular, the compiler checks that the types referred to by the format string match the types of the rest of the arguments passed to `printf`.

To check this, OCaml needs to analyze the contents of the format string at compile time, which means the format string needs to be available as a string literal at compile time. Indeed, if you try to pass an ordinary string to `printf`, the compiler will complain:

```
# let fmt = "%i is an integer\n";;
val fmt : string = "%i is an integer\n"
# printf fmt 3;;
Line 1, characters 8-11:
Error: This expression has type string but an expression was expected
    of type
         ('a -> 'b, Stdio.Out_channel.t, unit) format =
           ('a -> 'b, Stdio.Out_channel.t, unit, unit, unit, unit)
    format6
```

If OCaml infers that a given string literal is a format string, then it parses it at compile time as such, choosing its type in accordance with the formatting directives it finds. Thus, if we add a type annotation indicating that the string we're defining is actually a format string, it will be interpreted as such. (Here, we open the CamlinternalFormatBasics so that the representation of the format string that's printed out won't fill the whole page.)

```
# open CamlinternalFormatBasics;;
# let fmt : ('a, 'b, 'c) format =
  "%i is an integer\n";;
val fmt : (int -> 'c, 'b, 'c) format =
  Format
   (Int (Int_i, No_padding, No_precision,
     String_literal (" is an integer\n", End_of_format)),
   "%i is an integer\n")
```

And accordingly, we can pass it to `printf`:

```
# printf fmt 3;;
3 is an integer
- : unit = ()
```

If this looks different from everything else you've seen so far, that's because it is. This is really a special case in the type system. Most of the time, you don't need to know about this special handling of format strings—you can just use `printf` and not worry about the details. But it's useful to keep the broad outlines of the story in the back of your head.

Now let's see how we can rewrite our time conversion program to be a little more concise using `printf`:

```
open Core

let () =
  printf "Pick a timezone: %!";
  match In_channel.input_line In_channel.stdin with
  | None -> failwith "No timezone provided"
  | Some zone_string ->
    let zone = Time.Zone.find_exn zone_string in
    let time_string = Time.to_string_abs (Time.now ()) ~zone in
    printf "The time in %s is %s.\n%!" (Time.Zone.to_string zone)
    time_string
```

In the preceding example, we've used only two formatting directives: `%s`, for including a string, and `%!` which causes `printf` to flush the channel.

`printf`'s formatting directives offer a significant amount of control, allowing you to specify things like:

- Alignment and padding
- Escaping rules for strings
- Whether numbers should be formatted in decimal, hex, or binary
- Precision of float conversions

There are also `printf`-style functions that target outputs other than `stdout`, including:

- `eprintf`, which prints to `stderr`
- `fprintf`, which prints to an arbitrary `out_channel`
- `sprintf`, which returns a formatted string

All of this, and a good deal more, is described in the API documentation for the `Printf` module in the OCaml Manual.

9.6.3 File I/O

Another common use of `in_channels` and `out_channels` is for working with files. Here are a couple of functions—one that creates a file full of numbers, and the other that reads in such a file and returns the sum of those numbers:

```
# let create_number_file filename numbers =
    let outc = Out_channel.create filename in
    List.iter numbers ~f:(fun x -> Out_channel.fprintf outc "%d\n" x);
    Out_channel.close outc;;
val create_number_file : string -> int list -> unit = <fun>
# let sum_file filename =
    let file = In_channel.create filename in
    let numbers = List.map ~f:Int.of_string (In_channel.input_lines
    file) in
    let sum = List.fold ~init:0 ~f:(+) numbers in
    In_channel.close file;
```

```
    sum;;
val sum_file : string -> int = <fun>
# create_number_file "numbers.txt" [1;2;3;4;5];;
- : unit = ()
# sum_file "numbers.txt";;
- : int = 15
```

For both of these functions, we followed the same basic sequence: we first create the channel, then use the channel, and finally close the channel. The closing of the channel is important, since without it, we won't release resources associated with the file back to the operating system.

One problem with the preceding code is that if it throws an exception in the middle of its work, it won't actually close the file. If we try to read a file that doesn't actually contain numbers, we'll see such an error:

```
# sum_file "/etc/hosts";;
Exception:
(Failure
  "Int.of_string: \"127.0.0.1   localhost localhost.localdomain
    localhost4 localhost4.localdomain4\"")
```

And if we do this over and over in a loop, we'll eventually run out of file descriptors:

```
# for i = 1 to 10000 do try ignore (sum_file "/etc/hosts") with _ ->
    () done;;
- : unit = ()
# sum_file "numbers.txt";;
Error: I/O error: ...: Too many open files
```

And now, you'll need to restart your toplevel if you want to open any more files!

To avoid this, we need to make sure that our code cleans up after itself. We can do this using the `protect` function described in Chapter 8 (Error Handling), as follows:

```
# let sum_file filename =
    let file = In_channel.create filename in
    Exn.protect ~f:(fun () ->
      let numbers = List.map ~f:Int.of_string (In_channel.input_lines
    file) in
      List.fold ~init:0 ~f:(+) numbers)
      ~finally:(fun () -> In_channel.close file);;
val sum_file : string -> int = <fun>
```

And now, the file descriptor leak is gone:

```
# for i = 1 to 10000 do try ignore (sum_file "/etc/hosts" : int) with
    _ -> () done;;
- : unit = ()
# sum_file "numbers.txt";;
- : int = 15
```

This is really an example of a more general issue with imperative programming and exceptions. If you're changing the internal state of your program and you're interrupted by an exception, you need to consider quite carefully if it's safe to continue working from your current state.

`In_channel` has functions that automate the handling of some of these details. For

example, `In_channel.with_file` takes a filename and a function for processing data from an `in_channel` and takes care of the bookkeeping associated with opening and closing the file. We can rewrite `sum_file` using this function, as shown here:

```
# let sum_file filename =
    In_channel.with_file filename ~f:(fun file ->
      let numbers = List.map ~f:Int.of_string (In_channel.input_lines
    file) in
      List.fold ~init:0 ~f:(+) numbers);;
val sum_file : string -> int = <fun>
```

Another misfeature of our implementation of `sum_file` is that we read the entire file into memory before processing it. For a large file, it's more efficient to process a line at a time. You can use the `In_channel.fold_lines` function to do just that:

```
# let sum_file filename =
    In_channel.with_file filename ~f:(fun file ->
      In_channel.fold_lines file ~init:0 ~f:(fun sum line ->
        sum + Int.of_string line));;
val sum_file : string -> int = <fun>
```

This is just a taste of the functionality of `In_channel` and `Out_channel`. To get a fuller understanding, you should review the API documentation.

9.7 Order of Evaluation

The order in which expressions are evaluated is an important part of the definition of a programming language, and it is particularly important when programming imperatively. Most programming languages you're likely to have encountered are *strict*, and OCaml is too. In a strict language, when you bind an identifier to the result of some expression, the expression is evaluated before the variable is bound. Similarly, if you call a function on a set of arguments, those arguments are evaluated before they are passed to the function.

Consider the following simple example. Here, we have a collection of angles, and we want to determine if any of them have a negative `sin`. The following snippet of code would answer that question:

```
# let x = Float.sin 120. in
  let y = Float.sin 75.  in
  let z = Float.sin 128. in
  List.exists ~f:(fun x -> Float.O.(x < 0.)) [x;y;z];;
- : bool = true
```

In some sense, we don't really need to compute the `sin 128` because `sin 75` is negative, so we could know the answer before even computing `sin 128`.

It doesn't have to be this way. Using the `lazy` keyword, we can write the original computation so that `sin 128` won't ever be computed:

```
# let x = lazy (Float.sin 120.) in
  let y = lazy (Float.sin 75.)  in
  let z = lazy (Float.sin 128.) in
```

```
  List.exists ~f:(fun x -> Float.0.(Lazy.force x < 0.)) [x;y;z];;
- : bool = true
```

We can confirm that fact by a few well-placed `printf`s:

```
# let x = lazy (printf "1\n"; Float.sin 120.) in
  let y = lazy (printf "2\n"; Float.sin 75.)  in
  let z = lazy (printf "3\n"; Float.sin 128.) in
  List.exists ~f:(fun x -> Float.0.(Lazy.force x < 0.)) [x;y;z];;
1
2
- : bool = true
```

OCaml is strict by default for a good reason: lazy evaluation and imperative programming generally don't mix well because laziness makes it harder to reason about when a given side effect is going to occur. Understanding the order of side effects is essential to reasoning about the behavior of an imperative program.

Because OCaml is strict, we know that expressions that are bound by a sequence of `let` bindings will be evaluated in the order that they're defined. But what about the evaluation order within a single expression? Officially, the answer is that evaluation order within an expression is undefined. In practice, OCaml has only one compiler, and that behavior is a kind of *de facto* standard. Unfortunately, the evaluation order in this case is often the opposite of what one might expect.

Consider the following example:

```
# List.exists ~f:(fun x -> Float.0.(x < 0.))
    [ (printf "1\n"; Float.sin 120.);
      (printf "2\n"; Float.sin 75.);
      (printf "3\n"; Float.sin 128.); ];;
3
2
1
- : bool = true
```

Here, you can see that the subexpression that came last was actually evaluated first! This is generally the case for many different kinds of expressions. If you want to make sure of the evaluation order of different subexpressions, you should express them as a series of `let` bindings.

9.8 Side Effects and Weak Polymorphism

Consider the following simple, imperative function:

```
# let remember =
    let cache = ref None in
    (fun x ->
       match !cache with
       | Some y -> y
       | None -> cache := Some x; x);;
val remember : '_weak1 -> '_weak1 = <fun>
```

`remember` simply caches the first value that's passed to it, returning that value on

every call. That's because `cache` is created and initialized once and is shared across invocations of `remember`.

`remember` is not a terribly useful function, but it raises an interesting question: what should its type be?

On its first call, `remember` returns the same value it's passed, which means its input type and return type should match. Accordingly, `remember` should have type `t -> t` for some type `t`. There's nothing about `remember` that ties the choice of `t` to any particular type, so you might expect OCaml to generalize, replacing `t` with a polymorphic type variable. It's this kind of generalization that gives us polymorphic types in the first place. The identity function, as an example, gets a polymorphic type in this way:

```
# let identity x = x;;
val identity : 'a -> 'a = <fun>
# identity 3;;
- : int = 3
# identity "five";;
- : string = "five"
```

As you can see, the polymorphic type of `identity` lets it operate on values with different types.

This is not what happens with `remember`, though. As you can see from the above examples, the type that OCaml infers for `remember` looks almost, but not quite, like the type of the identity function. Here it is again:

```
val remember : '_weak1 -> '_weak1 = <fun>
```

The underscore in the type variable `'_weak1` tells us that the variable is only *weakly polymorphic*, which is to say that it can be used with any *single* type. That makes sense because, unlike `identity`, `remember` always returns the value it was passed on its first invocation, which means its return value must always have the same type.

OCaml will convert a weakly polymorphic variable to a concrete type as soon as it gets a clue as to what concrete type it is to be used as:

```
# let remember_three () = remember 3;;
val remember_three : unit -> int = <fun>
# remember;;
- : int -> int = <fun>
# remember "avocado";;
Line 1, characters 10-19:
Error: This expression has type string but an expression was expected
    of type
         int
```

Note that the type of `remember` was settled by the definition of `remember_three`, even though `remember_three` was never called!

9.8.1 The Value Restriction

So, when does the compiler infer weakly polymorphic types? As we've seen, we need weakly polymorphic types when a value of unknown type is stored in a persistent mutable cell. Because the type system isn't precise enough to determine all cases

where this might happen, OCaml uses a rough rule to flag cases that don't introduce any persistent mutable cells, and to only infer polymorphic types in those cases. This rule is called *the value restriction*.

The core of the value restriction is the observation that some kinds of expressions, which we'll refer to as *simple values*, by their nature can't introduce persistent mutable cells, including:

- Constants (i.e., things like integer and floating-point literals)
- Constructors that only contain other simple values
- Function declarations, i.e., expressions that begin with `fun` or `function`, or the equivalent let binding, `let f x = ...`
- `let` bindings of the form `let` *`var`* `=` *`expr1`* `in` *`expr2`*, where both *`expr1`* and *`expr2`* are simple values

Thus, the following expression is a simple value, and as a result, the types of values contained within it are allowed to be polymorphic:

```
# (fun x -> [x;x]);;
- : 'a -> 'a list = <fun>
```

But, if we write down an expression that isn't a simple value by the preceding definition, we'll get different results.

```
# identity (fun x -> [x;x]);;
- : '_weak2 -> '_weak2 list = <fun>
```

In principle, it would be safe to infer a fully polymorphic variable here, but because OCaml's type system doesn't distinguish between pure and impure functions, it can't separate those two cases.

The value restriction doesn't require that there is no mutable state, only that there is no *persistent* mutable state that could share values between uses of the same function. Thus, a function that produces a fresh reference every time it's called can have a fully polymorphic type:

```
# let f () = ref None;;
val f : unit -> 'a option ref = <fun>
```

But a function that has a mutable cache that persists across calls, like `memoize`, can only be weakly polymorphic.

9.8.2 Partial Application and the Value Restriction

Most of the time, when the value restriction kicks in, it's for a good reason, i.e., it's because the value in question can actually only safely be used with a single type. But sometimes, the value restriction kicks in when you don't want it. The most common such case is partially applied functions. A partially applied function, like any function application, is not a simple value, and as such, functions created by partial application are sometimes less general than you might expect.

Consider the `List.init` function, which is used for creating lists where each element is created by calling a function on the index of that element:

```
# List.init;;
- : int -> f:(int -> 'a) -> 'a list = <fun>
# List.init 10 ~f:Int.to_string;;
- : string list = ["0"; "1"; "2"; "3"; "4"; "5"; "6"; "7"; "8"; "9"]
```

Imagine we wanted to create a specialized version of `List.init` that always created lists of length 10. We could do that using partial application, as follows:

```
# let list_init_10 = List.init 10;;
val list_init_10 : f:(int -> '_weak3) -> '_weak3 list = <fun>
```

As you can see, we now infer a weakly polymorphic type for the resulting function. That's because there's nothing that guarantees that `List.init` isn't creating a persistent `ref` somewhere inside of it that would be shared across multiple calls to `list_init_10`. We can eliminate this possibility, and at the same time get the compiler to infer a polymorphic type, by avoiding partial application:

```
# let list_init_10 ~f = List.init 10 ~f;;
val list_init_10 : f:(int -> 'a) -> 'a list = <fun>
```

This transformation is referred to as *eta expansion* and is often useful to resolve problems that arise from the value restriction.

9.8.3 Relaxing the Value Restriction

OCaml is actually a little better at inferring polymorphic types than was suggested previously. The value restriction as we described it is basically a syntactic check: you can do a few operations that count as simple values, and anything that's a simple value can be generalized.

But OCaml actually has a relaxed version of the value restriction that can make use of type information to allow polymorphic types for things that are not simple values.

For example, we saw that a function application, even a simple application of the identity function, is not a simple value and thus can turn a polymorphic value into a weakly polymorphic one:

```
# identity (fun x -> [x;x]);;
- : '_weak4 -> '_weak4 list = <fun>
```

But that's not always the case. When the type of the returned value is immutable, then OCaml can typically infer a fully polymorphic type:

```
# identity [];;
- : 'a list = []
```

On the other hand, if the returned type is mutable, then the result will be weakly polymorphic:

```
# [||];;
- : 'a array = [||]
# identity [||];;
- : '_weak5 array = [||]
```

A more important example of this comes up when defining abstract data types. Consider the following simple data structure for an immutable list type that supports constant-time concatenation:

```
# module Concat_list : sig
    type 'a t
    val empty : 'a t
    val singleton : 'a -> 'a t
    val concat  : 'a t -> 'a t -> 'a t  (* constant time *)
    val to_list : 'a t -> 'a list      (* linear time   *)
  end = struct

    type 'a t = Empty | Singleton of 'a | Concat of 'a t * 'a t

    let empty = Empty
    let singleton x = Singleton x
    let concat x y = Concat (x,y)

    let rec to_list_with_tail t tail =
      match t with
      | Empty -> tail
      | Singleton x -> x :: tail
      | Concat (x,y) -> to_list_with_tail x (to_list_with_tail y tail)

    let to_list t =
      to_list_with_tail t []

  end;;
module Concat_list :
  sig
    type 'a t
    val empty : 'a t
    val singleton : 'a -> 'a t
    val concat : 'a t -> 'a t -> 'a t
    val to_list : 'a t -> 'a list
  end
```

The details of the implementation don't matter so much, but it's important to note that a `Concat_list.t` is unquestionably an immutable value. However, when it comes to the value restriction, OCaml treats it as if it were mutable:

```
# Concat_list.empty;;
- : 'a Concat_list.t = <abstr>
# identity Concat_list.empty;;
- : '_weak6 Concat_list.t = <abstr>
```

The issue here is that the signature, by virtue of being abstract, has obscured the fact that `Concat_list.t` is in fact an immutable data type. We can resolve this in one of two ways: either by making the type concrete (i.e., exposing the implementation in the `mli`), which is often not desirable; or by marking the type variable in question as *covariant*. We'll learn more about covariance and contravariance in Chapter 13 (Objects), but for now, you can think of it as an annotation that can be put in the interface of a pure data structure.

In particular, if we replace `type 'a t` in the interface with `type +'a t`, that will make it explicit in the interface that the data structure doesn't contain any persistent references to values of type `'a`, at which point, OCaml can infer polymorphic types for expressions of this type that are not simple values:

```
# module Concat_list : sig
```

```
    type +'a t
    val empty : 'a t
    val singleton : 'a -> 'a t
    val concat  : 'a t -> 'a t -> 'a t  (* constant time *)
    val to_list : 'a t -> 'a list      (* linear time   *)
  end = struct

    type 'a t = Empty | Singleton of 'a | Concat of 'a t * 'a t

    let empty = Empty
    let singleton x = Singleton x
    let concat x y = Concat (x,y)

    let rec to_list_with_tail t tail =
      match t with
      | Empty -> tail
      | Singleton x -> x :: tail
      | Concat (x,y) -> to_list_with_tail x (to_list_with_tail y tail)

    let to_list t =
      to_list_with_tail t []

  end;;
module Concat_list :
  sig
    type +'a t
    val empty : 'a t
    val singleton : 'a -> 'a t
    val concat : 'a t -> 'a t -> 'a t
    val to_list : 'a t -> 'a list
  end
```

Now, we can apply the identity function to `Concat_list.empty` without losing any polymorphism:

```
# identity Concat_list.empty;;
- : 'a Concat_list.t = <abstr>
```

9.9 Summary

This chapter has covered quite a lot of ground, including:

- Discussing the building blocks of mutable data structures as well as the basic imperative constructs like `for` loops, `while` loops, and the sequencing operator `;`
- Walking through the implementation of a couple of classic imperative data structures
- Discussing so-called benign effects like memoization and laziness
- Covering OCaml's API for blocking I/O
- Discussing how language-level issues like order of evaluation and weak polymorphism interact with OCaml's imperative features

The scope and sophistication of the material here is an indication of the importance of OCaml's imperative features. The fact that OCaml defaults to immutability shouldn't

obscure the fact that imperative programming is a fundamental part of building any serious application, and that if you want to be an effective OCaml programmer, you need to understand OCaml's approach to imperative programming.

10 GADTs

Generalized Algebraic Data Types, or GADTs for short, are an extension of the variants we saw in Chapter 7 (Variants). GADTs are more expressive than regular variants, which helps you create types that more precisely match the shape of the program you want to write. That can help you write code that's safer, more concise, and more efficient.

At the same time, GADTs are an advanced feature of OCaml, and their power comes at a distinct cost. GADTs are harder to use and less intuitive than ordinary variants, and it can sometimes be a bit of a puzzle to figure out how to use them effectively. All of which is to say that you should only use a GADT when it makes a big qualitative improvement to your design.

That said, for the right use-case, GADTs can be really transformative, and this chapter will walk through several examples that demonstrate the range of use-cases that GADTs support.

At their heart, GADTs provide two extra features above and beyond ordinary variants:

- They let the compiler learn more type information when you descend into a case of a pattern match.
- They make it easy to use *existential types*, which let you work with data of a specific but unknown type.

It's a little hard to understand these features without working through some examples, so we'll do that next.

10.1 A Little Language

One classic use-case for GADTs is for writing typed expression languages, similar to the boolean expression language described in Chapter 7.3 (Variants and Recursive Data Structures). In this section, we'll create a slightly richer language that lets us mix arithmetic and boolean expressions. This means that we have to deal with the possibility of ill-typed expressions, e.g., an expression that adds a `bool` and an `int`.

Let's first try to do this with an ordinary variant. We'll declare two types here: `value`, which represents a primitive value in the language (i.e., an `int` or a `bool`), and `expr`, which represents the full set of possible expressions.

```
open Base

type value =
  | Int of int
  | Bool of bool

type expr =
  | Value of value
  | Eq of expr * expr
  | Plus of expr * expr
  | If of expr * expr * expr
```

We can write a recursive evaluator for this type in a pretty straight-ahead style. First, we'll declare an exception that can be thrown when we hit an ill-typed expression, e.g., when encountering an expression that tries to add a bool and an int.

```
exception Ill_typed
```

With that in hand, we can write the evaluator itself.

```
# let rec eval expr =
    match expr with
    | Value v -> v
    | If (c, t, e) ->
      (match eval c with
       | Bool b -> if b then eval t else eval e
       | Int _ -> raise Ill_typed)
    | Eq (x, y) ->
      (match eval x, eval y with
       | Bool _, _ | _, Bool _ -> raise Ill_typed
       | Int f1, Int f2 -> Bool (f1 = f2))
    | Plus (x, y) ->
      (match eval x, eval y with
       | Bool _, _ | _, Bool _ -> raise Ill_typed
       | Int f1, Int f2 -> Int (f1 + f2));;
val eval : expr -> value = <fun>
```

This implementation is a bit ugly because it has a lot of dynamic checks to detect type errors. Indeed, it's entirely possible to create an ill-typed expression which will trip these checks.

```
# let i x = Value (Int x)
  and b x = Value (Bool x)
  and (+:) x y = Plus (x,y);;
val i : int -> expr = <fun>
val b : bool -> expr = <fun>
val ( +: ) : expr -> expr -> expr = <fun>
# eval (i 3 +: b false);;
Exception: Ill_typed.
```

This possibility of ill-typed expressions doesn't just complicate the implementation: it's also a problem for users, since it's all too easy to create ill-typed expressions by mistake.

10.1.1 Making the Language Type-Safe

Let's consider what a type-safe version of this API might look like in the absence of GADTs. To even express the type constraints, we'll need expressions to have a type parameter to distinguish integer expressions from boolean expressions. Given such a parameter, the signature for such a language might look like this.

```
module type Typesafe_lang_sig = sig
  type 'a t

  (** functions for constructing expressions *)

  val int : int -> int t
  val bool : bool -> bool t
  val if_ : bool t -> 'a t -> 'a t -> 'a t
  val eq : 'a t -> 'a t -> bool t
  val plus : int t -> int t -> int t

  (** Evaluation functions *)

  val int_eval : int t -> int
  val bool_eval : bool t -> bool
end
```

The functions `int_eval` and `bool_eval` deserve some explanation. You might expect there to be a single evaluation function, with this signature.

```
val eval : 'a t -> 'a
```

But as we'll see, we're not going to be able to implement that, at least, not without using GADTs. So for now, we're stuck with two different evaluators, one for each type of expression.

Now let's write an implementation that matches this signature.

```
module Typesafe_lang : Typesafe_lang_sig = struct
  type 'a t = expr

  let int x = Value (Int x)
  let bool x = Value (Bool x)
  let if_ c t e = If (c, t, e)
  let eq x y = Eq (x, y)
  let plus x y = Plus (x, y)

  let int_eval expr =
    match eval expr with
    | Int x -> x
    | Bool _ -> raise Ill_typed

  let bool_eval expr =
    match eval expr with
    | Bool x -> x
    | Int _ -> raise Ill_typed
end
```

As you can see, the ill-typed expression we had trouble with before can't be constructed, because it's rejected by OCaml's type-system.

```
# let expr = Typesafe_lang.(plus (int 3) (bool false));;
Line 1, characters 40-52:
Error: This expression has type bool t but an expression was expected
    of type
         int t
       Type bool is not compatible with type int
```

So, what happened here? How did we add the type-safety we wanted? The fundamental trick is to add what's called a *phantom type*. In this definition:

```
type 'a t = expr
```

the type parameter `'a` is the phantom type, since it doesn't show up in the body of the definition of `t`.

Because the type parameter is unused, it's free to take on any value. That means we can constrain the use of that type parameter arbitrarily in the signature, which is a freedom we use to add the type-safety rules that we wanted.

This all amounts to an improvement in terms of the API, but the implementation is if anything worse. We still have the same evaluator with all of its dynamic checking for type errors. But we've had to write yet more wrapper code to make this work.

Also, the phantom-type discipline is quite error prone. You might have missed the fact that the type on the `eq` function above is wrong!

```
# Typesafe_lang.eq;;
- : 'a Typesafe_lang.t -> 'a Typesafe_lang.t -> bool Typesafe_lang.t =
    <fun>
```

It looks like it's polymorphic over the type of expressions, but the evaluator only supports checking equality on integers. As a result, we can still construct an ill-typed expression, phantom-types notwithstanding.

```
# let expr = Typesafe_lang.(eq (bool true) (bool false));;
val expr : bool Typesafe_lang.t = <abstr>
# Typesafe_lang.bool_eval expr;;
Exception: Ill_typed.
```

This highlights why we still need the dynamic checks in the implementation: the types within the implementation don't necessarily rule out ill-typed expressions. The same fact explains why we needed two different `eval` functions: the implementation of eval doesn't have any type-level guarantee of when it's handling a bool expression versus an int expression, so it can't safely give results where the type of the result varies based on the result of the expression.

10.1.2 Trying to Do Better with Ordinary Variants

To see why we need GADTs, let's see how far we can get without them. In particular, let's see what happens when we try to encode the typing rules we want for our DSL directly into the definition of the expression type. We'll do that by putting an ordinary type parameter on our `expr` and `value` types, in order to represent the type of an expression or value.

```
type 'a value =
  | Int of 'a
  | Bool of 'a

type 'a expr =
  | Value of 'a value
  | Eq of 'a expr * 'a expr
  | Plus of 'a expr * 'a expr
  | If of bool expr * 'a expr * 'a expr
```

This looks promising at first, but it doesn't quite do what we want. Let's experiment a little.

```
# let i x = Value (Int x)
  and b x = Value (Bool x)
  and (+:) x y = Plus (x,y);;
val i : 'a -> 'a expr = <fun>
val b : 'a -> 'a expr = <fun>
val ( +: ) : 'a expr -> 'a expr -> 'a expr = <fun>
# i 3;;
- : int expr = Value (Int 3)
# b false;;
- : bool expr = Value (Bool false)
# i 3 +: i 4;;
- : int expr = Plus (Value (Int 3), Value (Int 4))
```

So far so good. But if you think about it for a minute, you'll realize this doesn't actually do what we want. For one thing, the type of the outer expression is always just equal to the type of the inner expression, which means that some things that should type-check don't.

```
# If (Eq (i 3, i 4), i 0, i 1);;
Line 1, characters 9-12:
Error: This expression has type int expr
       but an expression was expected of type bool expr
       Type int is not compatible with type bool
```

Also, some things that shouldn't typecheck do.

```
# b 3;;
- : int expr = Value (Bool 3)
```

The problem here is that the way we want to use the type parameter isn't supported by ordinary variants. In particular, we want the type parameter to be populated in different ways in the different tags, and to depend in non-trivial ways on the types of the data associated with each tag. That's where GADTs can help.

10.1.3 GADTs to the Rescue

Now we're ready to write our first GADT. Here's a new version of our `value` and `expr` types that correctly encode our desired typing rules.

```
type _ value =
  | Int : int -> int value
  | Bool : bool -> bool value
```

```
type _ expr =
  | Value : 'a value -> 'a expr
  | Eq : int expr * int expr -> bool expr
  | Plus : int expr * int expr -> int expr
  | If : bool expr * 'a expr * 'a expr -> 'a expr
```

The syntax here requires some decoding. The colon to the right of each tag is what tells you that this is a GADT. To the right of the colon, you'll see what looks like an ordinary, single-argument function type, and you can almost think of it that way; specifically, as the type signature for that particular tag, viewed as a type constructor. The left-hand side of the arrow states the types of the arguments to the constructor, and the right-hand side determines the type of the constructed value.

In the definition of each tag in a GADT, the right-hand side of the arrow is an instance of the type of the overall GADT, with independent choices for the type parameter in each case. Importantly, the type parameter can depend both on the tag and on the type of the arguments. `Eq` is an example where the type parameter is determined entirely by the tag: it always corresponds to a `bool expr`. `If` is an example where the type parameter depends on the arguments to the tag, in particular the type parameter of the `If` is the type parameter of the then and else clauses.

Let's try some examples.

```
# let i x = Value (Int x)
  and b x = Value (Bool x)
  and (+:) x y = Plus (x,y);;
val i : int -> int expr = <fun>
val b : bool -> bool expr = <fun>
val ( +: ) : int expr -> int expr -> int expr = <fun>
# i 3;;
- : int expr = Value (Int 3)
# b 3;;
Line 1, characters 3-4:
Error: This expression has type int but an expression was expected of
    type
         bool
# i 3 +: i 6;;
- : int expr = Plus (Value (Int 3), Value (Int 6))
# i 3 +: b false;;
Line 1, characters 8-15:
Error: This expression has type bool expr
       but an expression was expected of type int expr
       Type bool is not compatible with type int
```

What we see here is that the type-safety rules we previously enforced with signature-level restrictions on phantom types are now directly encoded in the definition of the expression type.

These type-safety rules apply not just when constructing an expression, but also when deconstructing one, which means we can write a simpler and more concise evaluator that doesn't need any type-safety checks.

```
# let eval_value : type a. a value -> a = function
    | Int x -> x
```

```
    | Bool x -> x;;
val eval_value : 'a value -> 'a = <fun>
# let rec eval : type a. a expr -> a = function
    | Value v -> eval_value v
    | If (c, t, e) -> if eval c then eval t else eval e
    | Eq (x, y) -> eval x = eval y
    | Plus (x, y) -> eval x + eval y;;
val eval : 'a expr -> 'a = <fun>
```

Note that we now have a single polymorphic eval function, as opposed to the two type-specific evaluators we needed when using phantom types.

10.1.4 GADTs, Locally Abstract Types, and Polymorphic Recursion

The above example lets us see one of the downsides of GADTs, which is that code using them needs extra type annotations. Look at what happens if we write the definition of `value` without the annotation.

```
# let eval_value = function
    | Int x -> x
    | Bool x -> x;;
Line 3, characters 7-13:
Error: This pattern matches values of type bool value
       but a pattern was expected which matches values of type int
    value
      Type bool is not compatible with type int
```

The issue here is that OCaml by default isn't willing to instantiate ordinary type variables in different ways in the body of the same function, which is what is required here. We can fix that by adding a *locally abstract type*, which doesn't have that restriction.

```
# let eval_value (type a) (v : a value) : a =
    match v with
    | Int x -> x
    | Bool x -> x;;
val eval_value : 'a value -> 'a = <fun>
```

This isn't the same annotation we wrote earlier, and indeed, if we try this approach with `eval`, we'll see that it doesn't work.

```
# let rec eval (type a) (e : a expr) : a =
    match e with
    | Value v -> eval_value v
    | If (c, t, e) -> if eval c then eval t else eval e
    | Eq (x, y) -> eval x = eval y
    | Plus (x, y) -> eval x + eval y;;
Line 4, characters 43-44:
Error: This expression has type a expr but an expression was expected
    of type
         bool expr
       The type constructor a would escape its scope
```

This is a pretty unhelpful error message, but the basic problem is that `eval` is recursive, and inference of GADTs doesn't play well with recursive calls.

More specifically, the issue is that the type-checker is trying to merge the locally abstract type `a` into the type of the recursive function `eval`, and merging it into the outer scope within which `eval` is defined is the way in which `a` is escaping its scope.

We can fix this by explicitly marking `eval` as polymorphic, which OCaml has a handy type annotation for.

```
# let rec eval : 'a. 'a expr -> 'a =
    fun (type a) (x : a expr) ->
     match x with
     | Value v -> eval_value v
     | If (c, t, e) -> if eval c then eval t else eval e
     | Eq (x, y) -> eval x = eval y
     | Plus (x, y) -> eval x + eval y;;
val eval : 'a expr -> 'a = <fun>
```

This works because by marking `eval` as polymorphic, the type of `eval` isn't specialized to `a`, and so `a` doesn't escape its scope.

It's also helpful here because `eval` itself is an example of *polymorphic recursion*, which is to say that `eval` needs to call itself at multiple different types. This comes up, for example, with `If`, since the `If` itself must be of type `bool`, but the type of the then and else clauses could be of type `int`. This means that when evaluating `If`, we'll dispatch `eval` at a different type than it was called on.

As such, `eval` needs to see itself as polymorphic. This kind of polymorphism is basically impossible to infer automatically, which is a second reason we need to annotate `eval`'s polymorphism explicitly.

The above syntax is a bit verbose, so OCaml has syntactic sugar to combine the polymorphism annotation and the creation of the locally abstract types:

```
# let rec eval : type a. a expr -> a = function
    | Value v -> eval_value v
    | If (c, t, e) -> if eval c then eval t else eval e
    | Eq (x, y) -> eval x = eval y
    | Plus (x, y) -> eval x + eval y;;
val eval : 'a expr -> 'a = <fun>
```

This type of annotation is the right one to pick when you write any recursive function that makes use of GADTs.

10.2 When Are GADTs Useful?

The typed language we showed above is a perfectly reasonable example, but GADTs are useful for a lot more than designing little languages. In this section, we'll try to give you a broader sampling of the kinds of things you can do with GADTs.

10.2.1 Varying Your Return Type

Sometimes, you want to write a single function that can effectively have different types in different circumstances. In some sense, this is totally ordinary. After all, OCaml's

polymorphism means that values can take on different types in different contexts. `List.find` is a fine example. The signature indicates that the type of the result varies with the type of the input list.

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

And of course you can use `List.find` to produce values of different types.

```
# List.find ~f:(fun x -> x > 3) [1;3;5;2];;
- : int option = Some 5
# List.find ~f:(Char.is_uppercase) ['a';'B';'C'];;
- : char option = Some B
```

But this approach is limited to simple dependencies between types that correspond to how data flows through your code. Sometimes you want types to vary in a more flexible way.

To make this concrete, let's say we wanted to create a version of `find` that is configurable in terms of how it handles the case of not finding an item. There are three different behaviors you might want:

- Throw an exception.
- Return `None`.
- Return a default value.

Let's try to write a function that exhibits these behaviors without using GADTs. First, we'll create a variant type that represents the three possible behaviors.

```
module If_not_found = struct
  type 'a t =
    | Raise
    | Return_none
    | Default_to of 'a
end
```

Now we can write `flexible_find`, which takes an `If_not_found.t` as a parameter and varies its behavior accordingly.

```
# let rec flexible_find list ~f (if_not_found : _ If_not_found.t) =
    match list with
    | hd :: tl ->
      if f hd then Some hd else flexible_find ~f tl if_not_found
    | [] ->
      (match if_not_found with
      | Raise -> failwith "Element not found"
      | Return_none -> None
      | Default_to x -> Some x);;
val flexible_find :
  'a list -> f:('a -> bool) -> 'a If_not_found.t -> 'a option = <fun>
```

Here are some examples of the above function in action:

```
# flexible_find ~f:(fun x -> x > 10) [1;2;5] Return_none;;
- : int option = None
# flexible_find ~f:(fun x -> x > 10) [1;2;5] (Default_to 10);;
- : int option = Some 10
```

```
# flexible_find ~f:(fun x -> x > 10) [1;2;5] Raise;;
Exception: (Failure "Element not found")
# flexible_find ~f:(fun x -> x > 10) [1;2;20] Raise;;
- : int option = Some 20
```

This mostly does what we want, but the problem is that `flexible_find` always returns an option, even when it's passed `Raise` or `Default_to`, which guarantees that the `None` case is never used.

To eliminate the unnecessary option in the `Raise` and `Default_to` cases, we're going to turn `If_not_found.t` into a GADT. In particular, we'll mint it as a GADT with two type parameters: one for the type of the list element, and one for the return type of the function.

```
module If_not_found = struct
  type (_, _) t =
    | Raise : ('a, 'a) t
    | Return_none : ('a, 'a option) t
    | Default_to : 'a -> ('a, 'a) t
end
```

As you can see, `Raise` and `Default_to` both have the same element type and return type, but `Return_none` provides an optional return value.

Here's a definition of `flexible_find` that takes advantage of this GADT.

```
# let rec flexible_find
   : type a b. f:(a -> bool) -> a list -> (a, b) If_not_found.t -> b =
   fun ~f list if_not_found ->
    match list with
    | [] ->
      (match if_not_found with
      | Raise -> failwith "No matching item found"
      | Return_none -> None
      | Default_to x -> x)
    | hd :: tl ->
      if f hd
      then (
        match if_not_found with
        | Raise -> hd
        | Return_none -> Some hd
        | Default_to _ -> hd)
      else flexible_find ~f tl if_not_found;;
val flexible_find :
  f:('a -> bool) -> 'a list -> ('a, 'b) If_not_found.t -> 'b = <fun>
```

As you can see from the signature of `flexible_find`, the return value now depends on the type of `If_not_found.t`, which means it can depend on the particular variant of `If_not_found.t` that's in use. As a result, `flexible_find` only returns an option when it needs to.

```
# flexible_find ~f:(fun x -> x > 10) [1;2;5] Return_none;;
- : int option = Base.Option.None
# flexible_find ~f:(fun x -> x > 10) [1;2;5] (Default_to 10);;
- : int = 10
# flexible_find ~f:(fun x -> x > 10) [1;2;5] Raise;;
Exception: (Failure "No matching item found")
```

```
# flexible_find ~f:(fun x -> x > 10) [1;2;20] Raise;;
- : int = 20
```

10.2.2 Capturing the Unknown

Code that works with unknown types is routine in OCaml, and comes up in the simplest of examples:

```
# let tuple x y = (x,y);;
val tuple : 'a -> 'b -> 'a * 'b = <fun>
```

The type variables `'a` and `'b` indicate that there are two unknown types here, and these type variables are *universally quantified*. Which is to say, the type of `tuple` is: *for all* types `a` and `b`, `a -> b -> a * b`.

And indeed, we can restrict the type of `tuple` to any `'a` and `'b` we want.

```
# (tuple : int -> float -> int * float);;
- : int -> float -> int * float = <fun>
# (tuple : string -> string * string -> string * (string * string));;
- : string -> string * string -> string * (string * string) = <fun>
```

Sometimes, however, we want type variables that are *existentially quantified*, meaning that instead of being compatible with all types, the type represents a particular but unknown type.

GADTs provide one natural way of encoding such type variables. Here's a simple example.

```
type stringable =
  Stringable : { value: 'a; to_string: 'a -> string } -> stringable
```

This type packs together a value of some arbitrary type, along with a function for converting values of that type to strings.

We can tell that `'a` is existentially quantified because it shows up on the left-hand side of the arrow but not on the right, so the `'a` that shows up internally doesn't appear in a type parameter for `stringable` itself. Essentially, the existentially quantified type is bound within the definition of `stringable`.

The following function can print an arbitrary `stringable`:

```
# let print_stringable (Stringable s) =
    Stdio.print_endline (s.to_string s.value);;
val print_stringable : stringable -> unit = <fun>
```

We can use `print_stringable` on a collection of `stringables` of different underlying types.

```
# let stringables =
    (let s value to_string = Stringable { to_string; value } in
      [ s 100 Int.to_string
      ; s 12.3 Float.to_string
      ; s "foo" Fn.id
      ]);;
val stringables : stringable list =
  [Stringable {value = <poly>; to_string = <fun>};
```

```
    Stringable {value = <poly>; to_string = <fun>};
    Stringable {value = <poly>; to_string = <fun>}]
# List.iter ~f:print_stringable stringables;;
100
12.3
foo
- : unit = ()
```

The thing that lets this all work is that the type of the underlying object is existentially bound within the type `stringable`. As such, the type of the underlying values can't escape the scope of `stringable`, and any function that tries to return such a value won't type-check.

```
# let get_value (Stringable s) = s.value;;
Line 1, characters 32-39:
Error: This expression has type $Stringable_'a
       but an expression was expected of type 'a
       The type constructor $Stringable_'a would escape its scope
```

It's worth spending a moment to decode this error message, and the meaning of the type variable `$Stringable_'a` in particular. You can think of this variable as having three parts:

- The `$` marks the variable as an existential.
- `Stringable` is the name of the GADT tag that this variable came from.
- `'a` is the name of the type variable from inside that tag.

10.2.3 Abstracting Computational Machines

A common idiom in OCaml is to combine small components into larger computational machines, using a collection of component-combining functions, or *combinators*.

GADTs can be helpful for writing such combinators. To see how, let's consider an example: *pipelines*. Here, a pipeline is a sequence of steps where each step consumes the output of the previous step, potentially does some side effects, and returns a value to be passed to the next step. This is analogous to a shell pipeline, and is useful for all sorts of system automation tasks.

But, can't we write pipelines already? After all, OCaml comes with a perfectly serviceable pipeline operator:

```
# open Core;;
# let sum_file_sizes () =
    Sys.ls_dir "."
    |> List.filter ~f:Sys.is_file_exn
    |> List.map ~f:(fun file_name -> (Unix.lstat file_name).st_size)
    |> List.sum (module Int) ~f:Int64.to_int_exn;;
val sum_file_sizes : unit -> int = <fun>
```

This works well enough, but the advantage of a custom pipeline type is that it lets you build extra services beyond basic execution of the pipeline, e.g.:

- Profiling, so that when you run a pipeline, you get a report of how long each step of the pipeline took.

- Control over execution, like allowing users to pause the pipeline mid-execution, and restart it later.
- Custom error handling, so, for example, you could build a pipeline that kept track of where it failed, and offered the possibility of restarting it.

The type signature of such a pipeline type might look something like this:

```
module type Pipeline = sig
  type ('input,'output) t

  val ( @> ) : ('a -> 'b) -> ('b,'c) t -> ('a,'c) t
  val empty : ('a,'a) t
end
```

Here, the type `('a,'b) t` represents a pipeline that consumes values of type `'a` and emits values of type `'b`. The operator `@>` lets you add a step to a pipeline by providing a function to prepend on to an existing pipeline, and `empty` gives you an empty pipeline, which can be used to seed the pipeline.

The following shows how we could use this API for building a pipeline like our earlier example using `|>`. Here, we're using a *functor*, which we'll see in more detail in Chapter 11 (Functors), as a way to write code using the pipeline API before we've implemented it.

```
# module Example_pipeline (Pipeline : Pipeline) = struct
    open Pipeline
    let sum_file_sizes =
      (fun () -> Sys.ls_dir ".")
      @> List.filter ~f:Sys.is_file_exn
      @> List.map ~f:(fun file_name -> (Unix.lstat file_name).st_size)
      @> List.sum (module Int) ~f:Int64.to_int_exn
      @> empty
  end;;
module Example_pipeline :
  functor (Pipeline : Pipeline) ->
    sig val sum_file_sizes : (unit, int) Pipeline.t end
```

If all we want is a pipeline capable of a no-frills execution, we can define our pipeline itself as a simple function, the `@>` operator as function composition. Then executing the pipeline is just function application.

```
module Basic_pipeline : sig
   include Pipeline
   val exec : ('a,'b) t -> 'a -> 'b
 end= struct
  type ('input, 'output) t = 'input -> 'output

  let empty = Fn.id

  let ( @> ) f t input =
    t (f input)

  let exec t input = t input
end
```

But this way of implementing a pipeline doesn't give us any of the extra services we

discussed. All we're really doing is step-by-step building up the same kind of function that we could have gotten using the |> operator.

We could get a more powerful pipeline by simply enhancing the pipeline type, providing it with extra runtime structures to track profiles, or handle exceptions, or provide whatever else is needed for the particular use-case. But this approach is awkward, since it requires us to pre-commit to whatever services we're going to support, and to embed all of them in our pipeline representation.

GADTs provide a simpler approach. Instead of concretely building a machine for executing a pipeline, we can use GADTs to abstractly represent the pipeline we want, and then build the functionality we want on top of that representation.

Here's what such a representation might look like.

```
type (_, _) pipeline =
  | Step : ('a -> 'b) * ('b, 'c) pipeline -> ('a, 'c) pipeline
  | Empty : ('a, 'a) pipeline
```

The tags here represent the two building blocks of a pipeline: `Step` corresponds to the `@>` operator, and `Empty` corresponds to the `empty` pipeline, as you can see below.

```
# let ( @> ) f pipeline = Step (f,pipeline);;
val ( @> ) : ('a -> 'b) -> ('b, 'c) pipeline -> ('a, 'c) pipeline = <fun>
# let empty = Empty;;
val empty : ('a, 'a) pipeline = Empty
```

With that in hand, we can do a no-frills pipeline execution easily enough.

```
# let rec exec : type a b. (a, b) pipeline -> a -> b =
   fun pipeline input ->
    match pipeline with
    | Empty -> input
    | Step (f, tail) -> exec tail (f input);;
val exec : ('a, 'b) pipeline -> 'a -> 'b = <fun>
```

But we can also do more interesting things. For example, here's a function that executes a pipeline and produces a profile showing how long each step of a pipeline took.

```
# let exec_with_profile pipeline input =
    let rec loop
        : type a b.
          (a, b) pipeline -> a -> Time_ns.Span.t list -> b *
    Time_ns.Span.t list
      =
     fun pipeline input rev_profile ->
      match pipeline with
      | Empty -> input, rev_profile
      | Step (f, tail) ->
        let start = Time_ns.now () in
        let output = f input in
        let elapsed = Time_ns.diff (Time_ns.now ()) start in
        loop tail output (elapsed :: rev_profile)
    in
    let output, rev_profile = loop pipeline input [] in
    output, List.rev rev_profile;;
```

```
val exec_with_profile : ('a, 'b) pipeline -> 'a -> 'b * Time_ns.Span.t
    list =
  <fun>
```

The more abstract GADT approach for creating a little combinator library like this has several advantages over having combinators that build a more concrete computational machine:

- The core types are simpler, since they are typically built out of GADT tags that are just reflections of the types of the base combinators.
- The design is more modular, since your core types don't need to contemplate every possible use you want to make of them.
- The code tends to be more efficient, since the more concrete approach typically involves allocating closures to wrap up the necessary functionality, and closures are more heavyweight than GADT tags.

10.2.4 Narrowing the Possibilities

Another use-case for GADTs is to narrow the set of possible states for a given data-type in different circumstances.

One context where this can be useful is when managing complex application state, where the available data changes over time. Let's consider a simple example, where we're writing code to handle a logon request from a user, and we want to check if the user in question is authorized to logon.

We'll assume that the user logging in is authenticated as a particular name, but that in order to authenticate, we need to do two things: to translate that user-name into a numeric user-id, and to fetch permissions for the service in question; once we have both, we can check if the user-id is permitted to log on.

Without GADTs, we might model the state of a single logon request as follows.

```
type logon_request =
  { user_name : User_name.t
  ; user_id : User_id.t option
  ; permissions : Permissions.t option
  }
```

Here, `User_name.t` represents a textual name, `User_id.t` represents an integer identifier associated with a user, and a `Permissions.t` lets you determine which `User_id.t`'s are authorized to log in.

Here's how we might write a function for testing whether a given request is authorized.

```
# let authorized request =
    match request.user_id, request.permissions with
    | None, _ | _, None ->
      Error "Can't check authorization: data incomplete"
    | Some user_id, Some permissions ->
      Ok (Permissions.check permissions user_id);;
val authorized : logon_request -> (bool, string) result = <fun>
```

The intent is to only call this function once the data is complete, i.e., when the `user_id` and `permissions` fields have been filled in, which is why it errors out if the data is incomplete.

The code above works just fine for a simple case like this. But in a real system, your code can get more complicated in multiple ways, e.g.,

- more fields to manage, including more optional fields,
- more operations that depend on these optional fields,
- multiple requests to be handled in parallel, each of which might be in a different state of completion.

As this kind of complexity creeps in, it can be useful to be able to track the state of a given request at the type level, and to use that to narrow the set of states a given request can be in, thereby removing some extra case analysis and error handling, which can reduce the complexity of the code and remove opportunities for mistakes.

One way of doing this is to mint different types to represent different states of the request, e.g., one type for an incomplete request where various fields are optional, and a different type where all of the data is mandatory.

While this works, it can be awkward and verbose. With GADTs, we can track the state of the request in a type parameter, and have that parameter be used to narrow the set of available cases, without duplicating the type.

A Completion-Sensitive Option Type

We'll start by creating an option type that is sensitive to whether our request is in a complete or incomplete state. To do that, we'll mint types to represent the states of being complete and incomplete.

```
type incomplete = Incomplete
type complete = Complete
```

The definition of the types doesn't really matter, since we're never instantiating these types, just using them as markers of different states. All that matters is that the types are distinct.

Now we can mint a completeness-sensitive option type. Note the two type variables: the first indicates the type of the contents of the option, and the second indicates whether this is being used in an incomplete state.

```
type (_, _) coption =
  | Absent : (_, incomplete) coption
  | Present : 'a -> ('a, _) coption
```

We use `Absent` and `Present` rather than `Some` or `None` to make the code less confusing when both `option` and `coption` are used together.

You might notice that we haven't used `complete` here explicitly. Instead, what we've done is to ensure that only an `incomplete coption` can be `Absent`. Accordingly, a `coption` that's `complete` (and therefore not `incomplete`) can only be `Present`.

This is easier to understand with some examples. Consider the following function for getting the value out of a `coption`, returning a default value if `Absent` is found.