2013-07-04
10:28:29

# Real World OCaml

*Jason Hickey, Anil Madhavapeddy, and Yaron Minsky*

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

2013-07-04
10:28:29

**Real World OCaml**
by Jason Hickey, Anil Madhavapeddy, and Yaron Minsky

Copyright © 2013 O'Reilly Media . All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

**Editor:** Andy Oram
**Production Editor:**
**Copyeditor:**
**Proofreader:** FIX ME!

**Indexer:**
**Cover Designer:**
**Interior Designer:** FIX ME!
**Illustrator:** Robert Romano

August 2013: First Edition.

**Revision History for the First Edition:**
YYYY-MM-DD First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449323912* for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. !!FILL THIS IN!! and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-32391-2

[?]

1372948108

2013-07-04
10:28:29

# Table of Contents

2013-07-04
10:28:29

2013-07-04
10:28:29

2013-07-04
10:28:29

2013-07-04
10:28:29

## Part II. Tools and Techniques

2013-07-04
10:28:29

2013-07-04
10:28:29

2013-07-04
10:28:29

2013-07-04
10:28:29

# Preface

## Conventions Used in This Book

The following typographical conventions are used in this book:

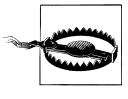
*Italic*
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

**`Constant width bold`**
: Shows commands or other text that should be typed literally by the user.

*`Constant width italic`*
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

## Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does

2013-07-04
10:28:29

require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

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

## Safari® Books Online


Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business. Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

## How to Contact Us

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

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

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

*http://www.oreilly.com/catalog/<catalog page>*

Don't forget to update the <url> attribute, too.

2013-07-04
10:28:29

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

*bookquestions@oreilly.com*

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

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

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

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

2013-07-04
10:28:29

2013-07-04
10:28:29

# Prologue

## Why OCaml?

The programming languages that you use affect the software you create. They influence your software's reliability, security and efficiency, and how easy it is to read, refactor, and extend. The languages you know can also deeply affect how you think about programming and software design.

But not all ideas about how to design a programming language are created equal. Over the last 40 years, a few key language features have emerged that together form a kind of sweet-spot in language design. These features include:

- *Garbage collection* for automatic memory management, now a feature of almost every modern high-level language.
- *Higher-order functions* that can be passed around as first-class values, as seen in Javascript or Scala.
- *Static type-checking* to reduce runtime errors, such as Java or Scala interfaces or variable type declarations in C#, Ada and Pascal.
- *Parametric polymorphism* to enable abstractions to be constructed across different datatypes, available as C++ templates or generics in Java or C#.
- *Immutable data structures* that cannot be destructively updated, famously enforced in Haskell but also a common feature of many distributed big data frameworks.
- *Algebraic datatypes* and *pattern matching* to define and manipulate complex data structures, available in Miranda, F# and Standard ML.
- *Automatic type inference* to avoid having to laboriously define the type of every single variable in a program, and instead have them inferred based on how a value is used. Available in Standard ML, F# and even modern C++11 via its `auto` keyword.

Some of you will know and love these features, and others will be completely new to them. 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, it turns out that there is something transformative about having them all together and able to interact in a single language.

Despite their importance, these ideas have made only limited inroads into mainstream languages and when they do arrive there, like higher-order functions in C# or parametric polymorphism in Java, it's typically in a limited and awkward form. The only languages that completely embody these ideas are statically-typed functional programming languages like OCaml, F#, Haskell, Scala and Standard ML.

Among this worthy set of languages, OCaml stands apart because it manages to provide a great deal of power while remaining highly pragmatic. The compiler has a straightforward compilation strategy without excessive optimization passes, and its strict evaluation model makes runtime behaviour easy to predict. The garbage collector is an incremental, precise implementation with no dynamic JIT compilation, and the runtime is simple and portable across platforms.

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

## A brief history from the 1960s

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 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 gradually turned into a fully fledged system of its own by the 1980s.

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 parsing system. Efficiency further improved with a fast native code compiler that made OCaml's performance competitive with mainstream languages such as C++. A module system inspired by Standard ML also provided powerful facilities for abstraction and larger scale programs.

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 OO 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 and variants and labelled and optional arguments.

The last decade has seen OCaml attract a significant user base, and language improvements have been steadily added to support the growing codebases that use the language both commercially and for academic use. First-class modules, Generalized Algebraic Data Types (GADTs) and dynamic linking have improved the flexibility of the lan-

guage, and there is fast native code support for x86_64, ARM, PowerPC and Sparc, making OCaml a good choice for systems where resource usage, predictability and performance all matter.

## The Core Standard Library

A language on its own isn't enough. You also need a rich set of libraries to base your applications on. A common source of frustration for those learning OCaml is that the standard library that ships with the compiler doesn't provide a lot of features. This standard library was actually developed for use within the compiler itself, and therefore by design covers only a small subset of the functionality you expect for more general-purpose use.

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

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

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

## The OCaml Platform

Core is a comprehensive and effective standard library, but there's a lot more out software out there. A large community of programmers have been using OCaml since its first release in 1996 and have generated a lot of useful libraries and tools.

In Real World OCaml, we'll introduce some of these libraries for you to experiment with realistic examples. The installation and management of these third-party libraries is made much easier via a package management tool known as OPAM. We'll explain more about OPAM as the book unfolds, but it forms the basis of the Platform, which is a set of tools and libraries that, along with the OCaml compiler, let you build realistic applications quickly and effectively.

Another big improvement in Core is 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 instead of the normal OCaml toplevel. It can, of course, be installed using OPAM, and Appendix A guides you through that process.

2013-07-04
10:28:29

# 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*. The world of dynamic scripting languages such as Javascript, Ruby and Python have all adopted healthy elements of functional programming, but not all of it. Real World OCaml takes you through the full lifecycle of how to construct software with static typing, including the powerful module system that makes code reuse so much more robust.

At the same time, OCaml is not Haskell. It takes a much more pragmatic approach by being strictly evaluated by default and permitting arbitrary side-effects. In fact, you can write OCaml code that looks very similar to imperative C, but that remains perfectly type-safe. One of the major strengths of OCaml for systems programming is that, with some experience, you can predict the runtime behaviour of a block of code very easily. We'll explain some of these tricks to you as we go through the book.

If you've learnt some OCaml before, this book may surprise you with some differences from your past experience. Core redefines most of the standard modules to be much more consistent, and so you'll need to adapt older code. We believe the Core model is worth learning; it's been successfully used on large, million-line codebases and removes a big barrier to more widespread OCaml adoption. There will always exist code that uses only the compiler standard library of course, but there are other online resources available to learn that. Real World OCaml focuses on the techniques the authors have used in their personal experience to construct scalable, robust computer systems.

## What to expect

Real World OCaml is split into three parts and appendices:

- Part I covers the basic language concepts you'll need to know when building OCaml programs. You won't need to memorise all of this (objects, for example, are used rarely in practice), but understanding the concepts and examples is important. This part opens up with a guided tour to give you a quick overview of the language. It then moves onto modules, functors and objects, which may take some time to digest. Persevere though; even though these concepts may be difficult at first, they will put you in good stead even when switching to other languages, many of which have drawn inspiration from ML.
- Part II builds on the basics by working through useful tools and techniques. Here you'll pick up useful techniques for building networked systems, as well as functional design patterns that help combine different features of the language to good effect. The focus throughout this section is on networked systems, and among other examples we'll build a running example that will perform Internet queries using the DuckDuckGo search engine.

- Part III is all about understanding the runtime system in OCaml. It's a remarkably simple system in comparison to other language runtimes (such as Java or the .NET CLR), and you'll need to read this to build very high performance systems that have to minimise resource usage or interface to C libraries. This is also where we talk about profiling and debugging techniques using tools such as GNU `gdb`.

Contributing your code back to the community is also important (if only to get bug fixes from other people!), and our appendices explain how to do this via OPAM and GitHub.

**Note to reviewers**

Real World OCaml uses some tools that we've developed while writing this book. Some of these resulted in improvements to the OCaml compiler, which means that you will need to ensure that you have an up-to-date development environment (using the 4.01.0 compiler). We've automated everything you need via the OPAM package manager, so please do follow the installation instructions in Appendix A carefully.

At this stage, the Windows operating system is also unsupported, and only Mac OS X, Linux, FreeBSD and OpenBSD can be expected to work reliably. We realize this is a concern; there are no fundamental barriers to Windows support, but we're focussed on getting the main content finished before getting stuck into the porting effort.

# About the Authors

## Jason Hickey

Jason Hickey is a Software Engineer at Google Inc. in Mountain View, California. He is part of the team that designs and develops the global computing infrastructure used to support Google services, including the software systems for managing and scheduling massively distributed computing resources.

Prior to joining Google, Jason was an Assistant Professor of Computer Science at Caltech, where his research was in reliable and fault-tolerant computing systems, including programming language design, formal methods, compilers, and new models of distributed computation. He obtained his PhD in Computer Science from Cornell University, where he studied programming languages. He is the author of the MetaPRL system, a logical framework for design and analysis of large software systems; and OMake, an advanced build system for large software projects. He is the author of the textbook, *An Introduction to Objective Caml* (unpublished).

2013-07-04
10:28:29

## Anil Madhavapeddy

Anil Madhavapeddy is a Senior Research Fellow at the University of Cambridge, based in the Systems Research Group. He was on the original team that developed the Xen hypervisor, and helped develop an industry-leading cloud management toolstack written entirely in OCaml. This XenServer product has been deployed on millions of physical hosts, and drives critical infrastructure for many Fortune 500 companies.

Prior to obtaining his PhD in 2006 from the University of Cambridge, Anil had a diverse background in industry at NetApp, NASA and Internet Vision. He is an active member of the open-source development community with the OpenBSD operating system, is on the steering committee of the Commercial Uses of Functional Programming ACM workshop, and serves on the boards of startup companies where OCaml is extensively used. He has also developed the Mirage unikernel system that is written entirely in OCaml from the device drivers up.

## Yaron Minsky

Yaron Minsky heads the Technology group at Jane Street, a proprietary trading firm that is the largest industrial user of OCaml. He was responsible for introducing OCaml to the company and for managing the company's transition to using OCaml for all of its core infrastructure. Today, billions of dollars worth of securities transactions flow each day through those systems.

Yaron obtained his PhD in Computer Science from Cornell University, where he studied distributed systems. Yaron has lectured, blogged and written about OCaml for years, with articles published in Communications of the ACM and the Journal of Functional Programming. He chairs the steering committee of the Commercial Users of Functional Programming, and is a member of the steering committee for the International Conference on Functional Programming.

# PART I

# Language Concepts

Part I covers the basic language concepts you'll need to know when building OCaml programs. You won't need to memorise all of this (objects, for example, are used rarely in practice) but understanding the concepts and examples is important.

This part opens up with a guided tour to give you a quick overview of the language using an interactive command-line interface. It then moves onto covering language features such as records, algebraic data types and the module system.

The final portion covers more advanced features such as functors, objects and first-class modules, which may all take some time to digest. Persevere though; even though these concepts may be difficult at first, they will put you in good stead even when switching to other languages, many of which have drawn inspiration from ML.

2013-07-04
10:28:29

2013-07-04
10:28:29

# CHAPTER 1
# A Guided Tour

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

We'll present this guided tour using the Core standard library and the `utop` OCaml toplevel, a shell that lets you type in expressions and evaluate them interactively. `utop` is an easier-to-use version of the standard toplevel (which you can start by typing `ocaml` at the command line). These instructions will assume you're using `utop` specifically.

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

### Installing `utop`

The easiest way to get the examples running is to set up the OPAM package manager, which is explained in Appendix A. In a nutshell, you need to have a working C compilation environment and the PCRE library installed, and then:

```
$ opam init
$ opam switch 4.01.0dev+trunk
$ opam install utop core_extended
$ eval `opam config -env`
```

Note that the above commands will take some time to run. When they're done, you should have a file called `~/.ocamlinit` in your home directory, to which you should add at least the following.

```
#use "topfind"
#camlp4o
#thread
```

```
#require "core.top"
#require "core.syntax"
```

Then type in `utop`, and you'll be in an interactive toplevel environment. OCaml phrases are only evaluated when you enter a double semicolon (`;;`), so you can split your typing over multiple lines. You can exit `utop` by pressing `control-d`. For complete instructions, please refer to Appendix A.

## OCaml as a calculator

Let's spin up `utop`. Throughout the book we're going to use Core, a more full-featured and capable replacement for OCaml's standard library. Accordingly, we'll start by opening the `Core.Std` module to get access to Core's libraries. If you don't open `Core.Std` many of the examples below will fail.

```
$ utop
# open Core.Std;;
```

Now that we have Core open, 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
# sqrt 9.;;
- : float = 3.
```

By and large, this is pretty similar to what you'd find in any programming language, but there are a few things that 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 stand-alone programs (though it is sometimes helpful to include `;;` to improve OCaml's error reporting).
- After evaluating an expression, the toplevel prints first the result and then the type of the result.
- 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 like C or Java.
- OCaml allows you to place underscores in the middle of your integer literals, as a way of improving 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 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 `0`, but `1 / 3.0` is a third. OCaml requires you to be explicit about which operation you're doing.

We can also create a variable to name the value of a given expression, using the `let` keyword (also 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
# let _x' = x' + x';;
# _x';;
- : int = 8
```

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

The following examples, however, are not legal.

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

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

## Functions and type inference

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

```
# 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. If no arguments are given, then we just have the ordinary definition of a variable that we saw earlier.

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

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

As a side note, the above is our first use of OCaml modules. Here, `Float.of_int` refers to the `of_int` function contained in the `Float` module, and not, as you might expect from an object-oriented language, accessing a method of an object. The `Float` module in particular contains `of_int` as well as many other useful functions for dealing with floats. It's also worth noting that module names always start with a capital letter.

The notation for the type-signature of a multi-argument function may be a little surprising at first, but we'll explain where it comes from when we get to function currying in "Multi-argument functions" on page 32. 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 even 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 mod 2 = 0 ;;
val even : int -> bool = <fun>
# sum_if_true even 3 4;;
- : int = 4
# sum_if_true even 2 4;;
- : int = 6
```

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

## Type inference

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

OCaml determines the type of an expression using a technique called *type inference*, by which it infers the type of a given expression based on what it already knows about the types of variables used in the expression, and on constraints on the types that arise from the structure of the code.

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

- OCaml requires that both arms of an `if` statement have the same type, so the expression `if test first then first else 0` requires that `first` must be the same type as `0`, which is `int`. Similarly, from `if test second then second else 0` we can conclude that `second` has type `int`.
- `test` is passed `first` as an argument. Since `first` has type `int`, the input type of `test` must be `int`.
- `test first` is used as the condition in an `if` statement, so the return type of `test` must be `bool`.
- 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. One way of making it easier to understand the types is to add 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. 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 actually go around any value in an OCaml program, and can be useful for figuring out why a given program is failing to compile.

## 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
  ;;
```

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

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

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.) 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*, and is very similar to generics in C# and Java.

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

```
# 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:

```
# 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";;
Characters 25-30:
  first_if_true big_number "short" "looooooong";;
                           ^^^^^^^
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 vs exceptions**

There's a big difference in OCaml (and really in any compiled language) between errors that are caught at compile time and those that are caught at run time. 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 run time and compile time errors, but that difference is still there. Generally, type errors, like this one:

```
# let add_potato x =
     x + "potato";;
  Characters 28-36:
       x + "potato";;
           ^^^^^^^^
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 mod y = 0 ;;
  val is_a_multiple : int -> int -> bool = <fun>
# is_a_multiple 8 2;;
```

```
- : bool = true
# is_a_multiple 8 0;;
Exception: Division_by_zero.
```

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

# Tuples, Lists, Options and Pattern Matching

## Tuples

So far we've encountered a handful of basic types like `int`, `float` and `string` as well as function types like `string -> int`. But we haven't yet talked about any data structures. We'll start by looking at a particularly simple data structure, the tuple. A tuple is an ordered collection of values that can each be of 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, the * character is used because the set of all pairs of type `t * s` corresponds to the Cartesian product of the set of elements of type `t` and the set of elements of type `s`.)

You can extract the components of a tuple using OCaml's pattern matching syntax. For example:

```
# 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, which 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 `float`s. The pattern matching syntax lets us get at the values we need with a minimum of fuss.

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

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

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

## Lists

Where tuples let you combine a fixed number of items, potentially of different types, lists let you hold any number of items of the same type. For 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, as we did with tuples.

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

### The `List` module

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

```
# 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. Note that `List.map` creates a new list and does not modify the original. Indeed, OCaml lists are immutable, so there are no operations for modifying lists in the language at all.

In this example, the function `String.length` is passed under the *labeled argument* `~f`. Labels allow you to specify arguments by name rather than by position. As you can see below, we can change the order of labeled arguments without changing the function's behavior.

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

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

#### Constructing lists with `::`

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

```
# "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 vs. 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 instead, 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.

Another thing that is uncovered by this example is 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 `::` operator can only be used for adding one element to the front of the list, with the list terminating at `[]`, the empty list. There's also a list concatenation operator, `@`, which can concatenate two lists.

```
# [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
  ;;
```

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.

If you try the above example in the toplevel, however, you'll see that it spits out a warning:

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

The warning indicates that the pattern is not exhaustive, meaning 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 non-empty list, and fails on empty ones.

```
# my_favorite_language ["English";"Spanish";"French"];;
```

```
- : string = "English"
# my_favorite_language [];;
Exception: Match_failure ("//toplevel//", 11, 10).
```

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

A `match` statement is a kind of juiced-up version of the switch statement found in `C` and `Java`. It essentially lets you list a sequence of patterns (separated by | characters --- the one before the first case is optional), and the compiler then dispatches to the code following the first matched pattern. And, as we've already seen, we can name new variables in our patterns that correspond to sub-structures 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"
```

Note that we included a comment in the above code. 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 (every list is either empty, or has at least one element), and the compiler can detect that exhaustiveness, which is why it doesn't spit out a warning.

#### Recursive list functions

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

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

```
# let rec sum l =
```

```
    match l with
    | [] -> 0                   (* base case *)
    | hd :: tl -> hd + sum tl   (* inductive case *)
  ;;
val sum : int list -> int
# 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 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 destutter list =
    match list with
    | [] -> []
    | hd1 :: hd2 :: tl ->
      if hd1 = hd2 then destutter (hd2 :: tl)
      else hd1 :: destutter (hd2 :: tl)
  ;;
```

Again, the first arm of the match is the base case, and the second is the inductive. Unfortunately, this code has a problem. If you type it into the toplevel, you'll see this error:

```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
_::[]
```

This indicates that we're missing a case, in particular we don't handle one-element lists. Note how the underscore is used to indicate the presence of a value without specifying what that value is.

We can fix this warning by adding another case to the match:

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

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

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

## 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, like `::` and `[]` for lists, which let you build optional values. You can think of an option as a specialized list that can only have zero or one element.

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

```
# let print_log_entry maybe_time message =
    let time =
      match maybe_time with
      | Some x -> x
      | None -> Time.now ()
    in
    printf "%s: %s\n" (Time.to_sec_string time) message ;;
val print_log_entry : Time.t option -> string -> unit
```

```
# print_log_entry (Some Time.epoch) "A long long time ago";;
1969-12-31 19:00:00: A long long time ago
- : unit = ()
# print_log_entry None "Up to the minute";;
2013-02-23 16:49:25: Up to the minute
- : unit = ()
```

We use a `match` statement for handling the two possible states of an option.

#### Nesting `lets` with `let` and `in`

As a side note, this is our first use of `let` to define a new variable within the body of a function. A `let` bounded 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 x = 7 in
  x + x
  ;;
- : int = 14
```

Note that the scope of the let binding is terminated by the double-semicolon.

We can also have multiple let statements 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.

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

## Records and Variants

So far, we've looked only at data structures that were predefined in the language, like lists and tuples. But OCaml also allows us to define new datatypes. Here's a toy example of a datatype representing a point in 2-dimensional space:

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

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

```
# 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 } =
    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*. When the name of the field and the name of the variable it is bound to in the match coincide, we don't have to write them both down. Using this, our magnitude function can be rewritten as follows.

```
# let magnitude { x; y } = sqrt (x ** 2. +. y ** 2.);;
```

We can also 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, as in the following types, each of which is a description of a different geometric object.

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

```
# 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 tag, like `Circle`, `Rect` and `Segment`, to distinguish that case from the others. Here's how we might write a function for testing whether a point is in the interior of some element of a list of `scene_element`s.

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

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

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

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

2013-07-04
10:28:29

# Imperative programming

So far, we've only written so-called *pure* or *functional* code, meaning that we didn't write any code that modified a variable or value after its creation. Indeed, almost all of the data structures we've encountered so far are *immutable*, meaning there's no way in the language to modify them at all. This is a quite different style from *imperative* programming, where computations are structured as sequences of instructions that operate by making modifications to the state of the program.

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

## Arrays

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

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

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

An odd type has cropped up in this example: `unit`. What makes `unit` different is that there is only one value of type `unit`, which is written `()`. Because there is only one value of type `unit`, that value doesn't really convey any information.

If it doesn't convey any information, then what is `unit` good for? Most of the time, `unit` acts as a placeholder. Thus, we use `unit` for the return value of an operation like setting a mutable field that operates by side effect rather than by returning a value. It's also used as the argument to functions that don't require an input value. This is similar to the role that `void` plays in languages like C and Java.

## Mutable record fields

The array is an important mutable data structure, but it's not the only one. Records, which are immutable by default, can be declared with specific fields as being mutable.

Here's a small example of a data structure for storing a running statistical summary of a collection of numbers. Here's the basic data structure:

```
# 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 below. (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 expressions.)

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

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

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

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

Note the use in the above code of single semi-colons 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 = 1.3333333333333326
```

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

## Refs

We can create a single mutable value by using a `ref`. The `ref` type comes pre-defined 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 = {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 }

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

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

Even though a `ref` is just another record type, it's important because it is the standard way of simulating the traditional mutable variable you'll find in most imperative 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
```

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

## For and while loops

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

```
# let permute array =
    let length = Array.length array in
    for i = 0 to length - 2 do
       (* pick a j that is after i and before the end of the array *)
       let j = i + 1 + Random.int (length - i - 1) in
       (* Swap i and j *)
       let tmp = array.(i) in
       array.(i) <- array.(j);
       array.(j) <- tmp
    done
  ;;
val permute : 'a array -> unit = <fun>
```

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

Here's an example run of this code.

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

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 above 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 above 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
  ;;
# find_first_negative_entry [|1;2;0;3|];;
Exception: (Invalid_argument "index out of bounds").
```

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

## A complete program

So far, we've played with the basic features of the language using the toplevel. Now we'll create a simple, complete stand-alone program that does something useful: sum 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.

```
(* file: sum.ml *)

open Core.Std

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

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

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

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

## Compiling and running

We can use `ocamlbuild` to compile the program. We'll need to create a file, in the same directory as `sum.ml`, called `_tags`. We can put the following in `_tags` to indicate that we're building against Core, and that threads should be enabled, which is required by Core.

```
true:package(core),thread
```

With our `_tags` file in place, we can build our executable by issuing this command.

```
ocamlbuild -use-ocamlfind sum.native
```

The `.native` suffix indicates that we're building a native-code executable, which we'll discuss more in Chapter 4. Once the build completes, we can use the resulting program like any command-line utility. In this example, we can just type in a sequence of numbers, one per line, hitting control-d to exit when the input is complete.

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

More work is needed to make a really usable command-line program, including a proper command-line parsing interface and better error handling, all of which is covered in Chapter 14.

## Where to go from here

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

2013-07-04
10:28:29

# CHAPTER 2
# Variables and Functions

Variables and functions are fundamental ideas that show up in virtually all programming languages. But OCaml has a different take on these basic concepts, so we'll spend some time digging into the details so you can understand OCaml's variables and functions and see how they differ from what you've encountered elsewhere.

## 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 into `utop` with the following syntax to bind a new variable. Note that variable names must start with a lowercase letter or an underscore.

```
let <identifier> = <expr>
```

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

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

Here's a simple example.

```
# 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 <identifier> = <expr1> in <expr2>
```

This first evaluates *expr1* and then evaluates *expr2* with *identifier* 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 top level, as we can see if we try to access it now.

```
# language_list;;
Characters 0-13:
  language_list;;
  ^^^^^^^^^^^^^
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 reappears.

```
# 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 = acos (-1.) in
     let area_of_circle r = pi *. r *. r in
     area_of_circle outer_radius -. area_of_circle inner_radius
  ;;
```

```
# area_of_ring 1. 3.;;
- : float = 25.1327412287183449
```

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 = acos (-1.) in
    let area_of_circle r = pi *. r *. r in
    let pi = 0. in
    area_of_circle outer_radius -. area_of_circle inner_radius
  ;;
```

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 you'd be wrong. In fact, the behavior of the function is unchanged. That's because the original definition of `pi` wasn't changed, it was just shadowed, so that any subsequent reference to `pi` would see the new definition of `pi` as zero. But there is no later use of `pi`, so the binding doesn't make a difference. Indeed, if you type the example above into the toplevel, OCaml will warn you that the definition is unused.

```
Characters 126-128:
    let pi = 0. in
        ^^
Warning 26: unused variable pi.
```

In OCaml, let bindings are immutable. As we'll see in Chapter 8, there are mutable values in OCaml, but no mutable variables.

**Why don't variables vary?**

One source of confusion for people new to functional languages 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 a functional language are really more like variables in an equation. If you think about the mathematical equation `x (y + z) = x y + x z`, there's no notion of mutating the variables `x`, `y` and `z`. They vary in the sense that you can instantiate this equation with different numbers for those variables, and it still holds.

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

## Pattern matching and `let`

Another useful feature of let bindings is that they support the use of *patterns* on the 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 identifiers to be bound. As we saw in "Tuples, Lists, Options and Pattern Matching" on page 10, 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 up-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)
  ;;
val upcase_first_entry : string -> string = <fun>
Characters 40-53:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
```

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

```
# 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 a function that is useful for throwing an exception in an impossible case. Asserts are discussed in more detail in Chapter 7

# Functions

OCaml being a functional language, it's no surprise that functions are an important and pervasive element of programming in OCaml. Indeed, we've seen functions pop up already in many of the examples we've looked at thus far. But while we've introduced the basics of functions, we're now going to cover them in more depth, starting from the foundations.

## 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 value that is declared without being named. They can be declared using the `fun` keyword, as shown here.

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

Anonymous functions aren't named, but they can be used for many different purposes nonetheless. You can, for example, 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]
```

Or even stuff them into a data structure.

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

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

The key thing to understand is that functions are ordinary values in OCaml, and you can do everything with them that you'd do with an ordinary value, including passing them to and returning them from other functions and storing them in data structures.

We even name functions in the same way that we name other values, by using a let binding.

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

Defining named functions is so common that there is a built in syntax for it. Thus, the following definition of `plusone` is equivalent to the definition above.

```
# 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 entirely 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 18.

## Multi-argument functions

OCaml of course also supports multi-argument functions, for example:

```
# 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 captured by the first function application, and `y`, which was captured by the second 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 above don't change the meaning of the signature, but it makes 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 definition above.

```
# let abs_diff = (fun x y -> abs (x - y));;
```

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 multi-argument 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 tradeoffs between these two approaches, but most of the time, one should stick to currying, since it's the default style in the OCaml world.

## Recursive functions

A function is *recursive* if it refers to itself in its definition. Recursion is important in any programming language, but is particularly important in functional languages, because it is the fundamental building block that is used for building looping constructs. (As we'll see in Chapter 8, OCaml also supports imperative looping constructs like `for` and `while`, but these are only useful when using OCaml's imperative features.)

In order to define a recursive function, you need to mark the let binding as recursive with the `rec` keyword, as shown in this example:

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

Note that in the above, the pattern `| [] | [_]` is what's called on *or-pattern*, which is the combination of two patterns. In this case, `[]`, matching the empty list, and `[_]`, matching 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 non-recursive definitions (using `let`) and recursive definitions (using `let rec`) largely for technical reasons: the type-inference algorithm needs

to know when a set of function definitions are mutually recursive, and for reasons that don't apply to a pure language like Haskell, these have to be marked explicitly by the programmer.

But this decision has some good effects. For one thing, recursive (and especially mutually recursive) definitions are harder to reason about than non-recursive definitions that proceed in order, each building on top of what has already been defined. It's therefore useful that, in the absence of an explicit marker, new definitions can only build upon ones that were previously defined.

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

## Prefix and Infix operators

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

```
# 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 above, we've partially applied `(+)` to gain a function that increments its single argument by `3`, and then applied that to all the elements of a list.

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:

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

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

We can define (or redefine) the meaning of an operator as follows. 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)
```

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

```
# let (***) x y = (x ** y) ** y;;
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. This table breaks the different operators and other syntactic forms into groups from highest to lowest precedence, explaining how each behaves syntactically. We write !... to indicate the class of operators beginning with !.

| Prefix | Usage |
|---|---|
| !..., ?..., ~... | Prefix |
| ., .(, .[ | |
| function application, constructor, assert, lazy | Left associative |
| -, -. | Prefix |
| **..., lsl, lsr, asr | Right associative |
| *..., /..., %..., mod, land, lor, lxor | Left associative |
| +, - | Left associative |
| :: | Right associative |
| @..., ^... | Right associative |
| =..., <..., >..., \|..., &..., $... | Left associative |
| &, && | Right associative |
| or, \|\| | Right associative |
| , | |
| <-, := | Right associative |
| if | |
| ; | Right associative |

There's one important special case: - and -., which are the integer and floating point subtraction operators, 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 below.

```
# Int.max 3 (-4);;
- : int = 3
# Int.max 3 -4;;
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;;
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 above. Here's the code.

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

It's not quite obvious at first what the purpose of this operator is: it just takes some value and a function, and applies the function to the value. But its utility is clearer when you see it in action. It works as a kind of sequencing operator, similar in spirit to using pipe in the UNIX shell. Consider, for example, the following code for printing out the unique elements of your `PATH`. Note that `List.dedup` below removes duplicates from a list by sorting the list using the provided comparison function.

```
# Sys.getenv_exn "PATH"
  |> String.split ~on:':'
  |> List.dedup ~compare:String.compare
  |> List.iter ~f:print_endline
  ;;
/bin
/opt/local/bin
/usr/bin
/usr/local/bin
- : unit = ()
```

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

```
# let path = Sys.getenv_exn "PATH" in
  let split_path = String.split ~on:':' path in
  let deduped_path = List.dedup ~compare:String.compare split_path in
  List.iter ~f:print_endline deduped_path
  ;;
/bin
/opt/local/bin
```

```
/usr/bin
/usr/local/bin
- : unit = ()
```

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

```
# 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 `|>` pipeline above.

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

```
# let (^!) = (|>);;
val ( ^! ) : 'a -> ('a -> 'b) -> 'b = <fun>
# Sys.getenv_exn "PATH"
  ^! String.split ~on:':'
  ^! List.dedup ~compare:String.compare
  ^! List.iter ~f:print_endline
  ;;
       Characters 93-119:
    ^! List.iter ~f:print_endline
       ^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: This expression has type string list -> unit
       but an expression was expected of type
         (string list -> string list) -> 'a
```

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

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

## Declaring functions with `function`

Another way to define a function is using the `function` keyword. Instead of having syntactic support for declaring multi-argument (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
  ;;
# some_or_default 3 (Some 5);;
- : int = 5
# List.map ~f:(some_or_default 100) [Some 3; None; Some 4];;
- : int list = [3; 100; 4]
```

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

## Labeled Arguments

Up until now, the functions we've defined have specified their arguments positionally, *i.e.*, by the order in which the arguments are passed to the function. OCaml also supports labeled arguments, which let you identify a function argument by name. Indeed, we've already encountered functions from Core like `List.map` that use labeled arguments. Labeled arguments are marked by a leading tilde, and a label (followed by a colon) are put in front of the variable to be labeled. Here's an example.

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

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

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

OCaml also supports *label punning*, meaning that you get to drop the text after the `:` if the name of the label and the name of the variable being used are the same. We've seen above how label punning works when defining a function. The following shows how it can be used when invoking a function.

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

Labeled arguments are useful in a few different cases:

- When defining a function with lots of arguments. Beyond a certain number, arguments are easier to remember by name than by position.
- When defining functions that have multiple arguments that might get confused with each other. This is most at issue when the arguments are of the same type. For example, consider this signature for a function for extracting a substring of another string.

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

  where the two ints are the starting position and length of the substring to extract. Labeled arguments can make this signature clearer:

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

  This improves the readability of both the signature and of client code that makes use of `substring`, and makes it harder to accidentally swap the position and the length.
- When the meaning of a particular argument is unclear from the type alone. For example, consider a function for creating a hash table where the first argument is the initial size of the table, and the second argument is a flag which, when true, indicates that the hash table will reduce its size when the hash table contains few elements. The following signature doesn't give you much of a hint as to the meaning of the arguments.

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

  but with labeled arguments, we can make the intent much clearer.

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

- When you want flexibility on the order in which arguments are passed. Consider a function like `List.iter`, that 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. This requires putting the function argument first.

```
# Sys.getenv_exn "PATH"
  |> String.split ~on:':'
  |> List.dedup ~compare:String.compare
  |> List.iter ~f:print_endline
  ;;
```

  In other cases, you want to put the function argument second. One common reason is readability. In particular, a multi-line function passed as an argument to another function is easiest to read when it is the final argument to that function.

#### Higher-order functions and labels

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

```
# 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 of listing 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);;
Characters 15-21:
  apply_to_tuple_2 divide (3,4);;
                   ^^^^^^
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
```

So, even though the order of labeled arguments usually doesn't matter, it will sometimes bite you in higher-ordered contexts, where you're passing functions as arguments to other functions as we were in the above examples.

## 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, optional 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 simple pairwise string concatenation.

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

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

In the above example, we had a bit of code to substitute in the empty string when no argument 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` even 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 that calls that function.

The downside is that the caller may be unaware that there is a choice to be made, and so may unknowingly (and wrongly) pick that default behavior. Optional arguments really only make sense when the extra concision of omitting the argument overwhelms the corresponding loss of explicitness.

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

#### Explicit passing of an optional argument

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

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

```
# 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 dimensions. The function takes an argument `delta` which determines the scale at which to compute the derivative, values `x` and `y` which determine which point to compute the derivative at, 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 =
  <fun>
```

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)
  ;;
Characters 131-132:
      let dx = (f ~y ~x:x' -. base) /. delta in
                ^
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:":";;
Characters 0-13:
  prepend_pound "a BASH comment" ~sep:":";;
  ^^^^^^^^^^^^^
Error: This function has type string -> string
       It is applied to too many arguments; maybe you forgot a `;'.
```

So when does OCaml decide to erase an optional argument?

The rule is: an optional argument is erased as soon as the first positional (*i.e.*, neither labeled nor optional) argument defined *after* the optional argument is passed in. That explains the behavior of `prepend_pound` above. 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 ;;
Characters 15-38:
  let concat x y ?(sep="") = x ^ sep ^ y ;;
                 ^^^^^^^^^^^^^^^^^^^^^^
Warning 16: this optional argument cannot be erased.
val concat : string -> string -> ?sep:string -> string = <fun>
```

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

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

# Exercises

## Typing

For each of the following expressions, is the expression well-typed? If it is well-typed, does it evaluate to a value? If so, what is the value?

- `1 - 2`

Well typed. The value is `-1`.

- `1 - 2 - 3`

Well typed. Subtraction is left-associative, so the value is `-4`.

- `1 - - 2`

Well typed. The value is `3`.

- `0b101 + 0x10`

Well typed. The value is `0x15` (`21` in decimal).

- `1073741823 + 1`

Well typed. On a 32-bit platform, `1073741823` is the maximum integer, so the value is `-1073741824`. On a 64-bit machine, the addition does not overflow, so the result is `1073741824`.

- `1073741823.0 + 1e2`

Ill typed. The operator `+` is for integer addition only.

- `1 ^ 1`

Ill typed. The operator `^` is string concatenation.

- `if true then 1`

Ill typed. The missing `else` branch has type `unit`, which is not compatible with `1`.

- `if false then ()`

Well typed. The result is `()`.

- `if 0.3 -. 0.2 = 0.1 then 'a' else 'b'`

Well-typed. On most platforms, `0.3 -. 0.2` is very close to, but different from, `0.1`, so the result is `'b'`.

- `true || (1 / 0 >= 0)`

Well-typed. The value is `true` (since disjunction `||` is a short-circuit operator).

- `1 > 2 - 1`

Well typed, because `-` has higher precedence than `>`. The result is `false`.

- `"Hello world".[6]`

Well typed. The value is `'w'`.

- `"Hello world".[11] <- 's'`

Well typed, but the index `11` is out of bounds, so the expression does not evaluate to a value.

- `String.lowercase "A" < "B"`

Well typed. The value is `false`.

- `Char.code 'a'`

Well typed. The ASCII character code for `'a'` is `97`.

- `(((())))`

Well typed. The value is the unit `()`.

- `((((*1*))))`

Well typed. The value is `()`.

- `((*(()*))`

Well typed. The value is `()`.

2013-07-04
10:28:30

# CHAPTER 3
# Lists and Patterns

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

## List Basics

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

```
# [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.

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

The `::` operator conveys something important about the nature of lists, which is that they are implemented as singly-linked lists. The following 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.

```
+---+---+   +---+---+   +---+---+
| 1 | *---->| 2 | *---->| 3 | *---->||
+---+---+   +---+---+   +---+---+
```

Each `::` essentially adds a new block to the picture above. 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 doesn't need to 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]
```

## Using patterns to extract data from a list

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

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

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

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

The fact that match statements can be used to bind new variables can be a source of confusion. To see how, imagine we wanted to write a function that filtered out from a list all elements equal to a particular value. You might be tempted to write that code as follows.

```
# 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
  ;;
```

But when we type this in, the compiler will immediately warn us that something is wrong.

```
Characters 114-122:
      | hd :: tl -> hd :: drop_value tl to_drop
        ^^^^^^^^
Warning 11: this match case is unused.
val drop_value : 'a list -> 'a -> 'a list = <fun>
```

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

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

So, what's going on?

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

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

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

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

```
# let rec drop_zero l =
    match l with
    | [] -> []
```

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

## Limitations (and blessings) of pattern matching

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

### Performance

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

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

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

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

Note the use of _ in the above match. This is a wild-card 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.Std;;
# [ Bench.Test.create ~name:"plus_one_match" (fun () ->
      ignore (plus_one_match 10))
  ; Bench.Test.create ~name:"plus_one_if" (fun () ->
      ignore (plus_one_if 10)) ]
  |> Bench.bench
  ;;
Estimated testing time 20s (change using -quota SECS).
+----------------+-----------+----------+
| Name           | Time (ns) | % of max |
|----------------+-----------+----------|
| plus_one_match |    369.38 |    83.40 |
| plus_one_if    |    442.88 |   100.00 |
+----------------+-----------+----------+
```

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

```
# 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 () -> ignore (sum_if numbers))
  ; Bench.Test.create ~name:"sum"    (fun () -> ignore (sum numbers)) ]
  |> Bench.bench
  ;;
Estimated testing time 20s (change using -quota SECS).
+--------+-----------+----------+
| Name   | Time (ns) | % of max |
|--------+-----------+----------|
| sum_if |   650_463 |   100.00 |
| sum    |   177_740 |    27.32 |
+--------+-----------+----------+
- : unit = ()
```

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

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

## Detecting errors

The error-detecting capabilities of match statements are if anything more important than their performance. We've already seen one example of OCaml's ability to find problems in a pattern match: in our broken implementation of `drop_value`, OCaml warned us that the final case was redundant. There are no algorithms for determining if a predicate written in a general-purpose language is redundant, but it can be solved reliably in the context of patterns.

OCaml also checks match statements for exhaustiveness. Consider what happens if we modify `drop_zero` by deleting the handler for one of the cases.

```
# let rec drop_zero l =
    match l with
    | [] -> []
    | 0  :: tl -> drop_zero tl
  ;;
```

The compiler will produce a warning that we've missed a case, along with an example of an unmatched pattern.

```
val drop_zero : int list -> 'a list = <fun>
Characters 26-84:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1::_
```

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

## Using the `List` module effectively

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

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

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

it would generate the following output.

```
| language | architect      | first release |
|----------+----------------+---------------|
| Lisp     | John McCarthy  | 1958          |
| C        | Dennis Ritchie | 1969          |
| ML       | Robin Milner   | 1973          |
| OCaml    | Xavier Leroy   | 1996          |
```

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 rev_map2_exn: 3 <> 4 ").
```

`List.fold` is the most complicated of the three, taking three arguments: a list to process, an initial accumulator value, and a function for updating the accumulator with the information from a list element. `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 list x -> x :: list) [1;2;3;4];;
- : int list = [4; 3; 2; 1]
```

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

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

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

**Performance of `String.concat` and ^**

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

```
let s = "." ^ "."  ^ "."  ^ "."  ^ "."  ^ "."  ^ "."
```

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

```
let s = String.concat [".";".";".";".";".";".";"."]
```

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 of 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 `pad` for padding out a string to a specified length plus one blank space on either side.

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

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

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

```
# render_row ["Hello";"World"] [10;15];;
- : string = "| Hello      | World           |"
```

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

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

## More useful list functions

The example we worked through above only touched on three of the function in `List`. We won't cover the entire interface, but there are a few more functions that 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, hwoever, you want something more that's simpler and thereby 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, one wants to restrict attention to just a subset of values. The `List.filter` function does just that.

```
# List.filter ~f:(fun x -> x mod 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. `List.filter_map` allows you to do just that. The function passed to `List.filter_map` returns an optional value, and `List.filter_map` drops all elements for which `None` is returned.

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

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

The above is also an example of an or-patterns, 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 function that is similar to `filter` is `partition_tf`, which takes a list and partitions it into a pair of lists based on a boolean condition. `tf` is a mnemonic to remind the reader that `true` elements go to the first bucket and `false` ones go to the second. Thus, one could write:

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

Note the use of a nested or-pattern in `is_ocaml_source`.

### Combining lists

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

```
# 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.

```
# let rec ls_rec s =
    if Sys.is_file_exn ~follow_symlinks:true s
    then [s]
    else
      Sys.ls_dir s
      |> List.map ~f:(fun sub -> ls_rec (s ^ "/" ^ sub))
      |> List.concat
  ;;
# ls_rec ".";;
- : string list =
["./_build/_digests"; "./_build/_log"; "./_build/example.ml";
 "./_build/example.ml.depends"; "./_build/ocamlc.where"; "./_tags";
 "./example.ml"]
```

The above 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 (s ^/ sub))
  ;;
val ls_rec : string -> string list = <fun>
```

## Tail recursion

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

```
# let rec length = function
```

```
    | [] -> 0
    | _ :: tl -> 1 + length tl
  ;;
# length [1;2;3];;
- : int = 3
```

This looks simple enough, but you'll discover that this implementation runs into problems on very large lists. Here are some examples, using another useful function from the `List` module, `List.init`, to create the lists. `List.init` 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.

```
# let make_list n = List.init n ~f:(fun x -> x);;
val make_list : int -> int list = <fun>
# make_list 10;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
# length (make_list 10_000_000);;
Stack overflow during evaluation (looping recursion?).
```

To understand what went wrong, 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 ten 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 of 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.

## More concise and faster patterns

Now that we know more about how lists and patterns work, let's consider how we can improve on an example from "Recursive list functions" on page 14: the function `destutter`, which removes sequential duplicates from a list. Here's the implementation that was described earlier.

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

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

First, let's consider efficiency. One problem with the `destutter` code above is that it in some cases recreates 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, which 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 sub-pattern. While we're at it, we'll use the `function` keyword to eliminate the need for an explicit match.

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

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

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

We can make the code slightly terser now by using a `when` clause. A `when` clause allows one to add an extra precondition on 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 destutter = function
    | [] | [_] as l -> l
    | hd :: (hd' :: _ as tl) when hd = hd' -> destutter tl
    | hd :: tl -> hd :: destutter tl
  ;;
val destutter : 'a list -> 'a list = <fun>
```

**Polymorphic compare**

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

```
# 3 = 4;;
- : bool = false
# [3;4;5] = [3;4;5];;
- : bool = true
```

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

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

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

OCaml actually 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 actually built-in to the runtime to a low level. These comparisons are polymorphic on the basis of ignoring almost everything about the types of the values that are being compared, paying attention only to the structure of the values as they're laid out in memory.

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

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

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

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

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

Note that `when` clauses have some downsides. As we noted earlier, the static checks associated with pattern matches rely on the fact that patterns are restricted in what they can express. Once we add the ability to add an arbitrary condition to a pattern, something will be lost. In particular, the ability for 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
  ;;
val count_some : 'a option list -> int = <fun>
Characters 30-169:
val count_some : 'a option list -> int = <fun>
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
_::_
(However, some guarded clause may match this value.)
```

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
  ;;
```

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
  ;;
```

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

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

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

CHAPTER 4

# Files, Modules and Programs

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

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

## Single File Programs

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

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

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

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

Now we can write down `freq.ml`.

```
(* freq.ml: basic implementation *)

open Core.Std

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

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

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

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

**Where is the main function?**

Unlike C, programs in OCaml do not have a unique `main` function. When an OCaml program is evaluated, all the statements in the implementation files are evaluated in the order in which they were linked together. These implementation files can contain arbitrary expressions, not just function definitions. In this example, the declaration starting with `let () =` plays the role of the `main` declaration, 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.

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

```
ocamlc freq.ml -o freq
```

But in this case, this command will fail with the error `Unbound module Core`. We need a somewhat more complex invocation to get Core linked in:

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

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

While this works well enough for a one-file project, more complicated builds will require a tool to orchestrate the build. One great tool for this task is `ocamlbuild`, which is shipped with the OCaml compiler. We'll talk more about `ocamlbuild` in Appendix B, but for now, we'll just walk through the steps required for this simple application. First, create a `_tags` file containing the following lines:

```
true:package(core),thread,annot,debugging
```

The purpose of the `_tags` file is to specify which compilation options are required for which files. In this case, we're telling `ocamlbuild` to link in the `core` package and to turn on threading, generation of annotation files, which are used by various editor modes for providing interactive inspection of the types inferred by the compiler, and debugging support. This is applied to all files, since the condition `true` evaluates to `true` on all files.

We can then invoke `ocamlbuild` to build the executable.

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

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

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

```
$ strings `which ocamlopt` | ./freq.byte
 13: movq
 10: cmpq
  8: ", &
  7: .globl
  6: addq
  6: leaq
  5: ", $
  5: .long
```

```
5: .quad
4: ", '
```

**Bytecode vs native code**

OCaml ships with two compilers: the `ocamlc` bytecode compiler and the `ocamlopt` native-code compiler. Programs compiled with `ocamlc` are interpreted by a virtual machine, while programs compiled with `ocamlopt` are compiled to native machine code to be run on a specific operating system and processor architecture.

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

As a general matter, production executables should usually be built using the native-code compiler, but it sometimes makes sense to use bytecode for development builds. And, of course, bytecode makes sense when targeting a platform not supported by the native-code compiler.

## Multi-file 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, for example, 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 describe the counts. The key function, called `touch`, updates the

association list with the information that a given line should be added to the frequency counts.

```
(* counter.ml: first version *)

open Core.Std

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

The file `counter.ml` will be compiled into a module named `Counter`. The name of the module is derived automatically from the filename. The module name is capitalized even if the file is not, and more generally module names must be capitalized.

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

```
(* freq.ml: using Counter *)
open Core.Std

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

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

## Signatures and Abstract Types

While we've pushed some of the logic to the `Counter` module, the code in `freq.ml` can still depend on the details of the implementation of `Counter`. Indeed, if you look at the definition of `build_counts`:

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

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 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.

```
(* filename: counter.mli *)
open Core.Std

val touch : (string * int) list -> string -> (string * int) list
```

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

**Auto-generating `mli` files**

If you don't want to construct an mli entirely by hand, you can ask OCaml to autogenerate one for you from the source, which you can then adjust to fit your needs. In this case, we can write:

```
$ ocamlbuild -use-ocamlfind counter.inferred.mli
```

Which will generate the file `_build/counter.inferred.mli`, with the following contents.

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

This is equivalent to the `mli` that we generated, but is a little more verbose. In general, you want to use autogenerated `mli`'s as a starting point only. There's no replacement for a careful consideration of what should be included in the interface of your module and of how that should be organized, documented and formatted.

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`:

```
(* counter.mli: abstract interface *)
open Core.Std
```

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

(** The empty set of frequency counts  *)
val empty : t

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

(* Converts the set of frequency counts to an association list.  Every strings
   in the list will show up at most once, and the integers will be at least
   1. *)
val to_list : t -> (string * int) list
```

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

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

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

```
(* counter.ml: implementation matching abstract interface *)

open Core.Std

type t = (string * int) list

let empty = []

let to_list x = x

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

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

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

This is because `freq.ml` depends on the fact that frequency counts are represented as association lists, a fact that we've just hidden. We just need to fix `build_counts` to use

`Counter.empty` instead of `[]` and `Counter.to_list` to get the association list out at the end for processing and printing. The resulting implementation is shown below.

```
(* filename: freq.ml *)
open Core.Std

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

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

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

```
(* counter.ml: efficient version *)

open Core.Std

type t = int String.Map.t

let empty = String.Map.empty

let to_list t = Map.to_alist t

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

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

## Concrete types in signatures

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

For example, imagine we wanted to add a function to `Counter` for returning the line with the median frequency count. If the number of lines is even, then there is no precise

median and the function would return the lines before and after the median instead. We'll use a custom type to represent the fact that there are two possible return values. Here's a possible implementation.

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

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

In the above, we use `failwith` to throw an exception for the case of the empty list. We'll discuss exceptions more in Chapter 7. Note also that the function `fst` simply returns the first element of any 2-tuple.

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

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

val median : t -> median
```

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

## Nested modules

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

A better approach is to mint new abstract types for each identifier, where those types are under the covers just implemented as strings. That way, the type system will prevent

you from confusing a username with a hostname, and if you do need to convert, you can do so using explicit conversions to and from the string type.

Here's how you might create such an abstract type, within a sub-module:

```
open Core.Std

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

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

The basic structure of a module declaration like this is:

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

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

```
(* file: buggy.ml *)
open Core.Std

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

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

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

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

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

The above 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.

```
File "buggy.ml", line 25, characters 12-19:
Error: This expression has type Hostname.t
       but an expression was expected of type Username.t
Command exited with code 2.
```

## Opening modules

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

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

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

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

Here's some general advice on how to deal with opens.

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

```
let average x y =
```

```
  let open Int64 in
  x + y / of_int 2
```

In the above, `of_int` and the infix operators are the ones from the `Int64` module.

There's another even more lightweight syntax for local opens, which is particularly useful for small expressions:

```
let average x y =
  Int64.(x + y / of_int 2)
```

- 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, instead of writing:

```
let print_median m =
   match m with
   | Counter.Median string -> printf "True median:\n   %s\n"
   | 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"
   | 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 toplevel of your module is usually a mistake.

## Including modules

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

```
# 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;;
Error: Unbound value Extended_interval.create
```

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

```
(* ext_list.ml: an extended list module *)

open Core.Std

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

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

Now, what about the interface of this new module? It turns out that include works on the signature language as well, so we can pull essentially the same trick to write an `mli` for this new module. The only trick is that we need to get our hands on the signature for the list module, which can be done using `module type of`.

```
(* ext_list.mli: an extended list module *)

open Core.Std

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

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

Note that the order of declarations in the `mli` does not need to match the order of declarations in the `ml`. Also, the order of declarations in the `ml` is quite important in that it determines what values are shadowed. If we wanted to replace a function in `List` with a new function of the same name, the declaration of that function in the `ml` would have to come after the `include List` declaration.

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

```
(* common.ml *)

module List = Ext_list
```

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

## Common errors with modules

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

### Type mismatches

The simplest kind of error is where the type specified in the signature does not match up with 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:

```
val touch : string -> t -> t
```

and then try to compile `Counter` (by writing `ocamlbuild -use-ocamlfind counter.cmo`. The `cmo` file is a compiled object file, containing the bytecode-compiled version of a module), we'll get the following error:

```
File "counter.ml", line 1, characters 0-1:
Error: The implementation counter.ml
       does not match the interface counter.cmi:
       Values do not match:
         val touch :
           ('a, int) Core.Std.Map.t -> 'a -> ('a, int) Core.Std.Map.t
       is not included in
         val touch : string -> t -> t
```

This error message is a bit intimidating at first, and it takes a bit of thought to see why the first type for touch (which comes from the implementation) doesn't match the second one (which comes from the interface). The key thing to remember is that `t` is a `Core.Std.Map.t`, at which point you can see that the error is a mismatch in the order of arguments to `touch`.

There's no denying that learning to decode such error messages is difficult at first, and takes some getting used to. But in time, decoding these errors becomes second nature.

## Missing definitions

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

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

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

```
File "counter.ml", line 1, characters 0-1:
Error: The implementation counter.ml
       does not match the interface counter.cmi:
       The field `count' is required but not provided
```

A missing type definition will lead to a similar error.

## Type definition mismatches

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

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

will lead to a compilation error:

```
File "counter.ml", line 1, characters 0-1:
```

```
Error: The implementation counter.ml
       does not match the interface counter.cmi:
       Type declarations do not match:
         type median = Before_and_after of string * string | Median of string
       is not included in
         type median = Median of string | Before_and_after of string * string
       Their first fields have different names, Before_and_after and Median.
```

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

## Cyclic dependencies

In most cases, OCaml doesn't allow cyclic dependencies, *i.e.*, a collection of definitions that all refer to each other. If you want to create such definitions, you typically have to mark them specially. For example, when defining a set of mutually recursive values (like the definition of `is_even` and `is_odd` in "Recursive functions" on page 34), 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 indeed, 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 case of this is that a module can not directly refer to itself (although definitions within a module can refer to each other in the ordinary way). So, if we tried to add a reference to `Counter` from within `counter.ml`:

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

then when we try to build, we'll get this error:

```
File "counter.ml", line 17, characters 18-31:
Error: Unbound module Counter
Command exited with code 2.
```

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

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

In this case, `ocamlbuild` will notice the error and complain:

```
Circular dependencies: "freq.cmo" already seen in
  [ "counter.cmo"; "freq.cmo" ]
```

2013-07-04
10:28:30

# CHAPTER 5
# Records

**A note to reviewers**

**IMPORTANT**: This section is going to describe how records and variants will behave in OCaml 4.01. If you want to follow along fully with the examples here, you'll need to make sure you install a bleeding-edge release, which you can do as follows:

```
opam switch 4.01.0dev+trunk
```

Once you make the switch, you'll need to install the packages you need. Note that switching back and forth between different compilers and their packages is a fast operation after the initial build.

One of OCaml's best features is its concise and expressive system for declaring new datatypes. Two key elements of that system are *records* and *variants*, both of which we discussed briefly in Chapter 1. In this chapter we'll cover records in more depth, covering more of the details of how they work, as well as advice on how to use them effectively in your software designs. Variants will be covered in more depth in Chapter 6.

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-name> : <type-name> ;
    <field-name> : <type-name> ;
    ...
  }
```

Note that record field names must start with a lower-case letter.

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

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

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

```
# #require "core_extended";;
# open Core_extended.Std;;
# let my_host =
    let sh = Shell.sh_one_exn in
    { hostname   = sh "hostname";
      os_name    = sh "uname -s";
      cpu_arch   = sh "uname -p";
      timestamp  = Time.now ();
    };;
val my_host : host_info =
  {hostname = "yevaud"; os_name = "Darwin"; cpu_arch = "i386";
   timestamp = 2013-06-29 11:05:48.358121+01:00}
```

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

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

```
# my_host.cpu_arch;;
- : string = "i386"
```

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

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

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

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

## Patterns and exhaustiveness

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

```
# let host_info_to_string { hostname = h; os_name = os;
                            cpu_arch = c; timestamp = ts;
                          } =
      sprintf "%s (%s / %s, on %s)" h os c (Time.to_sec_string ts);;
val host_info_to_string : host_info -> string = <fun>
# host_info_to_string my_host;;
- : string = "yevaud (Darwin / i386, on 2013-06-29 11:05:48)"
```

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

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

As an example, imagine that we wanted to add a new field to our `host_info` record called `os_release`, as shown below.

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

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

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

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

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

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, as shown below.

```
# let host_info_to_string { hostname = h; os_name = os;
                            cpu_arch = c; timestamp = ts; _
                          } =
    sprintf "%s (%s / %s, on %s)" h os c (Time.to_sec_string ts);;
val host_info_to_string : host_info -> string = <fun>
```

It's a good idea to enable the warning for incomplete record matches, and to explicitly disable it with an _ where necessary.

## 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 host_info_to_string { hostname; os_name; cpu_arch; timestamp; _ } =
     sprintf "%s (%s / %s) <%s>" hostname os_name cpu_arch
       (Time.to_string timestamp);;
  val host_info_to_string : host_info -> string = <fun>
```

Field punning can also be used to construct a record. Consider the following code for generating a `host_info` record.

```
# let my_host =
    let sh cmd = Shell.sh_one_exn cmd in
    let hostname   = sh "hostname" in
    let os_name    = sh "uname -s" in
    let cpu_arch   = sh "uname -p" in
    let os_release = sh "uname -r" in
    let timestamp  = Time.now () in
    { hostname; os_name; cpu_arch; os_release; timestamp };;
val my_host : host_info =
  {hostname = "yevaud"; os_name = "Darwin"; cpu_arch = "i386";
   os_release = "12.4.0"; timestamp = 2013-06-29 11:11:32.475361+01:00}
```

In the above 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, as shown below.

```
# let create_host_info ~hostname ~os_name ~cpu_arch ~os_release =
    { os_name; cpu_arch; os_release;
      hostname = String.lowercase hostname;
      timestamp = Time.now () };;
val create_host_info :
  hostname:string ->
  os_name:string -> cpu_arch:string -> os_release:string -> host_info = <fun>
```

This is considerably more concise than what you would get without punning at all.

```
# let create_host_info
    ~hostname:hostname ~os_name:os_name
    ~cpu_arch:cpu_arch ~os_release:os_release =
    { os_name = os_name;
      cpu_arch = cpu_arch;
      os_release = os_release;
      hostname = String.lowercase hostname;
      timestamp = Time.now () };;
val create_host_info :
  hostname:string ->
  os_name:string -> cpu_arch:string -> os_release:string -> host_info = <fun>
```

Together, labeled arguments, field names, and field and label punning, encourage a style where you propagate the same names throughout your code-base. This is generally good practice, since it encourages consistent naming, which makes it easier to navigate the source.

## Reusing field names

Defining records with the same field names can be problematic. Let's consider a simple example: building types to represent the protocol used for a logging server.

We'll describe three message types: `log_entry`, `heartbeat` and `logon`. The `log_entry` message is used to deliver a log entry to the server; the `logon` message is sent to initiate a connection, and includes the identity of the user connecting and credentials used for authentication; and the `heartbeat` message is periodically sent by the client to demonstrate to the server that the client is alive and connected. All of these messages include a session id and the time the message was generated.

```
# type log_entry =
    { session_id: string;
      time: Time.t;
      important: bool;
      message: string;
    }
  type heartbeat =
    { session_id: string;
```

```
      time: Time.t;
      status_message: string;
    }
  type logon =
    { session_id: string;
      time: Time.t;
      user: string;
      credentials: string;
    }
;;
```

Reusing field names can lead to some ambiguity. For example, if we want to write a function to grab the session_id from a record, what type will it have?

```
# 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_heatbeat_session_id (t:heartbeat) = t.session_id;;
val get_heatbeat_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);;
Error: The record type logon has no field status_message
# let session_and_status (t:heartbeat) = (t.session_id, t.status_message);;
val session_and_status : heartbeat -> string * string = <fun>
```

Why did the first definition succeed without a type annotation and the second one fail? The difference is that in the first case, the type-checker considered the `status_message` field first and thus concluded that the record was a `heartbeat`. When the order was switched, the `session_id` field was considered first, and so that drove the type to be considered to be a `logon`, at which point `t.status_message` no longer made sense.

We can avoid this ambiguity altogether, either by using non-overlapping field names or, more generally, by minting a module for each type. Packing types into modules is a broadly useful idiom (and one used quite extensively by Core), providing for each type a name-space within which to put related values. When using this style, it is standard practice to name the type associated with the module `t`. Using this style we would write:

```
# module Log_entry = struct
    type t =
```

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

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

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

The module name `Log_entry` is required to qualify the fields, because this function is outside of the `Log_entry` module where the record was defined. OCaml only requires the module qualification for one record field, however, so we can write this more concisely. Note that we are allowed to insert whitespace between the module-path and the field name.

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

This is not restricted to constructing a record; we can use the same trick when pattern matching.

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

When using dot-notation for accessing record fields, we can qualify the field by the module directly.

```
# let is_important t = t.Log_entry.important;;
val is_important : Log_entry.t -> bool = <fun>
```

The syntax here is a little surprising when you first encounter it. The thing to keep in mind is that the dot is being used in two ways: the first dot is a record field access, with everything to the right of the dot being interpreted as a field name; the second dot is accessing the contents of a module, referring to the record field `important` from within the module `Log_entry`. The fact that `Log_entry` is capitalized and so can't be a field name is what disambiguates the two uses.

For functions defined within the module where a given record is defined, the module qualification goes away entirely.

## Functional updates

Fairly often, you will find yourself wanting to create a new record that differs from an existing record in only a subset of the fields. For example, imagine our logging server had a record type for representing the state of a given client, including when the last heartbeat was received from that client. The following defines a type for representing this information, as well as a function for updating the client information when a new heartbeat arrives.

```
# type client_info =
   { addr: Unix.Inet_addr.t;
     port: int;
     user: string;
     credentials: string;
     last_heartbeat_time: Time.t;
   };;
# let register_heartbeat t hb =
      { addr = t.addr;
        port = t.port;
        user = t.user;
        credentials = t.credentials;
        last_heartbeat_time = hb.Heartbeat.time;
      };;
val register_heartbeat : client_info -> Heartbeat.t -> client_info = <fun>
```

This is fairly verbose, given that there's only one field that we actually want to change, and all the others are just being copied over from `t`. We can use OCaml's *functional update* syntax to do this more tersely. The syntax of a functional update is as follows.

```
{ <record> with <field> = <value>;
                <field> = <value>;
                ...
}
```

The purpose of the functional update is to create a new record based on an existing one, with a set of field changes layered on top.

Given this, we can rewrite `register_heartbeat` more concisely.

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

Functional updates make your code independent of the identity of the fields in the record that are not changing. This is often what you want, but it has downsides as well. In particular, if you change the definition of your record to have more fields, the type system will not prompt you to reconsider whether your update code should affect those 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.t;
     last_heartbeat_status: string;
   };;
```

The original implementation of `register_heartbeat` would now be invalid, and thus the compiler would 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>
```

## Mutable fields

Like most OCaml values, records are immutable by default. You can, however, declare individual record fields as mutable. For example, we could take the `client_info` type and make the fields that may need to change over time mutable, as follows.

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

```
    mutable last_heartbeat_status: string;
  };;
```

We then use the `<-` operator for actually changing the state. The side-effecting version of `register_heartbeat` would be written as follows.

```
# let register_heartbeat t hb =
    t.last_heartbeat_time   <- hb.Heartbeat.time;
    t.last_heartbeat_status <- hb.Heartbeat.status_message
  ;;
val register_heartbeat : client_info -> Heartbeat.t -> unit = <fun>
```

Note that mutable assignment, and thus the `<-` operator, is not needed for initialization, because all fields of a record, including mutable ones, are specified when the record is created.

OCaml's policy of immutable-by-default is a good one, but imperative programming does have its place. We'll discuss more about how (and when) to use OCaml's imperative features in "Imperative programming" on page 20.

## First-class fields

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

```
# let get_users logons =
    List.dedup (List.map logons ~f:(fun x -> x.Logon.user));;
val get_users : Logon.t list -> string list = <fun>
```

Here, we wrote a small function (`fun x -> x.Logon.user`) to access the `user` field. This kind of accessor function is a common enough pattern that it would be convenient to generate them automatically. The `fieldslib` syntax extension that ships with `Core` does just that.

You can enable the syntax extension by typing `#require "fieldslib.syntax"` into the top-level, at which point the `with fields` annotation at the end of the declaration of a record type will cause the extension to be applied to a given type declaration. So, for example, we could have defined `Logon` as follows.

```
# #require "fieldslib.syntax";;
# module Logon = struct
    type t =
      { session_id: string;
        time: Time.t;
        user: string;
        credentials: string;
      }
    with fields
  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 (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 sub-module 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`, which returns the name of a field
- `Field.get`, which returns the content of a field
- `Field.fset`, which does a functional update of a field
- `Field.setter`, which 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, first parameter of the `Field.t` corresponds to the record you pass to `get`, and the second argument 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 above, as you can see below.

```
# Field.get;;
- : ('b, 'r, 'a) Field.t_with_perm -> 'r -> 'a = <fun>
```

The type is `Field.t_with_perm` rather than a simple `Field.t` because fields have a notion of access control associated with them because there are some special cases where we may expose the ability to read a field but not the ability to do a functional update.

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) Field.t -> ('b -> string) -> 'a -> 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.now ();
                user = "yminsky";
                credentials = "Xy2d9W"; }
  ;;
# show_field Logon.Fields.user Fn.id logon;;
- : string = "user: yminsky"
# show_field Logon.Fields.time Time.to_string logon;;
- : string = "time: 2012-06-26 18:44:13.807826"
```

As a side note, the above 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 iterate over all 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.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.to_string)
      ~user:(print Fn.id)
      ~credentials:(print Fn.id)
  ;;
val print_logon : Logon.t -> unit = <fun>
# print_logon logon;;
session_id: 26685
time: 2012-06-26 18:44:13.807826
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.

This exhaustion guarantee is a valuable one. 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, and such applications can benefit from the guarantee that all fields of the record type in question have been considered.

2013-07-04
10:28:30

2013-07-04
10:28:30

# CHAPTER 6
# Variants

Variant types are one of the most useful features of OCaml, and also one of the most unusual. They let you represent data that may take on multiple different forms, where each form is marked by an explicit tag. As we'll see, when combined with pattern matching, variants give you a powerful way of representing complex data and of organizing the case-analysis on that information.

The basic syntax of a variant type declaration is as follows.

```
type <variant-name> =
  | <Tag> [ of <type> [* <type>]... ]
  | <Tag> [ of <type> [* <type>]... ]
  | ...
```

Each row starts with a tag that identifies that case, and in addition, there may be a collection of fields, each with its own type, that is associated with a given tag.

Let's consider a concrete example of how variants can be useful. Almost all terminals support a set of 8 basic colors, and we can represent those colors using a variant. Each color is declared as a simple tag, with pipes used to separate the different cases. Note that variant tags must be capitalized.

```
# type basic_color =
   | Black | Red | Green | Yellow | Blue | Magenta | Cyan | White ;;
# Cyan ;;
- : basic_color = Cyan
# [Blue; Magenta; Red] ;;
- : basic_color list = [Blue; Magenta; Red]
```

The following function uses pattern matching to convert a `basic_color` to a corresponding integer. The exhaustiveness checking on pattern matches means that the compiler will warn us if we miss a color.

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

```
val basic_color_to_int : basic_color -> int = <fun>
# List.map ~f:basic_color_to_int [Blue;Red];;
- : int list = [4; 1]
```

Using the above, we can generate escape codes to change the color of a given string displayed in a terminal.

```
# let color_by_number number text =
    sprintf "\027[38;5;%dm%s\027[0m" number text;;
  val color_by_number : int -> string -> string = <fun>
# let blue = color_by_number (basic_color_to_int Blue) "Blue";;
val blue : string = "\027[38;5;4mBlue\027[0m"
# printf "Hello %s World!\n" blue;;
Hello Blue World!
- : 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 a simple enumeration. Indeed, 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 8 basic colors, in regular and bold versions.
- A 6 × 6 × 6 RGB color cube
- A 24-level grayscale ramp

We'll also represent this more complicated color-space as a variant, but this time, the different tags will have arguments which 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 *)
;;
# [RGB (250,70,70); Basic (Green, Regular)];;
- : color list = [RGB (250, 70, 70); Basic (Green, Regular)]
```

Once again, we'll use pattern matching to convert a color to a corresponding integer. But in this case, the pattern matching does more than separate out the different cases; it also allows us to extract the data associated with each tag.

```
# 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 = ()
```

#### Catch-all cases and refactoring

OCaml's type system can act as a refactoring tool, by 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 ;;
Characters 40-60:
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 flag 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 ;;
Characters 19-154:
Warning 8: this pattern matching is not exhaustive.
Here is an example of a value 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 8 `basic_color`s in regular and bold) in the normal way, but rendering 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>
```

But because the catch-all case encompasses all possibilities, the type system will no longer warn us that we have missed the new `Bold` case when we change the type to include it. We can get this check back by avoiding the catch-all case, and instead being explicit about the tags that are ignored.

## 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 peculialrly 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 combination of sums and products. Let's see what we can achieve with this by revisiting the logging server types that were described in Chapter 5. We'll start by reminding ourselves of the definition of `Log_entry.t`.

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

This record type combines multiple pieces of data into one 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 acting as conjunctions. Variants, on the other hand, are disjunctions, letting you represent multiple possibilities, as in the following example.

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

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 dealt with.

You can increase the precision of your types by using variants to represent differences between types, and records to represent shared structure. Consider the following function that takes a list of `client_message`s 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:([],String.Set.empty)
        ~f:(fun ((messages,user_sessions) as acc) message ->
          match message with
          | Logon m ->
```

```
          if m.Logon.user = user then
            (message::messages, Set.add user_sessions m.Logon.session_id)
          else acc
        | Heartbeat _ | Log_entry _ ->
          let session_id = match message with
            | Logon     m -> m.Logon.session_id
            | Heartbeat m -> m.Heartbeat.session_id
            | Log_entry m -> m.Log_entry.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>
```

There's one awkward bit about the code above, which is the calculation of the session ids. In particular, we have the following repetitive snippet of code:

```
let session_id = match message with
  | Logon     m -> m.Logon.session_id
  | Heartbeat m -> m.Heartbeat.session_id
  | Log_entry m -> m.Log_entry.session_id
in
```

This code effectively computes the session id for each underlying message type. The repetition in this case isn't that bad, but would become problematic in larger and more complicated examples. Also, we had to include code for the `Logon` case, even though it can't actually come up.

We can improve the code by refactoring our types to explicitly separate the parts that are shared from those that are common. The first step is to cut down the definitions of the per-message records to just contain the unique components of each message.

```
# 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 covers the different possible unique components.

```
# 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.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 example above as follows:

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

Note that the more complex match statement for computing the session id has been replaced with the simple expression `common.Common.session_id`.

In addition, this design allows us to essentially downcast to the specific message type once we know what it is, and then 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,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)
;;
```

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

## 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 a `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 above, 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 such an expression. The following code shows how you could write a general-purpose evaluator for these expressions.

```
# 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. The following simplification code is based on having some simplifying constructors that mirror the tags used to construct a tree. Then the simplification function is reponsible for rebuilding the tree using these constructors.

```
# let and_ l =
    if List.mem l (Const false) then Const false
    else
      match List.filter l ~f:((<>) (Const true)) with
      | [] -> Const true
      | [ x ] -> x
      | l -> And l

  let or_ l =
    if List.mem l (Const true) then Const true
    else
      match List.filter l ~f:((<>) (Const false)) with
```

```
      | [] -> Const false
      | [x] -> x
      | l -> Or l

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

Now, we can write a simplification routine that brings these together.

```
# 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 of 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 non-trivial 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 handling 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.

## 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 above 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];;
Characters 14-18:
  [three; four; five];;
```

```
                ^^^^
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 `[> `Int of string | `Float of float]` as describing a variant whose tags include `` `Int of string `` and `` `Float of float ``, 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 -> 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 ``.

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 different upper and lower bounds. Note that `Ok` and `Error` in the following example come from the `Result.t` type from Core.

```
# let is_positive = function
    | `Int   x -> Ok (x > 0)
    | `Float x -> Ok (x > 0.)
    | `Not_a_number -> Error "not a number";;
val is_positive :
  [< `Float of float | `Int of int | `Not_a_number ] ->
  (bool, string) Result.t = <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.

## 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
  ;;
```

This looks reasonable enough, but it leads to the following type error.

```
Characters 93-98:
    | (Basic _ | RGB _ | Gray _) as color -> color_to_int color
                                                      ^^^^^
Error: This expression has type extended_color
       but an expression was expected of type color
```

The problem is that `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

  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 basic_color_to_int :
  [< `Black | `Blue | `Cyan | `Green | `Magenta | `Red | `White | `Yellow ] ->
  int = <fun>
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 above 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
  ;;
      Characters 125-130:
      | color -> color_to_int color
                             ^^^^^
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
```

**Polymorphic variants and catch-all cases**

As we saw with the definition of `is_positive`, a match statement 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 statement, we end up with a function with a lower bound.

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

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 4. Let's start with the `mli`.

```
(* file: terminal_color.mli *)

open Core.Std

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.

```
(* file: terminal_color.ml *)

open Core.Std

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 above code, we did something funny to the definition of `extended_color_to_int`, that underlines 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`.

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.

```
File "color.ml", line 29, characters 4-11:
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
```

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.

## 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:* As we've seen, 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 11, subtyping brings in a lot of complexity, and most of the time, that's complexity you want to avoid.

2013-07-04
10:28:30

# CHAPTER 7
# 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 planning out 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 code 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.

## 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.

```
# List.find;;
- : 'a list -> f:('a -> bool) -> 'a option
```

The option in the return type indicates that the function may not succeed in finding a suitable element, as you can see below.

```
# 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
```

Having errors be explicit in the return values of your functions tells the caller that there is an error that needs to be handled. The caller can then handle the error explicitly, either recovering from the error or propagating it onward.

Consider the `compute_bounds` function defined below. The function 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 ~cmp list =
    let sorted = List.sort ~cmp list in
    match List.hd sorted, List.last sorted with
    | None,_ | _, None -> None
    | Some x, Some y -> Some (x,y)
  ;;
val compute_bounds :
  cmp:('a -> 'a -> int) -> 'a list -> ('a * 'a) option = <fun>
```

The match statement 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 `find_mismatches` below, 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, 'b) Hashtbl.t -> ('a, 'b) Hashtbl.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.

## 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 promoted to the toplevel by `Core.Std`. As such, we can write:

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

without first opening the `Result` module.

## `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?

Core's answer to this question is the `Error.t` type, which tries to forge a good compromise between efficiency, convenience, and control over the presentation of errors.

It might not be obvious at first why efficiency is an issue at all. But generating error messages is an expensive business. An ASCII representation of a value can be quite time-consuming to construct, particularly if it includes expensive-to-convert numerical data.

`Error` gets around this issue through laziness. In particular, an `Error.t` allows you to put off generation of the error string until and unless you need it, which means a lot of the time you never have to construct it at all. You can of course construct an error directly from a string:

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

But you can also construct an `Error.t` from a *thunk*, *i.e.*, a function that takes a single argument of type `unit`.

```
# Error.of_thunk (fun () ->
    sprintf "something went wrong: %f" 32.3343);;
- : Error.t = something went wrong: 32.334300
```

In this case, we can benefit from the laziness of `Error`, since the thunk won't be called unless the `Error.t` is converted to a string.

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. Thus, the following is a simple s-expression:

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

S-expressions are supported by the `sexplib` package that is distributed with Core, and is the most common serialization format used in Core. Indeed, most types in Core come with built-in s-expression converters. Here's an example of creating an error using the sexp converter for times, `Time.sexp_of_t`.

```
# Error.create "Something failed a long time ago" Time.epoch Time.sexp_of_t;;
- : Error.t =
Something failed a long time ago: (1970-01-01 01:00:00.000000+01:00)
```

Note that the time isn't actually serialized into an s-expression until the error is printed out. We're not restricted to doing this kind of error reporting with built-in types. This will be discussed in more detail in Chapter 17, but Sexplib comes with a language extension that can auto-generate sexp-converters for newly generated types, as shown below.

```
# let custom_to_sexp = <:sexp_of<float * string list * int>>;;
val custom_to_sexp : float * string list * int -> Sexp.t = <fun>
# custom_to_sexp (3.5, ["a";"b";"c"], 6034);;
- : Sexp.t = (3.5 (a b c) 6034)
```

We can use this same idiom for generating an error.

```
# Error.create "Something went terribly wrong"
    (3.5, ["a";"b";"c"], 6034)
    <:sexp_of<float * string list * int>> ;;
- : Error.t = Something went terribly wrong: (3.5(a b c)6034)
```

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

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

The type `'a Or_error.t` is just a shorthand for `('a,Error.t) Result.t`, and it is, after `option`, the most common way of returning errors in Core.

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

The above 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 sub-interface called `Monad`, which we'll talk about more in Chapter 18.

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

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 ~cmp list =
    let sorted = List.sort ~cmp list in
    Option.both (List.hd sorted) (List.last sorted)
  ;;
val compute_bounds : cmp:('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 similar functionality is available in both `Result` and `Or_error`.

## 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 sub-computations.

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 part way through it's execution, never getting to the end of the list.

```
# List.map ~f:(fun x -> 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, as you can see below.

```
# let exceptions = [ Not_found; Division_by_zero; Key_not_found "b" ];;
val exceptions : exn list = [Not_found; Division_by_zero; Key_not_found("b")]
# List.filter exceptions  ~f:(function
    | Key_not_found _ | Not_found -> true
    | _ -> false);;
- : exn list = [Not_found; Key_not_found("b")]
```

All exceptions are of type `exn`, and that type is a similar to a variant type of the kind we encountered in Chapter 6. The biggest difference is that it is an open type, meaning that new tags can be added at any time, by any part of the program. As such, you can never have a match on an exception that is guaranteed to exhaustively list all values.

Here's an example of a function for looking up a key in an *association list*, *i.e.* a list of key/value pairs which uses this newly-defined exception:

```
# let rec find_exn alist key = match alist with
    | [] -> raise (Key_not_found key)
    | (key',data) :: tl -> if 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 Core.

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

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

The return type of `'a` suggests that `raise` could return a value of any type. That seems impossible, and it is. Really, `raise` has this type 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 is an infinite loop.

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

#### Declaring exceptions using `with sexp`

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

```
# exception Wrong_date of Date.t;;
exception Wrong_date of Date.t
# Wrong_date (Date.of_string "2011-02-23");;
- : exn = Wrong_date(_)
```

But if we declare the exception using `with sexp` (and the constituent types have sexp converters), we'll get something with more information.

```
# exception Wrong_date of Date.t with sexp;;
exception Wrong_date of Core.Std.Date.t
# Wrong_date (Date.of_string "2011-02-23");;
- : exn = (.Wrong_date 2011-02-23)
```

The period in front of `Wrong_date` is there because the representation generated by `with sexp` includes the full module path of the module where the exception in question is defined. In this case, since we've declared the exception at the toplevel, that module path is trivial.

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 17.

## Helper functions for throwing exceptions

OCaml and Core provide 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 Core.

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 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 15).
```

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

## Exception handlers

So far, we've only seen exceptions fully terminate the execution of a computation. But often, 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` statement. 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 statements 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.

## 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 code snippet:

```
let load_config filename =
  let inc = In_channel.create filename in
  let config = Config.t_of_sexp (Sexp.input_sexp inc) in
  In_channel.close inc;
  config
```

The problem with this code is that the function that loads the s-expression and parses it into a `Config.t` might throw an exception if the config 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 Core's `protect` function. The purpose of `protect` is to ensure that the `finally` thunk will 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 `load_config`.

```
let load_config filename =
  let inc = In_channel.create filename in
  protect ~f:(fun () -> Config.t_of_sexp (Sexp.input_sexp inc))
    ~finally:(fun () -> In_channel.close inc)
```

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

```
let load_config filename =
  In_channel.with_file filename ~f:(fun inc ->
    Config.t_of_sexp (Sexp.input_sexp inc))
```

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

## 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, `List.find_exn` throws `Not_found` when

the element in question can't be found. You can take advantage of this in your code, for example, let's define a function called `lookup_weight`, with the following signature.

```
val lookup_weight :
  compute_weight:('data -> float) -> ('key * 'data) list -> 'key -> float
```

`lookup_weight ~compute_weight alist key` should return a floating-point weight by applying `compute_weight` to the data associated with `key` by `alist`. If `key` is not found, then it should return 0.

We can implement `lookup_weight` as follows.

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

Let's think about the behavior of this code in the presence of exceptions. 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 `Not_found`, then that's not what will happen:

```
# lookup_weight ~compute_weight:(fun _ -> raise Not_found)
    ["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 generally better 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 (List.Assoc.find_exn alist key)
      with _ -> None
    with
    | None -> 0.
    | Some data -> compute_weight data ;;
val lookup_weight :
  compute_weight:('a -> float) ->
  ('b, 'a) Core.Std.List.Assoc.t -> 'b -> float = <fun>
```

At which point, it makes sense to simply use the non-exception throwing function, `List.Assoc.find`, instead.

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

## 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.

```
(* exn.ml *)

open Core.Std
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 give you some information about where the error occurred, and the stack of function calls that were in place at the time of the error.

```
$ ./exn.byte
3
Fatal error: exception Exn.Empty_list
Raised at file "exn.ml", line 7, characters 16-26
Called from file "exn.ml", line 12, characters 17-28
```

You can also capture a backtrace within your program by calling `Exn.backtrace`, 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. Core reverses the default, so if you're linking in Core, you will have backtraces enabled at runtime.

Even using Core and compiling with debugging symbols, you can turn backtraces off by setting the `OCAMLRUNPARAM` environment variable to be empty.

```
$ export OCAMLRUNPARAM=
```

```
$ ./exn.byte
3
Fatal error: exception Exn.Empty_list
```

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

```
(* file: exn_cost.ml *)

open Core.Std
open Core_bench.Std

let simple_computation () =
  List.range 0 10
  |> List.fold ~init:0 ~f:(fun sum x -> sum + x * x)
  |> ignore

let simple_with_handler () =
  try simple_computation () with Exit -> ()

let end_with_exn () =
  try
    simple_computation ();
    raise Exit
  with Exit -> ()

let () =
  [ Bench.Test.create ~name:"simple computation"
      (fun () -> simple_computation ());
    Bench.Test.create ~name:"simple computation w/handler"
      (fun () -> simple_with_handler ());
    Bench.Test.create ~name:"end with exn"
      (fun () -> end_with_exn ());
  ]
  |> Bench.make_command
  |> Command.run
```

We're testing three cases here: a simple computation with no exceptions; the same computation with an exception handler but no thrown exceptions; and finally the same computation where we use the exception to do the control flow back to the caller.

If we run this with stacktraces on, the benchmark results look like this.

```
$ ./exn_cost.native cycles
Estimated testing time 30s (change using -quota SECS).
┌──────────────────────────┬────────┬─────────────┬──────────┐
│ Name                     │ Cycles │ Time (ns)   │ % of max │
├──────────────────────────┼────────┼─────────────┼──────────┤
│ simple computation       │ 198.32 │      116.66 │    78.36 │
│ simple computation w/handler │ 219.23 │      128.96 │    86.62 │
```

```
| end with exn                  | 253.10 |     148.88 |    100.00 |
└──────────────────────────────┴────────┴────────────┴───────────┘
```

Here, we see that we lose something like 20 cycles to adding an exception handler, and 30 more to actually throwing and catching an exception. If we turn backtraces off, then the results look like this.

```
$ ./exn_cost.native cycles
Estimated testing time 30s (change using -quota SECS).
┌──────────────────────────────┬────────┬────────────┬───────────┐
| Name                         | Cycles | Time (ns) | % of max |
├──────────────────────────────┼────────┼────────────┼───────────┤
| simple computation           | 198.84 |     116.97 |     83.86 |
| simple computation w/handler | 217.17 |     127.75 |     91.60 |
| end with exn                 | 237.10 |     139.47 |    100.00 |
└──────────────────────────────┴────────┴────────────┴───────────┘
```

Here, the handler costs about the same, at 20 cycles, but the exception itself costs only 20, as opposed to 30 additional cycles. All told, this should only matter if you're using exceptions routinely as part of your flow control, which is in most cases a stylistic mistake anyway.

## 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, Core 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 = None
# find ["a",1; "b",2] "b";;
- : int option = Some 2
```

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

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

And then we can re-raise that exception:

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

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

## 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 tradeoff 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 same 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 tradeoff depends on your application. If you're writing a rough and ready program where getting to 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 old maxim of "use exceptions for exceptional conditions" applies. If an error occurs sufficiently rarely, then throwing an exception may well be the right behavior.

Also, for errors that are omnipresent, error-aware return types may also 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. And 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 forseeable and ordinary part of the execution of your production code and that are not omnipresent, error aware return types are typically the right solution.

2013-07-04
10:28:30

2013-07-04
10:28:30

CHAPTER 8

# Imperative Programming

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 where you need it. This chapter will walk you through OCaml's imperative features, and help you use them to their fullest.

## 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 really for illustration purposes; both Core and the standard library provide imperative dictionaries, and for most real world tasks, you should use one of those implementations. There's more advice on using Core's implementation in particular in Chapter 13.

Our dictionary, like those in Core and the standard library, will be implemented as a hash table. In particular, we'll use an *open hashing* scheme, which is to say 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 interface we'll match, provided as an `mli`. Here, the type `('a, 'b) t` is used for a dictionary with keys of type `'a` and data of type `'b`.

```
(* file: dictionary.mli *)
open Core.Std

type ('a, 'b) t

val create : unit -> ('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
```

The `mli` also includes a collection of helper functions whose purpose and behavior should be largely inferrable from their names and type signatures. Notice that a number of the functions, in particular, ones like `add` that modify the dictionary, return unit. This is typical of functions that act by side-effect.

We'll now walk through the implementation (contained in the corresponding `ml` file) 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 with two fields.

```
(* file: dictionary.ml *)
open Core.Std

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

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, as we'll see.

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

```
let num_buckets = 17

let hash_bucket key = (Hashtbl.hash key) mod num_buckets

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

```
  }

let length t = t.length

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

Note that `num_buckets` is a constant. That's because, for simplicity's sake, we're using a fixed-length bucket array. For a practical implementation, the length of the array would have to be able to grow as the number of elements in the dictionary increases.

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. It is implemented on top of `Hashtbl.hash`, which is a hash function provided by the OCaml runtime that can be applied to values of any type. Thus, its own type is polymorphic: `'a -> int`.

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 bit of syntax has popped up in `find`: we write `array.(index)` to grab a value from an array. Also, `find` uses `List.find_map`, which you can see the type of by typing it into the toplevel:

```
# 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 the value returned by `f` is returned by `find_map`. If `f` returns `None` on all values, then `None` is returned by `find_map`.

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 the context of imperative code.

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',_) -> key' = key)

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

This above 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 a single semicolon, `;`, as a sequencing operator, to allow us to do a sequence of side-effecting operations in a row: first, update the bucket, then update the count. We could have done this using let bindings:

```
  let () = t.buckets.(i) <- filtered_bucket in
  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), 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.

# 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.

## 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 below.

### 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.(index)
```

and for setting an element in an array:

```
array.(index) <- 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.

#### Strings

Strings are essentially byte-arrays which are often used for textual data. The main advantage of using a `string` in place of a `Char.t array` (a `Char.t` is an 8-bit character) is that the former is considerably more space efficient; an array uses one word --- 8 bytes on a 64-bit machine --- to store a single entry, whereas strings use one byte per character.

Strings also come with their own syntax for getting and setting values: `string.[index]` and `string.[index] <- expr` respectively, and string literals are bounded by quotes. There's also a module `String` where you'll find useful functions for working with strings.

#### 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 20. Bigarrays too have their own getting and setting syntax: `bigarray.{index}` and `bigarray.{index} <- expr`. There is no literal syntax for bigarrays.

## 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 11, 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 }
```

The standard library defines the following operators for working with refs.

- `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 below.

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

The above are just ordinary OCaml functions which could be defined as follows.

```
let ref x = { contents = x }
let (!) r = r.contents
let (:=) r x = r.contents <- x
```

### 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, like LAPACK bindings.

## `for` and `while` loops

OCaml provides support for traditional imperative looping constructs, in particular, `for` and `while` loops, even though neither of them is strictly necessary. Anything you can do with such a loop you can also do with a recursive function, and you can also write higher-order functions like `Array.iter` that cover much of the same ground.

Nonetheless, explicit `for` and `while` loops are both more idiomatic for imperative programming and often more concise.

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`.

```
# 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 = ()
```

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 *)
      incr i;
      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 above, we used `incr` and `decr`, which are build-in functions for incrementing and decrementing an `int ref` by one, respectively.

## 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.

```
(* file: dlist.mli *)
open Core.Std

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`.

```
(* file: dlist.ml *)
open Core.Std

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 = !t = None
```

```
let value elt = elt.value

let first t = !t
let next elt = elt.next
let prev elt = elt.prev
```

These all follow relatively straight-forwardly 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 afterwards.

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; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1;
   2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2; 3; 1; 2;
   ...]
```

This approach is quite limited, however. General purpose cyclic data structures require mutation.

## 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
  begin match !t with
  | Some old_first -> old_first.prev <- Some new_elt
  | None -> ()
  end;
  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 `begin ... end`. We could have used parenthesis for the same purpose. 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
  begin match elt.next with
  | Some old_next -> old_next.prev <- Some new_elt
  | None -> ()
  end;
  elt.next <- Some new_elt;
  new_elt
```

Finally, we need a `remove` function.

```
let remove t elt =
  let { prev; next; _ } = elt in
  begin match prev with
  | Some prev -> prev.next <- next
  | None -> t := next
  end;
  begin match next with
  | Some next -> next.prev <- prev;
  | None -> ()
  end;
  elt.prev <- None;
  elt.next <- None
```

Note that the above 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 above 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 that the code you write is careful about error detection.

## 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 values 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
```

## 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, without really changing anything else. 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 effect 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 prefix any expression, returning a value of type `'a Lazy.t`. The evaluation of that expression is delayed until forced with the `Lazy.force` function.

```
# let v = lazy (print_string "performing lazy computation\n"; 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
```

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 `ref` containing a `lazy_state`, where the `ref` 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 based on 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 create_lazy f = ref (Delayed f);;
val create_lazy : (unit -> 'a) -> 'a lazy_state ref = <fun>
# let v = create_lazy
    (fun () -> print_string "performing lazy computation\n"; sqrt 16.);;
  val v : float lazy_state ref = {contents = Delayed <fun>}
```

Now we just need a way to force a lazy value. The following function does just that.

```
# let force v =
    match !v with
    | Value x -> x
    | Exn e -> raise e
    | Delayed f ->
      try
        let x = f () in
        v := Value x;
        x
      with exn ->
        v := Exn exn;
        raise exn
   ;;
val force : 'a lazy_state ref -> 'a = <fun>
```

Which we can use in the same way we used `Lazy.force`:

```
# force v;;
performing lazy computation
- : float = 4.
# force v;;
- : float = 4.
```

The main user-visible difference between our implementation of laziness and the built-in version is syntax. Rather than writing `create_lazy (fun () -> sqrt 16.)`, we can with the built-in `lazy` just write `lazy (sqrt 16.)`.

## 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 Core's `Hashtbl` module, rather than our toy `Dictionary`.

```
# let memoize f =
    let table = Hashtbl.Poly.create () in
    (fun x ->
      match Hashtbl.find table x with
      | Some y -> y
      | None ->
        let y = f x in
        Hashtbl.add_exn table ~key:x ~data:y;
        y
    );;
val memoize : ('a -> 'b) -> 'a -> 'b = <fun>
```

The code above is a bit tricky. `memoize` takes as its argument a function `f`, and then allocates a hash table (called `table`) and returns a new function as the memoized version of `f`. When called, this new function looks in `table` first, and if it fails to find a value, calls `f` and stashes the result in `table`. Note that `table` doesn't go out of scope as long as the function returned by `memoize` is in scope.

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: every time you create a memoized function, there's something of a built-in memory leak. 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 spell checkers.

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 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:

```
edit_distance "OCam" "ocaml"
   edit_distance "OCa" "ocaml"
   edit_distance "OCam" "ocam"
   edit_distance "OCa" "ocam"
edit_distance "OCaml" "ocam"
   edit_distance "OCam" "ocam"
   edit_distance "OCaml" "oca"
   edit_distance "OCam" "oca"
edit_distance "OCam" "ocam"
   edit_distance "OCa" "ocam"
   edit_distance "OCam" "oca"
   edit_distance "OCa" "oca"
```

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.

```
# let time f =
    let start = Time.now () in
    let x = f () in
    let stop = Time.now () in
    printf "Time: %s\n" (Time.Span.to_string (Time.diff stop start));
```

```
    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: 5.11003ms
- : int = 2
# time (fun () -> edit_distance "OCaml 4.01" "ocaml 4.01");;
Time: 19.3322s
- : int = 2
```

Just those few extra characters made it almost four thousand 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. This technique is sometimes referred to as *dynamic programming*. To see how to do this, let's step away from `edit_distance`, and instead consider a much simpler example: computing the nth element of the Fibonacci sequence. The Fibonacci sequence by definition starts out with two `1`'s, 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 1 else fib (i - 1) + fib (i - 2);;
```

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: 5.17392ms
- : int = 10946
# time (fun () -> fib 40);;
Time: 51.4205s
- : int = 165580141
```

Here, `fib 40` takes almost a minute to compute, as opposed to five *milliseconds* for `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 (though of course repeated calls will be improved).

```
# let fib = memoize fib;;
val fib : int -> int = <fun>
# time (fun () -> fib 40);;
Time: 52.6s
- : int = 165580141
# time (fun () -> fib 40);;
```

```
Time: 0.00596046ms
- : int = 165580141
```

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, as shown below.

```
# let rec fib i = fib_norec fib i;;
val fib : int -> int = <fun>
# fib 5;;
- : int = 8
```

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 5;;
- : int = 8
```

This is a pretty strange piece of code, and it may take a few minutes of thought to figure out what's going on. Like `fib_norec`, the function `f_norec` passed into `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 variant on `make_rec` that inserts memoization when it ties the recursive knot. We'll call that function `memo_rec`.

```
# let memo_rec f_norec x =
     let fref = ref (fun _ -> assert false) in
     let f = memoize (fun x -> f_norec !fref x) in
     fref := f;
     f x
  ;;
val memo_rec : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
```

Note that `memo_rec` has 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 fib_norec;;
val fib : int -> int = <fun>
# time (fun () -> fib 40);;
Time: 0.236034ms
```

And 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 to memo_rec does not trigger a call to `memoize`. Only when the final argument to `fib` is presented does `memoize` get called, and the result of that call falls out of scope when the `fib` call returns. That means that, unlike ordinary memoization, 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 (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 in to `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`:

```
# let edit_distance = memo_rec (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 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>
```

This new version of `edit_distance` is much more efficient than the one we started with; the following call is about ten thousand times faster than it was without memoization.

```
# time (fun () -> edit_distance ("OCaml 4.01","ocaml 4.01"));;
Time: 2.14601ms
- : 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 f_norec =
     let rec f = memoize (fun x -> f_norec f x) in
     f
  ;;
      Characters 41-72:
       let rec f = memoize (fun x -> f_norec f x) in
```

```
                                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
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 to execute this code. In some sense, you could imagine it compiling down to an infinite loop, but there's no looping control structure to make that happen.

To avoid such 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 (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.

```
# Lazy.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 f_norec x =
     let rec f = lazy (memoize (fun x -> f_norec (Lazy.force f) x)) in
     (Lazy.force f) x
  ;;
val lazy_memo_rec : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
# time (fun () -> lazy_memo_rec fib_norec 40);;
Time: 0.298977ms
- : 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.

# 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 Core. 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 18. Most of the functionality in Core's `In_channel`, `Out_channel` (and in Core's `Unix` module) derives from the standard library, but we'll use Core's interfaces here.

## 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. `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 global name-space 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 timezone, and then prints out the current time in that timezone. Here, we use Core's `Zone` module for looking up a timezone, and the `Time` module for computing the current time and printing it out in the timezone in question.

```
(* file: time_converter.ml *)
```

```
open Core.Std

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 = Zone.find_exn zone_string in
    let time_string = Time.to_localized_string (Time.now ()) zone in
    Out_channel.output_string stdout
      (String.concat
         ["The time in ";Zone.to_string zone;" is "; time_string;"\n"]);
    Out_channel.flush stdout
```

We can build this program (using `ocamlbuild` with the `_tags` file described in "Single File Programs" on page 67) and run it, you'll see that it prompts you for input, as follows:

```
$ ./time_converter.byte
Pick a timezone:
```

You can then type in the name of a timezone and hit return, and it will print out the current time in the timezone in question.

```
Pick a timezone: Europe/London
The time in Europe/London is 2013-03-06 02:15:13.602033
```

We called `Out_channel.flush` on `stdout` because `out_channel`s 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 flush is nonetheless good practice.

## 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 describe 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 = ()
```

Importantly, and 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;;
Characters 26-29:
  printf "An integer: %i\n" 4.5;;
                            ^^^
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 format strings, unlike their equivalent in C, are 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, %F is a float, \"%s\" is a string\n";;
val fmt : string = "%i is an integer, %F is a float, \"%s\" is a string\n"
# printf fmt 3 4.5 "five";;
Characters 7-10:
  printf fmt 3 4.5 "five";;
         ^^^
Error: This expression has type string but an expression was expected of type
         ('a -> 'b -> 'c -> 'd, out_channel, unit) format =
           ('a -> 'b -> 'c -> 'd, out_channel, 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:

```
# let fmt : ('a, 'b, 'c) format =
```

```
  "%i is an integer, %F is a float, \"%s\" is a string\n";;
val fmt : (int -> float -> string -> 'c, 'b, 'c) format = <abstr>
```

And accordingly, we can pass it to `printf`.

```
# printf fmt 3 4.5 "five";;
3 is an integer, 4.5 is a float, "five" is a string
- : 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 worry 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`.

```
(* file: time_converter.ml *)
open Core.Std

let () =
  printf "Pick a timezone: %!";
  match In_channel.input_line stdin with
  | None -> failwith "No timezone provided"
  | Some zone_string ->
    let zone = Zone.find_exn zone_string in
    let time_string = Time.to_localized_string (Time.now ()) zone in
    printf "The time in %s is %s.\n%!" (Zone.to_string zone) time_string
```

In the above 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.

## File I/O

Another common use of `in_channel`s and `out_channel`s is for working with files. Here's 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 -> 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 code above 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: \"##\"").
```

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";;
Exception: (Sys_error "numbers.txt: 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 7, as follows.

```
# let sum_file filename =
     let file = In_channel.create filename in
```

```
      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") with _ -> () done;;
- : unit = ()
# sum_file "numbers.txt";;
- : int = 15
```

This is really an example of a more general complexity of imperative programming. When programming imperatively, you need to be quite careful to make sure that exceptions don't leave you in an awkward state.

`In_channel` also supports some idioms that handle some of the details of this for you. For example, the `with_file` function takes a filename and a function for processing that file, and takes care of the opening and closing of the file transparently.

```
# 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 for those modules.

## 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 defined. 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 = sin 120. in
  let y = sin 75.  in
  let z = sin 128. in
  List.exists ~f:(fun x -> 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 (sin 120.) in
  let y = lazy (sin 75.)  in
  let z = lazy (sin 128.) in
  List.exists ~f:(fun x -> 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"; sin 120.) in
  let y = lazy (printf "2\n"; sin 75.)  in
  let z = lazy (printf "3\n"; sin 128.) in
  List.exists ~f:(fun x -> 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.

In a strict language, 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 -> x < 0.)
    [ (printf "1\n"; sin 120.);
      (printf "2\n"; sin 75.);
      (printf "3\n"; sin 128.); ]
  ;;
3
2
1
- : bool = true
```

Here, you can see that the sub-expression 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 sub-expressions, you should express them as a series of `let` bindings.

## 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)
  ;;
```

`remember` simply caches the first value that's passed to it, returning that value on every call. Note that we've carefully written `remember` so that `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 type should it have?

On its first call, `remember` returns the same value its passed, which means its input type and return type should match. Accordingly, `remember` should have the 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. Here's the type that OCaml infers for `remember`, which looks almost, but not quite, like the type of the identity function.

```
val remember : '_a -> '_a = <fun>
```

The underscore in the type variable `'_a` 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 it can only be used with one 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, as you can see below.

```
# let remember_three () = remember 3;;
val remember_three : unit -> int = <fun>
# remember;;
- : int -> int = <fun>
# remember "avocado";;
Characters 9-18:
  remember "avocado";;
           ^^^^^^^^^
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!

## 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 where it's sure there are no persistent refs, 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 simple values by their nature can't contain refs, including:

- Constants (*i.e.*, things like integer and floating point literals)
- Constructors that contain only 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 above definition, we'll get different results. For example, consider what happens if we try to memoize the function defined above.

```
# memoize (fun x -> [x;x]);;
- : '_a -> '_a list = <fun>
```

The memoized version of the function does in fact need to be restricted to a single type, because it uses mutable state behind the scenes to cache previous invocations of the function it has passed. But OCaml would make the same determination even if the function in question did no such thing. Consider this example:

```
# identity (fun x -> [x;x]);;
- : '_a -> '_a list = <fun>
```

It would be safe to infer a weakly 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.

## 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 -> '_a) -> '_a list = <fun>
```

As you can see, we now infer a weakly polymorphic type for the resulting function, and for good reason. There's nothing that tells us 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 using explicit variables rather than 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.

## Relaxing the value restriction

OCaml is actually a little better at inferring polymorphic types than is implied above. The value restriction as we described it above is basically a syntactic check: there are a few operations that you can do 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 some use of type information to allow polymorphic types for things that are not simple values.

For example, we saw above 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]);;
- : '_a -> '_a 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 potentially mutable, then the result will be weakly polymorphic.

```
# [||];;
- : 'a array = [||]
# identity [||];;
- : '_a 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;;
- : '_a 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 variance and covariance in Chapter 11, but for now, you can think of it as an annotation which can be put in the interface of a pure data structure.

Thus, 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.

```
# identity Concat_list.empty;;
- : 'a Concat_list.t = <abstr>
```

2013-07-04
10:28:30

2013-07-04
10:28:30

# CHAPTER 9
# Functors

Up until now, we've seen OCaml's modules play an important but limited role. In particular, we've seen them as a mechanism for organizing code into units with specified interfaces. But OCaml's module system can do much more than that, serving as a powerful tool for building generic code and structuring large-scale systems. Much of that power comes from functors.

Functors are, roughly speaking, functions from modules to modules, and they can be used to solve a variety of code-structuring problems, including:

- *Dependency injection*, or making the implementations of some components of a system swappable. This is particularly useful when you want to mock up parts of your system for testing and simulation purposes.
- *Auto-extension of modules*. Functors give you a way of extending existing modules with new functionality in a standardized way. For example, you might want to add a slew of comparison operators derived from a base comparison function. To do this by hand would require a lot of repetitive code for each type, but functors let you write this logic just once and apply it to many different types.
- *Instantiating modules with state*. Modules can contain mutable state, and that means that you'll occasionally want to have multiple instantiations of a particular module, each with its own separate and independent mutable state. Functors let you automate the construction of such modules.

## A trivial example

Let's create a functor that takes a module containing a single integer variable `x`, and returns a new module with `x` incremented by one. This is not actually a useful example, but it's a good way to walk through the basic mechanics of functors.

First, let's define a signature for a module that contains a single value of type `int`.

```
# module type X_int = sig val x : int end;;
module type X_int = sig val x : int end
```

Now we can define our functor. We'll use `X_int` both to constrain the argument to the functor, as well as to constraint module returned by the functor.

```
# module Increment (M : X_int) : X_int = struct
    let x = M.x + 1
  end;;
module Increment : functor (M : X_int) -> X_int
```

One thing that immediately jumps out is that functors are more syntactically heavy-weight than ordinary functions. For one thing, functors require explicit (module) type annotations, which ordinary functions do not. Technically, only the type on the input is mandatory, although in practice, you should usually constrain the module returned by the functor, just as you should use an `mli`, even though it's not mandatory.

The following shows what happens when we omit the module type for the output of the functor.

```
# module Increment (M : X_int) = struct
    let x = M.x + 1
  end;;
module Increment : functor (M : X_int) -> sig val x : int end
```

We can see that the inferred module type of the output is now written out explicitly, rather than being a reference to the named signature `X_int`.

We can now use `Increment` to define new modules.

```
# module Three = struct let x = 3 end;;
  module Three : sig val x : int end
# module Four = Increment(Three);;
module Four : sig val x : int end
# Four.x - Three.x;;
- : int = 1
```

In this case, we applied `Increment` to a module whose signature is exactly equal to `X_int`. But we can apply `Increment` to any module that *satisfies* the interface `X_int`, in the same way that the contents of an `ml` file must satisfy the `mli`. That means that the module type can omit some information available in the module, either by dropping fields or by leaving some fields abstract. Here's an example:

```
# module Three_and_more = struct
    let x = 3
    let y = "three"
  end;;
module Three_and_more : sig val x : int val y : string end
# module Four = Increment(Three_and_more);;
module Four : sig val x : int end
```

The rules for determining whether a module matches a given signature are similar in spirit to the rules in an object-oriented language that determine whether an object satisfies a given interface. As in an object-oriented context, the extra information that doesn't match the signature you're looking for (in this case, the variable `y`), is simply ignored.

## A bigger example: computing with intervals

Let's consider a more realistic example of how to use functors: a library for computing with intervals. Intervals are a common computational object, and they come up in different contexts and for different types. You might need to work with intervals of floating point values, or strings, or times, and in each of these cases, you want similar operations: testing for emptiness, checking for containment, intersecting intervals, and so on.

Let's see how to use functors to build a generic interval library that can be used with any type that supports a total ordering on the underlying set over which you want to build intervals.

First we'll define a module type that captures the information we'll need about the endpoints of the intervals. This interface, which we'll call `Comparable`, contains just two things: a comparison function, and the type of the values to be compared.

```
# module type Comparable = sig
    type t
    val compare : t -> t -> int
  end ;;
module type Comparable = sig type t val compare : t -> t -> int end
```

The comparison function follows the standard OCaml idiom for such functions, returning `0` if the two elements are equal, a positive number if the first element is larger than the second, and a negative number if the first element is smaller than the second. Thus, we could rewrite the standard comparison functions on top of `compare` as shown below.

```
compare x y < 0     (* x < y *)
compare x y = 0     (* x = y *)
compare x y > 0     (* x > y *)
```

The functor for creating the interval module is shown below. We represent an interval with a variant type, which is either `Empty` or `Interval (x,y)`, where `x` and `y` are the bounds of the interval. In addition to the type, the functor contains implementations of a number of useful primitives for interacting with intervals.

```
# module Make_interval(Endpoint : Comparable) = struct

    type t = | Interval of Endpoint.t * Endpoint.t
```

```
            | Empty

    (** [create low high] creates a new interval from [low] to
        [high].  If [low > high], then the interval is empty *)
    let create low high =
      if Endpoint.compare low high > 0 then Empty
      else Interval (low,high)

    (** Returns true iff the interval is empty *)
    let is_empty = function
      | Empty -> true
      | Interval _ -> false

    (** [contains t x] returns true iff [x] is contained in the
        interval [t] *)
    let contains t x =
      match t with
      | Empty -> false
      | Interval (l,h) ->
        Endpoint.compare x l >= 0 && Endpoint.compare x h <= 0

    (** [intersect t1 t2] returns the intersection of the two input
        intervals *)
    let intersect t1 t2 =
      let min x y = if Endpoint.compare x y <= 0 then x else y in
      let max x y = if Endpoint.compare x y >= 0 then x else y in
      match t1,t2 with
      | Empty, _ | _, Empty -> Empty
      | Interval (l1,h1), Interval (l2,h2) ->
        create (max l1 l2) (min h1 h2)

  end ;;
module Make_interval :
  functor (Endpoint : Comparable) ->
    sig
      type t = Interval of Endpoint.t * Endpoint.t | Empty
      val create : Endpoint.t -> Endpoint.t -> t
      val is_empty : t -> bool
      val contains : t -> Endpoint.t -> bool
      val intersect : t -> t -> t
    end
```

We can instantiate the functor by applying it to a module with the right signature. In the following, rather than name the module first and then call the functor, we provide the functor input as an anonymous module.

```
# module Int_interval =
    Make_interval(struct
      type t = int
      let compare = Int.compare
    end);;
module Int_interval :
  sig
    type t = Interval of int * int | Empty
```

```
  val create : int -> int -> t
  val is_empty : t -> bool
  val contains : t -> int -> bool
  val intersect : t -> t -> t
end
```

If the input interface for your functor is aligned with the standards of the libraries you use, then you don't need to construct a custom module to feed to the functor. In this case, we can directly use the `Int` or `String` modules provided by Core.

```
# module Int_interval = Make_interval(Int) ;;
# module String_interval = Make_interval(String) ;;
```

This works because many modules in Core, including `Int` and `String`, satisfy an extended version of the `Comparable` signature described above. Such standardized signatures are good practice, both because they makes functors easier to use, and because they make your codebase generally easier to navigate.

Now we can use the newly defined `Int_interval` module like any ordinary module.

```
# let i1 = Int_interval.create 3 8;;
val i1 : Int_interval.t = Int_interval.Interval (3, 8)
# let i2 = Int_interval.create 4 10;;
val i2 : Int_interval.t = Int_interval.Interval (4, 10)
# Int_interval.intersect i1 i2;;
- : Int_interval.t = Int_interval.Interval (4, 8)
```

This design gives us the freedom to use any comparison function we want for comparing the endpoints. We could, for example, create a type of integer interval with the order of the comparison reversed, as follows:

```
# module Rev_int_interval =
    Make_interval(struct
      type t = int
      let compare x y = Int.compare y x
    end);;
```

The behavior of `Rev_int_interval` is of course different from `Int_interval`, as we can see below.

```
# let interval = Int_interval.create 4 3;;
val interval : Int_interval.t = Int_interval.Empty
# let rev_interval = Rev_int_interval.create 4 3;;
val rev_interval : Rev_int_interval.t = Rev_int_interval.Interval (4, 3)
```

Importantly, `Rev_int_interval.t` is a different type than `Int_interval.t`, even though its physical representation is the same. Indeed, the type system will prevent us from confusing them.

```
# Int_interval.contains rev_interval 3;;
Characters 22-34:
```

```
  Int_interval.contains rev_interval 3;;
                        ^^^^^^^^^^^^
Error: This expression has type Rev_int_interval.t
       but an expression was expected of type
         Int_interval.t = Make_interval(Int).t
```

This is important, because confusing the two kinds of intervals would be a semantic error, and it's an easy one to make. The ability of functors to mint new types is a useful trick that comes up a lot.

## Making the functor abstract

There's a problem with `Make_interval`. The code we wrote depends on the invariant that the upper bound of an interval is greater than its lower bound, but that invariant can be violated. The invariant is enforced by the create function, but because `Interval.t` is not abstract, we can bypass the `create` function.

```
# Int_interval.create 4 3;; (* going through create *)
- : Int_interval.t = Int_interval.Empty
# Int_interval.Interval (4,3);; (* bypassing create *)
- : Int_interval.t = Int_interval.Interval (4, 3)
```

To make `Int_interval.t` abstract, we need to restrict it the output of the `Make_interval` with an interface. Here's an explicit interface that we can use for that purpose.

```
# module type Interval_intf = sig
   type t
   type endpoint
   val create : endpoint -> endpoint -> t
   val is_empty : t -> bool
   val contains : t -> endpoint -> bool
   val intersect : t -> t -> t
  end;;
```

This interface includes the type `endpoint` to give us a way of referring to the endpoint type. Given this interface, we can redo our definition of `Make_interval`. Notice that we added the type `endpoint` to the implementation of the module to match `Interval_intf`.

```
# module Make_interval(Endpoint : Comparable) : Interval_intf
  = struct

    type endpoint = Endpoint.t
    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty

    ....

  end ;;
module Make_interval : functor (Endpoint : Comparable) -> Interval_intf
```

## Sharing constraints

The resulting module is abstract, but it's unfortunately too abstract. In particular, we haven't exposed the type `endpoint`, which means that we can't even construct an interval anymore.

```
# module Int_interval = Make_interval(Int);;
module Int_interval : Interval_intf
# Int_interval.create 3 4;;
Characters 20-21:
  Int_interval.create 3 4;;
                      ^
Error: This expression has type int but an expression was expected of type
         Int_interval.endpoint
```

To fix this, we need to expose the fact that `endpoint` is equal to `Int.t` (or more generally, `Endpoint.t`, where `Endpoint` is the argument to the functor). One way of doing this is through a *sharing constraint*, which allows you to tell the compiler to expose the fact that a given type is equal to some other type. The syntax for a simple sharing constraint is as follows.

```
S with type s = t
```

where `S` is a module type, `s` is a type inside of `S`, and `t` is a type defined outside of `S`. The result of this expression is a new signature that's been modified so that it exposes the fact that `s` is equal to `t`. One can also apply multiple sharing constraints to the same signature.

```
S with type s = t and s' = t'
```

We can use a sharing constraint to create a specialized version of `Interval_intf` for integer intervals.

```
# module type Int_interval_intf =
    Interval_intf with type endpoint = int;;
module type Int_interval_intf =
  sig
    type t
    type endpoint = int
    val create : endpoint -> endpoint -> t
    val is_empty : t -> bool
    val contains : t -> endpoint -> bool
    val intersect : t -> t -> t
  end
```

We can also use sharing constraints in the context of a functor. The most common use case is where you want to expose that some of the types of the module being generated by the functor are related to the types in the module fed to the functor.

In this case, we'd like to expose an equality between the type `endpoint` in the new module and the type `Endpoint.t`, from the module `Endpoint` that is the functor argument. We can do this as follows.

```
# module Make_interval(Endpoint : Comparable)
      : (Interval_intf with type endpoint = Endpoint.t)
  = struct

    type endpoint = Endpoint.t
    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty

    ...

  end ;;
module Make_interval :
  functor (Endpoint : Comparable) ->
    sig
      type t
      type endpoint = Endpoint.t
      val create : endpoint -> endpoint -> t
      val is_empty : t -> bool
      val contains : t -> endpoint -> bool
      val intersect : t -> t -> t
    end
```

So now, the interface is as it was, except that `endpoint` is now known to be equal to `Endpoint.t`. As a result of that type equality, we can again do things that require that `endpoint` be exposed, like constructing intervals.

```
# let i = Int_interval.create 3 4;;
val i : Int_interval.t = <abstr>
# Int_interval.contains i 5;;
- : bool = false
```

## Destructive substitution

Sharing constraints basically do the job, but they have some downsides. In particular, we've now been stuck with the useless type declaration of `endpoint` that clutters up both the interface and the implementation. A better solution would be to modify the `Interval_intf` signature by replacing `endpoint` with `Endpoint.t` everywhere it shows up, and deleting the definition of `endpoint` from the signature. We can do just this using what's called *destructive substitution*. Here's the basic syntax.

```
S with type s := t
```

The following shows how we could use this with `Make_interval`.

```
# module type Int_interval_intf =
    Interval_intf with type endpoint := int;;
```

```
module type Int_interval_intf =
  sig
    type t
    val create : int -> int -> t
    val is_empty : t -> bool
    val contains : t -> int -> bool
    val intersect : t -> t -> t
  end
```

There's now no `endpoint` type: all of its occurrences of have been replaced by `int`. As with sharing constraints, we can also use this in the context of a functor.

```
# module Make_interval(Endpoint : Comparable)
    : Interval_intf with type endpoint := Endpoint.t =
  struct

    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty

    ....

  end ;;
module Make_interval :
  functor (Endpoint : Comparable) ->
    sig
      type t
      val create : Endpoint.t -> Endpoint.t -> t
      val is_empty : t -> bool
      val contains : t -> Endpoint.t -> bool
      val intersect : t -> t -> t
    end
```

The interface is precisely what we want: the type `t` is abstract, the type of the endpoint is exposed, so we can create values of type `Int_interval.t` using the creation function, but not directly using the constructors and thereby violating the invariants of the module, as you can see below.

```
# module Int_interval = Make_interval(Int);;
# Int_interval.create 3 4;;
- : Int_interval.t = <abstr>
# Int_interval.Interval (4,3);;
Characters 0-27:
  Int_interval.Interval (4,3);;
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: Unbound constructor Int_interval.Interval
```

In addition, the `endpoint` type is gone from the interface, meaning we no longer need to define the `endpoint` type alias in the body of the module.

It's worth noting that the name is somewhat misleading, in that there's nothing destructive about destructive substitution; it's really just a way of creating a new signature by transforming an existing one.

## Using multiple interfaces

Another feature that we might want for our interval module is the ability to *serialize*, *i.e.*, to be able to read and write intervals as a stream of bytes. In this case, we'll do this by converting to and from s-expressions, which were mentioned already in Chapter 7. To recall, an s-expression is essentially a parenthesized expression whose atoms are strings, and it is a serialization format that is used commonly in Core. Here's an example.

```
# Sexp.of_string "(This is (an s-expression))";;
- : Sexp.t = (This is (an s-expression))
```

Core comes with a syntax extension called `sexplib` which can auto-generate s-expression conversion functions from a type declaration. Attaching `with sexp` to a type definition signals to the extension to generate the converters. Thus, we can write:

```
# type some_type = int * string list with sexp;;
type some_type = int * string list
val some_type_of_sexp : Sexp.t -> int * string list = <fun>
val sexp_of_some_type : int * string list -> Sexp.t = <fun>
# sexp_of_some_type (33, ["one"; "two"]);;
- : Sexp.t = (33 (one two))
# Sexp.of_string "(44 (five six))" |> some_type_of_sexp;;
- : int * string list = (44, ["five"; "six"])
```

We'll discuss s-expressions and `sexplib` in more detail in Chapter 17, but for now, let's see what happens if attach the `with sexp` declaration to the definition of `t` within the functor.

```
# module Make_interval(Endpoint : Comparable)
    : (Interval_intf with type endpoint := Endpoint.t) = struct

    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty
    with sexp

    ....

  end ;;
Characters 120-123:
        type t = | Interval of Endpoint.t * Endpoint.t
                                ^^^^^^^^^^
Error: Unbound value Endpoint.t_of_sexp
```

The problem is that `with sexp` adds code for defining the s-expression converters, and that code assumes that `Endpoint` has the appropriate sexp-conversion functions for `Endpoint.t`. But all we know about `Endpoint` is that it satisfies the `Comparable` interface, which doesn't say anything about s-expressions.

Happily, Core comes with a built in interface for just this purpose called `Sexpable`, which is defined as follows:

```
module type Sexpable = sig
  type t = int
  val sexp_of_t : t -> Sexp.t
  val t_of_sexp : Sexp.t -> t
end
```

We can modify `Make_interval` to use the `Sexpable` interface, for both its input and its output. First, let's create an extended version of the `Interval_intf` interface that includes the functions from the Sexpable interface. We can do this using destructive substitution on the `Sexpable` interface, to avoid having multiple distinct type `t`'s clashing with each other.

```
# module type Interval_intf_with_sexp = sig
   include Interval_intf
   include Sexpable with type t := t
  end;;
module type Interval_intf_with_sexp =
  sig
    type t
    type endpoint
    val create : endpoint -> endpoint -> t
    val is_empty : t -> bool
    val contains : t -> endpoint -> bool
    val intersect : t -> t -> t
    val t_of_sexp : Sexp.t -> t
    val sexp_of_t : t -> Sexp.t
  end
```

Equivalently, we can define a type `t` within our new module, and apply destructive substitutions to all of the included interfaces, `Interval_intf` included, as shown below. This is somewaht cleaner when combining multiple interfaces, since it correctly reflects that all of the signatures are being handled equivalently.

```
# module type Interval_intf_with_sexp = sig
   type t
   include Interval_intf with type t := t
   include Sexpable      with type t := t
  end;;
```

Now we can write the functor itself. We have been careful to override the sexp-converter here to ensure that the data structure's invariants are still maintained when reading in from an s-expression.

```
# module Make_interval(Endpoint : sig
                         type t
                         include Comparable with type t := t
                         include Sexpable   with type t := t
                       end)
```

```
  : (Interval_intf_with_sexp with type endpoint := Endpoint.t)
= struct

  type t = | Interval of Endpoint.t * Endpoint.t
           | Empty
  with sexp

  (** [create low high] creates a new interval from [low] to
      [high].  If [low > high], then the interval is empty *)
  let create low high =
    if Endpoint.compare low high > 0 then Empty
    else Interval (low,high)

  (* put a wrapper round the auto-generated sexp_of_t to enforce
     the invariants of the data structure *)
  let t_of_sexp sexp =
    match t_of_sexp sexp with
    | Empty -> Empty
    | Interval (x,y) -> create x y

  (** Returns true iff the interval is empty *)
  let is_empty = function
    | Empty -> true
    | Interval _ -> false

  (** [contains t x] returns true iff [x] is contained in the
      interval [t] *)
  let contains t x =
    match t with
    | Empty -> false
    | Interval (l,h) ->
      Endpoint.compare x l >= 0 && Endpoint.compare x h <= 0

  (** [intersect t1 t2] returns the intersection of the two input
      intervals *)
  let intersect t1 t2 =
    let min x y = if Endpoint.compare x y <= 0 then x else y in
    let max x y = if Endpoint.compare x y >= 0 then x else y in
    match t1,t2 with
    | Empty, _ | _, Empty -> Empty
    | Interval (l1,h1), Interval (l2,h2) ->
      create (max l1 l2) (min h1 h2)
end;;
module Make_interval :
  functor
    (Endpoint : sig
                  type t
                  val compare : t -> t -> int
                  val t_of_sexp : Sexp.t -> t
                  val sexp_of_t : t -> Sexp.t
                end) ->
    sig
      type t
      val create : Endpoint.t -> Endpoint.t -> t
      val is_empty : t -> bool
```

```
    val contains : t -> Endpoint.t -> bool
    val intersect : t -> t -> t
    val t_of_sexp : Sexp.t -> t
    val sexp_of_t : t -> Sexp.t
  end
```

And now, we can use that sexp-converter in the ordinary way:

```
# module Int_interval = Make_interval(Int) ;;
# Int_interval.sexp_of_t (Int_interval.create 3 4);;
- : Sexp.t = (Interval 3 4)
# Int_interval.sexp_of_t (Int_interval.create 4 3);;
- : Sexp.t = Empty
```

## Extending modules

Another common use of functors is to generate type-specific functionality for a given module in a standardized way. Let's see how this works in the context of a functional queue, which is just a functional version of a FIFO (first-in, first-out) queue. Being functional, operations on the queue return new queues, rather than modifying the queues that were passed in.

Here's a reasonable `mli` for such a module.

```
(* file: fqueue.mli *)

type 'a t

val empty : 'a t

(** [enqueue el q] adds [el] to the back of [q] *)
val enqueue : 'a t -> 'a -> 'a t

(** [dequeue q] returns None if the [q] is empty, otherwise returns
    the first element of the queue and the remainder of the queue *)
val dequeue : 'a t -> ('a * 'a t) option

(** Folds over the queue, from front to back *)
val fold : 'a t -> init:'acc -> f:('acc -> 'a -> 'acc) -> 'acc
```

The `Fqueue.fold` function above requires some explanation. It follows the same pattern as the `List.fold` function we described in "Using the `List` module effectively" on page 55. Essentially, `Fqueue.fold ~q ~init ~f` walks over the elements of `q` from front to back, starting with an accumulator of `init` and using `f` to update the accumulator value as it walks over the queue, returning the final value of the accumulator at the end of the computation. Fold is a quite powerful operation, as we'll see.

Now let's implement `Fqueue`. A standard trick is for the `Fqueue` to maintain an input and an output list, so that one can efficiently `enqueue` on the input list and efficiently dequeue from the output list. If you attempt to dequeue when the output list is empty, the input

list is reversed and becomes the new output list. Here's an implementation that uses that trick.

```
(* file: fqueue.ml *)
open Core.Std

type 'a t = 'a list * 'a list

let empty = ([],[])

let enqueue (in_list, out_list) x =
  (x :: in_list,out_list)

let dequeue (in_list, out_list) =
  match out_list with
  | hd :: tl -> Some (hd, (in_list, tl))
  | [] ->
    match List.rev in_list with
    | [] -> None
    | hd :: tl -> Some (hd, ([], tl))

let fold (in_list, out_list) ~init ~f =
  let after_out = List.fold ~init ~f out_list in
  List.fold_right ~init:after_out ~f in_list
```

One problem with `Fqueue` is that the interface is quite skeletal. There are lots of useful helper functions that one might want that aren't there. The list module, by way of contrast, has functions like `List.iter`, which runs a function on each element; and `List.for_all`, which returns true if and only if the given predicate evaluates to try on every element of the list. Such helper functions come up for pretty much every container type, and implementing them over and over is a dull and repetitive affair.

As it happens, many of these helper functions can be derived mechanically from just the fold function we already implemented. Rather than write all of these helper functions by hand for every new container type, we can instead use a functor that will let us add this functionality to any container that has a `fold` function.

We'll create a new module, `Foldable` that automates the process of adding helper functions to a fold-supporting container. As you can see, `Foldable` contains a module signature `S` which defines the signature that is required to support folding; and a functor `Extend` that allows one to extend any module that matches `Foldable.S`.

```
(* file: foldable.ml *)

open Core.Std

module type S = sig
  type 'a t
  val fold : 'a t -> init:'acc -> f:('acc -> 'a -> 'acc) -> 'acc
end
```

```
module type Extension = sig
  type 'a t
  val iter    : 'a t -> f:('a -> unit) -> unit
  val length  : 'a t -> int
  val count   : 'a t -> f:('a -> bool) -> int
  val for_all : 'a t -> f:('a -> bool) -> bool
  val exists  : 'a t -> f:('a -> bool) -> bool
end

(* For extending a Foldable module *)
module Extend(Arg : S)
  : (Extension with type 'a t := 'a Arg.t) =
struct
  open Arg

  let iter t ~f =
    fold t ~init:() ~f:(fun () a -> f a)

  let length t =
    fold t ~init:0  ~f:(fun acc _ -> acc + 1)

  let count t ~f =
    fold t ~init:0  ~f:(fun count x -> count + if f x then 1 else 0)

  exception Short_circuit

  let for_all c ~f =
    try iter c ~f:(fun x -> if not (f x) then raise Short_circuit); true
    with Short_circuit -> false

  let exists c ~f =
    try iter c ~f:(fun x -> if f x then raise Short_circuit); false
    with Short_circuit -> true
end
```

Now we can apply this to `Fqueue`. We can rewrite the interface of `Fqueue` as follows.

```
(* file: fqueue.mli *)
open Core.Std

type 'a t
val empty : 'a t
val enqueue : 'a t -> 'a -> 'a t
val dequeue : 'a t -> ('a * 'a t) option
val fold : 'a t -> init:'acc -> f:('acc -> 'a -> 'acc) -> 'acc

include Foldable.Extension with type 'a t := 'a t
```

In order to apply the functor, we'll put the definition of `Fqueue` in a sub-module called `T`, and then call `Foldable.Extend` on `T`.

```
(* file: fqueue.ml *)

open Core.Std
```

```
module T = struct
  type 'a t = 'a list * 'a list

  let empty = [],[]

  let enqueue (in_list,out_list) x =
    (x :: in_list,out_list)

  let dequeue (in_list, out_list) =
    match out_list with
    | hd :: tl -> Some (hd, (in_list, tl))
    | [] ->
      match List.rev in_list with
      | [] -> None
      | hd :: tl -> Some (hd, ([], tl))

  let fold (in_list,out_list) ~init ~f =
    let after_out = List.fold ~init ~f out_list in
    List.fold_right ~init:after_out ~f in_list
end

include T
include Foldable.Extend(T)
```

Core comes with a number of functors for extending modules that follow this same basic pattern, including:

- `Container.Make`, which is very similar to `Foldable.Extend`.
- `Comparable.Make`, which adds a variety of helper functions and types for types that have a comparison function.
- `Hashable.Make` for adding hash-based data structures like hash sets and hash heaps for types that have hash functions.
- `Monad.Make` for so-called monadic libraries, like the ones discussed in Chapter 7 and Chapter 18. Here, the functor is used to provide a collection of standard helper functions based on the `bind` and `return` operators.

These functors come in handy when you want to add the same kind of functionality that is commonly available in Core to your own types.

2013-07-04
10:28:30

CHAPTER 10

# First class modules

You can think of OCaml as being broken up into two parts: a core language that is concerned with values and types, and a module language that is concerned with modules and module signatures. These sub-languages are stratified, in that modules can contain types and values, but ordinary values can't contain modules or module types. That means you can't do things like define a variable whose value is a module, or a function that takes a module as an argument.

OCaml provides a way around this stratification in the form of *first-class modules*. First-class modules are ordinary values that can be created from and converted back to regular modules.

First-class modules are a sophisticated technique, and you'll need to get comfortable with some advanced aspects of the language to use them effectively. But it's worth learning, because letting modules into the core language is quite powerful, increasing the range of what you can express and making it easier to build flexible and modular systems.

## Working with first-class modules

We'll start out by covering the basic mechanics of first-class modules by working through some toy examples. We'll get to more realistic examples in the next section.

In that light, consider the following signature of a module with a single integer variable.

```
# module type X_int = sig val x : int end;;
module type X_int = sig val x : int end
```

We can also create a module that matches this signature.

```
# module Three : X_int = struct let x = 3 end;;
module Three : X_int
# Three.x;;
- : int = 3
```

A first-class module is created by packaging up a module with a signature that it satisfies. This is done using the `module` keyword, using the following syntax:

```
(module <Module_name> : <Module_type>)
```

So, we can convert `Three` into a first-class module as follows.

```
# let three = (module Three : X_int);;
val three : (module X_int) = <module>
```

The module type doesn't need to be part of the construction of a first-class module if it can be inferred. Thus, we can write:

```
# module Four = struct let x = 4 end;;
module Four : sig val x : int end
# let numbers = [ three; (module Four) ];;
val numbers : (module X_int) list = [<module>; <module>]
```

We can also create a first-class module from an anonymous module:

```
# let numbers = [three; (module struct let x = 4 end)];;
val numbers : (module X_int) list = [<module>; <module>]
```

In order to access the contents of a first-class module, you need to unpack it into an ordinary module. This can be done using the `val` keyword, using this syntax:

```
(val <first_class_module> : <Module_type>)
```

And here's an example.

```
# module New_three = (val three : X_int) ;;
module New_three : X_int
# New_three.x;;
- : int = 3
```

**Equality of first-class module types**

The type of the first-class module, *e.g.*, `(module X_int)`, is based on the fully-qualified name of the signature that was used to construct it. A first class module based on a signature with a different name, even if it is substantively the same signature, will result in a distinct type, as you can see below.

```
# module type Y_int = X_int;;
module type Y_int = X_int
# let five = (module struct let x = 5 end : Y_int);;
val five : (module Y_int) = <module>
# [three; five];;
```

```
Error: This expression has type (module Y_int)
       but an expression was expected of type (module X_int)
```

Even though their types as first-class modules are distinct, the underlying module types are compatible (indeed, identical), so we can unify the types by unpacking and repacking the module.

```
# [three; (module (val five))];;
- : (module X_int) list = [<module>; <module>]
```

The way in which type equality for first-class modules is determined can be confusing. One common and problematic case is that of creating an alias of a module type defined elsewhere. This is often done to improve readability, and can happen both through an explicit declaration of a module type or implicitly through an `include` declaration. In both cases, this has the unintended side effect of making first class modules built off of the alias incompatible with those built off of the original module type. To deal with this, one should be disciplined in how one refers to signatures when constructing first-class modules.

We can also write ordinary functions which consume and create first class modules. The following shows the definition of two functions: `to_int`, which converts a `(module X_int)` into an `int`; and `plus`, which returns the sum of two `(module X_int)`s.

```
# let to_int m =
    let module M = (val m : X_int) in
    M.x
  ;;
val to_int : (module X_int) -> int = <fun>
# let plus m1 m2 =
    (module struct
       let x = to_int m1 + to_int m2
     end : X_int)
  ;;
val plus : (module X_int) -> (module X_int) -> (module X_int) = <fun>
```

With these functions in hand, we can now work with values of type `(module X_int)` in a more natural style, taking advantage of the concision and simplicity of the core language.

```
# let six = plus three three;;
val six : (module X_int) = <module>
# to_int (List.fold ~init:six ~f:plus [three;three]);;
- : int = 12
```

There are some useful syntactic shortcuts when dealing with first class modules. One notable one is that you can do the conversion to an ordinary module within a pattern match. Thus, we can rewrite the `to_int` function as follows.

```
# let to_int (module M : X_int) = M.x ;;
val to_int : (module X_int) -> int = <fun>
```

First-class modules can contain types and functions in addition to simple values like `int`. Here's an interface that contains a type and a corresponding `bump` operation that takes a value of the type and produces a new one.

```
# module type Bumpable = sig
    type t
    val bump : t -> t
  end;;
module type Bumpable = sig type t val val bump : t -> t end
```

We can create multiple instances of this module with different undelrying types.

```
# module Int_bumper = struct
    type t = int
    let bump n = n + 1
  end;;
module Int_bumper : sig type t = int val bump : t -> t end
# module Float_bumper = struct
     type t = float
     let bump n = n +. 1.
  end;;
module Float_bumper : sig type t = float val bump : t -> t end
```

And we can convert these to first-class modules.

```
# let int_bumper = (module Int_bumper : Bumpable);;
val int_bumper : (module Bumpable) = <module>
```

But you can't do much with `int_bumper`, since `int_bumper` is fully abstract, so that we can no longer recover the fact that the type in question is `int`. This means you can't really do much with it, as you can see below.

```
# let (module Bumpable) = int_bumper in Bumpable.bump 3;;
Error: This expression has type int but an expression was expected of type
         Bumpable.t
```

To make `int_bumber` usable, we need to expose the type, which we can do as follows.

```
# let int_bumper = (module Int_bumper : Bumpable with type t = int);;
val int_bumper : (module Bumpable with type t = int) = <module>
# let float_bumper = (module Float_bumper : Bumpable with type t = float);;
val float_bumper : (module Bumpable with type t = float) = <module>
```

The sharing constraints we've added above make the resulting first-class modules polymorphic in the type `t`. As a result, we can now use these values on values of the matching type.

```
# let (module Bumpable) = int_bumper in Bumpable.bump 3;;
- : int = 4
# let (module Bumpable) = float_bumper in Bumpable.bump 3.5;;
- : float = 4.5
```

We can also write functions that use such first-class modules polymorphically. The following function takes two arguments: a `Bumpable` module, and a list of elements of the same type as the type `t` of the module.

```
# let bump_list
       (type a)
       (module B : Bumpable with type t = a)
       (l: a list)
    =
    List.map ~f:B.bump l
  ;;
val bump_list : (module Bumpable with type t = 'a) -> 'a list -> 'a list =
  <fun>
```

Here, we used a feature of OCaml that hasn't come up before: a *locally abstract type*. For any function, you can declare a pseudo-parameter of the form `(type a)` for any type name `a` which introduces a fresh type that acts like an abstract type within the context of the function. Here, we used that type as part of a sharing constraint that ties the type `B.t` with the type of the elements of the list passed in.

The resulting function is polymorphic in both the type of the list element and the type `Bumpable.t`. We can see this function in action below.

```
# bump_list int_bumper [1;2;3];;
- : int list = [2; 3; 4]
# bump_list float_bumper [1.5;2.5;3.5];;
- : float list = [2.5; 3.5; 4.5]
```

Polymorphic first-class modules are important because they allow you to connect the types associated with a first-class module to the types of other values you're working with.

**More on locally abstract types**

One of the key properties of locally abstract types is that they are dealt with as abstract types within the function they're defined within, but are polymorphic from the outside. Consider the following example.

```
# let wrap_in_list (type a) (x:a) = [x];;
- : 'a -> 'a list = <fun>
```

This compiles successfully because the type `a` is used in a way that is compatible with it being abstract, but the type of the function that is inferred is polymorphic.

If, on the other hand, we try to use the type `a` as equivalent to some concrete type, say, `int`, then the compiler will complain.

```
# let wrap_int_in_list (type a) (x:a) = x + x;;
Error: This expression has type a but an expression was expected of type int
```

One common use of locally abstract types is to create a new type that can be used in constructing a module. Here's an example of doing this to create a new first-class module.

```
# module type Comparable = sig
    type t
    val compare : t -> t -> int
  end ;;
module type Comparable = sig type t val compare : t -> t -> int end
# let create_comparable (type a) compare =
    (module struct
       type t = a
       let compare = compare
     end : Comparable with type t = a)
  ;;
# create_comparable Int.compare;;
- : (module Comparable with type t = int) = <module>
# create_comparable Float.compare;;
- : (module Comparable with type t = float) = <module>
```

Here, what we effectively do is capture a polymorphic type and export it as a concrete type within a module.

This technique is useful beyond first-class modules. For example, we can use the same approach to construct a local module to be fed to a functor.

## Example: A query handling framework

Now let's look at first class modules in the context of a more complete and realistic example. In particular, consider the following signature for a module that implements a query handler.

```
# module type Query_handler = sig
```

```
    (** Configuration for a query handler.  Note that this can be
        converted to and from an s-expression *)
    type config with sexp

    (** The name of the query-handling service *)
    val name : string

    (** The state o fthe query handler *)
    type t

    (** Create a new query handler from a config *)
    val create : config -> t

    (** Evaluate a given query, where both input and output are
        s-expressions *)
    val eval : t -> Sexp.t -> Sexp.t Or_error.t
  end;;
module type Query_handler =
  sig
    type config
    val name : string
    type t
    val create : config -> t
    val eval : t -> Sexp.t -> Sexp.t Or_error.t
    val config_of_sexp : Sexp.t -> config
    val sexp_of_config : config -> Sexp.t
  end
```

In the above we use s-expressions as the format for queries and responses, as well for the config. S-expressions are a simple, flexible, and human-readable serialization format commonly used in Core. We'll cover s-expressions in more detail in Chapter 17, but for now, it's enough to think of them as balanced parenthetical expressions whose atomic values are strings, *e.g.*, `(this (is an) (s expression))`.

In addition, we use the `sexplib` syntax extension which extends OCaml by adding the `with sexp` declaration. When attached to a type in a signature, `with sexp` adds declarations of s-expression converters, *e.g.*,

```
# module type M = sig type t with sexp end;;
module type M =
  sig type t val t_of_sexp : Sexp.t -> t val sexp_of_t : t -> Sexp.t end
```

In a module, `with sexp` adds the implementation of those functions. Thus, we can write

```
# type u = { a: int; b: float } with sexp;;
type u = { a : int; b : float; }
val u_of_sexp : Sexp.t -> u = <fun>
val sexp_of_u : u -> Sexp.t = <fun>
# sexp_of_u {a=3;b=7.};;
- : Sexp.t = ((a 3) (b 7))
# u_of_sexp (Sexp.of_string "((a 43) (b 3.4))");;
- : u = {a = 43; b = 3.4}
```

This is all described in more detail in Chapter 17.

## Implementing a query handler

Let's look at some examples of query handlers that satisfy this interface. The following handler produces unique integer ids by keeping an internal counter which it bumps every time it produces a new value. The input to the query in this case is just the trivial s-expression `()`, otherwise known as `Sexp.unit`.

```
# module Unique = struct
    type config = int with sexp
    type t = { mutable next_id: int }

    let name = "unique"
    let create start_at = { next_id = start_at }

    let eval t sexp =
      match Or_error.try_with (fun () -> unit_of_sexp sexp) with
      | Error _ as err -> err
      | Ok () ->
        let response = Ok (Int.sexp_of_t t.next_id) in
        t.next_id <- t.next_id + 1;
        response
  end;;
module Unique :
  sig
    type config = int
    val config_of_sexp : Sexp.t -> config
    val sexp_of_config : config -> Sexp.t
    type t = { mutable next_id : config; }
    val name : string
    val create : config -> t
    val eval : t -> Sexp.t -> (Sexp.t, Error.t) Result.t
  end
```

We can use this module to create an instance of the `Unique` query handler and interact with it.

```
# let unique = Unique.create 0;;
val unique : Unique.t = {Unique.next_id = 0}
# Unique.eval unique Sexp.unit;;
- : (Sexp.t, Error.t) Result.t = Ok 0
# Unique.eval unique Sexp.unit;;
- : (Sexp.t, Error.t) Result.t = Ok 1
```

Here's another example: a query handler that does directory listings. Here, the config is the default directory that relative paths are interpreted within.

```
# module List_dir = struct
    type config = string with sexp
    type t = { cwd: string }
```

```
    (** [is_abs p] Returns true if [p] is an absolute path  *)
    let is_abs p =
      String.length p > 0 && p.[0] = '/'

    let name = "ls"
    let create cwd = { cwd }

    let eval t sexp =
      match Or_error.try_with (fun () -> string_of_sexp sexp) with
      | Error _ as err -> err
      | Ok dir ->
        let dir =
          if is_abs dir then dir
          else Filename.concat t.cwd dir
        in
        Ok (Array.sexp_of_t String.sexp_of_t (Sys.readdir dir))
  end;;
module List_dir :
  sig
    type config = string
    val name : config
    type t
    val create : config -> t
    val eval : t -> Sexp.t -> Sexp.t Or_error.t
    val config_of_sexp : Sexp.t -> config
    val sexp_of_config : config -> Sexp.t
  end
```

Again, we can create an instance of this query handler and interact with it directly.

```
# let list_dir = List_dir.create "/var";;
val list_dir : List_dir.t = <abstr>
# List_dir.eval list_dir (sexp_of_string ".");;
- : (Sexp.t, Error.t) Result.t =
Ok
 (agentx at audit backups db empty folders jabberd lib log mail msgs named
  netboot pgsql_socket_alt root rpc run rwho spool tmp vm yp)
# List_dir.eval list_dir (sexp_of_string "yp");;
- : (Sexp.t, Error.t) Result.t =
Ok (binding binding~orig Makefile.main Makefile.yp)
```

## Dispatching to multiple query handlers

Now, what if we want to dispatch queries to any of an arbitrary collection of handlers? Ideally, we'd just like to pass in the handlers as a simple data structure like a list. This is awkward to do with modules and functors alone, but it's quite natural with first-class modules. The first thing we'll need to do is create a signature that combines a `Query_handler` module with an instantiated query handler.

```
# module type Query_handler_instance = sig
    module Query_handler : Query_handler
```

```
    val this : Query_handler.t
  end;;
module type Query_handler_instance =
  sig module Query_handler : Query_handler val this : Query_handler.t end
```

With this signature, we can create a first-class module that encompasses both an instance of the query and the matching operations for working with that query.

We can create an instance as follows.

```
# let unique_instance =
    (module struct
       module Query_handler = Unique
       let this = Unique.create 0
     end : Query_handler_instance);;
val unique_instance : (module Query_handler_instance) = <module>
```

Constructing instances in this way is a little verbose, but we can write a function that eliminates most of this boilerplate. Note that we are again making use of a locally abstract type.

```
# let build_instance
        (type a)
        (module Q : Query_handler with type config = a)
        config
    =
    (module struct
       module Query_handler = Q
       let this = Q.create config
     end : Query_handler_instance)
  ;;
val build_instance :
  (module Query_handler with type config = 'a) ->
  'a -> (module Query_handler_instance) = <fun>
```

Using `build_instance`, constructing a new instance becomes a one-liner:

```
# let unique_instance = build_instance (module Unique) 0;;
val unique_instance : (module Query_handler_instance) = <module>
# let list_dir_instance = build_instance (module List_dir)  "/var";;
val list_dir_instance : (module Query_handler_instance) = <module>
```

The following code lets you dispatch queries to one of a list of query handler instances. We assume that the shape of the query is as follows:

```
(<query-name> <query>)
```

where `<query-name>` is used to determine which query handler to dispatch the query to.

The first thing we'll need is a function that takes a list of query handler instances and constructs a dispatch table from it.

```
# let build_dispatch_table handlers =
    let table = String.Table.create () in
    List.iter handlers
      ~f:(fun ((module I : Query_handler_instance) as instance) ->
        Hashtbl.replace table ~key:I.Query_handler.name ~data:instance);
    table
  ;;
val build_dispatch_table :
  (module Query_handler_instance) list ->
  (module Query_handler_instance) String.Table.t = <fun>
```

Now, we need a function that dispatches to a handler using a dispatch table.

```
# let dispatch dispatch_table name_and_query =
    match name_and_query with
    | Sexp.List [Sexp.Atom name; query] ->
      begin match Hashtbl.find dispatch_table name with
      | None ->
        Or_error.error "Could not find matching handler"
          name String.sexp_of_t
      | Some (module I : Query_handler_instance) ->
        I.Query_handler.eval I.this query
      end
    | _ ->
      Or_error.error_string "malformed query"
  ;;
val dispatch :
  (string, (module Query_handler_instance)) Hashtbl.t ->
  Sexp.t -> Sexp.t Or_error.t = <fun>
```

This function interacts with an instance by unpacking it into a module `I` and then using the query handler instance (`I.this`) in concert with the associated module (`I.Query_handler`).

The bundling together of the module and the value is in many ways reminisicent of object-oriented languages. One key difference, is that first-class modules allow you to package up more than just a functions or methods. As we've seen, you can also include types and even modules. We've only used it in a small way here, but this extra power allows you to build more sophisticated components that involve multiple interdependent types and values.

Now let's turn this into a complete, running example, by adding a command-line interface, as shown below.

```
# let rec cli dispatch_table =
    printf ">>> %!";
    let result =
      match In_channel.input_line stdin with
      | None -> `Stop
      | Some line ->
        match Or_error.try_with (fun () -> Sexp.of_string line) with
        | Error e -> `Continue (Error.to_string_hum e)
```

```
        | Ok (Sexp.Atom "quit") -> `Stop
        | Ok query ->
          begin match dispatch dispatch_table query with
          | Error e -> `Continue (Error.to_string_hum e)
          | Ok s    -> `Continue (Sexp.to_string_hum s)
          end;
    in
    match result with
    | `Stop -> ()
    | `Continue msg ->
      printf "%s\n%!" msg;
      cli dispatch_table
  ;;
val cli : (string, (module Query_handler_instance)) Hashtbl.t -> unit = <fun>
```

We can most effectively run this command-line interface from a stand-alone program, which we can do by putting the above code in a file along with following command to launch the interface.

```
let () =
  cli [unique_instance; list_dir_instance]
```

Here's an example of a session with this program.

```
$ ./query_handler.byte
>>> (unique ())
0
>>> (unique ())
1
>>> (ls .)
(agentx at audit backups db empty folders jabberd lib log mail msgs named
 netboot pgsql_socket_alt root rpc run rwho spool tmp vm yp)
>>> (ls vm)
(sleepimage swapfile0 swapfile1 swapfile2 swapfile3 swapfile4 swapfile5
 swapfile6)
```

## Loading and unloading query handlers

Let's show off the dynamism and flexibility of first-class modules by showing how to build a query handler whose job is to load and unload other query handlers.

The module in question will be called `Loader`, and its configuration is a list of known `Query_handler` modules. Here are the basic types.

```
module Loader = struct
  type config = (module Query_handler) list sexp_opaque
  with sexp

  type t = { known  : (module Query_handler)          String.Table.t
           ; active : (module Query_handler_instance) String.Table.t
           }
```

```
let name = "loader"
```

Note that a `Loader.t` has two hash tables: one containing the known query handler modules, and one containing the active query handler instances. The `Loader.t` will be responsible for creating new instances and adding them to the table, as well as for removing instances, all in response to user queries.

Next, we'll need a function for creating a `Loader.t`. This function requires the list of known query handler modules. Note that the table of active modules starts out as empty.

```
let create known_list =
  let active = String.Table.create () in
  let known  = String.Table.create () in
  List.iter known_list
    ~f:(fun ((module Q : Query_handler) as q) ->
      Hashtbl.replace known ~key:Q.name ~data:q);
  { known; active }
```

Now we'll start writing out the functions for manipulating the table of active query handlers. We'll start with the function for loading an instance. Note that it takes as an argument both the name of the query handler, and the configuration for instantiating that handler, in the form of an s-expression. These are used for creating a first-class module of type `(module Query_handler_instance)`, which is then added to the active table.

```
let load t handler_name config =
  if Hashtbl.mem t.active handler_name then
    Or_error.error "Can't re-register an active handler"
      handler_name String.sexp_of_t
  else
    match Hashtbl.find t.known handler_name with
    | None ->
      Or_error.error "Unknown handler" handler_name String.sexp_of_t
    | Some (module Q : Query_handler) ->
      let instance =
        (module struct
           module Query_handler = Q
           let this = Q.create (Q.config_of_sexp config)
         end : Query_handler_instance)
      in
      Hashtbl.replace t.active ~key:handler_name ~data:instance;
      Ok Sexp.unit
```

Since the `load` function will refuse to `load` an already active handler, we also need the ability to unload a handler. Note that the handler explicitly refuses to unload itself.

```
let unload t handler_name =
  if not (Hashtbl.mem t.active handler_name) then
    Or_error.error "Handler not active" handler_name String.sexp_of_t
```

```
    else if handler_name = name then
      Or_error.error_string "It's unwise to unload yourself"
    else (
      Hashtbl.remove t.active handler_name;
      Ok Sexp.unit
    )
```

Finally, we need to implement the `eval` function, which will determine the query interface presented to the user. We'll do this by creating a variant type, and using the s-expression converter generated for that type to parse the query from the user.

```
type request =
  | Load of string * Sexp.t
  | Unload of string
  | Known_services
  | Active_services
with sexp
```

The eval function itself is fairly straight-forward, dispatching to the appropriate functions to respond to each type of query. Note that we use write `<sexp_of<string list>>` to auto-generate a function for converting a list of strings to an s-expression. This is part of the sexplib package described in Chapter 17.

This function ends the definition of the `Loader` module.

```
let eval t sexp =
  match Or_error.try_with (fun () -> request_of_sexp sexp) with
  | Error _ as err -> err
  | Ok resp ->
    match resp with
    | Load (name,config) -> load   t name config
    | Unload name        -> unload t name
    | Known_services ->
      Ok (<:sexp_of<string list>> (Hashtbl.keys t.known))
    | Active_services ->
      Ok (<:sexp_of<string list>> (Hashtbl.keys t.active))
end
```

Finally, we can put this all together with the command line interface. We first create an instance of the loader query handler, and then add that instance to the loader's active table. We can then just launch the command-line interface, passing it the active table.

```
let () =
  let loader = Loader.create [(module Unique); (module List_dir)] in
  let loader_instance =
    (module struct
       module Query_handler = Loader
       let this = loader
     end : Query_handler_instance)
  in
  Hashtbl.replace loader.Loader.active
```

```
    ~key:Loader.name ~data:loader_instance;
  cli loader.Loader.active
```

The resulting command line interface behaves much as you'd expect, starting out with no query handlers available, but giving you the ability to load and unload them. Here's an example of it in action. As you can see, we start out with `loader` itself as the only active handler.

```
>>> (loader known_services)
(ls unique)
>>> (loader active_services)
(loader)
```

If we try to use one of the inactive queries, it will fail.

```
>>> (ls .)
Could not find matching handler: ls
```

But, we can load the `ls` handler with a config of our choice, at which point, it will be available for use. And once we unload it, it will be unavailable yet again, and could be reloaded with a different config.

```
>>> (loader (load ls /var))
()
>>> (ls /var)
(agentx at audit backups db empty folders jabberd lib log mail msgs named
 netboot pgsql_socket_alt root rpc run rwho spool tmp vm yp)
>>> (loader (unload ls))
()
>>> (ls /var)
Could not find matching handler: ls
```

Notably, the loader can't be itself loaded (since it's not on the list of known handlers), and can't be unloaded.

```
>>> (loader (unload loader))
It's unwise to unload yourself
```

We can push this dynamism yet further using libraries like `ocaml_plugin`, which use OCaml's dynamic linking facilities to allow a program to compile and load an OCaml source file as a first-class module. Thus, one could extend `Loader` to loads entirely new query handlers from disk on demand.

## Living without first-class modules

It's worth noting that most designs that can be done with first class modules can be simulated without them, with some level of awkwardness. For example, we could re-write our query handler example without first-class modules using the following types:

```
# type query_handler_instance = { name : string
                                ; eval : Sexp.t -> Sexp.t Or_error.t
                                }
  type query_handler = Sexp.t -> query_handler_instance
  ;;
type query_handler_instance = {
  name : string;
  eval : Sexp.t -> Sexp.t Or_error.t;
}
type query_handler = Sexp.t -> query_handler_instance
```

The idea here is that we hide the true types of the objects in question behind the functions stored in the closure. Thus, we could put the `Unique` query handler into this framework as follows.

```
# let unique_handler config_sexp =
    let config = Unique.config_of_sexp config_sexp in
    let unique = Unique.create config in
    { name = Unique.name
    ; eval = (fun config -> Unique.eval unique config)
    }
  ;;
val unique_handler : Sexp.t -> query_handler_instance = <fun>
```

For an example on this scale, the above approach is completely reasonable, and first-class modules are not really necessary. But the more functionality you need to hide away behind a set of closures, and the more complicated the relationships between the different types in question, the more awkward this approach becomes, and the better it is to use first-class modules.

2013-07-04
10:28:30

# CHAPTER 11
# Objects

We've already seen several tools that OCaml provides for organizing programs, particularly modules. In addition, OCaml also supports object-oriented programming. There are objects, classes, and their associated types. In this chapter, we'll introduce you to OCaml objects and subtyping. In the next chapter Chapter 12, we'll introduce you to classes and inheritance.

**What is Object-Oriented Programming?**

Object-oriented programming (often shorted to OOP) is a programming style that encapsulates computation and data within logical *objects*. Each object contains some data stored in *fields*, and has *method* functions that can be invoked against the data within the object. The code definition behind an object is called a *class*, and objects are constructed from a class definition by calling a constructor with the data that the object will use to build itself.

There are five fundamental properties that differentiate OOP from other styles:

- *Abstraction*: the details of the implementation are hidden in the object, and the external interface is just the set of publicly-accessible methods.
- *Dynamic lookup*: when a message is sent to an object, the method to be executed is determined by the implementation of the object, not by some static property of the program. In other words, different objects may react to the same message in different ways.
- *Subtyping*: if an object `a` has all the functionality of an object `b`, then we may use `a` in any context where `b` is expected.
- *Inheritance*: the definition of one kind of object can be reused to produce a new kind of object. This new definition can override some behaviour, but also share code with its parent.
- *Open recursion*: an object's methods can invoke another method in the same object using a special variable (often called self). These method calls use dynamic lookup, allowing a method defined in one object to invoke methods defined in another object that inherits from the first.

Almost every notable modern programming language has been influenced by OOP, and you'll have run across these terms if you've ever used C++, Java, C#, Ruby, Python or JavaScript.

# OCaml objects

If you already know about object oriented programming in a language like Java or C++, the OCaml object system may come as a surprise. Foremost is the complete separation of objects, and their types, from the class system in OCaml. In a language like Java, a class name is also used as the type of objects created by instantiating it, and the relationships between these object types correspond to inheritance. For example, if we implement a class `Deque` in Java by inheriting from a class `Stack`, we would be allowed to pass a deque anywhere a stack is expected.

OCaml is entirely different. Classes are used to construct objects and support inheritance, but classes are not types. Instead, objects have *object types*, and if you want to use objects, you aren't required to use classes at all. Here's an example of a simple object.

```
# let s = object
    val mutable v = [0; 2]

    method pop =
      match v with
        hd :: tl ->
          v <- tl;
          Some hd
      | [] -> None

    method push hd =
      v <- hd :: v
  end;;
val s : < pop : int option; push : int -> unit > = <obj>
```

The object has an integer list value `v`, a method `pop` that returns the head of `v`, and a method `push` that adds an integer to the head of `v`.

The object type is enclosed in angle brackets `< ... >`, containing just the types of the methods. Fields, like `v`, are not part of the public interface of an object. All interaction with an object is through its methods. The syntax for a method invocation (also called "sending a message" to the object) uses the `#` character.

```
# s#pop;;
- : int option = Some 0
# s#push 4;;
- : unit = ()
# s#pop;;
- : int option = Some 4
```

Objects can also be constructed by functions. If we want to specify the initial value of the object, we can define a function that takes the value and returns an object.

```
# let stack init = object
    val mutable v = init

    method pop =
      match v with
        hd :: tl ->
          v <- tl;
          Some hd
      | [] -> None

    method push hd =
      v <- hd :: v
  end;;
val stack : 'a list -> < pop : 'a option; push : 'a -> unit > = <fun>
# let s = stack [3; 2; 1];;
val s : < pop : int option; push : int -> unit > = <obj>
# s#pop;;
- : int option = Some 3
```

Note that the types of the function `stack` and the returned object now use the polymorphic type `'a`. When `stack` is invoked on a concrete value `[3; 2; 1]`, we get the same object type as before, with type `int` for the values on the stack.

## Object Polymorphism

Like polymorphic variants, methods can be used without an explicit type declaration.

```
# let area sq = sq#width ** 2.;;
val area : < width : float; .. > -> float = <fun>
# let minimize sq : unit = sq#resize 1.;;
val minimize : < resize : float -> unit; .. > -> unit = <fun>
# let limit sq =
    if (area sq) > 10. then minimize sq;;
val limit : < resize : float -> unit; width : float; .. > -> unit = <fun>
```

As you can see object types are inferred automatically from the methods that are invoked on them.

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

```
# let toggle sq b : unit =
    if b then sq#resize `Fullscreen
    else minimize sq;;
Characters 76-78:
    else minimize sq;;
                  ^^
Error: This expression has type < resize : [> `Fullscreen ] -> unit; .. >
       but an expression was expected of type < resize : float -> unit; .. >
       Types for method resize are incompatible
```

The `..` in the inferred object types are ellipsis, standing for any other methods. The type `< width : float; .. >` specifies an object that must have at least an `width` method, and possibly some others as well. Such object types are said to be *open*.

We can manually *close* an object type using a type annotation:

```
# let area_closed (r: < width : float >) = r#width ** 2.;;
val area_closed : < width : float > -> float = <fun>
# let sq = object
    method width = 3.
    method name = "sq"
  end;;
val sq : < name : string; width : float > = <obj>
# area_closed sq;;
Characters 12-14:
  area_closed sq;;
              ^^
Error: This expression has type < name : string; width : float >
       but an expression was expected of type < width : float >
       The second object type has no method name
```

**Elisions are polymorphic**

The `..` in an open object type is an elision, standing for "possibly more methods." It may not be apparent from the syntax, but an elided object type is actually polymorphic. If we try to write a type definition, we get an obscure error.

```
# type square = < width : float; ..>;;
Characters 5-34:
  type square = < width : float; ..>;;
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: A type variable is unbound in this type declaration.
In type < width : float; .. > as 'a the variable 'a is unbound
```

A `..` in an object type is called a *row variable* and this typing scheme is called *row polymorphism*. Even though `..` doesn't look like a type variable, it actually is. Row polymorphism is also used in polymorphic variant types, and there is a close relationship between objects and polymorphic variants: objects are to records what polymorphic variants are to ordinary variants.

An object of type `< pop : int option; .. >` can be any object with a method `pop : int option`, it doesn't matter how it is implemented. When the method `#pop` is invoked, the actual method that is run is determined by the object.

```
# let print_pop st = Option.iter ~f:(printf "Popped: %d\n") st#pop;;
val print_pop : < pop : int Core.Std.Option.t; .. > -> unit = <fun>
# print_pop (stack [5;4;3;2;1]);;
Popped: 5
- : unit = ()
# let t = object
    method pop = Some (Float.to_int (Time.to_float (Time.now ())))
  end;;
val t : < pop : int option > = <obj>
# print_pop t;;
Popped: 1372618419
- : unit = ()
```

## Immutable objects

Many people consider object-oriented programming to be intrinsically imperative, where an object is like a state machine. Sending a message to an object causes it to change state, possibly sending messages to other objects.

Indeed, in many programs, this makes sense, but it is by no means required. Let's define a function that creates immutable stack objects.

```
# let imm_stack init = object
    val v = init
```

```
    method pop =
      match v with
        hd :: tl -> Some (hd, {< v = tl >})
      | [] -> None

    method push hd =
      {< v = hd :: v >}
  end;;
val imm_stack :
  'a list -> (< pop : ('a * 'b) option; push : 'a -> 'b > as 'b) = <fun>
```

The key parts of this implementation are in the `pop` and `push` methods. The expression `{< ... >}` produces a copy of the current object, with the same type, and the specified fields updated. In other words, the `push hd` method produces a copy of the object, with `v` replaced by `hd :: v`. The original object is not modified.

```
# let s = imm_stack [3; 2; 1];;
val s : < pop : (int * 'a) option; push : int -> 'a > as 'a = <obj>
# let t = s#push 4;;
val t : < pop : (int * 'a) option; push : int -> 'a > as 'a = <obj>
# s#pop;;
- : (int * (< pop : 'a; push : int -> 'b > as 'b)) option as 'a =
Some (3, <obj>)
# t#pop;;
- : (int * (< pop : 'a; push : int -> 'b > as 'b)) option as 'a =
Some (4, <obj>)
```

There are some restriction on the use of the expression `{< ... >}`. It can be used only within a method body, and only the values of fields may be updated. Method implementations are fixed at the time the object is created, they cannot be changed dynamically.

## When to use objects

You might wonder when to use objects in OCaml, which has a multitude of alternative mechanisms to express the similar concepts. First-class modules are more expressive (a module can include types, while classes and objects cannot). Modules, functors, and datatypes also offer a wide range of ways to express program structure. In fact, many seasoned OCaml programmers rarely use classes and objects, if at all.

Objects have some advantages over records: they don't require type definitions and their support for row polymorphism makes them more flexible. However, the heavy syntax and additional runtime cost means that objects are rarely used in place of records.

The real benefits of objects come from the class system. Classes support inheritance and open recursion. Open recursion allows parts of an object to be defined separately. This works because calls between the methods of an object are determined when the object is instantiated, a form of *dynamic* binding. This makes it possible (and necessary)

for one method to refer to other methods in the object without knowing statically how they will be implemented.

In contrast, modules use static binding. If you want to parameterize your module code so that some part of it can be implemented later, you would write a function or functor. This is more explicit, but often more verbose than overriding a method in a class.

In general, a rule of thumb is: use classes and objects in situations where open recursion is a big win. Two good examples are Xavier Leroy's Cryptokit (*http://gallium.inria.fr/~xleroy/software.html#cryptokit*), which provides a variety of cryptographic primitives that can be combined in building-block style, and the Camlimages (*http://cristal.inria.fr/camlimages/*) library which manipulates various graphical file formats.

We'll introduce you to classes, and examples using open recursion, in Chapter 12.

# Subtyping

Subtyping is a central concept in object-oriented programming. It governs when an object with one type *A* can be used in an expression that expects an object of another type *B*. When this is true, we say that *A* is a *subtype* of *B*. Actually, more concretely, subtyping determines when the coercion operator `e :> t` can be applied. This coercion works only if the expression `e` has some type `s` and `s` is a subtype of `t`.

## Width subtyping

To explore this, let's define some simple object types for geometric shapes. The generic type `shape` has a method to compute the area, and `square` and `circle` are specific kinds of shape.

```
type shape = < area : float >;;

type square = < area : float; width : float >;;

let square w = object
  method area = w *. w
  method width = w
end;;

type circle = < area : float; radius : float >;;

let circle r = object
  method area = 3.14 *. r ** 2.0
  method radius = r
end;;
```

A `square` has a method `area` just like a `shape`, and an additional method `width`. Still, we expect a `square` to be a `shape`, and it is. The coercion `:>` must be explicit.

```
# let shape w : shape = square w;;
Characters 22-30:
  let shape w : shape = square w;;
                        ^^^^^^^^
Error: This expression has type square but an expression was expected of type shape
       The second object type has no method width
# let shape w : shape = (square w :> shape);;
val shape : float -> shape = <fun>
```

This form of object subtyping is called *width* subtyping. Width subtyping means that an object type *A* is a a subtype of *B*, if *A* has all of the methods of *B*, and possibly more. A `square` is a subtype of `shape` because it implements all of the methods of `shape` (the `area` method).

## Depth subtyping

We can also use *depth* subtyping with objects. Depth subtyping, in its most general form, says that an object type `< m: t1 >` is a subtype of `< m: t2 >` iff `t1` is a subtype of `t2`.

For example, we can create two objects with a `shape` method:

```
# let coin = object
    method shape = circle 0.5
    method color = "silver"
  end
val coin : < color : string; shape : circle > = <obj>
# let map = object
    method shape = square 1.0
  end
val map : < shape : square > = <obj>
```

Both these objects have a `shape` method whose type is a subtype of the `shape` type, so they can both be coerced into the object type `< shape : shape >`

```
# type item = < shape : shape >;;
type item = < shape : shape >
# let items = [ (coin :> item) ; (map :> item) ];;
val items : item list = [<obj>; <obj>]
```

## Polymorphic variant subtyping

Subtyping can also be used to coerce a polymorphic variant into a larger polymorphic variant type.

```
# type num = [ `Int of int | `Float of float ];;
type num = [ `Float of float | `Int of int ]
# type const = [ num | `String of string ];;
type const = [ `Float of float | `Int of int | `String of string ]
```

```
# let n : num = `Int 3;;
val n : num = `Int 3
# let c : const = (n :> const);;
val c : const = `Int 3
```

## Variance

What about types built from object types? If a `square` is a `shape`, we expect a `square list` to be a `shape list`. OCaml does indeed allow such coercions:

```
# let squares: square list = [ square 1.0; square 2.0 ];;
val squares : square list = [<obj>; <obj>]
# let shapes: shape list = (squares :> shape list);;
val shapes : shape list = [<obj>; <obj>]
```

Note that this relies on lists being immutable. It would not be safe to treat a `square array` as a `shape array` because it would allow you to store non-square shapes into what should be an array of squares. OCaml recognises this and does not allow the coercion.

```
# let square_array: square array = [| square 1.0; square 2.0 |];;
val square_array : square array = [|<obj>; <obj>|]
# let shape_array: shape array = (square_array :> shape array);;
Characters 31-60:
  let shape_array: shape array = (square_array :> shape array);;
                                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: Type square array is not a subtype of shape array
Type square = < area : float; width : float > is not compatible with type
  shape = < area : float >
The second object type has no method width
```

We say that `'a list` is *covariant* (in `'a`), whilst `'a array` is *invariant*.

Subtyping function types requires a third class of variance. A function with type `square -> string` cannot be used with type `shape -> string` because it expects its argument to be a `square` and would not know what to do with a `circle`. However, a function with type `shape -> string` *can* safely be used with type `square -> string`.

```
# let shape_to_string: shape -> string =
    fun s -> sprintf "Shape(%F)" s#area;;
val shape_to_string : shape -> string = <fun>
# let square_to_string: square -> string =
    (shape_to_string :> square -> string);;
val square_to_string : square -> string = <fun>
```

We say that `'a -> string` is *contravariant* in `'a`. In general, function types are contravariant in their arguments and covariant in their results.

**Variance annotations**

OCaml works out the variance of a type using that type's definition.

```
# module Either = struct
  type ('a, 'b) t =
      Left of 'a
    | Right of 'b
end;;
module Either : sig type ('a, 'b) t = Left of 'a | Right of 'b end
# let sq : (square, circle) Either.t = Either.Left (square 4.0);;
val sq : (square, circle) Either.t = Either.Left <obj>
# let sh : (shape, shape) Either.t = (sq :> (shape, shape) Either.t);;
val sh : (shape, shape) Either.t = Either.Left <obj>
```

However, if the definition is hidden by a signature then OCaml is forced to assume that the type is invariant.

```
# module Either : sig
    type ('a, 'b) t
    val left: 'a -> ('a, 'b) t
    val right: 'b -> ('a, 'b) t
  end = struct
    type ('a, 'b) t =
        Left of 'a
      | Right of 'b
    let left x = Left x
    let right x = Right x
  end;;
module Either :
  sig
    type ('a, 'b) t
    val left : 'a -> ('a, 'b) t
    val right : 'b -> ('a, 'b) t
  end
# let sq : (square, circle) Either.t = Either.left (square 4.0);;
val sq : (square, circle) Either.t = <abstr>
# let sh : (shape, shape) Either.t = (sq :> (shape, shape) Either.t);;
Characters 35-66:
  let sh : (shape, shape) Either.t = (sq :> (shape, shape) Either.t);;
                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: Type (square, circle) Either.t is not a subtype of
         (shape, shape) Either.t
       Type square = < area : float; width : float > is not compatible with type
         shape = < area : float >
       The second object type has no method width
```

We can fix this by adding *variance annotations* to the type's parameters in the signature: + for covariance or - for contravariance.

```
# module Either : sig
    type (+'a, +'b) t
    val left: 'a -> ('a, 'b) t
    val right: 'b -> ('a, 'b) t
  end = struct
    type ('a, 'b) t =
        Left of 'a
      | Right of 'b
    let left x = Left x
```

```
    let right x = Right x
  end;;
module Either :
  sig
    type (+'a, +'b) t
    val left : 'a -> ('a, 'b) t
    val right : 'b -> ('a, 'b) t
  end
# let sq : (square, circle) Either.t = Either.left (square 4.0);;
val sq : (square, circle) Either.t = <abstr>
# let sh : (shape, shape) Either.t = (sq :> (shape, shape) Either.t);;
val sh : (shape, shape) Either.t = <abstr>
```

For a more concrete example of variance, let's create some stacks containing shapes by applying our `stack` function to some squares and some circles.

```
# type 'a stack = < pop: 'a option; push: 'a -> unit >;;
type 'a stack = < pop : 'a option; push : 'a -> unit >
# let square_stack: square stack = stack [square 3.0; square 1.0];;
val square_stack : square stack = <obj>
# let circle_stack: circle stack = stack [circle 2.0; circle 4.0];;
val circle_stack : circle stack = <obj>
```

If we wanted to write a function that took a list of such stacks and found the total area of their shapes, we might try:

```
# let total_area (shape_stacks: shape stack list) =
    let stack_area acc st =
      let rec loop acc =
        match st#pop with
          Some s -> loop (acc +. s#area)
        | None -> acc
      in
        loop acc
    in
      List.fold ~init:0.0 ~f:stack_area shape_stacks;;
val total_area : shape stack list -> float = <fun>
```

However, when we try to apply this function to our objects we get an error:

```
# total_area [(square_stack :> shape stack); (circle_stack :> shape stack)];;
Characters 12-41:
  total_area [(square_stack :> shape stack); (circle_stack :> shape stack)];;
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: Type square stack = < pop : square option; push : square -> unit >
       is not a subtype of
         shape stack = < pop : shape option; push : shape -> unit >
       Type shape = < area : float > is not a subtype of
         square = < area : float; width : float >
```

As you can see, `square stack` and `circle stack` are not subtypes of `shape stack`. The problem is with the `push` method. For `shape stack`, the `push` method takes an arbitrary

shape. So if we could coerce a square stack to a shape stack, then it would be possible to push an arbitrary shape onto square stack, which would be an error.

Another way of looking at this is that < push: 'a -> unit; .. > is contravariant in 'a, so < push: square -> unit; pop: square option > cannot be a subtype of < push: shape -> unit; pop: shape option >.

Still, the total_area function should be fine, in principle. It doesn't call push, so it isn't making that error. To make it work, we need to use a more precise type that indicates we are not going to be using the set method. We define a type readonly_stack and confirm that we can coerce the list of stacks to it.

```
# type 'a readonly_stack = < pop : 'a option >;;
type 'a readonly_stack = < pop : 'a option >
# let total_area (shape_stacks: shape readonly_stack list) =
    let stack_area acc st =
      let rec loop acc =
        match st#pop with
          Some s -> loop (acc +. s#area)
        | None -> acc
      in
        loop acc
    in
      List.fold ~init:0.0 ~f:stack_area shape_stacks;;
val total_area : shape readonly_stack list -> float = <fun>
# total_area [(square_stack :> shape readonly_stack); (circle_stack :> shape readonly_stack)];;
- : float = 72.8000000000000114
```

Aspects of this section may seem fairly complicated, but it should be pointed out that this typing *works*, and in the end the type annotations are fairly minor. In most typed object-oriented languages, these coercions would simply not be possible. For example, in C++, a STL type list<T> is invariant in T, it is simply not possible to use list<square> where list<shape> is expected (at least safely). The situation is similar in Java, although Java has an escape hatch that allows the program to fall back to dynamic typing. The situation in OCaml is much better; it works, it is statically checked, and the annotations are pretty simple.

## Narrowing

Narrowing, also called *down casting*, is the ability to coerce an object to one of its subtypes. For example, if we have a list of shapes shape list, we might know (for some reason) what the actual type of each shape is. Perhaps we know that all objects in the list have type square. In this case, *narrowing* would allow the re-casting of the object from type shape to type square. Many languages support narrowing through dynamic type checking. For example, in Java, a coercion (Square) x is allowed if the value x has type Square or one of its subtypes; otherwise the coercion throws an exception.

Narrowing is *not permitted* in OCaml. Period.

Why? There are two reasonable explanations, one based on a design principle, and another technical (the technical reason is simple: it is hard to implement).

The design argument is this: narrowing violates abstraction. In fact, with a structural typing system like in OCaml, narrowing would essentially provide the ability to enumerate the methods in an object. To check whether an object `obj` has some method `foo : int`, one would attempt a coercion `(obj :> < foo : int >)`.

More commonly, narrowing leads to poor object-oriented style. Consider the following Java code, which returns the name of a shape object.

```
String GetShapeName(Shape s) {
  if (s instanceof Square) {
    return "Square";
  } else if (s instanceof Circle) {
    return "Circle";
  } else {
    return "Other";
  }
}
```

Most programmers would consider this code to be "wrong." Instead of performing a case analysis on the type of object, it would be better to define a method to return the name of the shape. Instead of calling `GetShapeName(s)`, we should call `s.Name()` instead.

However, the situation is not always so obvious. The following code checks whether an array of shapes looks like a "barbell," composed to two `Circle` objects separated by a `Line`, where the circles have the same radius.

```
boolean IsBarbell(Shape[] s) {
  return s.length == 3 && (s[0] instanceof Circle) &&
    (s[1] instanceof Line) && (s[2] instanceof Circle) &&
    ((Circle) s[0]).radius() == ((Circle) s[2]).radius();
}
```

In this case, it is much less clear how to augment the `Shape` class to support this kind of pattern analysis. It is also not obvious that object-oriented programming is well-suited for this situation. Pattern matching seems like a better fit.

```
let is_barbell = function
 | [Circle r1; Line _; Circle r2] when r1 == r2 -> true
 | _ -> false;;
```

Regardless, there is a solution if you find yourself in this situation, which is to augment the classes with variants. You can define a method `variant` that injects the actual object into a variant type.

```
type shape = < variant : repr; area : float>
and circle = < variant : repr; area : float; radius : float >
and line = < variant : repr; area : float; length : float >
```

```
and repr =
 | Circle of circle
 | Line of line;;

let is_barbell = function
 | [s1; s2; s3] ->
   (match s1#variant, s2#variant, s3#variant with
     | Circle c1, Line _, Circle c2 when c1#radius == c2#radius -> true
     | _ -> false)
 | _ -> false;;
```

This pattern works, but it has drawbacks. In particular, the recursive type definition should make it clear that this pattern is essentially equivalent to using variants, and that objects do not provide much value here.

## Subtyping vs. Row Polymorphism

There is a great deal of overlap between subtyping and row polymorphism. Row polymorphism is in general preferred over subtyping because it does not require explicit coercions, and it preserves more type information, allowing functions like the following:

```
# let remove_large l =
    List.filter ~f:(fun s -> s#area < 10.0) l;;
val remove_large :
  (< area : float; .. > as 'a) Core.Std.List.t -> 'a Core.Std.List.t = <fun>
```

The return type of this function is built from the open object type of its argument, preserving any additional methods that it may have.

```
# let squares : < area : float; width : float > list =
    [square 3.0; square 4.0; square 2.0];;
val squares : < area : float; width : float > list = [<obj>; <obj>; <obj>]
# remove_large squares;;
- : < area : float; width : float > Core.Std.List.t = [<obj>; <obj>]
```

Writing a similar function with a closed type and applying it using subtyping does not preserve the methods of the argument: the returned object is only known to have an `area` method.

```
# let remove_large (l: < area : float > list) =
    List.filter ~f:(fun s -> s#area < 10.0) l;;
val remove_large : < area : float > list -> < area : float > Core.Std.List.t =
  <fun>
# remove_large (squares :> < area : float > list );;
- : < area : float > Core.Std.List.t = [<obj>; <obj>]
```

However, there are some situations where we cannot use row polymorphism. For example, lists of heterogeneous elements can not be created using row polymorphism:

```
# let hlist: < area: float; ..> list = [square 1.0; circle 3.0];;
Characters 50-60:
  let hlist: < area: float; ..> list = [square 1.0; circle 3.0];;
                                                    ^^^^^^^^^^
Error: This expression has type circle but an expression was expected of type
         square
       The second object type has no method radius
```

Since row polymorphism is a form of polymorphism, it also does not work well with references:

```
# let shape_ref: < area: float; ..> ref = ref (square 4.0);;
val shape_ref : < area : float; width : float > ref = {contents = <obj>}
# shape_ref := circle 2.0;;
Characters 13-23:
  shape_ref := circle 2.0;;
               ^^^^^^^^^^
Error: This expression has type circle but an expression was expected of type
         < area : float; width : float >
       The first object type has no method width
```

In both these cases we must use subtyping:

```
# let hlist: shape list = [(square 1.0 :> shape); (circle 3.0 :> shape)];;
val hlist : shape list = [<obj>; <obj>]
# let shape_ref: shape ref = ref (square 4.0 :> shape);;
val shape_ref : shape ref = {contents = <obj>}
# shape_ref := (circle 2.0 :> shape);;
- : unit = ()
```

**Production note**

This chapter contains significant external contributions from Leo White.

2013-07-04
10:28:30

2013-07-04
10:28:30

# CHAPTER 12
# Classes

Programming with objects directly is great for encapsulation, but one of the main goals of object-oriented programming is code reuse through inheritance. For inheritance, we need to introduce *classes*. In object-oriented programming, a class is a "recipe" for creating objects. The recipe can be changed by adding new methods and fields, or it can be changed by modifying existing methods.

## OCaml Classes

In OCaml, class definitions must be defined as toplevel statements in a module. A class is not an object, and a class definition is not an expression. The syntax for a class definition uses the keyword `class`.

```
# class point =
  object
    val mutable x = 0
    method get = x
    method set y = x <- y
  end;;
class point :
  object
    val mutable x : int
    method get : int
    method set : int -> unit
  end
```

The type `class point : ... end` is a *class type*. This particular type specifies that the `point` class defines a mutable field `x`, a method `get` that returns an `int`, and a method `set` with type `int -> unit`.

To produce an object, classes are instantiated with the keyword `new`.

```
# let p = new point;;
val p : point = <obj>
# p#get;;
```

```
- : int = 0
# p#set 5;;
- : unit = ()
# p#get;;
- : int = 5
```

Inheritance uses an existing class to define a new one. For example, the following class definition supports an addition method `moveby` that moves the point by a relative amount.

```
# class movable_point =
  object (self : 'self)
    inherit point
    method moveby dx = self#set (self#get + dx)
  end;;
class movable_point :
  object
    val mutable x : int
    method get : int
    method moveby : int -> unit
    method set : int -> unit
  end
```

This new `movable_point` class also makes use of the `(self : 'self)` binding after the `object` keyword. The variable `self` stands for the current object, allowing self-invocation, and the type variable `'self` stands for the type of the current object (which in general is a subtype of `movable_point`).

## An Example: Cryptokit

Let's take a break from describing the object system with a more practical example that uses the OCaml cryptographic library.

**Installing the Cryptokit library**

The Cryptokit library can be installed via OPAM via `opam install cryptokit`. Once that's finished compiling and installing, you just need to `#require "cryptokit"` in your toplevel to load the library and make the modules available.

Our first example mimics the `md5` command, which reads in an input file and returns a hexadecimal representation of its MD5 cryptographic hash. Cryptokit defines a number of different functions and collects them together under the `Cryptokit.hash` class type:

```
class type hash = object
  method add_byte : int -> unit
  method add_char : char -> unit
  method add_string : string -> unit
  method add_substring : string -> int -> int -> unit
```

```
  method hash_size : int
  method result : string
  method wipe : unit
end

val hash_string : hash -> string -> string
```

Concrete hash objects can be instantiated from various sub-modules in Cryptokit. The simplest ones such as MD5 or SHA1 do not take any special input parameters to build the object. The `hmac_sha1` takes a string key to initialise the Message Authenticate Code for that particular hash function.

```
# Cryptokit.Hash.md5;;
- : unit -> Cryptokit.hash = <fun>
# Cryptokit.Hash.sha1;;
- : unit -> Cryptokit.hash = <fun>
# Cryptokit.MAC.hmac_sha1;;
- : string -> Cryptokit.hash = <fun>
```

Hash objects hold state and are thus naturally imperative. Once instantiated, data is fed into them by the addition functions, the `result` is computed and finally the contents erased via `wipe`. The `hash_string` convenience function applies the hash function fully to a string, and returns the result. The `md5` command is quite straight-forward now:

```
open Core.Std
open Cryptokit

let () =
  In_channel.(input_all stdin)
  |> hash_string (Hash.md5 ())
  |> transform_string (Hexa.encode ())
  |> print_endline
```

After opening the right modules, we read in the entire standard input into an OCaml string. This is then passed onto the MD5 hash function, which returns a binary string. This binary is passed through the `Hexa` hexadecimal encoder, which returns an ASCII representation of the input. The output of this command will be the same as the `md5` command (or `md5sum` in some systems).

We can extend this simple example by selecting either the `md5` or `sha1` hash function at runtime depending on the name of our binary. `Sys.argv` is an array containing the arguments the command was invoked with, and the first entry is the name of the binary itself.

```
open Core.Std
open Cryptokit

let () =
  let hash_fn =
    match Filename.basename Sys.argv.(0) with
    |"md5" -> Hash.md5 ()
```

```
    |"sha1" -> Hash.sha1 ()
    |_ -> Hash.md5 ()
  in
  In_channel.(input_all stdin)
  |> hash_string hash_fn
  |> transform_string (Hexa.encode ())
  |> print_endline
```

Now let's try something more advanced. The `openssl` library is installed on most systems, and can be used to encrypt plaintext using several encryption strategies. At its simplest, it will take a secret phrase and derive an appropriate key and initialisation vector.

```
$ openssl enc -nosalt -aes-128-cbc -base64 -k "ocaml" -P
key=6217C07FF169F6AB2EB2731F855095F1
iv =8164D5477E66E6A9EC99A8D58ACAADAF
```

We've selected the `-nosalt` option here to make the output deterministic, and the `-P` option prints out the derived key and IV and exits. The algorithm used to derive these results is described in the `man EVP_BytesToKey` manual page (you may need to install the OpenSSL documentation packages on your system first). We can implement this derivation function using an imperative style:

```
let md5 s = hash_string (Hash.md5 ()) s

let evp_byte_to_key password tlen =
  let o = Hexa.encode () in
  let v = ref (md5 password) in
  o#put_string !v;
  while o#available_output/2 < tlen do
    let n = md5 (!v ^ password) in
    o#put_string n;
    v := n;
  done;
  String.uppercase o#get_string

let () =
  let secret = "ocaml" in
  let key_len = 16 * 2 in
  let iv_len = 16 * 2 in
  let x = evp_byte_to_key secret (key_len+iv_len) in
  let key = String.sub x ~pos:0 ~len:key_len in
  let iv = String.sub x ~pos:key_len ~len:iv_len in
  Printf.printf "key=%s\niv =%s\n%!" key iv
```

The derivation algorithm takes an input password and desired total length (the addition of the key and IV length). It initialises a `Hexa.encode` transformer, which will accept arbitrary binary data and output a hexadecimal string (with two output bytes per input byte). A reference stores the last digest that's been calculated, and then the algorithm iterates until it has sufficient data to satisfy the required key length.

Notice how the encoder object is used as an accumulator, by using the `put_string` and `available_output` to keep track of progress. Objects don't *require* an imperative style though, and the same algorithm can be written more functionally:

```
let evp_byte_to_key password tlen =
  let rec aux acc v =
    match String.length acc < tlen with
    | true ->
      let v = md5 (v ^ password) in
      aux (acc^v) v
    | false -> acc
  in
  let v = md5 password in
  String.uppercase (transform_string (Hexa.encode ()) (aux v v))
```

In this version, we don't use any references, and instead a recursive function keeps track of the last digest in use and the accumulated result string. This version isn't quite as efficient as the previous one due to the careless use of string concatenation for the accumulator, but this can easily be fixed by using the `Buffer` module instead.

## Class parameters and polymorphism

A class definition serves as the *constructor* for the class. In general, a class definition may have parameters that must be provided as arguments when the object is created with `new`.

Let's build an example of an imperative singly-linked list using object-oriented techniques. First, we'll want to define a class for a single element of the list. We'll call it a `node`, and it will hold a value of type `'a`. When defining the class, the type parameters are placed in square brackets before the class name in the class definition. We also need a parameter `x` for the initial value.

```
class ['a] node x =
object
  val mutable value : 'a = x
  val mutable next_node : 'a node option = None

  method get = value
  method set x = value <- x

  method next = next_node
  method set_next node = next_node <- node
end;;
```

The `value` is the value stored in the node, and it can be retrieved and changed with the `get` and `set` methods. The `next_node` field is the link to the next element in the stack. Note that the type parameter `['a]` in the definition uses square brackets, but other uses of the type can omit them (or use parentheses if there is more than one type parameter).

The type annotations on the `val` declarations are used to constrain type inference. If we omit these annotations, the type inferred for the class will be "too polymorphic," `x` could have some type `'b` and `next_node` some type `'c option`.

```
  class ['a] node x =
  object
    val mutable value = x
    val mutable next_node = None

    method get = value
    method set x = value <- x

    method next = next_node
    method set_next node = next_node <- node
  end;;
Error: Some type variables are unbound in this type:
         class ['a] node :
           'b ->
           object
             val mutable next_node : 'c option
             val mutable value : 'b
             method get : 'b
             method next : 'c option
             method set : 'b -> unit
             method set_next : 'c option -> unit
           end
       The method get has type 'b where 'b is unbound
```

In general, we need to provide enough constraints so that the compiler will infer the correct type. We can add type constraints to the parameters, to the fields, and to the methods. It is a matter of preference how many constraints to add. You can add type constraints in all three places, but the extra text may not help clarity. A convenient middle ground is to annotate the fields and/or class parameters, and add constraints to methods only if necessary.

Next, we can define the list itself. We'll keep a field `head` that refers to the first element in the list, and `last` that refers to the final element in the list. The method `insert` adds an element to the end of the list.

```
class ['a] slist =
object
   val mutable first : ('a) node option = None
   val mutable last : ('a) node option = None

   method is_empty = first = None

   method insert x =
      let new_node = Some (new node x) in
      match last with
         Some last_node ->
            last_node#set_next new_node;
            last <- new_node
```

```
        | None ->
            first <- new_node;
            last <- new_node
end;;
```

## Object types

This definition of the class `slist` is not complete, we can construct lists, but we also need to add the ability to traverse the elements in the list. One common style for doing this is to define a class for an `iterator` object. An iterator provides a generic mechanism to inspect and traverse the elements of a collection. This pattern isn't restricted to lists, it can be used for many different kinds of collections.

There are two common styles for defining abstract interfaces like this. In Java, an iterator would normally be specified with an interface, which specifies a set of method types. In languages without interfaces, like C++, the specification would normally use *abstract* classes to specify the methods without implementing them (C++ uses the "= 0" definition to mean "not implemented").

```
// Java-style iterator, specified as an interface.
interface <T> iterator {
  T Get();
  boolean HasValue();
  void Next();
};

// Abstract class definition in C++.
template<typename T>
class Iterator {
 public:
  virtual ~Iterator() {}
  virtual T get() const = 0;
  virtual bool has_value() const = 0;
  virtual void next() = 0;
};
```

OCaml support both styles. In fact, OCaml is more flexible than these approaches because an object type can be implemented by any object with the appropriate methods; it does not have to be specified by the object's class *a priori*. We'll leave abstract classes for later. Let's demonstrate the technique using object types.

First, we'll define an object type `iterator` that specifies the methods in an iterator.

```
type 'a iterator = < get : 'a; has_value : bool; next : unit >;;
```

Next, we'll define an actual iterator for the class `slist`. We can represent the position in the list with a field `current`, following links as we traverse the list.

```
class ['a] slist_iterator cur =
object
```

```
  val mutable current : 'a node option = cur

  method has_value = current <> None

  method get =
     match current with
        Some node -> node#get
      | None -> raise (Invalid_argument "no value")

  method next =
     match current with
        Some node -> current <- node#next
      | None -> raise (Invalid_argument "no value")
end;;
```

Finally, we add a method `iterator` to the slist class to produce an iterator. To do so, we construct an `slist_iterator` that refers to the first node in the list, but we want to return a value with the object type `iterator`. This requires an explicit coercion using the `:>` operator.

```
class ['a] slist = object
...
   method iterator = (new slist_iterator first :> 'a iterator)
end

# let l = new slist;;
# l.insert 5;;
# l.insert 4;;
# let it = l#iterator;;
# it#get;;
- : int = 5
# it#next;;
- : unit = ()
# it#get;;
- : int = 4
# it#next;;
- : unit = ()
# it#has_value;;
- : bool = false
```

We may also wish to define functional-style methods, `iter f` takes a function `f` and applies it to each of the elements of the list.

```
method iter f =
  let it = self#iterator in
  while it#has_value do
    f it#get
    it#next
  done
```

What about functional operations similar to `List.map` or `List.fold`? In general, these methods take a function that produces a value of some other type than the elements of the set. For example, the function `List.fold` has type `'a list -> ('b -> 'a -> 'b) -`

`> 'b -> 'b`, where `'b` is an arbitrary type. To replicate this in the `slist` class, we need a method type `('b -> 'a -> 'b) -> 'b -> 'b`, where the method type is polymorphic over `'b`.

The solution is to use a type quantifier, as shown in the following example. The method type must be specified directly after the method name, which means that method parameters must be expressed using a `fun` or `function` expression.

```
method fold : 'b. ('b -> 'a -> 'b) -> 'b -> 'b =
   (fun f x ->
         let y = ref x in
         let it = self#iterator in
         while it#has_value do
            y := f !y it#get;
            it#next
         done;
         !y)
```

## Class types

Once we have defined the list implementation, the next step is to wrap it in a module or `.ml` file and give it a type so that it can be used in the rest of our code. What is the type?

Before we begin, let's wrap up the implementation in an explicit module (we'll use explicit modules for illustration, but the process is similar when we want to define a `.mli` file). In keeping with the usual style for modules, we define a type `'a t` to represent the type of list values.

```
module SList = struct
   type 'a iterator = < get : 'a; has_value : bool; next : unit >
   type 'a t = < is_empty : bool; insert : 'a -> unit; iterator : 'a iterator >

   class ['a] node x = object ... end
   class ['a] slist_iterator cur = object ... end
   class ['a] slist = object ... end

   let make () = new slist
end;;
```

We have multiple choices in defining the module type, depending on how much of the implementation we want to expose. At one extreme, a maximally-abstract signature would completely hide the class definitions.

```
module AbstractSList : sig
   type 'a iterator = < get : 'a; has_value : bool; next : unit >
   type 'a t = < is_empty : bool; insert : 'a -> unit; iterator : 'a iterator >

   val make : unit -> 'a t
end = SList
```

The abstract signature is simple because we ignore the classes. But what if we want to include them in the signature, so that other modules can inherit from the class definitions? For this, we need to specify types for the classes, called *class types*. Class types do not appear in mainstream object-oriented programming languages, so you may not be familiar with them, but the concept is pretty simple. A class type specifies the type of each of the visible parts of the class, including both fields and methods. Just like for module types, you don't have to give a type for everything; anything you omit will be hidden.

```
module VisibleSList : sig
  type 'a iterator = < get : 'a; has_value : bool; next : unit >
  type 'a t = < is_empty : bool; insert : 'a -> unit; iterator : 'a iterator >

  class ['a] node : 'a ->
  object
     method get : 'a
     method set : 'a -> unit
     method next : 'a node option
     method set_next : 'a node option -> unit
  end

  class ['a] slist_iterator : 'a node option ->
  object
     method has_value : bool
     method get : 'a
     method next : unit
  end

  class ['a] slist :
  object
    val mutable first : 'a node option
    val mutable last : 'a node option
    method is_empty : bool
    method insert : 'a -> unit
    method iterator : 'a iterator
  end

  val make : unit -> 'a slist
end = SList
```

In this signature, we've chosen to make nearly everything visible. The class type for `slist` specifies the types of the fields `first` and `last`, as well as the types of each of the methods. We've also included a class type for `slist_iterator`, which is of somewhat more questionable value, since the type doesn't appear in the type for `slist` at all.

One more thing, in this example the function `make` has type `unit -> 'a slist`. But wait, we've stressed *classes are not types*, so what's up with that? In fact, what we've said is entirely true, classes and class names *are not* types. However, class names can be used to stand for types. When the compiler sees a class name in type position, it automatically constructs an object type from it by erasing all the fields and keeping only the method types. In this case, the type expression `'a slist` is exactly equivalent to `'a t`.

## Binary methods

A *binary method* is a method that takes an object of `self` type. One common example is defining a method for equality.

```
# class square w =
  object (self : 'self)
    method width = w
    method area = self#width * self#width
    method equals (other : 'self) = other#width = self#width
  end;;
class square : int ->
  object ('a)
    method area : int
    method equals : 'a -> bool
    method width : int
  end
# class rectangle w h =
  object (self : 'self)
    method width = w
    method height = h
    method area = self#width * self#height
    method equals (other : 'self) = other#width = self#width && other#height = self#height
  end;;
...
# (new square 5)#equals (new square 5);;
- : bool = true
# (new rectangle 5 6)#equals (new rectangle 5 7);;
- : bool = false
```

This works, but there is a problem lurking here. The method `equals` takes an object of the exact type `square` or `rectangle`. Because of this, we can't define a common base class `shape` that also includes an equality method.

```
# type shape = < equals : shape -> bool; area : int >;;
# let sq = new square 5;;
# (sq :> shape);;
Characters 0-13:
  (sq :> shape);;
  ^^^^^^^^^^^^^
Error: Type square = < area : int; equals : square -> bool; width : int >
       is not a subtype of shape = < area : int; equals : shape -> bool >
Type shape = < area : int; equals : shape -> bool > is not a subtype of
  square = < area : int; equals : square -> bool; width : int >
```

The problem is that a `square` expects to be compared with a `square`, not an arbitrary shape; similarly for `rectangle`.

This problem is fundamental. Many languages solve it either with narrowing (with dynamic type checking), or by method overloading. Since OCaml has neither of these, what can we do?

One proposal we could consider is, since the problematic method is equality, why not just drop it from the base type `shape` and use polymorphic equality instead? Unfortunately, the builtin equality has very poor behavior when applied to objects.

```
# (object method area = 5 end) = (object method area = 5 end);;
- : bool = false
```

The problem here is that the builtin polymorphic equality compares the method implementations, not their return values. The method implementations (the function values that implement the methods) are different, so the equality comparison is false. There are other reasons not to use the builtin polymorphic equality, but these false negatives are a showstopper.

If we want to define equality for shapes in general, the remaining solution is to use the same approach as we described for narrowing. That is, introduce a *representation* type implemented using variants, and implement the comparison based on the representation type.

```
type shape_repr =
 | Square of int
 | Circle of int
 | Rectangle of int * int;;

type shape = < repr : shape_repr; equals : shape -> bool; area : int >;;

class square w =
object (self : 'self)
  method width = w
  method area = self#width * self#width
  method repr = Square self#width
  method equals (other : shape) = self#repr = other#repr
end;;
```

The binary method `equals` is now implemented in terms of the concrete type `shape_repr`. In fact, the objects are now isomorphic to the `shape_repr` type. When using this pattern, you will not be able to hide the `repr` method, but you can hide the type definition using the module system.

```
module Shapes : sig
  type shape_repr
  type shape = < repr : shape_repr; equals : shape -> bool; area -> int >

  class square : int ->
    object
      method width : int
      method area : int
      method repr : shape_repr
      method equals : shape -> bool
    end
end = struct
  type shape_repr = Square of int | Circle of int | Rectangle of int * int
```

```
  ...
end;;
```

## Private methods

Methods can be declared *private*, which means that they may be called by subclasses, but they are not visible otherwise (similar to a *protected* method in C++).

To illustrate, let's build a class `vector` that contains an array of integers, resizing the storage array on demand. The field `values` contains the actual values, and the `get`, `set`, and `length` methods implement the array access. For clarity, the resizing operation is implemented as a private method `ensure_capacity` that resizes the array if necessary.

```
# class vector =
  object (self : 'self)
     val mutable values : int array = [||]

     method get i = values.(i)
     method set i x =
        self#ensure_capacity i;
        values.(i) <- x
     method length = Array.length values

     method private ensure_capacity i =
        if self#length <= i then
           let new_values = Array.create (i + 1) 0 in
           Array.blit values 0 new_values 0 (Array.length values);
           values <- new_values
  end;;
# let v = new vector;;
# v#set 5 2;;
# v#get 5;;
- 2 : int
# v#ensure_capacity 10;;
Characters 0-1:
  v#ensure_capacity 10;;
  ^
Error: This expression has type vector
       It has no method ensure_capacity
```

To be precise, the method `ensure_capacity` is part of the class type, but it is not part of the object type. This means the object `v` has no method `ensure_capacity`. However, it is available to subclasses. We can extend the class, for example, to include a method `swap` that swaps two elements.

```
# class swappable_vector =
  object (self : 'self)
     inherit vector

     method swap i j =
        self#ensure_capacity (max i j);
```

```
        let tmp = values.(i) in
        values.(i) <- values.(j);
        values.(j) <- tmp
  end;;
```

Yet another reason for private methods is to factor the implementation and support recursion. Moving along with this example, let's build a binary heap, which is a binary tree in heap order: where the label of parent elements is smaller than the labels of its children. One efficient implementation is to use an array to represent the values, where the root is at index 0, and the children of a parent node at index `i` are at indexes `2 * i` and `2 * i + 1`. To insert a node into the tree, we add it as a leaf, and then recursively move it up the tree until we restore heap order.

```
class binary_heap =
object (self : 'self)
   val values = new swappable_vector

   method min =
      if values#length = 0 then
         raise (Invalid_argument "heap is empty");
      values#get 0

   method add x =
      let pos = values#length in
      values#set pos x;
      self#move_up pos

   method private move_up i =
      if i > 0 then
         let parent = (i - 1) / 2 in
            if values#get i < values#get parent then begin
               values#swap i parent;
               self#move_up parent
            end
end;;
```

The method `move_up` implements the process of restoring heap order as a recursive method (though it would be straightforward avoid the recursion and use iteration here).

The key property of private methods is that they are visible to subclasses, but not anywhere else. If you want the stronger guarantee that a method is *really* private, not even accessible in subclasses, you can use an explicit typing that omits the method. In the following code, the `move_up` method is explicitly omitted from the object type, and it can't be invoked in subclasses.

```
# class binary_heap :
  object
    method min : int
    method add : int -> unit
  end =
  object (self : 'self) {
    ...
```

```
  method private move_up i = ...
end;;
```

## Virtual classes and methods

A *virtual* class is a class where some methods or fields are declared, but not implemented. This should not be confused with the word "virtual" as it is used in C++. In C++, a "virtual" method uses dynamic dispatch, regular non-virtual methods use static dispatched. In OCaml, *all* methods use dynamic dispatch, but the keyword *virtual* means the method or field is not implemented.

In the previous section, we defined a class `swappable_vector` that inherits from `array_vector` and adds a `swap` method. In fact, the `swap` method could be defined for any object with `get` and `set` methods; it doesn't have to be the specific class `array_vector`.

One way to do this is to declare the `swappable_vector` abstractly, declaring the methods `get` and `set`, but leaving the implementation for later. However, the `swap` method can be defined immediately.

```
class virtual abstract_swappable_vector =
object (self : 'self)
   method virtual get : int -> int
   method virtual set : int -> int -> unit
   method swap i j =
      let tmp = self#get i in
      self#set i (self#get j);
      self#set j tmp
end;;
```

At some future time, we may settle on a concrete implementation for the vector. We can inherit from the `abstract_swappable_bvector` to get the `swap` method "for free." Here's one implementation using arrays.

```
class array_vector =
object (self : 'self)
   inherit abstract_swappable_vector

   val mutable values = [||]
   method get i = values.(i)
   method set i x =
      self#ensure_capacity i;
      values.(i) <- x
   method length = Array.length values

   method private ensure_capacity i =
      if self#length <= i then
         let new_values = Array.create (i + 1) 0 in
            Array.blit values 0 new_values 0 (Array.length values);
```

```
            values <- new_values
end
```

Here's a different implementation using `HashTbl`.

```
class hash_vector =
object (self : 'self)
   inherit abstract_swappable_vector

   val table = Hashtbl.create 19

   method get i =
      try Hashtbl.find table i with
         Not_found -> 0

   method set = Hashtbl.add table
end;;
```

One way to view a `virtual` class is that it is like a functor, where the "inputs" are the declared, but not defined, virtual methods and fields. The functor application is implemented through inheritance, when virtual methods are given concrete implementations.

We've been mentioning that fields can be virtual too. Here is another implementation of the swapper, this time with direct access to the array of values.

```
class virtual abstract_swappable_array_vector =
object (self : 'self)
   val mutable virtual values : int array
   method private virtual ensure_capacity : int -> unit

   method swap i j =
      self#ensure_capacity (max i j);
      let tmp = values.(i) in
      values.(i) <- values.(j);
      values.(j) <- tmp
end;;
```

This level of dependency on the implementation details is possible, but it is hard to justify the use of a virtual class -- why not just define the `swap` method as part of the concrete class? Virtual classes are better suited for situations where there are multiple (useful) implementations of the virtual parts. In most cases, this will be public virtual methods.

## Multiple inheritance

When a class inherits from more than one superclass, it is using *multiple inheritance*. Multiple inheritance extends the variety of ways in which classes can be combined, and it can be quite useful, particularly with virtual classes. However, it can be tricky to use,

particularly when the inheritance hierarchy is a graph rather than a tree, so it should be used with care.

## How names are resolved

The main "trickiness" of multiple inheritance is due to naming -- what happens when a method or field with some name is defined in more than one class?

If there is one thing to remember about inheritance in OCaml, it is this: inheritance is like textual inclusion. If there is more than one definition for a name, the last definition wins. Let's look at some artificial, but illustrative, examples.

First, let's consider what happens when we define a method more than once. In the following example, the method `get` is defined twice; the second definition "wins," meaning that it overrides the first one.

```
# class m1 =
object (self : 'self)
   method get = 1
   method f = self#get
   method get = 2
end;;
class m1 : object method f : int method get : int end
# (new m1)#f;;
- : int = 2
```

Fields have similar behavior, though the compiler produces a warning message about the override.

```
# class m2 =
# class m2 =
  object (self : 'self)
     val x = 1
     method f = x
     val x = 2
  end;;
Characters 69-74:
     val x = 2
         ^^^^^
Warning 13: the instance variable x is overridden.
The behaviour changed in ocaml 3.10 (previous behaviour was hiding.)
class m2 : object val x : int method f : int end
# (new m2)#f;;
- : int = 2
```

Of course, it is unlikely that you will define two methods or two fields of the same name in the same class. However, the rules for inheritance follow the same pattern: the last definition wins. In the following definition, the `inherit` declaration comes last, so the method definition `method get = 2` overrides the previous definition, always returning 2.

```
# class m4 = object method get = 2 end;;
# class m5 =
  object
    val mutable x = 1
    method get = x
    method set x' = x <- x'
    inherit m4
  end;;
class m5 : object val mutable x : int method get : int method set : int -> unit end
# let x = new m5;;
val x : m5 = <obj>
# x#set 5;;
- : unit = ()
# x#get;;
- : int = 2
```

To reiterate, to understand what inheritance means, replace each `inherit` directive with its definition, and take the last definition of each method or field. This holds even for private methods. However, it does *not* hold for private methods that are "really" private, meaning that they have been hidden by a type constraint. In the following definitions, there are three definitions of the private method `g`. However, the definition of `g` in `m8` is not overridden, because it is not part of the class type for `m8`.

```
# class m6 =
  object (self : 'self)
     method f1 = self#g
     method private g = 1
  end;;
class m6 : object method f1 : int method private g : int end
# class m7 =
  object (self : 'self)
     method f2 = self#g
     method private g = 2
  end;;
class m7 : object method f2 : int method private g : int end
# class m8 : object method f3 : int end =
  object (self : 'self)
     method f3 = self#g
     method private g = 3
  end;;
class m8 : object method f3 : int end
# class m9 =
  object (self : 'self)
     inherit m6
     inherit m7
     inherit m8
  end;;
# class m9 :
  object
    method f1 : int
    method f2 : int
    method f3 : int
    method private g : int
```

```
  end
# let x = new m9;;
val x : m9 = <obj>
# x#f1;;
- : int = 2
# x#f3;;
- : int = 3
```

## Mixins

When should you use multiple inheritance? If you ask multiple people, you're likely to get multiple (perhaps heated) answers. Some will argue that multiple inheritance is overly complicated; others will argue that inheritance is problematic in general, and one should use object composition instead. But regardless of who you talk to, you will rarely hear that multiple inheritance is great and you should use it widely.

In any case, if you're programming with objects, there's one general pattern for multiple inheritance that is both useful and reasonably simple, the *mixin* pattern. Generically, a *mixin* is just a virtual class that implements a feature based on another one. If you have a class that implements methods *A*, and you have a mixin *M* that provides methods *B* from *A*, then you can inherit from *M* -- "mixing" it in -- to get features *B*.

That's too abstract, so let's give an example based on collections. In Section XXX:Objecttypes, we introduced the *iterator* pattern, where an *iterator* object is used to enumerate the elements of a collection. Lots of containers can have iterators, singly-linked lists, dictionaries, vectors, etc.

```
type 'a iterator = < get : 'a; has_value : bool; next : unit >;;
class ['a] slist : object ... method iterator : 'a iterator end;;
class ['a] vector : object ... method iterator : 'a iterator end;;
class ['a] deque : object ... method iterator : 'a iterator end;;
class ['a, 'b] map : object ... method iterator : 'b iterator end;;
...
```

The collections are different is some ways, but they share a common pattern for iteration that we can reuse. For a simple example, let's define a mixin that implements an arithmetic sum for a collection of integers.

```
# class virtual int_sum_mixin =
  object (self : 'self)
     method virtual iterator : int iterator
     method sum =
        let it = self#iterator in
        let total = ref 0 in
        while it#has_value do
           total := !total + it#get;
           it#next
        done;
        !total
  end;;
```

```
# class int_slist =
  object
     inherit [int] slist
     inherit int_sum_mixin
  end;;
# let l = new int_slist;;
val l : int_slist = <obj>
# l#insert 5;;
# l#insert 12;;
# l#sum;;
- : int = 17
# class int_deque =
  object
     inherit [int] deque
     inherit int_sum_mixin
  end;;
```

In this particular case, the mixin works only for a collection of integers, so we can't add the mixin to the polymorphic class definition `['a] slist` itself. However, the result of using the mixin is that the integer collection has a method `sum`, and it is done with very little of the fuss we would need if we used object composition instead.

The mixin pattern isn't limited to non-polymorphic classes, of course. We can use it to implement generic features as well. The following mixin defines functional-style iteration in terms of the imperative iterator pattern.

```
class virtual ['a] fold_mixin =
object (self : 'self)
   method virtual iterator : 'a iterator
   method fold : 'b. ('b -> 'a -> 'b) -> 'b -> 'b =
      (fun f x ->
            let y = ref x in
            let it = self#iterator in
            while it#has_value do
               y := f !y it#get;
               it#next
            done;
            !y)
end;;

class ['a] slist_with_fold =
object
   inherit ['a] slist
   inherit ['a] fold_mixin
end;;
```

**Production note**

This chapter contains significant external contributions from Leo White.

# PART II

# Tools and Techniques

Part II builds on the basics by working through useful tools and techniques for using OCaml. Here you'll pick up useful techniques for building networked systems, as well as functional design patterns that help combine different features of the language to good effect.

The focus throughout this section is on networked systems, and among other examples we'll build a running example that will perform Internet queries using the DuckDuckGo search engine.

2013-07-04
10:28:30

2013-07-04
10:28:30

# CHAPTER 13
# Maps and Hash Tables

Lots of programming problems require dealing with data organized as key/value pairs. Maybe the simplest way of representing such data in OCaml is an *association list*, which is simply a list of pairs of keys and values. For example, you could represent a mapping between the 10 digits and their English names as follows.

```
# let digit_alist =
    [ 0, "zero"; 1, "one"; 2, "two"  ; 3, "three"; 4, "four"
    ; 5, "five"; 6, "six"; 7, "seven"; 8, "eight"; 9, "nine" ]
  ;;
```

We can use functions from the `List.Assoc` module to manipulate such an association list.

```
# List.Assoc.find digit_alist 6;;
- : string option = Some "six"
# List.Assoc.find digit_alist 22;;
- : string option = None
# List.Assoc.add digit_alist 0 "zilch";;
- : (int, string) List.Assoc.t =
[(0, "zilch"); (1, "one"); (2, "two"); (3, "three"); (4, "four");
 (5, "five"); (6, "six"); (7, "seven"); (8, "eight"); (9, "nine")]
```

Association lists are simple and easy to use, but their performance is not ideal, since almost every non-trivial operation on an association list requires a linear-time scan of the list.

In this chapter, we'll talk about two more efficient alternatives to association lists: *maps* and *hash tables*. A map is an immutable tree-based data structure where most operations take time logarithmic in the size of the map, whereas a hash table is a mutable data structure where most operations have constant time complexity. We'll describe both of these data structures in detail, and provide some advice as to how to choose between them.

## Maps

Let's consider an example of how one might use a map in practice. In Chapter 4, we showed a module `Counter` for keeping frequency counts on a set of strings. Here's the interface.

```
(* counter.mli *)
open Core.Std

type t

val empty : t
val touch : t -> string -> t
val to_list : t -> (string * int) list
```

The intended behavior here is straightforward. `Counter.empty` represents an empty collection of frequency counts; `touch` increments the frequency count of the specified string by 1; and `to_list` returns the list of non-zero frequencies.

Here's the implementation.

```
(* counter.ml *)
open Core.Std

type t = int String.Map.t

let empty = String.Map.empty

let to_list t = Map.to_alist t

let touch t s =
  let count = Option.value ~default:0 (Map.find t s) in
  Map.add t ~key:s ~data:(count + 1)
```

Note that in some places the above code refers to `String.Map.t`, and in others `Map.t`. This has to do with the fact that maps are implemented as ordered binary trees, and as such, need a way of comparing keys.

To deal with this, a map, once created, stores the necessary comparison function within the data structure. Thus, operations like `Map.find` or `Map.add` that access the contents of a map or create a new map from an existing one, do so by using the comparison function embedded within the map.

But in order to get a map in the first place, you need to get your hands on the comparison function somehow. For this reason, modules like `String` contain a `Map` sub-module that have values like `String.Map.empty` and `String.Map.of_alist` that are specialized to strings, and thus have access to a string comparison function. Such a `Map` sub-module is included in every module that satisfies the `Comparable.S` interface from Core.

## Creating maps with comparators

The specialized `Map` sub-module is convenient, but it's not the only way of creating a `Map.t`. The information required to compare values of a given type is wrapped up in a value called a *comparator*, that can be used to create maps using the `Map` module directly.

```
# let digit_map = Map.of_alist_exn digit_alist
                    ~comparator:Int.comparator;;
val digit_map : (int, string, Int.comparator) Map.t = <abstr>
# Map.find digit_map 3;;
- : string option = Some "three"
```

The above uses `Map.of_alist_exn` which creates a map from an association list, throwing an exception if there are duplicate keys in the list.

The comparator is only required for operations that create maps from scratch. Operations that update an existing map simply inherit the comparator of the map they start with.

```
# let zilch_map = Map.add digit_map ~key:0 ~data:"zilch";;
val zilch_map : (int, string, Int.comparator) Map.t = <abstr>
```

The type `Map.t` has three type parameters: one for the key, one for the value, and one to identify the comparator. Indeed, the type `'a Int.Map.t` is just a type alias for `(int,'a,Int.comparator) Map.t`

Including the comparator in the type is important because because operations that work on multiple maps at the same time often require that the maps share their comparison function. Consider, for example, `Map.symmetric_diff`, which computes a summary of the differences between two maps.

```
# let left = String.Map.of_alist_exn ["foo",1; "bar",3; "snoo", 0]
  let right = String.Map.of_alist_exn ["foo",0; "snoo", 0]
  let diff = Map.symmetric_diff ~data_equal:Int.equal left right
  ;;
val left : int String.Map.t = <abstr>
val right : int String.Map.t = <abstr>
val diff :
  (string * [ `Left of int | `Right of int | `Unequal of int * int ]) list =
  [("foo", `Unequal (1, 0)); ("bar", `Left 3)]
```

The type of `Map.symmetric_diff`, shown below, requires that the two maps it compares have the same comparator type. Each comparator has a fresh abstract type, so the type of a comparator identifies the comparator uniquely.

```
# Map.symmetric_diff;;
- : ('k, 'v, 'cmp) Map.t ->
    ('k, 'v, 'cmp) Map.t ->
    data_equal:('v -> 'v -> bool) ->
```

```
    ('k * [ `Left of 'v | `Right of 'v | `Unequal of 'v * 'v ]) list
= <fun>
```

This constraint is important because the algorithm that `Map.symmetric_diff` uses depends on the fact that both maps have the same comparator.

We can create a new comparator using the `Comparator.Make` functor, which takes as its input a module containing the type of the object to be compared, sexp-converter functions, and a comparison function. The sexp converters are included in the comparator to make it possible for users of the comparator to generate better error messages. Here's an example.

```
# module Reverse = Comparator.Make(struct
    type t = string
    let sexp_of_t = String.sexp_of_t
    let t_of_sexp = String.t_of_sexp
    let compare x y = String.compare y x
  end);;
module Reverse :
  sig
    type t = string
    val compare : t -> t -> int
    val t_of_sexp : Sexp.t -> t
    val sexp_of_t : t -> Sexp.t
    type comparator
    val comparator : (t, comparator) Comparator.t_
  end
```

As you can see below, both `Reverse.comparator` and `String.comparator` can be used to create maps with a key type of `string`.

```
# let alist = ["foo", 0; "snoo", 3];;
val alist : (string * int) list = [("foo", 0); ("snoo", 3)]
# let ord_map = Map.of_alist_exn ~comparator:String.comparator alist;;
val ord_map : (string, int, String.comparator) Map.t = <abstr>
# let rev_map = Map.of_alist_exn ~comparator:Reverse.comparator alist;;
val rev_map : (string, int, Reverse.comparator) Map.t = <abstr>
```

`Map.min_elt` returns the key and value for the smallest key in the map, which lets us see that these two maps do indeed use different comparison functions.

```
# Map.min_elt ord_map;;
- : (string * int) option = Some ("foo", 0)
# Map.min_elt rev_map;;
- : (string * int) option = Some ("snoo", 3)
```

And accordingly, if we try to use `Map.symmetric_diff` on these two maps, we'll get a compile-timer error.

```
# Map.symmetric_diff ord_map rev_map;;

Error: This expression has type (string, int, Reverse.comparator) Map.t
```

```
but an expression was expected of type
  ('a, 'b, 'c) Map.t = (string, int, String.comparator) Map.t
Type Reverse.comparator is not compatible with type String.comparator
```

## Trees

As we've discussed, maps carry within them the comparator that they were created with. Sometimes, often for space efficiency reasons, you want a version of the map data structure that doesn't include the comparator. You can get such a representation with `Map.to_tree`, which returns just the tree that the map is built out of, and not including the comparator.

```
# let ord_tree = Map.to_tree ord_map;;
val ord_tree : (string, int, String.comparator) Map.Tree.t = <abstr>
```

Even though a `Map.Tree.t` doesn't physically include a comparator, it does include the comparator in its type. This is what is known as a *phantom type parameter*, because it reflects something about the logic of value in question, even though it doesn't correspond to any values directly represented in the underlying physical structure of the value.

Since the comparator isn't included in the tree, we need to provide the comparator explicitly when we, say, search for a key, as shown below.

```
# Map.Tree.find ~comparator:String.comparator ord_tree "snoo";;
- : int option = Some 3
```

The algorithm of `Map.Tree.find` depends on the fact that it's using the same comparator when looking a value up as you were when you stored it. That's the invariant that the phantom type is there to enforce. As you can see below, using the wrong comparator will lead to a type error.

```
# Map.Tree.find ~comparator:Reverse.comparator ord_tree "snoo";;

Error: This expression has type (string, int, String.comparator) Map.Tree.t
       but an expression was expected of type
         ('a, 'b, 'c) Map.Tree.t = (string, 'b, Reverse.comparator) Map.Tree.t
       Type String.comparator is not compatible with type Reverse.comparator
```

## The polymorphic comparator

We don't need to generate specialized comparators for every type we want to build a map on. We can instead use a comparator based on OCaml's build-in polymorphic comparison function, which was discussed in Chapter 3. This comparator is found in the `Comparator.Poly` module, allowing us to write:

```
# Map.of_alist_exn ~comparator:Comparator.Poly.comparator digit_alist;;
- : (int, string, Comparator.Poly.comparator) Map.t = <abstr>
```

Or, equivalently:

```
# Map.Poly.of_alist_exn digit_alist;;
- : (int, string) Map.Poly.t = <abstr>
```

Note that maps based on the polymorphic comparator are not equivalent to those based on the type-specific comparators from the point of view of the type system. Thus, the compiler rejects the following:

```
# Map.symmetric_diff (Map.Poly.singleton 3 "three")
                     (Int.Map.singleton  3 "four" ) ;;

Error: This expression has type 'a Int.Map.t = (int, 'a, Int.comparator) Map.t
       but an expression was expected of type
         ('b, 'c, 'd) Map.t = (int, string, Z.Poly.comparator) Map.t
       Type Int.comparator is not compatible with type Z.Poly.comparator
```

This is rejected for good reason: there's no guarantee that the comparator associated with a given type will order things in the same way that polymorphic compare does.

#### The difference between = and ==, and `phys_equal` in Core

If you come from a C/C++ background, you'll probably reflexively use == to test two values for equality. In OCaml, the == operator tests for *physical* equality while the = operator tests for *structural* equality.

The physical equality test will match if two data structures have precisely the same pointer in memory. Two data structures that have identical contents but are constructed separately will not match using ==.

The = structural equality operator recursively inspects each field in the two values and tests them individually for equality. Crucially, if your data structure is cyclical (that is, a value recursively points back to another field within the same structure), the = operator will never terminate, and your program will hang! You therefore must use the physical equality operator or write a custom comparison function when comparing recursive values.

It's quite easy to mix up the use of = and ==, so Core disables the == operator and provides the more explicit `phys_equal` function instead. You'll see a type error if you use == anywhere in code that uses opens the Core standard module.

```
# open Core.Std;;
# 1 == 2;;
Error: This expression has type int but an expression was expected of type
         [ `Consider_using_phys_equal ]
```

```
# phys_equal 1 2;;
- : bool = false
```

If you feel like hanging your OCaml interpreter, you can verify what happens with recursive values and structural equality for yourself:

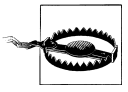
```
# type t1 = { foo1:int; bar1:t2 } and t2 = { foo2:int; bar2:t1 } ;;
type t1 = { foo1 : int; bar1 : t2; }
and t2 = { foo2 : int; bar2 : t1; }
# let rec v1 = { foo1=1; bar1=v2 } and v2 = { foo2=2; bar2=v1 };;
<lots of text>
# v1 == v1;;
- : bool = true
# phys_equal v1 v1;;
- : bool = true
# v1 = v1 ;;
<press ^Z and kill the process now>
```

## Sets

Sometimes, instead of keeping track of a set of key/value pairs, you just want a data-type for keeping track of a set of keys. You could build this on top of a map by representing a set of values by a map whose data type is `unit`. But a more idiomatic (and efficient) solution is to use Core's set type, which is similar in design and spirit to the map type, while having an API better tuned to working with sets, and a lower memory footprint. Here's a simple example:

```
# let dedup ~comparator l =
    List.fold l ~init:(Set.empty ~comparator) ~f:Set.add
    |> Set.to_list
  ;;
val dedup : comparator:('a, 'b) Core.Comparator.t_ -> 'a list -> 'a list =
  <fun>
# dedup ~comparator:Int.comparator [8;3;2;3;7;8;10];;
- : int list = [2; 3; 7; 8; 10]
```

In addition to the operators you would expect to have for maps, sets support the traditional set operations, including union, intersection and set difference. And, as with maps, we can create sets based on type-specific comparators or on the polymorphic comparator.



#### The perils of polymorphic compare

Polymorphic compare is highly convenient, but it has serious downsides as well, and should be used with care. In particular, polymorphic compare has a fixed algorithm for comparing values of any type, and that algorithm can sometimes yield surprising results.

To understand what's wrong with polymorphic compare, you need to understand a bit about how it works. Polymorphic compare is *structural*, in that it operates directly on the runtime-representation of OCaml values, walking the structure of the values in question without regard for their type.

This is convenient because it provides a comparison function that works for most OCaml values, and largely behaves as you would expect. For example, on `int`s and `float`s it acts as you would expect a numeric comparison function to act. For simple containers like strings and lists and arrays it operates as a lexicographic comparison. And except for closures and values from outside of the OCaml heap, it works on almost every OCaml type.

But sometimes, a structural comparison is not what you want. Sets are a great example of this. Consider the following two sets.

```
# let (s1,s2) = (Int.Set.of_list [1;2],
                 Int.Set.of_list [2;1]);;
```

```
val s1 : Int.Set.t = <abstr>
val s2 : Int.Set.t = <abstr>
```

Logically, these two sets should be equal, and that's the result that you get if you call `Set.equal` on them.

```
# Set.equal s1 s2;;
- : bool = true
```

But because the elements were added in different orders, the layout of the trees underlying the sets will be different. As such, a structural comparison function will conclude that they're different.

Let's see what happens if we use polymorphic compare to test for equality by way of the `=` operator. Comparing the maps directly will fail at runtime because the comparators stored within the sets contain function values.

```
# s1 = s2;;
Exception: (Invalid_argument "equal: functional value").
```

We can however use the function `Set.to_tree` to expose the underlying tree without the attached comparator.

```
# Set.to_tree s1 = Set.to_tree s2;;
- : bool = false
```

This can cause real and quite subtle bugs. If, for example, you use a map whose keys contain sets, then the map built with the polymorphic comparator will behave incorrectly, separating out keys that should be aggregated together. Even worse, it will work sometimes and fail others, since if the sets are built in a consistent order, then they will work as expected, but once the order changes, the behavior will change.

For this reason, it's preferable to avoid polymorphic compare for serious applications.

## Satisfying the `Comparable.S` interface

Core's `Comparable.S` interface includes a lot of useful functionality, including support for working with maps and sets. In particular, `Comparable.S` requires the presence of the `Map` and `Set` sub-modules as well as a comparator.

`Comparable.S` is satisfied by most of the types in Core, but the question arises of how to satisfy the comparable interface for a new type that you design. Certainly implementing all of the required functionality from scratch would be an absurd amount of work.

The module `Comparable` contains a number of functors to help you do just this. The simplest one of these is `Comparable.Make`, which takes as an input any module that satisfies the following interface:

```
sig
  type t
  val sexp_of_t : t -> Sexp.t
  val t_of_sexp : Sexp.t -> t
  val compare : t -> t -> int
end
```

In other words, it expects a type with a comparison function as well as functions for converting to and from *s-expressions*. S-expressions are a serialization format used commonly in Core, which we'll discuss more in Chapter 17. In the meantime, we can just use the `with sexp` declaration that comes from the `sexplib` syntax extension to create s-expression converters for us. S-expression converters can also be written by hand.

The following example shows how this all fits together, following the same basic pattern for using functors described in "Extending modules" on page 177.

```
# module Foo_and_bar : sig
    type t = { foo: Int.Set.t; bar: string }
    include Comparable.S with type t := t
  end = struct
    module T = struct
      type t = { foo: Int.Set.t; bar: string } with sexp
      let compare t1 t2 =
        let c = Int.Set.compare t1.foo t2.foo in
        if c <> 0 then c else String.compare t1.bar t2.bar
    end
    include T
    include Comparable.Make(T)
  end;;
```

We don't include the full response from the top-level because it is quite lengthy, but `Foo_and_bar` does satisfy `Comparable.S`.

In the above, we wrote the comparison function by hand, but this isn't strictly necessary. Core ships with a syntax extension called `comparelib` which will create a comparison function from a type definition. Using it, we can rewrite the above example as follows.

```
# module Foo_and_bar : sig
    type t = { foo: Int.Set.t; bar: string }
    include Comparable.S with type t := t
  end = struct
    module T = struct
      type t = { foo: Int.Set.t; bar: string } with sexp, compare
    end
    include T
    include Comparable.Make(T)
  end;;
```

The comparison function created by `comparelib` for a given type will call out to the comparison functions for its component types. As a result, the `foo` field will be com-

pared using `Int.Set.compare`. This is different, and saner, than the structural comparison done by polymorphic compare.

If you want your comparison function to behave in a specific way, you should still write your own comparison function by hand; but if all you want is a total order suitable for creating maps and sets with, then `comparelib` is a good way to go.

You can also satisfy the `Comparable.S` interface using polymorphic compare.

```
# module Foo_and_bar : sig
    type t = { foo: int; bar: string }
    include Comparable.S with type t := t
  end = struct
    module T = struct
      type t = { foo: int; bar: string } with sexp
    end
    include T
    include Comparable.Poly(T)
  end;;
```

That said, for reasons we discussed earlier, polymorphic compare should be used sparingly.

## Hash tables

Hash tables are the imperative cousin of maps. We walked over a basic hash table implementation in Chapter 8, so in this section we'll mostly discuss the pragmatics of Core's `Hashtbl` module. We'll cover this material more briefly than we did with maps, because many of the concepts are shared.

Hash tables differ from maps in a few key ways. First, hash tables are mutable, meaning that adding a key/value pair to a hash table modifies the table, rather than creating a new table with the binding added. Second, hash tables generally have better time-complexity than maps, providing constant time lookup and modifications as opposed to logarithmic for maps. And finally, just as maps depend on having a comparison function for creating the ordered binary tree that underlies a map, hash tables depend on having a *hash function*, *i.e.*, a function for converting a key to an integer.

**Time complexity of hash tables**

The statement that hash tables provide constant-time access hides some complexities. First of all, any hash table implementation, OCaml's included, needs to resize the table when it gets too full. A resize requires allocating a new backing array for the hash table and copying over all entries, and so it is quite an expensive operation. That means adding a new element to the table is only *amortized* constant, which is to say, it's constant on average over a long sequence of additions, but some of the indivdual additions can be quite expensive.

Another hidden cost of hash tables has to do with the hash function you use. If you end up with a pathologically bad hash function that hashes all of your data to the same number, then all of your insertions will hash to the same underlying bucket, meaning you no longer get constant-time access at all. Core's hash table implementation uses binary trees for the hash-buckets, so this case only leads to logarithmic time, rather than quadratic for a traditional hash table.

When creating a hash table, we need to provide a value of type *hashable* which includes among other things the function for hashing the key type. This is analogous to the comparator used for creating maps.

```
# let table = Hashtbl.create ~hashable:String.hashable ();;
val table : (string, '_a) Hashtbl.t = <abstr>
# Hashtbl.replace table ~key:"three" ~data:3;;
- : unit = ()
# Hashtbl.find table "three";;
- : int option = Some 3
```

The `hashable` value is included as part of the `Hashable.S` interface, which is satisfied by most types in Core. The `Hashable.S` interface also includes a `Table` sub-module which provides more convenient creation functions.

```
# let table = String.Table.create ();;
val table : '_a String.Table.t = <abstr>
```

There is also a polymorphic `hashable` value, corresponding to the polymorphic hash function provided by the OCaml runtime, for cases where you don't have a hash function for your specific type.

```
# let table = Hashtbl.create ~hashable:Hashtbl.Poly.hashable ();;
val table : ('_a, '_b) Hashtbl.t = <abstr>
```

Or, equivalently:

```
# let table = Hashtbl.Poly.create ();;
val table : ('_a, '_b) Hashtbl.t = <abstr>
```

Note that, unlike the comparators used with maps and sets, hashables don't show up in the type of a `Hashtbl.t`. That's because hash tables don't have operations that operate on multiple hash tables that depend on those tables having the same hash function, in that way that `Map.symmetric_diff` and `Set.union` depend on their arguments using the same comparison function.

**Collisions with the polymorphic hash function**

OCaml's polymorphic hash function works by walking over the data-structure its given using a breadth-first traversal that is bounded in the number of nodes its willing to traverse. By default, that bound is set at 10 "meaningful" nodes, essentially...

The bound on the traversal, means that the hash function may ignore part of the data-structure, and this can lead to pathological cases where every value you store has the same hash value. By default, OCaml's hash function will stop after it has found ten nodes it can extract data from. We'll demonstrate this below, using the function `List.range` to allocate lists of integers of different length.

```
# Caml.Hashtbl.hash (List.range 0 9);;
- : int = 209331808
# Caml.Hashtbl.hash (List.range 0 10);;
- : int = 182325193
# Caml.Hashtbl.hash (List.range 0 11);;
- : int = 182325193
# Caml.Hashtbl.hash (List.range 0 100);;
- : int = 182325193
```

As you can see, the hash function stops after the first 10 elements. The same can happen with any large data structure, including records and arrays. When building hash functions over large custom data-structures, it is generally a good idea to write one's own hash function, *e.g.*,

## Satisfying the `Hashable.S` interface

Most types in Core satisfy the `Hashable.S` interface, but as with the `Comparable.S` interface, the question remains of how one should satisfy this interface with a new type. Again, the answer is to use a functor to build the necessary functionality; in this case, `Hashable.Make`. Note that we use OCaml's `lxor` operator for doing the "logical" (*i.e.*, bit-wise) exclusive-or of the hashes from the component values.

```
# module Foo_and_bar : sig
    type t = { foo: int; bar: string }
    include Hashable.S with type t := t
  end = struct
    module T = struct
      type t = { foo: int; bar: string } with sexp, compare
      let hash t =
        (Int.hash t.foo) lxor (String.hash t.bar)
    end
```

```
  include T
  include Hashable.Make(T)
end;;
```

Note that in order to satisfy hashable, one also needs to provide a comparison function. That's because Core's hash tables use ordered binary tree data-structure for the hash-buckets, so that performance of the table degrades gracefully in the case of pathologically bad choice of hash function.

There is currently no analogue of `comparelib` for auto-generation of hash-functions, so you do need to either write the hash-function by hand, or use the built-in polymorphic hash function, `Hashtbl.hash`.

## Choosing between maps and hash tables

Maps and hash tables overlap enough in functionality that it's not always clear when to choose one or the other. Maps, by virtue of being immutable, are generally the default choice in OCaml by virtue of fitting most naturally with otherwise functional code. OCaml also has good support for imperative programming, though, and when programming in an imperative idiom, hash tables are often the more natural choice.

Programming idioms aside, there are significant performance differences between maps and hash tables as well. For code that is dominated by updates and lookups, hash tables are a clear performance win, and the win is clearer the larger the size of the tables.

The best way of answering a performance question is by running a benchmark, so let's do just that. The following benchmark uses the `core_bench` library, and it compares maps and hash tables under a very simple workload. Here, we're keeping track of a set of 1000 different integer keys, and cycling over the keys and updating the values they contain. Note that we use the `Map.change` and `Hashtbl.change` functions to update the respective data structures.

```
(* file: map_vs_hash.ml *)

open Core.Std
open Core_bench.Std

let map_iter ~num_keys ~iterations =
  let rec loop i map =
    if i <= 0 then ()
    else loop (i - 1)
           (Map.change map (i mod num_keys) (fun current ->
              Some (1 + Option.value ~default:0 current)))
  in
  loop iterations Int.Map.empty

let table_iter ~num_keys ~iterations =
  let table = Int.Table.create ~size:num_keys () in
  let rec loop i =
    if i <= 0 then ()
```

```
    else (
      Hashtbl.change table (i mod num_keys) (fun current ->
        Some (1 + Option.value ~default:0 current));
      loop (i - 1)
    )
  in
  loop iterations

let tests ~num_keys ~iterations =
  let test name f = Bench.Test.create f ~name in
  [ test "map"   (fun () -> map_iter   ~num_keys ~iterations)
  ; test "table" (fun () -> table_iter ~num_keys ~iterations)
  ]

let () =
  tests ~num_keys:1000 ~iterations:100_000
  |> Bench.make_command
  |> Command.run
```

The results, shown below, show the hash table version to be around four times faster than the map version.

```
bench $ ./map_vs_hash.native -clear-columns name time speedup
Estimated testing time 20s (change using -quota SECS).
┌─────────┬──────────────┬─────────┐
│ Name    │ Time (ns)    │ Speedup │
├─────────┼──────────────┼─────────┤
│ map     │ 31_584_468   │    1.00 │
│ table   │  8_157_439   │    3.87 │
└─────────┴──────────────┴─────────┘
```

We can make the speedup smaller or larger depending on the details of the test; for example, it will very with the number of distinct keys. But overall, for code that is heavy on sequences of querying and updating a set of key/value pairs, hash tables will significantly outperform maps.

Hash tables are not always the faster choice, though. In particular, maps are often more performant in situations where you want to take advantage of maps as a persistent data-structure. In particular, if you create map `m'` by calling `Map.add` on some other map `m`, then `m` and `m'` can be used independently, and in fact share most of their underlying storage. Thus, if you need to keep in memory at the same time multiple different related collections of key/value pairs, then a map is typically a much more efficient data structure to do it with.

Here's a benchmark to demonstrates this. In it, we create a list of maps (or hash tables) that are built up by iteratively applying updates, starting from an empty map. In the hash table implementation, we do this by calling `Hashtbl.copy` to get the list entries.

```
(* file: map_vs_hash2.ml *)

open Core.Std
open Core_bench.Std
```

```
let create_maps ~num_keys ~iterations =
  let rec loop i map =
    if i <= 0 then []
    else
      let new_map =
        Map.change map (i mod num_keys) (fun current ->
          Some (1 + Option.value ~default:0 current))
      in
      new_map :: loop (i - 1) new_map
  in
  loop iterations Int.Map.empty

let create_tables ~num_keys ~iterations =
  let table = Int.Table.create ~size:num_keys () in
  let rec loop i =
    if i <= 0 then []
    else (
      Hashtbl.change table (i mod num_keys) (fun current ->
        Some (1 + Option.value ~default:0 current));
      let new_table = Hashtbl.copy table in
      new_table :: loop (i - 1)
    )
  in
  loop iterations

let tests ~num_keys ~iterations =
  let test name f = Bench.Test.create f ~name in
  [ test "map"   (fun () -> ignore (create_maps   ~num_keys ~iterations))
  ; test "table" (fun () -> ignore (create_tables ~num_keys ~iterations))
  ]

let () =
  tests ~num_keys:50 ~iterations:1000
  |> Bench.make_command
  |> Command.run
```

Unsurprisingly, maps perform far better than hash tables on this benchmark, in this case by more than a factor of ten.

```
$ ./map_vs_hash2.native -clear-columns name time speedup
Estimated testing time 20s (change using -quota SECS).
┌─────────┬──────────────┬─────────┐
│ Name    │ Time (ns)    │ Speedup │
├─────────┼──────────────┼─────────┤
│ map     │    208_438   │   12.62 │
│ table   │  2_630_707   │    1.00 │
└─────────┴──────────────┴─────────┘
```

These numbers can be made more extreme by increasing the size of the tables or the length of the list.

2013-07-04
10:28:30

As you can see, the relative performance of trees and maps depends a great deal on the details of how they're used, and so whether to choose one data structure or the other will depend on the details of the application.

2013-07-04
10:28:30

2013-07-04
10:28:30

# CHAPTER 14
# Command Line Parsing

Many of the OCaml programs that you'll write will end up as binaries that need to be run from a command prompt. Any non-trivial command-line program needs a few features:

- program options and file inputs need to be parsed from the command line arguments.
- sensible error messages have to be generated in response to incorrect inputs.
- help needs to be shown for all the available options.
- interactive auto-completion of commands to assist the user.

It's tedious and error-prone to code all this manually for every program you write. Core provides the Command library that simplifies all this by letting you declare all your command-line options in one place. Command takes care of parsing the arguments, generating help text and provides interactive auto-completion to the user of the library.

Command also copes as you add more features to your programs. It's a simple library to use for small applications, and provides a sophisticated subcommand mode that grouping related commands together as the number of options grow. You may already be familiar with this command-line style from the Git or Mercurial version control systems.

In this chapter, we'll:

- learn how to use Command to construct basic and grouped command-line interfaces.
- read examples that extend the cryptographic utility from Chapter 12 and build a simple equivalent to the `md5` and `shasum` utilities.
- demonstrate how *functional combinators* can be used to declare complex data structures in a type-safe and elegant way.

## Basic command-line parsing

Let's start by cloning the `md5` binary that is present on most Linux and Mac OS X installations. It reads in the contents of a file, applies the MD5 one-way cryptographic hash function to the data, and outputs an ASCII hex representation of the result.

```
(* basic_md5.ml : calculate MD5 hash of input *)
open Core.Std

let do_hash file =
  let open Cryptokit in
  In_channel.read_all file
  |> hash_string (Hash.md5 ())
  |> transform_string (Hexa.encode ())
  |> print_endline

let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    ~readme:(fun () -> "More detailed information")
    Command.Spec.(
      empty
      +> anon ("filename" %: string)
    )
  (fun filename () -> do_hash filename)

let () = Command.run ~version:"1.0" ~build_info:"RWO" command
```

You can compile this file the usual way with `ocamlfind` by passing an additional `cryptokit` package. Install Cryptokit via OPAM if you didn't do so earlier.

```
$ opam install cryptokit
$ ocamlfind ocamlopt -thread -package cryptokit -package core -linkpkg \
  -o md5 basic_md5.ml
$ ./md5
```

The `do_hash` function accepts a filename parameter and prints the human-readable MD5 string to the console standard output.

The subsequent `command` variable defines how to invoke `do_hash` by parsing the command-line arguments. When you compile and run this program, it already defines a number of useful default command-line options.

For instance, query the version information for the binary you just compiled.

```
$ ./md5 -version
1.0
$ ./md5 -build-info
RWO
```

The actual versions are defined via optional arguments to `Command.run`. You can leave these blank or get your build system to generate them directly from your version control

system (e.g. by running `hg tip` to generate a build revision number, in the case of Mercurial).

```
$ ./md5
Generate an MD5 hash of the input data

  md5 filename

More detailed information

=== flags ===

  [-build-info]  print info about this build and exit
  [-version]     print the version of this build and exit
  [-help]        print this help text and exit
                 (alias: -?)

missing anonymous argument: filename
```

When we invoke this binary without any arguments, it helpfully displays a help screen that informs you that a required argument `filename` is missing.

If you do supply the filename argument, then `do_hash` is called with the argument and the MD5 output is displayed to the standard output.

```
$ ./md5 ./md5
59562f5e4f790d16f1b2a095cd5de844
```

# Defining parsing specifications

So how does all this work? Most of the interesting logic lies in how the specifications are constructed.

The `Command.Spec` module defines several combinators that can be chained together to define flags and anonymous arguments, what types they should map to, and whether to take special actions (such as interactive input) if certain fields are encountered.

## Anonymous arguments

Let's build the specification for a single argument that is specified directly on the command-line (this is known as an *anonymous* argument).

```
Command.Spec.(
  empty
  +> anon ("filename" %: string)
)
```

The specification above begins with an `empty` value and then adds more parameters via the `+>` combinator. Our example uses the `anon` function to define a single anonymous parameter.

Anonymous parameters are created using the `%:` operator, which binds a textual string name (used in the help text) to an OCaml conversion function. The conversion function is responsible for parsing the command-line fragment into an OCaml data type. In the example above, this is just a `string`, but we'll see more complex conversion options below.

## Callback functions

This specification is usually combined with a callback function that accepts all the command-line parameters as its function arguments. Multiple arguments are passed to the callback in the same order as they appeared in the specification (using the `+>` operator).

The callback function is where all the actual work happens after the command-line parsing is complete. This function is applied with the arguments containing the parsed command-line arguments, and takes over as the main thread of the application.

In our example, we had just one anonymous argument, so the callback function just has a single `string` parameter applied to it:

```
(fun file () -> do_hash file)
```

**The extra `unit` argument to callbacks**

The callback above needs an extra `unit` argument after `file`. This is to ensure that specifications can work even when they are empty (i.e. the `Command.Spec.empty` value).

Every OCaml function needs at least one argument, so the final `unit` guarantees that it will not be evaluated immediately as a value if there are no other arguments.

## Creating basic commands

The specification and the function callback are glued together using the `basic` command. Let's see how this looks in our `md5` command.

```
Command.basic
  ~summary:"Generate an MD5 hash of the input data"
  ~readme:(fun () -> "More detailed information")
  Command.Spec.(
    empty
    +> anon ("filename" %: string)
```

```
  )
  (fun filename () -> do_hash filename)
```

The `basic` function takes the following arguments:

- `summary` is a required one-line description to go at the top of the command help screen.
- `readme` is for longer help text when the command is called with `-help`. The `readme` argument is a function that is only evaluated when the help text is actually needed.
- The specification and the callback function follow, with the arguments to the callback matching the order in which the specification binds arguments.

## Command argument types

You aren't just limited to parsing command lines as strings, of course. The `Command.Spec` module defines several other conversion functions that validate and parse input into various types:

| Argument type | OCaml type | Example |
|---|---|---|
| `string` | `string` | `foo` |
| `int` | `int` | `123` |
| `float` | `float` | `123.01` |
| `bool` | `bool` | `true` |
| `date` | `Date.t` | `2013-12-25` |
| `time_span` | `Span.t` | `5s` |
| `file` | `string` | `/etc/passwd` |

A more realistic `md5` function might also read from the standard input if a filename isn't specified. We can change our specification with a single line to reflect this by writing:

```
Command.Spec.(
  empty
  +> anon (maybe ("filename" %: string))
)
```

The anonymous parameter has been prefixed with a `maybe` that indicates the value is now optional. If you compile the example, you'll get a type error though:

```
File "md5_broken.ml", line 18, characters 26-30:
Error: This expression has type string option
       but an expression was expected of type string
Command exited with code 2.
```

This is because the type of the callback function has changed. It now wants a `string option` instead of a `string` since the value is optional. We can quickly adapt our example to use the new information and read from standard input if no file is specified.

```
(* md5.ml : calculate md5 with an optional filename *)
open Core.Std

let get_file_data = function
  | None
  | Some "-" -> In_channel.(input_all stdin)
  | Some file -> In_channel.read_all file

let do_hash file =
  let open Cryptokit in
  get_file_data file
  |> hash_string (Hash.md5 ())
  |> transform_string (Hexa.encode ())
  |> print_endline

let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    Command.Spec.(
      empty
      +> anon (maybe ("filename" %: string))
    )
  (fun file () -> do_hash file)

let () = Command.run command
```

There are several other transformations you can do on anonymous arguments. We've shown you `maybe`, and you can also obtain lists of arguments or supply default values. Try altering the example above to take a list of files and output checksums for all of them, just as the `md5` command does.

| Anonymous argument | OCaml type |
|---|---|
| sequence | `list` of arguments |
| maybe | `option` argument |
| maybe_with_default | argument with a default value if argument is missing |

**Defining your own argument types**

You can also define your own argument parsers by using the `Spec.Arg_type` module directly. For instance, the `time_span` converter is defined as follows.

```
# open Command.Spec ;;
# let time_span = Command.Spec.Arg_type.create Time.Span.of_string;;
val time_span : Core.Span.t Command.Spec.Arg_type.t = <abstr>
```

```
# Command.Spec.(empty +> anon ("span" %: time_span));;
- : (Core.Span.t -> '_a, '_a) Command.Spec.t = <abstr>
```

## Adding flags to label the command line

You aren't just limited to anonymous arguments on the command-line. A *flag* is a named field that can be followed by an optional argument. These flags can appear in any order on the command-line, or multiple times, depending on how they're declared in the specification.

Let's add two arguments to our `md5` command that mimics the Linux version. A `-s` flag specifies the string to be hashed directly on the command-line and a `-t` runs a self-test. The complete example is:

```
(* mlmd5.ml : generate an MD5 hash of the input data *)
open Core.Std

let get_file_data file checksum =
  match file, checksum with
  | None, Some buf -> buf
  | _, Some buf -> eprintf "Warning: ignoring file\n"; buf
  | (None|Some "-"), None -> In_channel.(input_all stdin)
  | Some file, None -> In_channel.read_all file

let do_hash file checksum =
  let open Cryptokit in
  get_file_data file checksum
  |> hash_string (Hash.md5 ())
  |> transform_string (Hexa.encode ())
  |> print_endline

let command =
  Command.basic
    ~summary:"Generate an MD5 hash of the input data"
    Command.Spec.(
      empty
      +> flag "-s" (optional string) ~doc:"string Checksum the given string"
      +> flag "-t" no_arg ~doc:" run a built-in time trial"
      +> anon (maybe ("filename" %: string))
    )
  (fun checksum trial file () ->
    match trial with
    | true -> printf "Running time trial\n"
    | false -> do_hash file checksum)

let () = Command.run command
```

The specification now uses the `flag` command. The first argument to `flag` is its name on the command-line, and the `doc` argument supplies the help text.

The `doc` string is formatted so that the first word is the short name that appears in the usage text, with the remainder being the full help text. Notice that the `-t` flag has no

argument, and so we prepend its `doc` text with a blank space. The help text for the above code looks like this:

```
$ ./mlmd5 -help
Generate an MD5 hash of the input data

  ./mlmd5 [filename]

=== flags ===

  [-s string]    Checksum the given string
  [-t]           run a built-in time trial
  [-build-info]  print info about this build and exit
  [-version]     print the version of this build and exit
  [-help]        print this help text and exit
                 (alias: -?)

$ ./mlmd5 -s "ocaml rocks"
5a118fe92ac3b6c7854c595ecf6419cb
```

The `-s` flag in our specification requires a `string` argument, and the parser outputs an error message if it isn't supplied. Here's a list of some of the functions that you can wrap flags in to control how they are parsed:

| Flag function | OCaml type |
|---|---|
| `required` *arg* | *arg* and error if not present |
| `optional` *arg* | *arg* `option` |
| `optional_with_default` | *arg* with a default if not present |
| `listed` *arg* | *arg* `list`, flag may appear multiple times |
| `no_arg` | `bool` that is true if flag is present. |

The flags affect the type of the callback function in exactly the same way as anonymous arguments do. This lets you change the specification and ensure that all the callback functions are updated appropriately, without runtime errors.

Notice that the `get_file_data` function now pattern matches across the `checksum` flag and the `file` anonymous argument. It selects the flag in preference to the anonymous argument, but emits a warning if there's ambiguity and both are specified.

## Grouping sub-commands together

You can get pretty far by combining flags and anonymous arguments to assemble complex command-line interfaces. After a while though, too many options can make the program very confusing for newcomers to your application. One way to solve this is by grouping common operations together and adding some hierarchy to the command-line interface.

You'll have run across this style already when using the OPAM package manager (or, in the non-OCaml world, the Git or Mercurial commands). OPAM exposes commands in this form:

```
opam config env
opam remote list -kind git
opam install --help
opam install xmlm
```

The `config`, `remote` and `install` keywords form a logical grouping of commands, and factor out flags and arguments that are specific to that particular operation. It's really simple to extend your application to do this in Command: just swap `Command.basic` for `Command.group`:

```
val group :
  summary:string ->
  ?readme:(unit -> string) ->
  (string * t) list -> t
```

The `group` signature accepts a list of basic `Command.t` values and their corresponding names. When executed, it looks for the appropriate sub-command from the name list, and dispatches it to the right command handler.

Let's build the beginning of a calendar tool that does a few operations over dates from the command line. We first define a command that adds days to an input date and prints the resulting date.

```
open Core.Std

let add =
  Command.basic
    ~summary:"Add [days] to the [base] date and print day"
    Command.Spec.(
      empty
      +> anon ("base" %: date)
      +> anon ("days" %: int)
    )
  (fun base span () ->
    Date.add_days base span
    |> Date.to_string
    |> print_endline
  )

let () = Command.run add
```

Once we've tested this and made sure it works, we can define another command that takes the difference of two dates. Both of the commands are now grouped as sub-commands using `Command.group`.

```
open Core.Std
```

```
let add =
  Command.basic ~summary:"Add [days] to the [base] date"
    Command.Spec.(
      empty
      +> anon ("base" %: date)
      +> anon ("days" %: int)
    )
  (fun base span () ->
    Date.add_days base span
    |> Date.to_string
    |> print_endline
  )

let diff =
  Command.basic ~summary:"Show days between [date1] and [date2]"
    Command.Spec.(
      empty
      +> anon ("date1" %: date)
      +> anon ("date2" %: date)
    )
  (fun date1 date2 () ->
    Date.diff date1 date2
    |> printf "%d days\n"
  )

let command =
  Command.group ~summary:"Manipulate dates"
    [ "add", add; "diff", diff ]

let () = Command.run command
```

And that's all you need to add sub-command support! The help page for our calendar now reflects the two commands we just added:

```
$ cal
Manipulate dates

  cal SUBCOMMAND

=== subcommands ===

  add      Add [days] to the [base] date
  diff     Show days between [date1] and [date2]
  version  print version information
  help     explain a given subcommand (perhaps recursively)

missing subcommand for command cal
```

We can invoke the two commands we just defined to verify that they work and see the date parsing in action.

```
$ cal add 2012-12-25 40
2013-02-03
```

```
$ cal diff 2012-12-25 2012-11-01
54 days
```

## Advanced control over parsing

The use of the spec combinators has been somewhat magic so far: we just build them up with the '+>' combinator and things seem to work. As your programs get larger and more complex, you'll want to factor out common functionality between specifications. Some other times, you'll need to interrupt the parsing to perform special processing, such as requesting an interactive passphrase from the user before proceeding. We'll show you some new combinators that let you do this now.

### The types behind Command.Spec

The Command module's safety relies on the specification's output values precisely matching the callback function which invokes the main program. Any mismatch here will inevitably result in a dynamic failure, and so Command uses some interesting type abstraction to guarantee they remain in sync. You don't have to understand this section to use the more advanced combinators, but it'll help you debug type errors as you use `Command` more.

The type of `Command.t` looks deceptively simple:

```
type ('main_in, 'main_out) t
```

You can think of `('a, 'b) t` as a function of type `'a -> 'b`, but embellished with information about:

- how to parse the command line
- what the command does and how to call it
- how to auto-complete a partial command line

The type of a specification transforms a `'main_in` to a `'main_out` value. For instance, a value of `Spec.t` might have type:

```
(arg1 -> ... -> argN -> 'r, 'r) Spec.t
```

Such a value transforms a main function of type `arg1 -> ... -> argN -> 'r` by supplying all the argument values, leaving a main function that returns a value of type `'r`. Let's look at some examples of specs, and their types:

```
# Command.Spec.empty ;;
- : ('m, 'm) Spec.t = <abstr>
# Command.Spec.(empty +> anon ("foo" %: int)) ;;
- : (int -> '_a, '_a) Command.Spec.t = <abstr>
```

The empty specification is simple as it doesn't add any parameters to the callback type. The second example adds an `int` anonymous parameter that is reflected in the inferred type. One forms a command by combining a spec of type `('main, unit) Spec.t` with

a function of type `'main`. The combinators we've shown so far incrementally build the type of `'main` according to the command-line parameters it expects, so the resulting type of `'main` is something like `arg1 -> ... -> argN -> unit`.

The type of `Command.basic` should make more sense now:

```
val basic :
  summary:string ->
  ?readme:(unit -> string) ->
  ('main, unit -> unit) Spec.t -> 'main -> t
```

The final line is the important one. It shows that the callback function for a spec should consume identical arguments to the supplied `main` function, expect for an additional `unit` argument. This final `unit` is there to make sure the callback is evaluated as a function, since if zero command-line arguments are specified (i.e. `Spec.empty`), the callback would otherwise have no arguments and be evaluated immediately. That's why you have to supply an additional `()` to the callback function in all the previous examples.

## Composing specification fragments together

If you want to factor out common command-line operations, the `++` operator will append two specifications together. Let's add some dummy verbosity and debug flags to our calendar application to illustrate this.

```
open Core.Std

let add ~common =
  Command.basic ~summary:"Add [days] to the [base] date"
    Command.Spec.(
      empty
      +> anon ("base" %: date)
      +> anon ("days" %: int)
      ++ common
    )
  (fun base span debug verbose () ->
    Date.add_days base span
    |> Date.to_string
    |> print_endline
  )

let diff ~common =
  Command.basic ~summary:"Show days between [date2] and [date1]"
    Command.Spec.(
      empty
      +> anon ("date1" %: date)
      +> anon ("date2" %: date)
      ++ common
    )
  (fun date1 date2 debug verbose () ->
    Date.diff date1 date2
```

```
    |> printf "%d days\n"
  )
```

The definitions of the specifications are very similar to the earlier example, except that they append a `common` parameter after each specification. We can supply these flags when defining the groups:

```
let () =
  let common =
    Command.Spec.(
      empty
      +> flag "-d" (optional_with_default false bool) ~doc:" Debug mode"
      +> flag "-v" (optional_with_default false bool) ~doc:" Verbose output"
    )
  in
  List.map ~f:(fun (name, cmd) -> (name, cmd ~common))
    [ "add", add; "diff", diff ]
  |> Command.group ~summary:"Manipulate dates"
  |> Command.run
```

Both of these flags will now be applied and passed to all the callback functions. This makes code refactoring a breeze by using the compiler to spot places where you use commands. Just add a parameter to the common definition, run the compiler, and fix type errors until everything works again.

For example, if we remove the `verbose` flag above and compile, we'll get this impressively long type error:

```
File "cal_compose_error.ml", line 39, characters 38-45:
Error: This expression has type
         (bool -> unit -> unit -> unit, unit -> unit -> unit)
         Command.Spec.t =
           (bool -> unit -> unit -> unit, unit -> unit -> unit)
           Command.Spec.t
       but an expression was expected of type
         (bool -> unit -> unit -> unit, unit -> unit) Command.Spec.t
           = (bool -> unit -> unit -> unit, unit -> unit) Command.Spec.t
       Type unit -> unit is not compatible with type unit
```

While this does look scary, the key line to scan is the last one, where it's telling you that you have supplied too many arguments in the callback function (`unit -> unit` vs `unit`). If you started with a working program and made this single change, you typically don't even need to read the type error, as the filename and location information is sufficient to make the obvious fix.

## Prompting for interactive input

The `step` combinator lets you control the normal course of parsing by supplying a function that maps callback arguments to a new set of values. For instance, let's revisit our first calendar application that added a number of days onto a supplied base date.

```
(* cal_add.ml *)
open Core.Std

let add_days base span () =
  Date.add_days base span
  |> Date.to_string
  |> print_endline

let add =
  Command.basic
    ~summary:"Add [days] to the [base] date and print day"
    Command.Spec.(
      empty
      +> anon ("base" %: date)
      +> anon ("days" %: int)
    )
    add_days

let () = Command.run add
```

This `cal_add` program requires you to specify both the `base` date and the number of `days` to add onto it. If `days` isn't supplied on the command-line, an error is output. Now let's modify it to interactively prompt for a number of days if only the `base` date is supplied.

```
(* cal_add_interactive.ml *)
open Core.Std

let add_days base span () =
  Date.add_days base span
  |> Date.to_string
  |> print_endline

let add =
  Command.basic
    ~summary:"Add [days] to the [base] date and print day"
    Command.Spec.(
      step
        (fun m base days ->
           match days with
           | Some days ->
             m base days
           | None ->
             print_endline "enter days: ";
             read_int ()
             |> m base
        )
      +> anon ("base" %: date)
      +> anon (maybe ("days" %: int))
    )
    add_days

let () = Command.run add
```

The `days` anonymous argument is now an optional integer in the spec, and we want to transform it into a non-optional value before calling our `add_days` callback. The `step` combinator in the specification performs this transformation. It applies its supplied callback function first, which checks if `day` is defined. If it's undefined, then it interactively reads an integer from the standard input. The first `m` argument to the `step` callback is the next callback function in the chain. The transformation is completed by calling `m base days` to continue processing with the new values we've just calculated. The `days` value that is passed onto the next callback now has a non-optional `int` type.

```
$ cal_add_interactive 2013-12-01
enter days:
35
2014-01-05
```

The transformation means that the `add_days` callback can just keep its original definition of `Date.t -> int -> unit`. The `step` function transformed the `int option` argument from the parsing into an `int` suitable for `add_days`. This transformation is explicitly represented in the type of the `step` return value:

```
# open Core.Std ;;
# open Command.Spec ;;
# step (fun m (base:Date.t) days ->
  match days with
  | Some days -> m base days
  | None ->
     print_endline "enter days: ";
     m base (read_int ()));;
- : (Date.t -> int -> '_a, Date.t -> int option -> '_a) Spec.t = <abstr>
```

The first half of the `Spec.t` shows that the callback type is `Date.t -> int`, whereas the resulting value expected from the next specification in the chain is a `Date.t -> int option`.

## Adding labelled arguments to callbacks

The `step` chaining lets you control the types of your callbacks very easily. This can help you match existing interfaces or make things more explicit by adding labelled arguments.

```
(* cal_add_labels.ml *)
open Core.Std

let add_days ~base_date ~num_days () =
  Date.add_days base_date num_days
  |> Date.to_string
  |> print_endline

let add =
  Command.basic
```

```
    ~summary:"Add [days] to the [base] date and print day"
    Command.Spec.(
      step (fun m base days -> m ~base_date:base ~num_days:days)
      +> anon ("base" %: date)
      +> anon ("days" %: int)
    )
  add_days

let () = Command.run add
```

This `cal_add_labels` example goes back to our non-interactive calendar addition program, but the `add_days` main function now expects labelled arguments. The `step` function in the specification simply converts the default `base` and `days` arguments into a labelled function, and everything compiles again.

Labelled arguments are more verbose, but also help prevent errors with command-line arguments with similar types but different names and purposes. It's good form to use them when you have a lot of otherwise anonymous `int` and `string` arguments.

## Command-line auto-completion with bash

Modern UNIX shells usually have a tab-completion feature to interactively help you figure out how to build a command-line. These work by pressing the `<tab>` key in the middle of typing a command, and seeing the options that pop up. You've probably used this most often to find the files in the current directory, but it can actually be extended for other parts of the command too.

The precise mechanism for autocompletion varies depending on what shell you are using, but we'll assume you are using the most common one: `bash`. This is the default interactive shell on most Linux distributions and Mac OS X, but you may need to switch to it on *BSD or Windows (when using Cygwin). The rest of this section assumes that you're using `bash`.

Bash autocompletion isn't always installed by default, so check your OS package manager to see if you have it available.

| Operating System | Package Manager | Package |
|---|---|---|
| Debian Linux | `apt` | `bash-completion` |
| Mac OS X | Homebrew | `bash-completion` |
| FreeBSD | Ports System | `/usr/ports/shells/bash-completion` |

Once bash completion is installed and configured, check that it works by typing the `ssh` command, and pressing `<tab>`. This should show you the list of known hosts from your `~/.ssh/known_hosts` file. If it lists some hosts that you've recently connected to, you can continue on. If it lists the files in your current directory instead, then check your OS documentation to configure completion correctly.

One last bit of information you'll need to find is the location of the `bash_completion.d` directory. This is where all the shell fragments that contain the completion logic are held. On Linux, this is often in `/etc/bash_completion.d`, and in Homebrew on Mac OS X it would be `/usr/local/etc/bash_completion.d` by default.

## Generating completion fragments from Command

The Command library has a declarative description of all the possible valid options, and it can use this information to generate a shell script which provides completion support for that command. To generate the fragment, just run the command with the `COMMAND_OUTPUT_INSTALLATION_BASH` environment variable set to any value.

For example, let's try it on our calendar example from earlier, assuming that the binary is called `cal` in the current directory:

```
$ COMMAND_OUTPUT_INSTALLATION_BASH=1 ./cal

function _jsautocom_41790 {
  export COMP_CWORD
  COMP_WORDS[0]=./cal
  COMPREPLY=($("${COMP_WORDS[@]}"))
}
complete -F _jsautocom_41790 ./cal
```

## Installing the completion fragment

You don't need to worry about what this script actually does (unless you have an unhealthy fascination with shell scripting, that is). Instead, redirect the output to a file in your `bash_completion.d` directory, named after the command you're installing.

```
$ sudo env COMMAND_OUTPUT_INSTALLATION_BASH=1 ./cal \
    > /etc/bash_completion.d/cal
$ bash -l
$ ./cal <tab>
add      diff     help     version
```

The first line above redirects the earlier output into your `bash_completion.d` directory. The `bash -l` loads the new configuration as a fresh login shell, and then the final line shows the four valid commands by pressing the tab key.

Command completion support works for flags and grouped commands, and is very useful when building larger command-line interfaces.

#### Installing a generic completion handler

Sadly, `bash` doesn't support installing a generic handler for all Command-based applications. This means that you have to install the completion script for every application, but you should be able to automate this in the build and packaging system for your application.

It will help to check out how other applications that install tab-completion scripts and following their lead, as the details are very OS-specific.

2013-07-04
10:28:30

CHAPTER 15

# Handling JSON data

Data serialization, *i.e.* converting data to and from a sequence of bytes that's suitable for writing to disk or sending across the network, is an important and common programming task. You often have to match someone else's data format (such as XML), sometimes you need a highly efficient format, and other times you want something that is easy for humans to edit. To this end, OCaml comes with several techniques for data serialization depending on what your problem is.

We'll start by using the popular and simple JSON data format, and then look at other serialization formats later in in the book. This chapter introduces you to a couple of new techniques that glue together the basic ideas from Part I of the book:

- Using polymorphic variants to write more extensible libraries and protocols (but still retain the ability to extend them if needed)
- The use of *combinators* to compose common operations over data structures in a type-safe way.
- Using external tools to generate boilerplate OCaml modules and signatures from external specification files.

## JSON Basics

JSON is a lightweight data-interchange format often used in web services and browsers. It's described in RFC4627 (*http://www.ietf.org/rfc/rfc4627.txt*), and is easier to parse and generate than alternatives such as XML. You'll run into JSON very often when working with modern web APIs, so we'll cover several different ways to manipulate it in this chapter.

JSON consists of two basic structures: an unordered collection of key/value pairs, and an ordered list of values. Values can be strings, booleans, floats, integers or null. Let's see what a JSON record for an example book description looks like:

```
{
```

```
  "title": "Real World OCaml",
  "tags" : [ "functional programming", "ocaml", "algorithms" ],
  "pages": 450,
  "authors": [
    { "name": "Jason Hickey", "affiliation": "Google" },
    { "name": "Anil Madhavapeddy", "affiliation": "Cambridge"},
    { "name": "Yaron Minsky", "affiliation": "Jane Street"}
  ],
  "is_online": true
}
```

The outermost JSON value is usually a record (delimited by the curly braces) and contains an unordered set of key/value pairs. The keys must be strings but values can be any JSON type. In the example above, `tags` is a string list, while the `authors` field contains a list of records. Unlike OCaml lists, JSON lists can contain multiple different JSON types within a single list.

This free-form nature of JSON types is both a blessing and a curse. It's very easy to generate JSON values, but code that parses them also has to handle subtle variations in how the values are represented. For example, what if the `pages` value above is actually represented as a string value of `"450"` instead of an integer?

Our first task is to parse the JSON into a more structured OCaml type so that we can use static typing more effectively. When manipulating JSON in Python or Ruby, you might write unit tests to check that you have handled unusual inputs. The OCaml model prefers compile-time static checking as well as unit tests. For example, using pattern matching can warn you if you've not checked that a value can be `Null` as well as contain an actual value.

**Installing the Yojson library**

There are several JSON libraries available for OCaml. For this chapter, we've picked the `Yojson` (*http://mjambon.com/yojson.html*) library by Martin Jambon. It's easiest to install via OPAM.

```
$ opam install yojson
```

See Appendix A for installation instructions if you haven't already got OPAM. Once installed, you can open it in the `utop` toplevel by:

```
# #require "yojson" ;;
# open Yojson ;;
```

# Parsing JSON with Yojson

The JSON specification has very few data types, and the `Yojson.Basic.json` type shown below is sufficient to express any valid JSON structure.

```
(* json/yojson_basic.mli  *)
```

```
type json = [
  | `Assoc of (string * json) list
  | `Bool of bool
  | `Float of float
  | `Int of int
  | `List of json list
  | `Null
  | `String of string
]
```

Some interesting properties should leap out at you after reading this definition:

- Some of the type definitions are *recursive* (that is, one of the algebraic data types includes a reference to the name of the type being defined). Fields such as `Assoc` can contain references to more JSON fields, and thus precisely describe the underlying JSON data structure. The JSON `List` can contain fields of different types, unlike the OCaml `list` whose contents must be uniform.
- The definition specifically includes a `Null` variant for empty fields. OCaml doesn't allow null values by default, so this must be encoded like any other value.
- The type definition uses polymorphic variants and not normal variants. This will become significant later when we extend it with custom extensions to the JSON format.

Let's parse the earlier JSON example into this type now. The first stop is the `Yojson.Basic` documentation, where we find these helpful functions:

```
(* json/yojson_basic.mli (starting from part 1) *)
val from_string : ?buf:Bi_outbuf.t -> ?fname:string -> ?lnum:int -> string -> json
(* Read a JSON value from a string.
   [buf]   : use this buffer at will during parsing instead of
             creating a new one.
   [fname] : data file name to be used in error messages. It does not
             have to be a real file.
   [lnum]  : number of the first line of input. Default is 1. *)

val from_file : ?buf:Bi_outbuf.t -> ?fname:string -> ?lnum:int -> string -> json
(* Read a JSON value from a file. See [from_string] for the meaning of the optional
   arguments. *)
```

When first reading these interfaces, you can generally ignore the optional arguments (which have the question marks in the type signature), as they will be filled in with sensible values. In the above signature, the optional arguments offer finer control over the memory buffer allocation and error messages from parsing incorrect JSON.

The type signature for these functions with the optional elements removed makes their purpose much clearer. The two ways of parsing the JSON are either directly from a string or from a file on a filesystem.

```
(* json/yojson_basic_simple.mli  *)
```

```
val from_string : string -> json
val from_file   : string -> json
```

**The standard OCaml `in_channel` and `out_channel`**

You'll notice when reading the Yojson interface that we've left out the `from_channel` function, which uses a `in_channel` type provided by the OCaml standard library. These OCaml channels are considered deprecated in Core, as they're primarily used by the compiler itself, but don't always play well with threading and asynchronous programming.

The Core equivalent are the `In_channel` and `Out_channel` modules instead, but you will still see the standard OCaml channel types being used in third-party libraries such as Yojson which weren't specifically designed to be used with Core. They won't do any harm if you don't use them in your code, so just ignore them and use strings or files instead.

The next example shows both the string and file functions in action, assuming the JSON record is stored in a file called `book.json`.

```
(* json/read_json.ml  *)
open Core.Std

let () =
  (* Read JSON file into an OCaml string *)
  let buf = In_channel.read_all "book.json" in
  (* Use the string JSON constructor *)
  let json1 = Yojson.Basic.from_string buf in
  (* Use the file JSON constructor *)
  let json2 = Yojson.Basic.from_file "book.json" in
  (* Test that the two values are the same *)
  print_endline (if json1 = json2 then "OK" else "FAIL")
```

You can build this by writing a `_tags` file to define the package dependencies, and then running `ocamlbuild`.

```
# running json/run_read_json.out.sh
$ ocamlbuild -use-ocamlfind -pkg core,yojson -tag thread read_json.native
$ ./read_json.native
OK
```

The `from_file` function accepts an input filename and takes care of opening and closing it for you. It's far more common to use `from_string` to construct JSON values though, since these strings come in via a network connection (we'll see more of this in Chapter 18) or a database. Finally, the example checks that the two input mechanisms actually resulted in the same OCaml data structure.

## Selecting values from JSON structures

Now that we've figured out how to parse the example JSON into an OCaml value, let's manipulate it from OCaml code and extract specific fields.

```
(* json/parse_book.ml  *)
open Core.Std

let () =
  (* Read the JSON file *)
  let json = Yojson.Basic.from_file "book.json" in

  (* Locally open the JSON manipulation functions *)
  let open Yojson.Basic.Util in
  let title = json |> member "title" |> to_string in
  let tags = json |> member "tags" |> to_list |> filter_string in
  let pages = json |> member "pages" |> to_int in
  let is_online = json |> member "is_online" |> to_bool_option in
  let is_translated = json |> member "is_translated" |> to_bool_option in
  let authors = json |> member "authors" |> to_list in
  let names = List.map authors ~f:(fun json -> member "name" json |> to_string) in

  (* Print the results of the parsing *)
  printf "Title: %s (%d)\n" title pages;
  printf "Authors: %s\n" (String.concat ~sep:", " names);
  printf "Tags: %s\n" (String.concat ~sep:", " tags);
  let string_of_bool_option =
    function
    | None -> "<none>"
    | Some true -> "yes"
    | Some false -> "no" in
  printf "Online: %s\n" (string_of_bool_option is_online);
  printf "Translated: %s\n" (string_of_bool_option is_translated)
```

Build this with the same `_tags` file as the earlier example, and run `ocamlbuild` on the new file.

```
# running json/run_parse_book.out.sh
$ ocamlbuild -use-ocamlfind -pkg core,yojson -tag thread parse_book.native
$ ./parse_book.native
Title: Real World OCaml (450)
Authors: Jason Hickey, Anil Madhavapeddy, Yaron Minsky
Tags: functional programming, ocaml, algorithms
Online: yes
Translated: <none>
```

This code introduces the `Yojson.Basic.Util` module, which contains *combinator* functions that let you easily map a JSON object into a more strongly-typed OCaml value.

### Functional Combinators

Combinators are a design pattern that crops up quite often in functional programming. John Hughes defines them as "a function which builds program fragments from pro-

gram fragments". In a functional language, this generally means higher-order functions that combine other functions to apply useful transformations over values.

You've already run across several of these in the `List` module:

```
(* json/list_excerpt.mli  *)
val map  : 'a list -> f:('a -> 'b)   -> 'b list
val fold : 'a list -> init:'accum -> f:('accum -> 'a -> 'accum) -> 'accum
```

`map` and `fold` are extremely common combinators that transform an input list by applying a function to each value of the list. The `map` combinator is simplest, with the resulting list being output directly. `fold` applies each value in the input list to a function that accumulates a single result, and returns that instead.

```
(* json/list_excerpt.mli (starting from part 1) *)
val iter : 'a list -> f:('a -> unit) -> unit
```

`iter` is a more specialised combinator that is only useful in OCaml due to side-effects being allowed. The input function is applied to every value, but no result is supplied. The function must instead apply some side-effect such as changing a mutable record field or printing to the standard output.

`Yojson` provides several combinators in the `Yojson.Basic.Util` module.

| Function | Type | Purpose |
|---|---|---|
| member | `string -> json -> json` | Select a named field from a JSON record. |
| to_string | `json -> string` | Convert a JSON value into an OCaml `string`, and raise exception if this impossible. |
| to_int | `json -> int` | Convert a JSON value into an OCaml `int`, and raise exception if this is impossible. |
| filter_string | `json list -> string list` | Filter valid strings from a list of JSON fields, and return them as OCaml strings. |

We'll go through each of these uses one-by-one now. The examples below also use the `|>` pipe-forward operator that we explained earlier in Chapter 2. This lets us chain together multiple JSON selection functions and feed the output from one into the next one, without having to create separate `let` bindings for each one.

Let's start with selecting a single `title` field from the record.

```
...part 1 of json/parse_book.topscript
# open Yojson.Basic.Util ;;
# let title = json |> member "title" |> to_string ;;
val title : string = "Real World OCaml"
```

The `member` function accepts a JSON object and named key and returns the JSON field associated with that key, or `Null`. Since we know that the `title` value is always a string in our example schema, we want to convert it to an OCaml string. The `to_string`

function performs this conversion, and raises an exception if there is an unexpected JSON type. The |> operator provides a convenient way to chain these operations together.

```
...part 2 of json/parse_book.topscript
# let tags = json |> member "tags" |> to_list |> filter_string ;;
val tags : string list = ["functional programming"; "ocaml"; "algorithms"]
# let pages = json |> member "pages" |> to_int ;;
val pages : int = 450
```

The `tags` field is similar to `title`, but the field is a list of strings instead of a single one. Converting this to an OCaml `string list` is a two stage process. First, we convert the JSON `List` to an OCaml list of JSON values, and then filter out the `String` values as an OCaml `string list`. Remember that OCaml lists must contain values of the same type, so any JSON values that cannot be converted to a `string` will be skipped from the output of `filter_string`.

```
...part 3 of json/parse_book.topscript
# let is_online = json |> member "is_online" |> to_bool_option ;;
val is_online : bool option = Some true
# let is_translated = json |> member "is_translated" |> to_bool_option ;;
val is_translated : bool option = None
```

The `is_online` and `is_translated` fields are optional in our JSON schema, so no error should be raised if they are not present. The OCaml type is a `string option` to reflect this, and can be extracted via `to_bool_option`. In our example JSON, only `is_online` is present and `is_translated` will be `None`.

```
...part 4 of json/parse_book.topscript
# let authors = json |> member "authors" |> to_list ;;
val authors : Yojson.Basic.json list =
  [`Assoc
     [("name", `String "Jason Hickey"); ("affiliation", `String "Google")];
   `Assoc
     [("name", `String "Anil Madhavapeddy");
      ("affiliation", `String "Cambridge")];
   `Assoc
     [("name", `String "Yaron Minsky");
      ("affiliation", `String "Jane Street")]]
```

The final use of JSON combinators is to extract all the `name` fields from the list of authors. We first construct the `author list`, and then `map` it into a `string list`. Notice that the example explicitly binds `authors` to a variable name. It can also be written more succinctly using the pipe-forward operator:

```
...part 5 of json/parse_book.topscript
# let names =
  json |> member "authors" |> to_list
  |> List.map ~f:(fun json -> member "name" json |> to_string) ;;
```

```
val names : string list =
  ["Jason Hickey"; "Anil Madhavapeddy"; "Yaron Minsky"]
```

This style of programming which omits variable names and chains functions together is known as *point-free programming*. It's a succinct style, but shouldn't be overused due to the increased difficulty of debugging intermediate values. If an explicit name is assigned to each stage of the transformations, debuggers in particular have an easier time making the program flow easier to represent to the programmer.

This technique of using chained parsing functions is very powerful in combination with the OCaml type system. Many errors that don't make sense at runtime (for example, mixing up lists and objects) will be caught statically via a type error.

## Constructing JSON values

Building and printing JSON values is pretty straightforward given the `Yojson.Basic.json` type. You can just construct values of type `json` and call the `to_string` function on them. Let's remind ourselves of the `Yojson.Basic.type` again.

```
type json = [
  | `Assoc of (string * json) list
  | `Bool of bool
  | `Float of float
  | `Int of int
  | `List of json list
  | `Null
  | `String of string
]
```

We can directly build a JSON value against this type and use the pretty-printing functions in the `Yojson.Basic` module to display JSON output.

```
...part 1 of json/build_json.topscript
# let person = `Assoc [ ("name", `String "Anil") ] ;;
val person : [> `Assoc of (string * [> `String of string ]) list ] =
  `Assoc [("name", `String "Anil")]
```

In the example above, we've constructed a simple JSON object that represents a single person. We haven't actually defined the type of `person` explicitly, as we're relying on the magic of polymorphic variants to make this all work.

The OCaml type system infers a type for `person` based on how you construct its value. In this case, only the `Assoc` and `String` variants are used to define the record, and so the inferred type only contains these fields without knowledge of the other possible allowed variants in JSON records that you haven't used yet (e.g. `Int` or `Null`).

```
...part 2 of json/build_json.topscript
# Yojson.Basic.pretty_to_string ;;
- : ?std:bool -> Yojson.Basic.json -> string = <fun>
```

The `pretty_to_string` function has a more explicit signature that requires an argument of type `Yojson.Basic.json`. When `person` is applied to `pretty_to_string`, the inferred type of `person` is statically checked against the structure of the `json` type to ensure that they're compatible.

```
...part 3 of json/build_json.topscript
# Yojson.Basic.pretty_to_string person ;;
- : string = "{ \"name\": \"Anil\" }"
# Yojson.Basic.pretty_to_channel stdout person ;;
- : unit = ()
```

In this case, there are no problems. Our `person` value has an inferred type that is a valid sub-type of `json`, and so the conversion to a string just works without us ever having to explicitly specify a type for `person`. Type inference lets you write more succinct code without sacrificing runtime reliability, as all the uses of polymorphic variants are still checked at compile-time.

## Polymorphic variants and easier type checking

One difficulty you will encounter is that type errors involving polymorphic variants can be quite verbose if you make a mistake in your code. For example, suppose you build an `Assoc` and mistakenly include a single value instead of a list of keys:

```
...part 4 of json/build_json.topscript
# let person = `Assoc ("name", `String "Anil");;
val person : [> `Assoc of string * [> `String of string ] ] =
  `Assoc ("name", `String "Anil")
# Yojson.Basic.pretty_to_string person ;;
File "", line 1, characters 30-36:
Error: This expression has type
         [> `Assoc of string * [> `String of string ] ]
       but an expression was expected of type Yojson.Basic.json
       Types for tag `Assoc are incompatible
```

The type error above is more verbose than it needs to be, which can be inconvenient to wade through for larger values. You can help the compiler to narrow down this error to a shorter form by adding explicit type annotations as a hint about your intentions.

```
...part 5 of json/build_json.topscript
# let (person : Yojson.Basic.json) =
  `Assoc ("name", `String "Anil");;
File "", line 2, characters 2-33:
Error: This expression has type 'a * 'b
       but an expression was expected of type
         (string * Yojson.Basic.json) list
```

We've annotated `person` as being of type `Yojson.Basic.json`, and as a result the compiler spots that the argument to the `Assoc` variant has the incorrect type. This illustrates the strengths and weaknesses of polymorphic variants: they make it possible to easily sub-type across module boundaries, but the error messages can be more confusing. How-

ever, a bit of careful manual type annotation is all it takes to make tracking down such issues much easier.

We'll discuss more techniques like this that help you interpret type errors more easily in Chapter 22.

## Using non-standard JSON extensions

The standard JSON types are *really* basic, and OCaml types are far more expressive. Yojson supports an extended JSON format for those times when you're not interoperating with external systems and just want a convenient human-readable local format. The `Yojson.Safe.json` type is a superset of the `Basic` polymorphic variant, and looks like this:

```
type json = [
  | `Assoc of (string * json) list
  | `Bool of bool
  | `Float of float
  | `Floatlit of string
  | `Int of int
  | `Intlit of string
  | `List of json list
  | `Null
  | `String of string
  | `Stringlit of string
  | `Tuple of json list
  | `Variant of string * json option
]
```

The `Safe.json` type includes all of the variants from `Basic.json` and extends it with a few more useful ones. A standard JSON type such as a `String` will type-check against both the `Basic` module and also the non-standard `Safe` module. If you use the extension values with the `Basic` module however, the compiler will reject your code until you make it compliant with the portable subset of JSON.

Yojson supports the following JSON extensions:

- The `lit` suffix denotes that the value is stored as a JSON string. For example, a `Floatlit` will be stored as `"1.234"` instead of `1.234`.
- The `Tuple` type is stored as `("abc", 123)` instead of a list.
- The `Variant` type encodes OCaml variants more explicitly, as `<"Foo">` or `<"Bar": 123>` for a variant with parameters.

The only purpose of these extensions is to have greater control over how OCaml values are represented in JSON (for instance, storing a floating pointer number as a JSON string). The output still obeys the same standard format that can be easily exchanged with other languages.

You can convert a `Safe.json` to a `Basic.json` type by using the `to_basic` function as follows.

```
val to_basic : json -> Yojson.Basic.json
(** Tuples are converted to JSON arrays, Variants are converted to
    JSON strings or arrays of a string (constructor) and a json value
    (argument). Long integers are converted to JSON strings.
    Examples:

    `Tuple [ `Int 1; `Float 2.3 ]   ->    `List [ `Int 1; `Float 2.3 ]
    `Variant ("A", None)            ->    `String "A"
    `Variant ("B", Some x)          ->    `List [ `String "B", x ]
    `Intlit "12345678901234567890"  ->    `String "12345678901234567890"
 *)
```

# Automatically mapping JSON to OCaml types

The combinators described earlier make it easy to write functions that extract fields from JSON records, but the process is still pretty manual. When you implement larger specifications, it's much easier to generate the mappings from JSON schemas to OCaml values more mechanically than writing conversion functions individually.

We'll cover an alternative JSON processing method that is better for larger-scale JSON handling now, using the ATD (*http://mjambon.com/atd-biniou-intro.html*) tool. This will introduce our first *Domain Specific Language* that compiles JSON specifications into OCaml modules, which are then used throughout your application.

**Installing the ATDgen library and tool**

ATDgen installs some OCaml libraries that interface with Yojson, and also a command-line tool that generates code. It can all be installed via OPAM:

```
# running json/install_atdgen.out.sh
$ opam install atdgen
$ atdgen -version
1.2.3
```

The command-line tool will be installed within your `~/.opam` directory, and will already be on your `PATH` from running `opam config env`. See Appendix A if this isn't working.

## ATD basics

The idea behind ATD is to specify the format of the JSON in a separate file, and then run a compiler (`atdgen`) that outputs OCaml code to construct and parse JSON values. This means that you don't need to write any OCaml parsing code at all, as it will all be auto-generated for you.

Let's go straight into looking at an example of how this works, by using a small portion of the GitHub API. GitHub is a popular code hosting and sharing website that provides a JSON-based web API (*http://developer.github.com*). The ATD code fragment below describes the GitHub authorization API (which is based on a pseudo-standard web protocol known as OAuth).

```
(* json/github.atd *)
type scope = [
    User <json name="user">
  | Public_repo <json name="public_repo">
  | Repo <json name="repo">
  | Repo_status <json name="repo_status">
  | Delete_repo <json name="delete_repo">
  | Gist <json name="gist">
]

type app = {
  name: string;
  url: string;
}  <ocaml field_prefix="app_">

type authorization_request = {
  scopes: scope list;
  note: string;
} <ocaml field_prefix="auth_req_">

type authorization_response = {
  scopes: scope list;
  token: string;
  app: app;
  url: string;
  id: int;
  ?note: string option;
  ?note_url: string option;
}
```

The ATD specification syntax is deliberately quite similar to OCaml type definitions. Every JSON record is assigned a type name (e.g. `app` in the example above). You can also define variants that are similar to OCaml's variant types (e.g. `scope` in the example).

## ATD annotations

ATD deviates significantly from OCaml syntax due to its support for annotations within the specification. The annotations can customise the code that is generated for a particular target (of which the OCaml backend is of most interest to us).

For example, the GitHub `scope` field above is defined as a variant type with each option starting with an uppercase letter as is conventional for OCaml variants. However, the the JSON values that come back from Github are actually lowercase, and so aren't exactly the same as the option name.

The annotation `<json name="user">` signals that the JSON value of the field should be treated as `name`, but the value of the parsed variant in OCaml should be `User`. These annotations are also useful to map JSON values to reserved keywords in OCaml (e.g. `type`).

## Compiling ATD specifications to OCaml

The ATD specification we defined above can be compiled to OCaml code using the `atdgen` command-line tool. Let's run the compiler twice, to generate some OCaml type definitions and also a JSON serializing module that converts between input data and those type definitions.

The `atdgen` command will generate some new files in your current directory. `Github_t.ml` and `Github_t.mli` will contain an OCaml module with types defines that correspond to the ATD file.

```
# running json/build_github_atd.out.sh
$ atdgen -t github.atd
$ atdgen -j github.atd
$ ocamlfind ocamlc -package atd -i github_t.mli
type scope =
    [ `Delete_repo | `Gist | `Public_repo | `Repo | `Repo_status | `User ]
type app = { app_name : string; app_url : string; }
type authorization_request = {
  auth_req_scopes : scope list;
  auth_req_note : string;
}
type authorization_response = {
  scopes : scope list;
  token : string;
  app : app;
  url : string;
  id : int;
  note : string option;
  note_url : string option;
}
```

There is an obvious correspondence to the ATD definition. Note that field names in OCaml records in the same module cannot shadow each other, and so we instruct ATDgen to prefix every field with a name that distinguishes it from other records in the same module. For example, `<ocaml field_prefix="auth_req_">` in the ATD spec prefixes every field name in the generated `authorization_request` record with `auth_req`.

The `Github_t` module only contains the type definitions, while `Github_j` provides serialization functions to and from JSON. You can read the `github_j.mli` to see the full interface, but the important functions for most uses are the conversion functions to and from a string. For our example above, this looks like:

```
(* json/github_j_excerpt.mli  *)
```

```
val string_of_authorization_request :
  ?len:int -> authorization_request -> string
  (** Serialize a value of type {!authorization_request}
      into a JSON string.
      @param len specifies the initial length
                 of the buffer used internally.
                 Default: 1024. *)

val string_of_authorization_response :
  ?len:int -> authorization_response -> string
  (** Serialize a value of type {!authorization_response}
      into a JSON string.
      @param len specifies the initial length
                 of the buffer used internally.
                 Default: 1024. *)
```

This is pretty convenient! We've now written a single ATD file, and all the OCaml boilerplate to convert between JSON and a strongly typed record has been generated for us. You can control various aspects of the serializer by passing flags to `atdgen`. The important ones for JSON are:

- `-j-std`: Convert tuples and variants into standard JSON and refuses to print NaN and infinities. You should specify this if you intend to interoperate with services that aren't using ATD.
- `-j-custom-fields FUNCTION`: call a custom function for every unknown field encountered, instead of raising a parsing exception.
- `-j-defaults`: always explicitly output a JSON value if possible. This requires the default value for that field to be defined in the ATD specification.

The full ATD specification is quite sophisticated and well documented online at its homepage. The ATD compiler can also target formats other than JSON, and outputs code for other languages such as Java if you need more interoperability.

There are also several similar projects that automate the code generation process: Piqi (*http://piqi.org*) uses the Google protobuf format, and Thrift (*http://thrift.apache.org*) supports many other programming languages and includes OCaml bindings.

## Example: Querying Github organization information

Let's finish up with an example of some live JSON parsing from Github, and build a tool to query organization information via their API. Start by looking at the online API documentation (*http://developer.github.com/v3/orgs/*) for Github to see what the JSON schema for retrieving the organization information looks like.

Now create an ATD file that covers the fields we need. Any extra fields present in the response will be ignored by the ATD parser, so we don't need a completely exhaustive specification of every field that Github might send back.

```
(* json/github_org.atd *)
```

```
type org = {
  login: string;
  id: int;
  url: string;
  ?name: string option;
  ?blog: string option;
  ?email: string option;
  public_repos: int
}
```

Let's build the OCaml type declaration first by calling `atdgen -t` on the specification file.

```
# running json/generate_github_org_types.out.sh
$ atdgen -t github_org.atd
$ cat github_org_t.mli
(* Auto-generated from "github_org.atd" *)


type org = {
  login: string;
  id: int;
  url: string;
  name: string option;
  blog: string option;
  email: string option;
  public_repos: int
}
```

The OCaml type has an obvious mapping to the ATD spec, but we still need the logic to convert JSON buffers to and from this type. Calling `atdgen -j` will generate this serialization code for us in a new file called `github_org_j.ml`.

```
# running json/generate_github_org_json.out.sh
$ atdgen -j github_org.atd
$ cat github_org_j.mli
(* Auto-generated from "github_org.atd" *)


type org = Github_org_t.org = {
  login: string;
  id: int;
  url: string;
  name: string option;
  blog: string option;
  email: string option;
  public_repos: int
}

val write_org :
  Bi_outbuf.t -> org -> unit
  (** Output a JSON value of type {!org}. *)

val string_of_org :
  ?len:int -> org -> string
```

```
  (** Serialize a value of type {!org}
      into a JSON string.
      @param len specifies the initial length
                 of the buffer used internally.
                 Default: 1024. *)

val read_org :
  Yojson.Safe.lexer_state -> Lexing.lexbuf -> org
  (** Input JSON data of type {!org}. *)

val org_of_string :
  string -> org
  (** Deserialize JSON data of type {!org}. *)
```

The `Github_org_j` serializer interface contains everything we need to map to-and-from the OCaml types and JSON. The easiest way to use this interface is by using the `string_of_org` and `org_of_string` functions, but there are also more advanced low-level buffer functions available if you need higher performance (but we won't go into that in this tutorial).

All we need to complete our example is an OCaml program that fetches the JSON and uses these modules to output a one-line summary. Our example below does just that.

The code below calls the cURL command-line utility by using the `Core_extended.Shell` interface to run an external command and capture its output. You'll need to ensure that you have cURL installed on your system before running the example. You might also need to `opam install core_extended` if you haven't installed it previously.

```
(* json/github_org_info.ml  *)
open Core.Std

let print_org file () =
  let url = sprintf "https://api.github.com/orgs/%s" file in
  Core_extended.Shell.run_full "curl" [url]
  |> Github_org_j.org_of_string
  |> fun org ->
  let open Github_org_t in
  let name = Option.value ~default:"???" org.name in
  printf "%s (%d) with %d public repos\n"
    name org.id org.public_repos

let () =
  Command.basic ~summary:"Print Github organization information"
    Command.Spec.(empty +> anon ("organization" %: string))
    print_org
  |> Command.run
```

Below is a short shell script that generates all of the OCaml code and also builds the final executable.

```
(* json/build_github_org.out  *)
```

```
$ atdgen -t github_org.atd
$ atdgen -j github_org.atd
$ ocamlbuild -use-ocamlfind -tag thread -pkg core_extended,yojson,atdgen github_org_info.native
```

You can now run the command-line tool with a single argument to specify the name of the organization, and it will dynamically fetch the JSON from the web, parse it, and render the summary to your console.

```
# running json/run_github_org.out.sh
$ ./_build/github_org_info.native mirage
Mirage account (131943) with 32 public repos
$ ./_build/github_org_info.native janestreet
??? (3384712) with 32 public repos
```

The JSON returned from the `janestreet` query is missing an organization name, but this is explicitly reflected in the OCaml type since the ATD spec marked `name` as an optional field. Our OCaml code explicitly handles this case and doesn't have to worry about null-pointer exceptions. Similarly, the JSON integer for the `id` is mapped into a native OCaml integer via the ATD conversion.

While this tool is obviously quite simple, the ability to specify optional and default fields is very powerful. Take a look at the full ATD specification for the GitHub API in the `ocaml-github` (*http://github.com/avsm/ocaml-github*) repository online, which has lots of quirks typical in real-world web APIs.

Our example shells out to `curl` on the command-line to obtain the JSON, which is rather inefficient. We'll explain how to integrate the HTTP fetch directly into your OCaml application later on in Chapter 18.

2013-07-04
10:28:30

2013-07-04
10:28:31

# CHAPTER 16
# Parsing with OCamllex and Menhir

OCaml provides lexer and parser generators modeled on lex and yacc. Similar tools are available in a variety of languages, and with them you can parse a variety of kinds of input, including web formats or full blown programming languages.

Let's be more precise about these terms. By *parsing*, we mean reading a textual input into a form that is easier for a program to manipulate. For example, suppose we want to read a file containing a value in JSON format. JSON has a variety of values, including numbers, strings, arrays, and objects, and each of these has a textual representation. For example, the following text represents an object containing a string labeled `title`, and an array containing two objects, each with a name and array of zip codes.

```
{ title: "Cities",
  cities: [{ name: "Chicago", zips: [60601] },
           { name: "New York", zips: [10004] }]
}
```

The input text is represented as a sequence of characters. Manipulating it in that form would be really hard, so what we want is to give it a structured type that is easier for our programs to manipulate. For our example, we'll use the following type to represent JSON *abstract syntax*.

```
type value = [
| `Assoc of (string * value) list
| `Bool of bool
| `Float of float
| `Int of int
| `List of value list
| `String of string
| `Null ]
```

The objective of *parsing* is to convert the text input into a value of type `value`. This is normally done in two phase. First, *lexical* analysis (or lexing, for short) is used to convert the text input into a sequence of tokens, or words. For example, the JSON input would

be tokenized into a sequence of tokens like the following. In most cases (and in this example), lexical analysis will choose to omit white space from the token stream.

```
LEFT_BRACE, ID("title"), COLON, STRING("Cities"), COMMA, ID("cities"), ...
```

The next step is to convert the token stream into a program value that represents the abstract syntax tree, like the type `value` above. This is called *parsing*.

```
`Assoc
  ["title", `String "Cities";
   "cities", `List
     [`Assoc ["name", `String "Chicago"; "zips", `List [Int 60601]];
       `Assoc ["name", `String "New York"; "zips", `List [Int 10004]]]]
```

There are many techniques for lexing and parsing. In the lex/yacc world, lexing is specified using regular expressions, and parsing is specified using context-free grammars. These are concepts from formal languages; the lex/yacc tools constructing the machinery for you. For `lex`, this means constructing a finite automaton; and for `yacc`, this means constructing a pushdown automaton.

Parsing is a broad and often intricate topic, and our purpose here is not to teach all of the ins and outs of yacc and lex, but to show how to use these tools in OCaml. There are online resources, and most experience you may have using lex/yacc in other languages will also apply in OCaml. However, there are differences, and we'll try to point out the larger ones here.

For illustration, let's continue with the JSON example. For lexing, we'll use `ocamllex`, and for parsing, we'll use `menhir`, which is somewhat easier to use than `ocamlyacc`.

## Defining a JSON parser with menhir

The process of building a parser is interleaved between constructing the lexer and parser; you will have to do them simultaneously. The first step is to define the set of tokens that will be produced by the lexer. For various reasons, the tokens are specified by the parser (to specify what it expects as input), so we'll start with the parser first.

A parser file has suffix `.mly` (we'll use the name `parser.mly`) and it contains several parts in the following sequence:

```
declarations
%%
rules
%%
optional OCaml code
```

The `%%` are section separators; they have to be on a line by themselves. The declarations include token and type specifications, precedence directives, and other things, but we start by declaring the tokens.

## Token declarations

A token is declared using the syntax `%token <type> uid`, where the `<type>` is optional, and *uid* is an capitalized identifier. For JSON, we need tokens for numbers, strings, identifiers, and punctuation. To start, let's define just the tokens in the `parser.mly` file. For technical reasons, we need to include a `%start` declaration. For now, we'll include just a dummy grammar specification `exp: { () }` (we'll replace this when we implement the grammar below).

```
%token <int> INT
%token <float> FLOAT
%token <string> ID
%token <string> STRING
%token TRUE
%token FALSE
%token NULL
%token LEFT_BRACE
%token RIGHT_BRACE
%token LEFT_BRACK
%token RIGHT_BRACK
%token COLON
%token COMMA
%token EOF

%start <unit> exp

%%

exp: { () }
```

The `<type>` specifications mean that a token carries a value. The `INT` token carries an integer value with it, `FLOAT` has a `float` value, etc. Most of the remaining tokens, like `TRUE`, `FALSE`, the punctuation, aren't associated with any value, so we omit the `<type>` specification.

Compile this file with `menhir`. It will issue multiple warnings about unused tokens because we haven't actually defined a grammar yet. It is ok to ignore the warnings for now.

```
$ menhir parser.mly
Warning: the token COLON is unused.
...
```

The `menhir` tool is a parser generator, meaning it generates the code to perform parsing from the `parser.mly` description. The `parser.ml` contains an automaton implementation, and is generally difficult to read. However, the `parser.mli` contains declarations that we need to build a lexer.

```
$ cat parser.mli
exception Error
```

```
type token =
  | TRUE
  | STRING of (string)
  | RIGHT_BRACK
  | RIGHT_BRACE
  | NULL
  | LEFT_BRACK
  | LEFT_BRACE
  | INT of (int)
  | ID of (string)
  | FLOAT of (float)
  | FALSE
  | EOF
  | COMMA
  | COLON


val exp: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> (unit)
```

## Specifying the grammar rules

The grammar itself is specified using a set of rules, where a rule contains a set of productions. Abstractly, a production looks like the following.

```
symbol: [ id1 = ] symbol1; [ id2 = ] symbol2; ...; [ idN = ] symbolN
   { OCaml code }
```

A production can be interpreted as follows: given values `id1`, ..., `idN` for the input symbols `symbol1`, ..., `symbolN`; the OCaml code computes a value for the target `symbol`. That's too abstract, so let's get down to defining productions for parsing JSON. Here is the main production for a JSON value.

```
value: LEFT_BRACE; obj = opt_object_fields; RIGHT_BRACE
    { `Assoc obj }
  | LEFT_BRACK; vl = list_values; RIGHT_BRACK
    { `List vl }
  | s = STRING
    { `String s }
  | i = INT
    { `Int i }
  | x = FLOAT
    { `Float x }
  | TRUE
    { `Bool true }
  | FALSE
    { `Bool false }
  | NULL
    { `Null }
  ;
```

We can read it like this: a JSON `value` is either an object bracketed by curly braces, or an array bracketed by square braces. or a string, integer, float, etc. In each of the pro-

ductions, the right hand side specfies the expected sequence. For example, the object is specified with the curly-bracket production.

```
value: LEFT_BRACE; obj = opt_object_fields; RIGHT_BRACE
    { `Assoc obj }
```

That is, an object value starts with a `LEFT_BRACE`, contains some optional object field values (to be defined), and end with a `RIGHT_BRACE`. The returned value is `Assoc obj`, where `obj` is the sequence of object fields. Note that we've left out bindings for `LEFT_BRACE` and `RIGHT_BRACE`, because their tokens don't have values.

Next, let's define the object fields. In the following rules, the `opt_object_fields` are either empty, or a non-empty sequence of fields in reverse order. Note that if you wish to have comments in the rule definitions, you have to use C comment delimiters. By convention, the C comment `/* empty */` is used to point out that a production has an empty right hand side.

```
opt_object_fields: /* empty */
    { [] }
  | obj = rev_object_fields
    { List.rev obj }
  ;

rev_object_fields: k = ID; COLON; v = value
    { [k, v] }
  | obj = rev_object_fields; COMMA; k = ID; COLON; v = value
    { (k, v) :: obj }
  ;
```

The rule `rev_object_fields` is defined recursively. It has either one key/value field, or it is a sequence of fields, followed by a `COMMA` and one more field definition.

The `rev_` prefixed is intended to point out that the fields are returned in reverse order. Why would we do that? One reason is that the `menhir` parser generator is left-recursive, which means that the constructed pushdown automoton uses less stack space with left-recursive definitions. The following right-recursive rule accepts the same input, but during parsing it requires linear stack space to read object field definitions.

```
/* Inefficient right-recursive rule */
object_fields: k = ID; COLON; v = value
    { [k, v] }
  | k = ID; COLON; v = value; COMMA; obj = object_fields
    { (k, v) :: obj }
```

Alternatively, we could keep the left-recursive definition and simply construct the returned value in left-to-right order. This is fine, though less efficient. You will have to choose you technique according to circumstances.

```
/* Quadratic left-recursive rule */
object_fields: k = ID; COLON; v = value
```

```
    { [k, v] }
  | obj = rev_object_fields; COMMA; k = ID; COLON; v = value
    { obj @ [k, v] }
  ;
```

Finally, we can finish off the grammar by defining the rules for arrays, and adding a correct `%start` production. For the `%start` production, we'll return a `value option`, using `None` to represent end of file. Here is the complete file.

```
%token <int> INT
%token <float> FLOAT
%token <string> ID
%token <string> STRING
%token TRUE
%token FALSE
%token NULL
%token LEFT_BRACE
%token RIGHT_BRACE
%token LEFT_BRACK
%token RIGHT_BRACK
%token COLON
%token COMMA
%token EOF

%type <Json.value option> prog

%start prog

%%

prog: v = value
    { Some v }
  | EOF
    { None }
  ;

value: LEFT_BRACE; obj = opt_object_fields; RIGHT_BRACE
    { `Assoc obj }
  | LEFT_BRACK; vl = list_values; RIGHT_BRACK
    { `List vl }
  | s = STRING
    { `String s }
  | i = INT
    { `Int i }
  | x = FLOAT
    { `Float x }
  | TRUE
    { `Bool true }
  | FALSE
    { `Bool false }
  | NULL
    { `Null }
  ;
```

```
opt_object_fields: /* empty */
    { [] }
  | obj = rev_object_fields
    { List.rev obj }
  ;

rev_object_fields: k = ID; COLON; v = value
    { [k, v] }
  | obj = rev_object_fields; COMMA; k = ID; COLON; v = value
    { (k, v) :: obj }
  ;

list_values: /* empty */
    { [] }
  | vl = rev_values
    { List.rev vl }
  ;

rev_values: v = value
    { [v] }
  | vl = rev_values; COMMA; v = value
    { v :: vl }
  ;
```

That's it. We can compile this with `menhir`, which will now no longer complain about unused symbols.

# Defining a lexer with ocamllex

For the next part, we need to define a lexer to tokenize the input text, meaning that we break the input into a sequence of words or tokens. For this, we'll define a lexer using `ocamllex`. In this case, the specification is placed in a file with a `.mll` suffix (we'll use the name `lexer.mll`). A lexer file has several parts in the following sequence.

```
{ OCaml code }
let definitions...
rules...
{ OCaml code }
```

## Let-definitions for regular expressions

The OCaml code for the header and trailer is optional. The let-definitions are used to ease the definition of regular expressions. They are optional, but very useful. To get started, we know that we'll need to match numbers and strings, so let's define names for the regular expressions that specify their form.

An integer is a sequence of digits, optionally preceded by a minus sign. Leading zeroes are not allowed. The question mark means that the preceding symbol - is optional. The square brackets ['1'-'9'] define a character range, meaning that the first digit of the

integer should be 1-9. The final range `['0'-'9']*` includes star `*`, which means zero-or-more occurrences of the characters 0-9. Read formally then, an `int` has an optional minus sign, followed by a digit in the range 1-9, followed by zero or more digits in the range 0-9.

```
let int = '-'? ['1'-'9'] ['0'-'9']*
```

Floating-point numbers are similar, but we deal with decimal points and exponents. We can use multiple let-definitions for the different parts.

```
let digits = ['0'-'9']+
let frac = '.' digits
let exp = ['e' 'E'] ['-' '+']? digits
let float = int (frac | exp | frac exp)
```

The `digits` expression has a `+` symbol, meaning that `digits` has one or more occurrences of digits in the range 0-9. A fractional part `frac` has a decimal point followed by some digits; an exponent `exp` begins with an `e` followed by some digits; and a `float` has an integer part, and one or both of a `frac` and `exp` part. The vertical bar is a choice; the expression `(frac | exp | frac exp)` is either a `frac`, or an `exp`, or a `frac` followed by an `exp`.

Finally, let's define identifiers and whitespace. An identifier (label), is an alphanumeric sequence not beginning with a digit.

```
let white = [' ' '\t']+
let newline = '\r' | '\n' | "\r\n"

let id = ['a'-'z' 'A'-'Z' '_'] ['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

## Lexing rules

The lexing rules are specified as a set of `parse` rules. A `parse` rule has a regular expression followed by OCaml code that defines a semantic action. Let's write JSON parse rule.

```
rule read = parse
| white { read lexbuf }
| newline { next_line lexbuf; read lexbuf }
| int { INT (int_of_string (Lexing.lexeme lexbuf)) }
| float { FLOAT (float_of_string (Lexing.lexeme lexbuf)) }
| "true" { TRUE }
| "false" { FALSE }
| "null" { NULL }
| id { ID (Lexing.lexeme lexbuf) }
| '"' { read_string (Buffer.create 17) lexbuf }
| '{' { LEFT_BRACE }
| '}' { RIGHT_BRACE }
| '[' { LEFT_BRACK }
| ']' { RIGHT_BRACK }
```

```
| ':' { COLON }
| ',' { COMMA }
| _ { raise (SyntaxError ("Unexpected character: " ^ Lexing.lexeme lexbuf)) }
| eof { EOF }
```

The OCaml code for the rules has a parameter called `lexbuf` that defines the input, including the position in the input file, as well as the text that was matched by the regular expression. Let's skip to the third action.

```
| int { INT (int_of_string (Lexing.lexeme lexbuf)) }
```

This action specifies that when the input matches the `int` regular expression (defined as `'-'? ['1'-'9'] ['0'-'9']*`, then the lexer should return the expression `INT (int_of_string (Lexing.lexeme lexbuf))`. The expression `Lexing.lexeme lexbuf` returns the complete string matched by the regular expression. In this case, the string represents a number, so we use the `int_of_string` function to convert it to a number.

Going back to the first actions, the first `white { read lexbuf }` calls the lexer recursively. That's, it skips the input whitespace and returns the following token. The action `newline { next_line lexbuf; read lexbuf }` is similar, but we use it to advance the line number for the lexer. Here is the definition of the `next_line` function, which updates the line number in the `lexbuf`.

```
let next_line lexbuf =
  let pos = lexbuf.lex_curr_p in
  lexbuf.lex_curr_p <-
    { pos with pos_bol = lexbuf.lex_curr_pos;
               pos_lnum = pos.pos_lnum + 1
    }
```

There are actions for each different kind of token. The string expressions like `"true" { TRUE }` are used for keywords, and the special characters have actions too, like `'{' { LEFT_BRACE }`.

Some of these patterns overlap. For example, the regular expression `"true"` is also matched by the `id` pattern. `ocamllex` used the following disambiguation when a prefix of the input is matched by more than one pattern.

- The longest match always wins. For example, the first input `trueX: 167` matches the regular expression `"true"` for 4 characters, and it matches `id` for 5 characters. The longer match wins, and the return value is `ID "trueX"`.
- If all matches have the same length, then the first action wins. If the input were `true: 167`, then both `"true"` and `id` match the first 4 characters; `"true"` is first, so the return value is `TRUE`.

## Recursive rules

Unlike many other lexer generators, `ocamllex` allows the definition of multiple lexer in the same file, and the definitions can be recursive. In this case, we use recursion to match string literals, using the following rule definition.

```
and read_string buf = parse
| '"' { STRING (Buffer.contents buf) }
| '\\' '/' { Buffer.add_char buf '/'; read_string buf lexbuf }
| '\\' '\\' { Buffer.add_char buf '\\'; read_string buf lexbuf }
| '\\' 'b' { Buffer.add_char buf '\b'; read_string buf lexbuf }
| '\\' 'f' { Buffer.add_char buf '\012'; read_string buf lexbuf }
| '\\' 'n' { Buffer.add_char buf '\n'; read_string buf lexbuf }
| '\\' 'r' { Buffer.add_char buf '\r'; read_string buf lexbuf }
| '\\' 't' { Buffer.add_char buf '\t'; read_string buf lexbuf }
| '\\' 'u' hex hex hex hex
  { let string_code = String.sub (Lexing.lexeme lexbuf) 2 4 in
    let code = int_of_string ("0x" ^ string_code) in
    add_utf8 buf code;
    read_string buf lexbuf
  }
| [^ '"' '\\']+
  { Buffer.add_string buf (Lexing.lexeme lexbuf);
    read_string buf lexbuf
  }
| _ { raise (SyntaxError ("Illegal string character: " ^ Lexing.lexeme lexbuf)) }
| eof { raise (SyntaxError ("String is not terminated")) }
```

This rule takes a `buf : Buffer.t` as an argument. If we reach the terminating double quote `"`, then we return the contents of the buffer as a `STRING`.

The other cases are for handling the string contents. The action `[^ '"' '\\']+ { ... }` matches normal input that does not contain a double-quote or backslash. The actions beginning with a backslash `\` define what to do for escape sequences. In each of these cases, the final step includes a recursive call to the lexer.

As specified by JSON, we also handle Unicode code points, `'\\' 'u' hex hex hex hex`. Ocaml doesn't have any built-in handling for Unicode, so in this case we choose to represent the code point in UTF-8. We define the following function for adding the UTF-8 encoding to the buffer.

```
let add_utf8 buf code =
  if code <= 0x7f then
    Buffer.add_char buf (Char.chr code)
  else if code <= 0x7ff then begin
    Buffer.add_char buf (Char.chr (0b11000000 lor ((code lsr 6) land 0x3f)));
    Buffer.add_char buf (Char.chr (0b10000000 lor (code land 0x3f)))
  end else begin
    Buffer.add_char buf (Char.chr (0b11100000 lor ((code lsr 12) land 0x3f)));
    Buffer.add_char buf (Char.chr (0b10000000 lor ((code lsr 6) land 0x3f)));
    Buffer.add_char buf (Char.chr (0b10000000 lor (code land 0x3f)))
  end
```

That covers the lexer. Next, we need to combine the lexer with the parser to bring it all together.

## Bringing it all together

For the final part, we need to compose the lexer and parser. As we saw in the type definition in `parser.mli`, the parsing function expects a lexer of type `Lexing.lexbuf -> token`, and it also expects a `lexbuf`.

```
val prog: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> (Json.value option)
```

The standard lexing library `Lexing` provides a function `from_channel` to read the input from a channel. The following function describes the structure, where the `Lexing.from_channel` function is used to construct a `lexbuf`, which is passed with the lexing function `Lexer.read` to the `Parser.prog` function. `Parsing.prog` returns `None` when it reaches end of file. We define a function `Json.output_value`, not shown here, to print a `Json.value`.

```
let rec parse_and_print lexbuf =
  match Parser.prog Lexer.read lexbuf with
  | Some value -> Json.output_value stdout value; parse_and_print lexbuf
  | None -> ()

let loop filename =
  let inx = open_in filename in
  let lexbuf = Lexing.from_channel inx in
  parse_and_print lexbuf;
  close_in inx
```

This isn't quite right yet -- we need to handle parsing errors. Currently there are two errors, `Parser.Error` and `Lexer.SyntaxError`. A simple solution when encountering an error is to print the error and give up.

```
let parse_with_error lexbuf =
  try Parser.prog Lexer.read lexbuf with
  | SyntaxError msg ->
      Printf.fprintf stderr "%a: %s\n" print_position lexbuf msg;
      None
  | Parser.Error ->
      Printf.fprintf stderr "%a: syntax error\n" print_position lexbuf;
      None

let rec parse_and_print lexbuf =
  match parse_with_error lexbuf with
  | Some value -> Json.output_value stdout value; parse_and_print lexbuf
  | None -> ()
```

This approach, "give up on the first error," is easy to implement, but it isn't very friendly. In general, error handling can be pretty intricate, and we won't discuss it here. However,

the menhir parser defines additional mechanisms you can use to try and recover from errors, describe it its reference manual.

Here is an example of a successful run on the following input file.

```
$ cat test1.json
true
false
null
[1, 2]
"Hello\r\n\t\b\\\/\u12345"
{ field1: "Hello",
  field2: 17e13,
  field3: [1, 2, 3],
  field4: { fieldA: 1, fieldB: "Hello" }
}

$ ./test test1.json
true
false
null
[1, 2]
"Hello
       \/ 5"
{ field1: "Hello", field2: 170000000000000.000000, field3: [1, 2, 3], field4: { fieldA: 1, fieldB: "Hell
```

With our simple error handling scheme, errors are fatal.

```
$ cat test2.json
{ name: "Chicago",
  zips: [12345,
}
{ name: "New York",
  zips: [10004]
}
$ ./test test2.json
test2.json:3:2: syntax error
```

2013-07-04
10:28:31

# CHAPTER 17
# Data Serialization with S-Expressions

We've already shown you how to parse third-party data formats into OCaml in earlier chapters. Sometimes though, you just want to quickly convert an OCaml type to and from a human-readable and editable form in your own code, and not worry about interoperability. Core's solution to this problem is to use s-expressions.

S-expressions are nested parenthetical expressions whose atomic values are strings. They were first popularized by the Lisp programming language in the 1960s, and have remained one of the simplest and most effective ways to encode structured data. There's a full definition of them available online (*http://people.csail.mit.edu/rivest/Sexp.txt*).

An example s-expression might look like this:

```
(this (is an) (s expression))
```

The OCaml type of an s-expression is quite simple:

```
module Sexp : sig
  type t = Atom of string | List of t list
end
```

An s-expression can be thought of as a tree where each node contains a list of its children, and where the leaves of the tree are strings.

The `Sexp` module in Core comes with functionality for parsing and printing s-expressions.

```
# let sexp =
    let a x = Sexp.Atom x and l x = Sexp.List x in
    l [a "this";l [a "is"; a "an"]; l [a "s"; a "expression"]];;
val sexp : Sexp.t = (this (is an) (s expression))
```

In addition, most of the base types in Core support conversion to and from s-expressions. For example, we can write:

```
# Int.sexp_of_t 3;;
```

```
- : Sexp.t = 3
# List.sexp_of_t;;
- : ('a -> Sexp.t) -> 'a List.t -> Sexp.t = <fun>
# List.sexp_of_t Int.sexp_of_t [1;2;3];;
- : Sexp.t = (1 2 3)
```

Notice that `List.sexp_of_t` is polymorphic, and takes as its first argument another conversion function to handle the elements of the list to be converted. Core uses this scheme more generally for defining sexp-converters for polymorphic types.

But what if you want a function to convert some brand new type to an s-expression? You can of course write it yourself manually:

```
# type t = { foo: int; bar: float };;
# let sexp_of_t t =
    let a x = Sexp.Atom x and l x = Sexp.List x in
    l [ l [a "foo"; Int.sexp_of_t t.foo  ];
        l [a "bar"; Float.sexp_of_t t.bar]; ]
  ;;
val sexp_of_t : t -> Core.Std.Sexp.t = <fun>
# sexp_of_t { foo = 3; bar = -5.5 };;
- : Core.Std.Sexp.t = ((foo 3) (bar -5.5))
```

This is somewhat tiresome to write, and it gets more so when you consider the parser, *i.e.*, `t_of_sexp`, which is considerably more complex. Writing this kind of parsing and printing code by hand is mechanical and error prone, not to mention a drag.

Given how mechanical the code is, you could imagine writing a program that inspected the type definition and auto-generated the conversion code for you. As it turns out, we can do just that using `Sexplib`. The `Sexplib` package, which is included with Core, provides both a library for manipulating s-expressions and a syntax extension for generating such conversion functions. With that syntax extension enabled, any type that has `with sexp` as an annotation will trigger the generation of the functions we want for free.

```
# type t = { foo: int; bar: float } with sexp;;
type t = { foo : int; bar : float; }
val t_of_sexp__ : Sexplib.Sexp.t -> t = <fun>
val t_of_sexp : Sexplib.Sexp.t -> t = <fun>
val sexp_of_t : t -> Sexplib.Sexp.t = <fun>
# t_of_sexp (Sexp.of_string "((bar 35) (foo 3))");;
- : t = {foo = 3; bar = 35.}
```

The `with sexp` is detected by a `Sexplib` syntax extension and replaced with the extra conversion functions you see above. You can ignore `t_of_sexp__`, which is a helper function that is needed in very rare cases.

The syntax extensions in Core almost all have this same basic structure: they auto-generate code based on type definitions, implementing functionality that you could in theory have implemented by hand, but with far less programmer effort.

**The `camlp4` preprocessor and `type_conv`**

OCaml doesn't directly support converting static type definitions to and from other data formats. Instead, it supplies a powerful syntax extension mechanism known as `camlp4`. This lets you extend the grammar of the language to mark types as requiring special action, and then mechanically generate boilerplate code over those types (such as converting to and from other data formats).

Many of the examples in the subsequent chapters depend on `camlp4`, but the examples all invoke it automatically for you via the `-pp` flag to the OCaml compiler. If you're interested in building your own generators, investigate the `type_conv` library which provides the basic extension mechanism used by the rest of this chapter.

## The Sexp format

The textual representation of s-expressions is pretty straightforward. An s-expression is written down as a nested parenthetical expression, with whitespace-separated strings as the atoms. Quotes are used for atoms that contain parentheses or spaces themselves; backslash is the escape character; and semicolons are used to introduce single-line comments. Thus, the following file, `example.scm`:

```
;; example.scm

((foo 3.3) ;; This is a comment
 (bar "this is () an \" atom"))
```

can be loaded using sexplib. As you can see, the commented data is not part of the resulting s-expression.

```
# Sexp.load_sexp "example.scm";;
- : Core.Std.Sexp.t = ((foo 3.3) (bar "this is () an \" atom"))
```

All in, the s-expression format actually supports three comment syntaxes:

- `;`, which comments out everything to the end of a line
- `#|` and `|#`, which are delimiters for commenting out a block
- `#;`, which comments out the first complete s-expression that follows.

The following example shows all of these in action.

```
;; comment_heavy_example.scm
((this is included)
 ; (this is commented out
 (this stays)
 #; (all of this is commented
     out (even though it crosses lines.))
  (and #| block delimiters #| which can be nested #|
```

```
    will comment out
   an arbitrary multi-line block))) |#
  now we're done
  ))
```

Again, loading the file as an s-expression drops the comments.

```
# Sexp.load_sexp "comment_heavy_example.scm";;
- : Core.Std.Sexp.t = ((this is included) (this stays) (and now we're done))
```

Note that the comments were dropped from the file upon reading. This is expected, since there's no place in the `Sexp.t` type to store comments.

If we introduce an error into our s-expression, by, say, deleting the open-paren in front of `bar`, we'll get a parse error:

```
# Exn.handle_uncaught ~exit:false (fun () ->
    ignore (Sexp.load_sexp "example.scm"));;
  Uncaught exception:

  (Sexplib.Sexp.Parse_error
   ((location parse) (err_msg "unexpected character: ')'") (text_line 4)
    (text_char 29) (global_offset 94) (buf_pos 94)))
```

In the above, we use `Exn.handle_uncaught` to make sure that the exception gets printed out in full detail. You should generally wrap every Core program in this handler to get good error messages for any unexpected exceptions.

## Sexp converters

The most important functionality provided by Sexplib is the auto-generation of converters for new types. We've seen a bit of how this works already, but let's walk through a complete example. Here's the source for the beginning of a library for representing integer intervals.

```
(* file: int_interval.ml *)
(* Module for representing closed integer intervals *)

open Core.Std

(* Invariant: For any Range (x,y), y >= x *)
type t = | Range of int * int
         | Empty
with sexp

let is_empty = function Empty -> true | Range _ -> false
let create x y = if x > y then Empty else Range (x,y)
let contains i x = match i with
   | Empty -> false
   | Range (low,high) -> x >= low && x <= high
```

We can now use this module as follows:

```
(* file: test_interval.ml *)

open Core.Std

let intervals =
  let module I = Int_interval in
  [ I.create 3 4;
    I.create 5 4; (* should be empty *)
    I.create 2 3;
    I.create 1 6;
  ]

let () =
  intervals
  |> List.sexp_of_t Int_interval.sexp_of_t
  |> Sexp.to_string_hum
  |> print_endline
```

But we're still missing something: we haven't created an `mli` signature for `Int_interval` yet. Note that we need to explicitly export the s-expression converters that were created within the ml. If we don't:

```
(* file: int_interval.mli *)
(* Module for representing closed integer intervals *)

type t

val is_empty : t -> bool
val create : int -> int -> t
val contains : t -> int -> bool
```

then we'll get the following error:

```
File "test_interval.ml", line 15, characters 20-42:
Error: Unbound value Int_interval.sexp_of_t
Command exited with code 2.
```

We could export the types by hand in the signature:

```
type t
val sexp_of_t : Sexp.t -> t
val t_of_sexp : t -> Sexp.t
```

But Sexplib has a shorthand for this as well, so that we can just use the same `with` shorthand in the `mli`.

```
type t with sexp
```

at which point `test_interval.ml` will compile again, and if we run it, we'll get the following output:

```
$ ./test_interval.native
((Range 3 4) Empty (Range 2 3) (Range 1 6))
```

## Preserving invariants

One easy mistake to make when dealing with sexp converters is to ignore the fact that those converters can violate the invariants of your code. For example, the `Int_interval` module depends for the correctness of the `is_empty` check on the fact that for any value `Range (x,y)`, `y` is greater than or equal to `x`. The `create` function preserves this invariant, but the `t_of_sexp` function does not.

We can fix this problem by overriding the autogenerated function and writing a custom sexp-converter that is based on the auto-generated converter.

```
type t = | Range of int * int
         | Empty
with sexp

let create x y = if x > y then Empty else Range (x,y)

let t_of_sexp sexp =
  let t = t_of_sexp sexp in
  begin match t with
  | Empty -> ()
  | Range (x,y) ->
    if y < x then of_sexp_error "Upper and lower bound of Range swapped" sexp
  end;
  t
```

This trick of overriding an existing function definition with a new one is perfectly acceptable in OCaml. Function definitions are only recursive if the `rec` keyword is specified, and so in this case the inner `t_of_sexp` call will go to the earlier auto-generated definition that resulted from the `type t with sexp` definition.

We call the function `of_sexp_error` to raise an exception because that improves the error reporting that Sexplib can provide when a conversion fails.

## Getting good error messages

There are two steps to deserializing a type from an s-expression: first, converting the bytes in a file to an s-expression, and the second, converting that s-expression into the type in question. One problem with this is that it can be hard to localize errors to the right place using this scheme. Consider the following example:

```
(* file: read_foo.ml *)

open Core.Std
```

```
type t = { a: string; b: int; c: float option } with sexp

let run () =
  let t =
    Sexp.load_sexp "example.scm"
    |> t_of_sexp
  in
  printf "b is: %d\n%!" t.b

let () =
  Exn.handle_uncaught ~exit:true run
```

If you were to run this on a malformatted file, say, this one:

```
;; example.scm
((a not-an-integer)
 (b not-an-integer)
 (c ()))
```

you'll get the following error:

```
read_foo $ ./read_foo.native
Uncaught exception:

  (Sexplib.Conv.Of_sexp_error
   (Failure "int_of_sexp: (Failure int_of_string)") not-an-integer)
```

If all you have is the error message and the string, it's not terribly informative. In particular, you know that the parsing error-ed out on the atom "not-an-integer", but you don't know which one! In a large file, this kind of bad error message can be pure misery.

But there's hope! If we make small change to the `run` function as follows:

```
let run () =
  let t = Sexp.load_sexp_conv_exn "example.scm" t_of_sexp in
  printf "b is: %d\n%!" t.b
```

and run it again, we'll get the following much more helpful error message:

```
read_foo $ ./read_foo.native
Uncaught exception:

  (Sexplib.Conv.Of_sexp_error
   (Sexplib.Sexp.Annotated.Conv_exn example.scm:3:4
    (Failure "int_of_sexp: (Failure int_of_string)"))
   not-an-integer)
```

In the above error, "example.scm:3:4" tells us that the error occurred on "example.scm", line 3, character 4, which is a much better start for figuring out what has gone wrong.

## Sexp-conversion directives

Sexplib supports a collection of directives for modifying the default behavior of the auto-generated sexp-converters. These directives allow you to customize the way in which types are represented as s-expressions without having to write a custom parser.

### sexp_opaque

The most commonly used directive is `sexp_opaque`, whose purpose is to mark a given component of a type as being unconvertible. Anything marked with `sexp_opaque` will be presented as the atom `<opaque>` by the to-sexp converter, and will trigger an exception from the from-sexp converter.

Note that the type of a component marked as opaque doesn't need to have a sexp-converter defined. Here, if we define a type without a sexp-converter, and then try to use another type with a sexp-converter, we'll error out:

```
# type no_converter = int * int;;
type no_converter = int * int
# type t = { a: no_converter; b: string } with sexp;;
Characters 14-26:
  type t = { a: no_converter; b: string } with sexp;;
                ^^^^^^^^^^^^
Error: Unbound value no_converter_of_sexp
```

But with `sexp_opaque`, we won't:

```
# type t = { a: no_converter sexp_opaque; b: string } with sexp;;
type t = { a : no_converter Core.Std.sexp_opaque; b : string; }
val t_of_sexp__ : Sexplib.Sexp.t -> t = <fun>
val t_of_sexp : Sexplib.Sexp.t -> t = <fun>
val sexp_of_t : t -> Sexplib.Sexp.t = <fun>
```

And if we now convert a value of this type to an s-expression, we'll see the contents of field `a` marked as opaque:

```
# sexp_of_t { a = (3,4); b = "foo" };;
- : Sexp.t = ((a <opaque>) (b foo))
```

### sexp_list

Sometimes, sexp-converters have more parentheses than one would ideally like. Consider, for example, the following variant type:

```
# type compatible_versions = | Specific of string list
                             | All
  with sexp;;
```

```
# sexp_of_compatible_versions (Specific ["3.12.0"; "3.12.1"; "3.13.0"]);;
- : Sexp.t = (Specific (3.12.0 3.12.1 3.13.0))
```

You might prefer to make the syntax a bit less parenthesis-laden by dropping the parentheses around the list. `sexp_list` gives us this alternate syntax:

```
# type compatible_versions = | Specific of string sexp_list
                             | All
  with sexp;;
# sexp_of_compatible_versions (Specific ["3.12.0"; "3.12.1"; "3.13.0"]);;
- : Sexp.t = (Specific 3.12.0 3.12.1 3.13.0)
```

## sexp_option

Another common directive is `sexp_option`, which is used to to make a record field optional in the s-expression. Normally, optional values are represented either as `()` for `None`, or as `(x)` for `Some x`, and a record field containing an option would be rendered accordingly. For example:

```
# type t = { a: int option; b: string } with sexp;;
# sexp_of_t { a = None; b = "hello" };;
- : Sexp.t = ((a ()) (b hello))
# sexp_of_t { a = Some 3; b = "hello" };;
- : Sexp.t = ((a (3)) (b hello))
```

But what if we want a field to be optional, *i.e.*, we want to allow it to be omitted from the record entirely? In that case, we can mark it with `sexp_option`:

```
# type t = { a: int sexp_option; b: string } with sexp;;
# sexp_of_t { a = Some 3; b = "hello" };;
- : Sexp.t = ((a 3) (b hello))
# sexp_of_t { a = None; b = "hello" };;
- : Sexp.t = ((b hello))
```

### Specifying defaults

The `sexp_option` declaration is really just an example of how one might want to deal with default values. With `sexp_option`, your type on the OCaml side is an option, with `None` representing the case where no value is provided. But you might want to allow other ways of filling in default values.

Consider the following type which represents the configuration of a very simple web-server.

```
# type http_server_config = {
    web_root: string;
    port: int;
    addr: string;
  } with sexp;;
```

One could imagine making some of these parameters optional; in particular, by default, we might want the web server to bind to port 80, and to listen as localhost. The sexp-syntax allows this as follows.

```
# type http_server_config = {
    web_root: string;
    port: int with default(80);
    addr: string with default("localhost");
  } with sexp;;
```

The top-level will echo back the type you just defined as usual, but also generate some additional conversion functions.

```
type http_server_config = { web_root : string; port : int; addr : string; }
val http_server_config_of_sexp__ : Sexplib.Sexp.t -> http_server_config =
  <fun>
val http_server_config_of_sexp : Sexplib.Sexp.t -> http_server_config = <fun>
val sexp_of_http_server_config : http_server_config -> Sexplib.Sexp.t = <fun>
```

These new functions let you convert to and from s-expressions.

```
# http_server_config_of_sexp (Sexp.of_string "((web_root /var/www/html))";;
# let cfg =
  http_server_config_of_sexp (Sexp.of_string "((web_root /var/www/html))");;
val cfg : http_server_config =
  {web_root = "/var/www/html"; port = 80; addr = "localhost"}
```

When we convert the configuration back out to an s-expression, you'll notice that no data is dropped.

```
# sexp_of_http_server_config cfg;;
- : Sexplib.Sexp.t = ((web_root /var/www/html) (port 80) (addr localhost))
```

We could make the generated s-expression also drop exported values, by using the `sexp_drop_default` directive.

```
# type http_server_config = {
    web_root: string;
    port: int with default(80), sexp_drop_default;
    addr: string with default("localhost"), sexp_drop_default;
  } with sexp;;
type http_server_config = { web_root : string; port : int; addr : string; }
val http_server_config_of_sexp__ : Sexplib.Sexp.t -> http_server_config =
  <fun>
val http_server_config_of_sexp : Sexplib.Sexp.t -> http_server_config = <fun>
val sexp_of_http_server_config : http_server_config -> Sexplib.Sexp.t = <fun>
# let cfg = http_server_config_of_sexp (Sexp.of_string "((web_root /var/www/html))");;
val cfg : http_server_config =
  {web_root = "/var/www/html"; port = 80; addr = "localhost"}
# sexp_of_http_server_config cfg;;
- : Sexplib.Sexp.t = ((web_root /var/www/html))
```

As you can see, the fields that are at their default values are simply omitted from the s-expression. On the other hand, if we convert a config with other values, then those values will be included in the s-expression.

```
# sexp_of_http_server_config { cfg with port = 8080 };;
- : Sexplib.Sexp.t = ((web_root /var/www/html) (port 8080))
# sexp_of_http_server_config { cfg with port = 8080; addr = "192.168.0.1" };;
- : Sexplib.Sexp.t =
((web_root /var/www/html) (port 8080) (addr 192.168.0.1))
```

This can be very useful in designing config file formats that are both reasonably terse and easy to generate and maintain. It can also be useful for backwards compatibility: if you add a new field to your config record, but you make that field optional, then you should still be able to parse older version of your config.

2013-07-04
10:28:31

2013-07-04
10:28:31

CHAPTER 18

# Concurrent Programming with Async

The logic of building programs that interact with the outside world is often dominated by waiting: waiting for the click of a mouse, or for data to be fetched from disk, or for space to be available on an outgoing network buffer. Even mildly sophisticated interactive applications are typically *concurrent*, needing to wait for multiple different events at the same time, responding immediately to whatever event happens first.

A common approach to concurrency is to use preemptive system threads, which is the most common solution in languages like Java or C#. In this model, each task that may require simultaneous waiting is given an operating system thread of its own, so it can block without stopping the entire program. Other language runtimes such as Javascript are single-threaded, and applications register function callbacks to be triggered upon external events such as a timeout or browser click.

Each of these mechanisms has its own trade-offs. Preemptive threads require significant memory and other resources per thread. Also, the operating system can arbitrarily interleave the execution of preemptive threads, requiring the programmer to carefully protect shared resources with locks and condition variables, which can be exceedingly error-prone.

Single-threaded event-driven systems, on the other hand, execute a single task at a time and do not require the same kind of complex synchronization that preemptive threads do. However, the inverted control structure of an event-driven program often means that your own control flow has to be threaded awkwardly through the system's event loop, leading to a maze of event callbacks.

This chapter covers the Async library, which offers a hybrid model that aims to provide the best of both worlds, avoiding the performance compromises and synchronization woes of preemptive threads without the confusing inversion of control that usually comes with event-driven systems.

## Async basics

Consider a typical function for doing I/O in Core.

```
# In_channel.read_all;;
- : string -> string = <fun>
```

Since the function returns a concrete string, it has to block until the read completes. The blocking nature of the call means that no progress can be made on anything else until the read is completed, as you can see below.

```
# Out_channel.write_all "test.txt" ~data:"This is only a test.";;
- : unit = ()
# In_channel.read_all "test.txt";;
- : string = "This is only a test."
```

In Async, well-behaved functions never block. Instead, they return a value of type `Deferred.t` that acts as a placeholder that will eventually be filled in with the result. As an example, consider the signature of the Async equivalent of `In_channel.read_all`.

```
# open Async.Std;;
# Reader.file_contents;;
- : string -> string Deferred.t = <fun>
```

Note that we opened `Async.Std`, which adds a number of new identifiers and modules into our environment that make using Async more convenient. Opening `Async.Std` is standard practice for writing programs using Async, much like opening `Core.Std` is for using Core.

A deferred is essentially a handle to a value that may be computed in the future. As such, if we call `Reader.file_contents`, the resulting deferred will initially be empty, as you can see by calling `Deferred.peek` on the resulting deferred.

```
# let contents = Reader.file_contents "test.txt";;
val contents : string Deferred.t = <abstr>
# Deferred.peek contents;;
- : string option = None
```

The value in `contents` isn't yet determined in part because there's nothing running that could do the necessary I/O. When using Async, processing of I/O and other events is handled by the Async scheduler. When writing a stand-alone program, you need to start the scheduler explicitly, but utop knows about Async, and can start the scheduler automatically. More than that, utop knows about deferred values, and when you type in an expression of type `Deferred.t`, it will make sure the scheduler is running and block until the deferred is determined. Thus, we can write:

```
# contents;;
- : string = "This is only a test.\n"
```

```
# Deferred.peek contents;;
- : string option = Some "This is only a test.\n"
```

In order to do real work with deferreds, we need a way of sequencing deferred computations, which we do using `Deferred.bind`. First, let's consider the type-signature of bind.

```
# Deferred.bind ;;
- : 'a Deferred.t -> ('a -> 'b Deferred.t) -> 'b Deferred.t = <fun>
```

Thus, `Deferred.bind d f` takes a deferred value `d` and a function f that is to be run with the value of `d` once it's determined. The call to `Deferred.bind` returns a new deferred that becomes determined when the deferred returned by `f` is determined. It also implicitly registers with the scheduler an *Async job* that is responsible for running `f` once `d` is determined.

Here's a simple use of bind for a function that replaces a file with an uppercase version of its contents.

```
# let uppercase_file filename =
    let text = Reader.file_contents filename in
    Deferred.bind text (fun text ->
      Writer.save filename ~contents:(String.uppercase text))
  ;;
val uppercase_file : string -> unit Deferred.t = <fun>
# uppercase_file "test.txt";;
- : unit = ()
# Reader.file_contents "test.txt";;
- : string = "THIS IS ONLY A TEST."
```

Writing out `Deferred.bind` explicitly can be rather verbose, and so `Async.Std` includes an infix operator for it: `>>=`. Using this operator, we can rewrite `uppercase_file` as follows.

```
# let uppercase_file filename =
    Reader.file_contents filename
    >>= fun text ->
    Writer.save filename ~contents:(String.uppercase text)
  ;;
val uppercase_file : string -> unit Deferred.t = <fun>
```

In the above we've dropped the parentheses around the function on the right-hand side of the bind, and we didn't add a level of indentation for the contents of that function. This is standard practice for using the bind operator.

Now let's look at another potential use of bind. In this case, we'll write a function that counts the number of lines in a file.

```
# let count_lines filename =
    Reader.file_contents filename
    >>= fun text ->
```

```
    List.length (String.split text ~on:'\n')
  ;;
```

This looks reasonable enough, but when we try to compile it, we get the following error.

```
Error: This expression has type int but an expression was expected of type
         'a Deferred.t
```

The issue here is that bind expects a function that returns a deferred, but we've provided it a function that simply returns the result. To make these signatures match, we need a function for taking an ordinary value and wrapping it in a deferred. This function is a standard part of Async, and is called `return`:

```
# return;;
- : 'a -> 'a Deferred.t = <fun>
# let three = return 3;;
val three : int Deferred.t = <abstr>
# three;;
- : int = 3
```

Using `return`, we can make `count_lines` compile.

```
# let count_lines filename =
    Reader.file_contents filename
    >>= fun text ->
    return (List.length (String.split text ~on:'\n'))
  ;;
val count_lines : string -> int Deferred.t = <fun>
```

Together, `bind` and `return` form a design pattern in functional programming known as a *monad*. You'll run across this signature in many applications beyond just threads. Indeed, we already ran across monads in "`bind` and other error-handling idioms" on page 118.

Calling `bind` and `return` together is a fairly common pattern, and as such there is a standard shortcut for it called `Deferred.map`, which has the following signature:

```
# Deferred.map;;
- : 'a Deferred.t -> f:('a -> 'b) -> 'b Deferred.t = <fun>
```

and comes with its own infix equivalent, `>>|`. Using it, we can rewrite `count_lines` again a bit more succinctly:

```
# let count_lines filename =
    Reader.file_contents filename
    >>| fun text ->
    List.length (String.split text ~on:'\n')
  ;;
val count_lines : string -> int Deferred.t = <fun>
```

## Ivars and upon

Deferreds are usually built using combinations of `bind`, `map` and `return`, but sometimes you want to construct a deferred that you can determine explicitly with user-code. This is done using an *ivar*, which is a handle that lets you control precisely when a deferred becomes determined.

There are three fundamental operations for working with an ivar; you can create one, using `Ivar.create`, you can read off the deferred that corresponds to the ivar in question, using `Ivar.read`, and you can fill an ivar, thus causing that deferred to become determined, using `Ivar.fill`. These operations are illustrated below.

```
# let ivar = Ivar.create ();;
val ivar : '_a Ivar.t = <abstr>
# let def = Ivar.read ivar;;
val def : '_a Ivar.Deferred.t = <abstr>
# Deferred.peek def;;
- : '_a option = None
# Ivar.fill ivar "Hello";;
- : unit = ()
# Deferred.peek def;;
- : string option = Some "Hello"
```

Ivars are something of a low-level feature; operators like map, bind and return are typically easier to use and think about. But ivars can be useful when you want to build complicated synchronization patterns that can't be constructed naturally otherwise.

As an example, imagine we wanted a way of scheduling a sequence of actions that would run after a fixed delay. In addition, we'd like to guarantee that these delayed actions are executed in the same order they were scheduled in. One could imagine building a module for handling this with the following interface.

```
# module type Delayer_intf = sig
    type t
    val create : Time.Span.t -> t
    val schedule : t -> (unit -> 'a Deferred.t) -> 'a Deferred.t
  end;;
```

An action is handed to `schedule` in the form of a deferred-returning thunk (a thunk is a function whose argument is of type `unit`). A deferred is handed back to the caller of `schedule` that will eventually be filled with the contents of the deferred value returned by the thunk to be scheduled. We can implement this using an ivar which we fill after the thunk is called and the deferred it returns becomes determined. Instead of using `bind` or `map` for scheduling these events, we'll use a different operator called `upon`. Here's the signature of `upon`:

```
# upon;;
- : 'a Deferred.t -> ('a -> unit) -> unit = <fun>
```

Like `bind` and `return`, `upon` schedules a callback to be executed when the deferred it is passed is determined; but unlike those calls, it doesn't create a new deferred for this callback to fill.

Our delayer implementation is organized around a queue of thunks, where every call to `schedule` adds a thunk to the queue, and also schedules a job in the future to grab a thunk off the queue and run it. The waiting will be done using the function `after` which takes a time span and returns a deferred which becomes determined after that time span elapses. The role of the ivar here is to take the value returned by the thunk and use it to fill the deferred returned by the provided thunk.

```
# module Delayer : Delayer_intf = struct
    type t = { delay: Time.Span.t;
               jobs: (unit -> unit) Queue.t;
             }

    let create delay =
      { delay; jobs = Queue.create () }

    let schedule t thunk =
      let ivar = Ivar.create () in
      Queue.enqueue t.jobs (fun () ->
        upon (thunk ()) (fun x -> Ivar.fill ivar x));
      upon (after t.delay) (fun () ->
        let job = Queue.dequeue_exn t.jobs in
        job ());
      Ivar.read ivar
  end;;
module Delayer : Delayer_intf
```

This code isn't particularly long, but it is a bit subtle. This is typical of code that involves ivars and `upon`, and because of this, you should stick to the simpler map/bind/return style of working with deferreds when you can.

## Examples: an echo server

Now that we have the basics of Async under our belt, let's look at a small complete stand-alone Async program. In particular, we'll write an echo server, *i.e.*, a program that accepts connections from clients and spits back every line of text sent to it.

The first step is to create a function that can copy data from an input to an output. Here, we'll use Async's `Reader` and `Writer` modules which provide a convenient abstraction for working with input and output channels.

```
(* filename: echo.ml *)
open Core.Std
open Async.Std

(* Copy data from the reader to the writer, using the provided buffer
```

```
   as scratch space *)
let rec copy_blocks buffer r w =
  Reader.read r buffer
  >>= function
  | `Eof -> return ()
  | `Ok bytes_read ->
    Writer.write w buffer ~len:bytes_read;
    Writer.flushed w
    >>= fun () ->
    copy_blocks buffer r w
```

Bind is used in the above code to sequence the operations: first, we call `Reader.read` to get a block of input. Then, when that's complete and if a new block was returned, we write that block to the writer. Finally, we wait until the writer's buffers are flushed, waiting on the deferred returned by `Writer.flushed`, at which point we recur. If we hit an end-of-file condition, the loop is ended. The deferred returned by a call to `copy_blocks` becomes determined only once the end-of-file condition is hit.

One important aspect of how this is written is that it uses *pushback*, which is to say that if the writer can't make progress writing, the reader will stop reading. If you don't implement pushback in your servers, then a stopped client can cause your program to leak memory, since you'll need to allocate space for the data that's been read in but not yet written out.

Another memory leak you might be concerned with is the chain of deferreds that is built up as you go through the loop. After all, this code constructs an ever-growing chain of binds, each of which creates a deferred. In this case, however, all of the deferreds should become determined precisely when the final deferred in the chain is determined, in this case, when the `Eof` condition is hit. Because of this, we could safely replace all of these deferreds with a single deferred. Async has logic to do just this, which is essentially a form of tail-call optimization.

`copy_blocks` provides the logic for handling a client connection, but we still need to set up a server to receive such connections and dispatch to `copy_blocks`. For this, we'll use Async's `Tcp` module, which has a collection of utilities for creating simple TCP clients and servers.

```
(** Starts a TCP server, which listens on the specified port, invoking
    copy_blocks every time a client connects. *)
let run () =
  let host_and_port =
    Tcp.Server.create
      ~on_handler_error:`Raise
      (Tcp.on_port 8765)
      (fun _addr r w ->
         let buffer = String.create (16 * 1024) in
         copy_blocks buffer r w)
  in
  ignore (host_and_port : (Socket.Address.Inet.t, int) Tcp.Server.t Deferred.t)
```

The result of calling `Tcp.Server.create` is a `Tcp.Server.t`, which is a handle to the server that lets you shut the server down. We don't use that functionality here, so we explicitly ignore [server] to suppress the unused-variables error. We put in a type annotation around the ignored value to make the nature of the value we're ignoring explicit.

The most important argument to `Tcp.Server.create` is the final one, which is the client connection handler. Notably, the above code does nothing explicit to close down the client connections when the communication is done. That's because the server will automatically shut down the connection once the deferred returned by the handler becomes determined.

Finally, we need to initiate the server and start the Async scheduler.

```
(* Call [run], and then start the scheduler *)
let () =
  run ();
  never_returns (Scheduler.go ())
```

One of the most common newbie errors with Async is to forget to run the scheduler. It can be a bewildering mistake, because without the scheduler, your program won't do anything at all; even calls to `printf` won't actually reach the terminal.

It's worth noting that even though we didn't spend much explicit effort on thinking about multiple clients, this server is able to handle many concurrent clients without further modification.

Now that we have the echo server, we can try it out using `netcat`.

```
echo_server $ ./echo.native &
[1] 25030
echo_server $ nc 127.0.0.1 8765
This is an echo server
This is an echo server
It repeats whatever I write.
It repeats whatever I write.
```

**Functions that never return**

You might wonder what's going on with the call to `never_returns` above. `never_returns` is an idiom that comes from `Core` that is used to mark functions that don't return. Typically, a function that doesn't return is inferred as having return type `'a`.

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

```
# let always_fail () = assert false;;
val always_fail : unit -> 'a = <fun>
```

This can be surprising when you call a function like this expecting it to return unit, and really it never returns. The type-checker won't necessarily complain in such a case.

```
# let do_stuff n =
    let x = 3 in
    if n > 0 then loop_forever ();
    x + n
  ;;
val do_stuff : int -> unit = <fun>
```

With a name like `loop_forever`, the meaning is clear enough in this case. But with something like `Scheduler.go`, the fact that it never returns is less clear, and so we use the type-system to make it more explicit by giving it a return type of `never_returns`. To make it clearer how this works, let's do the same trick with `loop_forever`.

```
# let rec loop_forever () : never_returns = loop_forever ();;
val loop_forever : unit -> never_returns = <fun>
```

The type `never_returns` is uninhabited, so a function can't return a value of type `never_returns`, which means only functions that never return can have it as their return type! Now, if we rewrite our `do_stuff` function, we'll get a helpful type error.

```
# let do_stuff n =
    let x = 3 in
    if n > 0 then loop_forever ();
    x + n
  ;;
Error: This expression has type unit but an expression was expected of type
         never_returns
```

We can resolve the error by calling the function `never_returns`.

```
# never_returns;;
- : never_returns -> 'a = <fun>
# let do_stuff n =
    let x = 3 in
    if n > 0 then never_returns (loop_forever ());
    x + n
  ;;
val do_stuff : int -> int = <fun>
```

Thus, we got the compilation to go through by explicitly marking in the source that the call to `loop_forever` never returns.

## Improving the echo server

Let's try to go a little bit farther with our echo server. Let's walk through a few small improvements:

- Add a proper command-line interface with `Command`
- Add a flag to specify the port to listen on, and a flag to make the server echo back the capitalized version of whatever was sent to it.
- Simplify the code using Async's `Pipe` interface.

Here's the improved code below.

```
let run ~uppercase ~port =
  let host_and_port =
    Tcp.Server.create
      ~on_handler_error:`Raise
      (Tcp.on_port port)
      (fun _addr r w ->
        Pipe.transfer (Reader.pipe r) (Writer.pipe w)
           ~f:(if uppercase then String.uppercase else Fn.id))
  in
  ignore (host_and_port : (Socket.Address.Inet.t, int) Tcp.Server.t Deferred.t);
  Deferred.never ()

let () =
  Command.async_basic
    ~summary:"Start an echo server"
    Command.Spec.(
      empty
      +> flag "-uppercase" no_arg
        ~doc:" Convert to uppercase before echoing back"
      +> flag "-port" (optional_with_default 8765 int)
        ~doc:" Port to listen on (default 8765)"
    )
    (fun uppercase port () -> run ~uppercase ~port)
  |> Command.run
```

The most notable change in this function is the use of Async's `Pipe`. A `Pipe` is a communication channel that's used for connecting different parts of your program. You can think of it as a consumer/producer queue that uses deferreds for communicating when the pipe is ready to be read from or written to. Our use of pipes is fairly minimal here, but they are an important part of Async, so it's worth discussing them in some detail.

Pipes are created in connected read/write pairs, as you can see below.

```
# let (r,w) = Pipe.create ();;
val r : '_a Pipe.Reader.t = <abstr>
val w : '_a Pipe.Writer.t = <abstr>
```

`r` and `w` are really just read and write handles to the same underlying object. Note that `r` and `w` have weakly polymorphic types. That's because a pipe is mutable and so can contain elements of only one type, which will be settled by the compiler once we try to use the pipe for anything.

If we just try and write to the writer, we'll see that we block indefinitely in utop. You can break out of the wait by hitting `Control-C`.

```
# Pipe.write w "Hello World!";;
Interrupted.
```

The deferred returned by write completes on its own once the value written into the pipe has been read out:

```
# let (r,w) = Pipe.create ();;
val r : '_a Pipe.Reader.t = <abstr>
val w : '_a Pipe.Writer.t = <abstr>
# let write_complete = Pipe.write w "Hello World!";;
val write_complete : unit Deferred.t = <abstr>
# Pipe.read r;;
- : [ `Eof | `Ok of string ] = `Ok "Hello World!"
# write_complete;;
- : unit = ()
```

In the function `run` above, we're taking advantage of one of the many utility functions provided for pipes in the `Pipe` module. In particular, we're using `Pipe.transfer` to set up a process that takes data from a reader-pipe and moves it to a writer-pipe. Here's the type of `Pipe.transfer`:

```
# Pipe.transfer;;
- : 'a Pipe.Reader.t -> 'b Pipe.Writer.t -> f:('a -> 'b) -> unit Deferred.t =
<fun>
```

The two pipes being connected are generated by the `Reader.pipe` and `Writer.pipe` call respectively. Note that pushback is preserved throughout the process, so that if the writer gets blocked, the writer's pipe will stop pulling data from the reader's pipe, which will prevent the reader from reading in more data.

Importantly, the deferred returned by `Pipe.transfer` becomes determined once the reader has been closed and the last element is transferred from the reader to the writer. Once that deferred becomes determined, the server will shut down that client connection. So, when a client disconnects, the rest of the shutdown happens transparently.

The command-line parsing for this program is based on the `Command` library that we introduced in Chapter 14. When you open `Async.Std`, the `Command` module has added to it the `async_basic` call:

```
# Command.async_basic;;
- : summary:string ->
    ?readme:(unit -> string) ->
    ('a, unit -> unit Deferred.t) Command.Spec.t -> 'a -> Command.t
= <fun>
```

This differs from the ordinary `Command.basic` call in that the main function must return a `Deferred.t`, and that the running of the command (using `Command.run`) automatically starts the async scheduler, without requiring an explicit call to `Scheduler.go`.

## Example: searching definitions with DuckDuckGo

DuckDuckGo is a search engine with a freely available search interface. In this section, we'll use Async to write a small command-line utility for querying DuckDuckGo to extract definitions for a collection of terms.

Our code is going to rely on a number of other libraries, all of which can be installed using OPAM. Refer to Appendix A if you need help on the installation. Here's the list of libraries we'll need.

- `textwrap`, a library for wrapping long lines. We'll use this for printing out our results.
- `uri`, a library for handling URI's, or "Uniform Resource Identifiers", of which HTTP URL's are an example.
- `yojson`, a JSON parsing library that was described in Chapter 15
- `cohttp`, a library for creating HTTP clients and servers. We need Async support, which comes with the `cohttp.async` package.

Now let's dive into the implementation.

### URI handling

You're probably familiar with HTTP URLs, which identify endpoints across the World Wide Web. These are actually part of a more general family known as Uniform Resource Identifiers (URIs). The full URI specification is defined in RFC3986 (*http://tools.ietf.org/html/rfc3986*), and is rather complicated. Luckily, the `uri` library provides a strongly-typed interface which takes care of much of the hassle.

We'll need a function for generating the URI's that we're going to use to query the DuckDuckGo servers.

```
(* file: search.ml *)
open Core.Std
open Async.Std

(* Generate a DuckDuckGo search URI from a query string *)
let query_uri query =
  let base_uri = Uri.of_string "http://api.duckduckgo.com/?format=json" in
  Uri.add_query_param base_uri ("q", [query])
```

A `Uri.t` is constructed from the `Uri.of_string` function, and a query parameter `q` is added with the desired search query. The library takes care of encoding the URI correctly when outputting it in the network protocol.

## Parsing JSON strings

The HTTP response from DuckDuckGo is in JSON, a common (and thankfully simple) format that is specified in RFC4627 (*http://www.ietf.org/rfc/rfc4627.txt*). We'll parse the JSON data using the Yojson library, which we already introduced in Chapter 15.

We expect the response from DuckDuckGo to come across as a JSON record, which is represented by the `Assoc` tag in Yojson's JSON variant. We expect the definition itself to come across under either the key "Abstract" or "Definition", and so the code below looks under both keys, returning the first one for which a non-empty value is defined.

```
(* Extract the "Definition" or "Abstract" field from the DuckDuckGo results *)
let get_definition_from_json json =
  match Yojson.Safe.from_string json with
  | `Assoc kv_list ->
    let find key =
      begin match List.Assoc.find kv_list key with
      | None | Some (`String "") -> None
      | Some s -> Some (Yojson.Safe.to_string s)
      end
    in
    begin match find "Abstract" with
    | Some _ as x -> x
    | None -> find "Definition"
    end
  | _ -> None
```

## Executing an HTTP client query

Now let's look at the code for dispatching the search queries over HTTP, using the Cohttp library.

```
(* Execute the DuckDuckGo search *)
let get_definition word =
  Cohttp_async.Client.get (query_uri word)
  >>= fun (_, body) ->
  Pipe.to_list body
  >>| fun strings ->
  (word, get_definition_from_json (String.concat strings))
```

To better understand what's going on, it's useful to look at the type for `Cohttp_async.Client.get`, which we can do in utop.

```
# #require "cohttp.async";;
# Cohttp_async.Client.get;;
- : ?interrupt:unit Deferred.t ->
    ?headers:Cohttp.Header.t ->
    Uri.t -> (Cohttp.Response.r * Cohttp_async.body) Deferred.t
= <fun>
```

The `get` call takes as a required argument a URI, and returns a deferred value containing a `Cohttp.Response.t` (which we ignore) and a pipe reader to which the body of the request will be written to as it is received.

In this case, the HTTP body probably isn't very large, so we call `Pipe.to_list` to collect the strings from the pipe as a single deferred list of strings. We then join those strings using `String.concat` and pass the result through our parsing function.

Running a single search isn't that interesting from a concurrency perspective, so let's write code for dispatching multiple searches in parallel. First, we need code for formatting and printing out the search result.

```
(* Print out a word/definition pair *)
let print_result (word,definition) =
  printf "%s\n%s\n\n%s\n\n"
    word
    (String.init (String.length word) ~f:(fun _ -> '-'))
    (match definition with
    | None -> "No definition found"
    | Some def ->
      String.concat ~sep:"\n"
        (Wrapper.wrap (Wrapper.make 70) def))
```

We use the `Wrapper` module from the `textwrap` package to do the line-wrapping. It may not be obvious that this routine is using Async, but it does: the version of `printf` that's called here is actually Async's specialized `printf` that goes through the Async scheduler rather than printing directly. The original definition of `printf` is shadowed by this new one when you open `Async.Std`. An important side effect of this is that if you write an Async program and forget to start the scheduler, calls like `printf` won't actually generate any output!

The next function dispatches the searches in parallel, waits for the results, and then prints.

```
(* Run many searches in parallel, printing out the results after they're all
   done. *)
let search_and_print words =
  Deferred.all (List.map words ~f:get_definition)
  >>| fun results ->
  List.iter results ~f:print_result
```

We used `List.map` to call `get_definition` on each word, and `Deferred.all` to wait for all the results. Here's the type of `Deferred.all`:

```
# Deferred.all;;
- : 'a Deferred.t list -> 'a list Deferred.t = <fun>
```

Note that the list returned by `Deferred.all` reflects the order of the deferreds passed to it. As such, the definitions will be printed out in the same order that the search words

are passed in, no matter what orders the queries return in. We could rewrite this code to print out the results as they're received (and thus potentially out of order) as follows.

```
(* Run many searches in parallel, printing out the results as you go *)
let search_and_print words =
  Deferred.all_unit (List.map words ~f:(fun word ->
    get_definition word >>| print_result))
```

The difference is that we both dispatch the query and print out the result in the closure passed to `map`, rather than waiting for all of the results to get back and then printing them out together. We use `Deferred.all_unit`, which takes a list of `unit` deferreds and returns a single `unit` deferred that becomes determined when every deferred on the input list is determined. We can see the type of this function in utop.

```
# Deferred.all_unit;;
- : unit Deferred.t list -> unit Deferred.t = <fun>
```

Finally, we create a command line interface using `Command.async_basic`.

```
let () =
  Command.async_basic
    ~summary:"Retrieve definitions from duckduckgo search engine"
    Command.Spec.(
      empty
      +> anon (sequence ("word" %: string))
    )
    (fun words () -> search_and_print words)
  |> Command.run
```

And that's all we need to create a simple but usable definition searcher.

```
$ ./search.native "Concurrent Programming" "OCaml"
Concurrent Programming
----------------------

"Concurrent computing is a form of computing in which programs are
designed as collections of interacting computational processes that
may be executed in parallel."

OCaml
-----

"OCaml, originally known as Objective Caml, is the main implementation
of the Caml programming language, created by Xavier Leroy, Jérôme
Vouillon, Damien Doligez, Didier Rémy and others in 1996."
```

## Exception handling

When programming with external resources, errors are everywhere: everything from a flaky server to a network outage to exhausting of local resources can lead to a runtime

error. When programming in OCaml, some of these errors will show up explicitly in a function's return type, and some of them will show up as exceptions. We covered exception handling in OCaml in “Exceptions” on page 120, but as we'll see, exception handling in a concurrent program presents some new challenges.

Let's get a better sense of how exceptions work in Async by creating an asynchronous computation that (sometimes) fails with an exception. The function `maybe_raise` below blocks for half a second, and then either throws an exception or returns unit, alternating between the two behaviors on subsequent calls.

```
# let maybe_raise =
    let should_fail = ref false in
    fun () ->
      let will_fail = !should_fail in
      should_fail := not will_fail;
      after (Time.Span.of_sec 0.5)
      >>= fun () ->
      if will_fail then raise Exit else return ()
 ;;
val maybe_raise : Core.Span.t -> unit Deferred.t = <fun>
# maybe_raise ();;
- : unit = ()
# maybe_raise ();;
Exception:
(lib/monitor.ml.Error_
 ((exn Exit) (backtrace ""))
  (monitor
   (((name block_on_async) (here ()) (id 5) (has_seen_error true)
     (someone_is_listening true) (kill_index 0))
    ((name main) (here ()) (id 1) (has_seen_error false)
     (someone_is_listening false) (kill_index 0))))).
```

In utop, the exception thrown by `maybe_raise ()` terminates the evaluation of just that expression, but in a stand-alone program, an uncaught exception would bring down the entire process.

So, how could we capture and handle such an exception? You might try to do this using OCaml's built-in `try/with` statement, but as you can see below, that doesn't quite do the trick.

```
# let handle_error () =
    try
      maybe_raise ()
      >>| fun () -> "success"
    with _ -> return "failure"
  ;;
val handle_error : unit -> string Deferred.t = <fun>
# handle_error ();;
- : string = "success"
# handle_error ();;
Exception:
(lib/monitor.ml.Error_
```

```
((exn Exit) (backtrace ""))
 (monitor
  (((name block_on_async) (here ()) (id 58) (has_seen_error true)
    (someone_is_listening true) (kill_index 0))
   ((name main) (here ()) (id 1) (has_seen_error false)
    (someone_is_listening false) (kill_index 0))))).
```

This didn't work because `try/with` only captures exceptions that are thrown in the code directly executed within it, while `maybe_raise` schedules an Async job to run in the future, and it's that job that throws an exception.

We can capture this kind of asynchronous error use the `try_with` function provided by Async:

```
# let handle_error () =
    try_with (fun () -> maybe_raise ())
    >>| function
    | Ok ()   -> "success"
    | Error _ -> "failure"
  ;;
# handle_error ();;
- : string = "success"
# handle_error ();;
  - : string = "failure"
```

`try_with f` takes as its argument a deferred-returning thunk `f`, and returns a deferred that becomes determined either as `Ok` of whatever `f` returned, or `Error exn` if `f` threw an exception before its return value became determined.

## Monitors

`try_with` is a a great way of handling exceptions in Async, but it's not the whole story. All of Async's exception-handling mechanisms, `try_with` included, are built on top of Async's system of *monitors*, which are inspired by the error-handling mechanism in Erlang of the same name. Monitors are fairly low-level and are only occasionally used directly, but it's nonetheless worth understanding how they work.

In Async, a monitor is a context that determines what to do when there is an unhandled exception. Every Async job runs within the context of some monitor, which, when the job is running, is referred to as the current monitor. When a new Async job is scheduled, say, using `bind` or `map`, it inherits the current monitor of the job that spawned it.

Monitors are arranged in a tree -- when a new monitor is created (say, using `Monitor.create`) it is a child of the current monitor. You can explicitly run jobs within a monitor using `within`, which takes a thunk that returns a non-deferred value, or `within'`, which takes a thunk that returns a deferred. Here's an example.

```
# let blow_up () =
    let monitor = Monitor.create ~name:"blow up monitor" () in
    within' ~monitor maybe_raise
```

```
  ;;
# blow_up ();;
- : unit = ()
# blow_up ();;
Exception:
(lib/monitor.ml.Error_
 ((exn Exit) (backtrace (""))
  (monitor
   (((name "blow up monitor") (here ()) (id 73) (has_seen_error true)
     (someone_is_listening false) (kill_index 0))
    ((name block_on_async) (here ()) (id 72) (has_seen_error false)
     (someone_is_listening true) (kill_index 0))
    ((name main) (here ()) (id 1) (has_seen_error false)
     (someone_is_listening false) (kill_index 0))))))).
```

In addition to the ordinary stack-trace, the exception displays the trace of monitors through which the exception traveled, starting at the one we created, called "blow up monitor". The other monitors you see come from utop's special handling of deferreds.

Monitors can do more than just augment the error-trace of an exception. You can also use a monitor to explicitly handle errors delivered to that monitor. The `Monitor.errors` call is a particularly important one. It detaches the monitor from its parent, handing back the stream of errors that would otherwise have been delivered to the parent monitor. This allows one to do custom handling of errors, which may include re-raising errors to the parent. Here is a very simple example of function that captures and ignores errors in the processes it spawns.

```
# let swallow_error () =
    let monitor = Monitor.create () in
    Stream.iter (Monitor.errors monitor) ~f:(fun _exn ->
      printf "an error happened\n");
    within' ~monitor (fun () ->
      after (Time.Span.of_sec 0.5) >>= fun () -> failwith "Kaboom!")
  ;;
val swallow_error : unit -> 'a Deferred.t = <fun>
# swallow_error ();;
an error happened
```

The message "an error happened" is printed out, but the deferred returned by `swallow_error` is never determined. This makes sense, since the calculation never actually completes, so there's no value to return. You can break out of this in utop by hitting `Control-C`.

Here's an example of a monitor which passes some exceptions through to the parent, and handles others. Exceptions are sent to the parent using `Monitor.send_exn`, with `Monitor.current` being called to find the current monitor, which is the parent of the newly created monitor.

```
# exception Ignore_me;;
exception Ignore_me
# let swallow_some_errors exn_to_raise =
```

```
    let child_monitor  = Monitor.create  () in
    let parent_monitor = Monitor.current () in
    Stream.iter (Monitor.errors child_monitor) ~f:(fun error ->
      match Monitor.extract_exn error with
      | Ignore_me -> printf "ignoring exn\n"
      | _ -> Monitor.send_exn parent_monitor error);
    within' ~monitor:child_monitor (fun () ->
       after (Time.Span.of_sec 0.5)
       >>= fun () -> raise exn_to_raise)
  ;;
val swallow_some_errors : exn -> 'a Deferred.t = <fun>
```

Note that we use `Monitor.extract_exn` to grab the underlying exception that was thrown. Async wraps exceptions it catches with extra information, including the monitor trace, so you need to grab the underlying exception to match on it.

If we pass in an exception other than `Ignore_me`, like, say, the built-in exception `Not_found`, then the exception will be passed to the parent monitor and delivered as usual.

```
# swallow_some_errors Not_found;;
Exception:
(lib/monitor.ml.Error_
 ((exn Not_found) (backtrace ""))
  (monitor
   (((name (id 3)) (here ()) (id 3) (has_seen_error true)
     (someone_is_listening true) (kill_index 0))
    ((name block_on_async) (here ()) (id 2) (has_seen_error true)
     (someone_is_listening true) (kill_index 0))
    ((name main) (here ()) (id 1) (has_seen_error false)
     (someone_is_listening false) (kill_index 0))))).
```

If instead we use `Ignore_me`, the exception will be ignored, and we again see that the deferred never returns, but the exception was caught and ignored.

```
# swallow_some_errors Ignore_me;;
ignoring exn
```

In practice, you should rarely use monitors directly, instead using functions like `try_with` and `Monitor.protect` that are built on top of monitors. One example of a library that uses monitors directly is `Tcp.Server.create`, which tracks both exceptions thrown by the logic that handles the network connection and by the callback for responding to an individual request, in either case responding to an exception by closing the connection. It is for building this kind of custom error handling that monitors can be helpful.

## Example: Handling exceptions with DuckDuckGo

Let's now go back and improve the exception handling of our DuckDuckGo client. In particular, we'll change it so that any individual queries that fail are reported as such, without preventing other queries from succeeding.

The search code as it is fails rarely, so let's make a change that allows us to trigger failures more predictably. We'll do this by making it possible to distribute the requests over multiple servers. Then, we'll handle the errors that occur when one of those servers is misspecified.

First we'll need to change `query_uri` to take an argument specifying the server to connect to, as follows.

```
(* Generate a DuckDuckGo search URI from a query string *)
let query_uri ~server query =
  let base_uri =
    Uri.of_string (String.concat ["http://";server;"/?format=json"])
  in
  Uri.add_query_param base_uri ("q", [query])
```

and then making the appropriate changes to get the list of servers on the command-line, and to distribute the search queries round-robin over the list of servers. Now, let's see what happens if we rebuild the application and run it giving it a list of servers, some of which won't respond to the query.

```
$ ./search_with_configurable_server.native \
     -servers localhost,api.duckduckgo.com \
     "Concurrent Programming" OCaml
("unhandled exception"
 ((lib/monitor.ml.Error_
   ((exn (Unix.Unix_error "Connection refused" connect 127.0.0.1:80))
    (backtrace
     ("Raised by primitive operation at file \"lib/unix_syscalls.ml\", line 793, characters 12-69"
      "Called from file \"lib/deferred.ml\", line 24, characters 62-65"
      "Called from file \"lib/scheduler.ml\", line 120, characters 6-17"
      "Called from file \"lib/jobs.ml\", line 73, characters 8-13" ""))
    (monitor
     (((name Tcp.close_sock_on_error) (here ()) (id 3) (has_seen_error true)
       (someone_is_listening true) (kill_index 0))
      ((name main) (here ()) (id 1) (has_seen_error true)
       (someone_is_listening false) (kill_index 0))))))
  (Pid 1352)))
```

As you can see, we got a "Connection refused" failure which ends the entire program, even though one of the two queries would have gone through successfully. We can handle the failures of individual connections separately by using the `try_with` function within each call to `get_definition`, as follows.

```
(* Execute the DuckDuckGo search *)
let get_definition ~server word =
```

```
  try_with (fun () ->
    Cohttp_async.Client.get (query_uri ~server word)
    >>= fun  (_, body) ->
    Pipe.to_list body
    >>| fun strings ->
    (word, get_definition_from_json (String.concat strings)))
  >>| function
  | Ok (word,result) -> (word, Ok result)
  | Error _          -> (word, Error "Unexpected failure")
```

Here, we use `try_with` to capture the exception, which we then use map (the `>>|` operator) to convert the error into the form we want: a pair whose first element is the word being searched for, and the second element is the (possibly erroneous) result.

Now we just need to change the code for `print_result` so that it can handle the new type.

```
(* Print out a word/definition pair *)
let print_result (word,definition) =
  printf "%s\n%s\n\n%s\n\n"
    word
    (String.init (String.length word) ~f:(fun _ -> '-'))
    (match definition with
     | Error s -> "DuckDuckGo query failed: " ^ s
     | Ok None -> "No definition found"
     | Ok (Some def) ->
       String.concat ~sep:"\n"
         (Wrapper.wrap (Wrapper.make 70) def))
```

Now, if we run that same query, we'll get individualized handling of the connection failures:

```
$ ./search_with_error_handling.native \
     -servers localhost,api.duckduckgo.com \
     "Concurrent Programming" OCaml
Concurrent Programming
----------------------

DuckDuckGo query failed unexpectedly

OCaml
-----

"OCaml, originally known as Objective Caml, is the main implementation
of the Caml programming language, created by Xavier Leroy, Jérôme
Vouillon, Damien Doligez, Didier Rémy and others in 1996."
```

Now, only the query that went to `localhost` failed.

Note that in this code, we're relying on the fact that `Cohttp_async.Client.get` will clean up after itself after an exception, in particular by closing its file descriptors. If you need to implement such functionality directly, you may want to use the `Monitor.protect` call,

which is analogous to the `protect` call described in "Cleaning up in the presence of exceptions" on page 124.

## Timeouts, Cancellation and Choices

In a concurrent program, one often needs to combine results from multiple distinct concurrent sub-computations going on in the same program. We already saw this in our DuckDuckGo example, where we used `Deferred.all` and `Deferred.all_unit` to wait for a list of deferreds to become determined. Another useful primitive is `Deferred.both`, which lets you wait until two deferreds of different types have returned, returning both values as a tuple. Here, we use the function `sec`, which is shorthand for creating a time-span equal to a given number of seconds.

```
# let string_and_float = Deferred.both
   (after (sec 0.5)  >>| fun () -> "A")
   (after (sec 0.25) >>| fun () -> 32.33);;
val string_and_float : (string * float) Deferred.t = <abstr>
# string_and_float;;
- : string * float = ("A", 32.33)
```

Sometimes, however, we want to wait only for the first of multiple events to occur. This happens particularly often when dealing with timeouts. In that case, we can use the call `Deferred.any`, which, given a list of deferreds, returns a single deferred that will become determined once any of the values on the list is determined.

```
# Deferred.any [ (after (sec 0.5) >>| fun () -> "half a second")
               ; (after (sec 10.) >>| fun () -> "ten seconds") ] ;;
- : string = "half a second"
```

Let's use this to add timeouts to our DuckDuckGo searches. We'll do this by writing a wrapper for `get_definition` that takes a timeout (in the form of a `Time.Span.t`) as an argument, and returns either the definition, or, if that takes too long, the timeout.

```
let get_definition_with_timeout ~server ~timeout word =
  Deferred.any
    [ (after timeout >>| fun () -> (word,Error "Timed out"))
    ; (get_definition ~server word
       >>| fun (word,result) ->
       let result' = match result with
         | Ok _ as x -> x
         | Error _ -> Error "Unexpected failure"
       in
       (word,result')
      )
    ]
```

We use `>>|` above to transform the deferred values we're waiting for so that `Deferred.any` can choose between values of the same type.

A problem with this code is that the HTTP query kicked off by `get_definition` is not actually shut down when the timeout fires. As such, `get_definition_with_timeout` essentially leaks an open connection. Happily, Cohttp does provide a way of shutting down a client. You can pass a deferred under the label `interrupt` to `Cohttp_async.Client.get`. Once `interrupt` is determined, the client connection will terminated and the corresponding connections closed.

The following code shows how you can change `get_definition` and `get_definition_with_timeout` to cancel the `get` call if the timeout expires.

```
(* Execute the DuckDuckGo search *)
let get_definition ~server ~interrupt word =
  try_with (fun () ->
    Cohttp_async.Client.get ~interrupt (query_uri ~server word)
    >>= fun  (_, body) ->
    Pipe.to_list body
    >>| fun strings ->
    (word, get_definition_from_json (String.concat strings)))
  >>| function
  | Ok (word,result) -> (word, Ok result)
  | Error exn        -> (word, Error exn)
```

Next, we'll modify `get_definition_with_timeout` to create a deferred to pass in to `get_definition` which will become determined when our timeout expires.

```
let get_definition_with_timeout ~server ~timeout word =
  get_definition ~server ~interrupt:(after timeout) word
  >>| fun (word,result) ->
  let result' = match result with
    | Ok _ as x -> x
    | Error _ -> Error "Unexpected failure"
  in
  (word,result')
```

This will work, and will cause the connection to shut-down cleanly when we time out; but our code no longer explicitly knows whether or not the timeout has kicked in. In particular, the error message on a timeout will now be `Unexpected failure` rather than `Timed out`, which it was in our previous implementation. This is a minor issue in this case, but if we wanted to have special behavior in the case of a timeout, it would be a more serious issue.

We can get more precise handling of timeouts using Async's `choose` operator, which lets you pick between a collection of different deferreds, reacting to exactly one of them. Each deferred is combined, using the function `choice`, with a function that is called if and only if that is the chosen deferred. Here's the type signature of `choice` and `choose`:

```
# choice;;
- : 'a Deferred.t -> ('a -> 'b) -> 'b Deferred.choice = <fun>
# choose;;
- : 'a Deferred.choice list -> 'a Deferred.t = <fun>
```

`choose` provides no guarantee that the `choice` built around the first deferred to become determined will in fact be chosen. But `choose` does guarantee that only one `choice` will be chosen, and only the chosen `choice` will execute the attached closure.

In the following, we use `choose` to ensure that the `interrupt` deferred becomes determined if and only if the timeout-deferred is chosen. Here's the code.

```
let get_definition_with_timeout ~server ~timeout word =
  let interrupt = Ivar.create () in
  choose
    [ choice (after timeout) (fun () ->
       Ivar.fill interrupt ();
       (word,Error "Timed out"))
    ; choice (get_definition ~server ~interrupt:(Ivar.read interrupt) word)
        (fun (word,result) ->
           let result' = match result with
             | Ok _ as x -> x
             | Error _ -> Error "Unexpected failure"
           in
           (word,result')
        )
    ]
```

Now, if we run this with a suitably small timeout, we'll see that some queries succeed and some fail, and the timeouts are reported as such.

```
$ ./search_with_timeout_no_leak.native "concurrent programming" ocaml -timeout 0.1s
concurrent programming
----------------------

DuckDuckGo query failed: Timed out

ocaml
-----

"OCaml or Objective Caml, is the main implementation of the Caml
programming language, created by Xavier Leroy, Jérôme Vouillon,
Damien Doligez, Didier Rémy and others in 1996."
```

## Working with system threads

OCaml does have built-in support for true system threads, *i.e.*, kernel-level threads that can be interleaved at any time with each other. We discussed in the beginning of the chapter why Async is generally a better choice than system threads, but even if you mostly use Async, OCaml's system threads are sometimes necessary, and it's worth understanding them.

The most surprising aspect of OCaml's system threads is that they don't afford you any access to physical parallelism on the machine. That's because OCaml's runtime has a single runtime lock which at most one thread can be holding at a time.

Given that threads don't provide physical parallelism, why are they useful at all?

The most common reason for using system threads is that there are some operating system calls that have no non-blocking alternative, which means that you can't run them directly in a system like Async without blocking out the entire system. For this reason, Async maintains a thread pool for running such calls. Most of the time, as a user of Async you don't need to think about this, but it's worth knowing that it's happening.

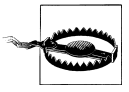
Another reasonably common reason to have multiple threads is to deal with non-OCaml libraries that have their own event loop or for whatever reason need their own threads. In that case, it's sometimes useful to run some OCaml code on the C thread as part of the communication between your main program and the C library. C bindings are discussed in more detail in Chapter 19.

Another occasional motivation for using true system threads is to interoperate with compute-intensive OCaml code that otherwise would block the Async runtime. In Async, if you have a long-running computation that never calls `bind` or `map`, then that computation will block out the entire system until it completes.

One way of dealing with this is to explicitly break up the calculation into smaller pieces that are separated by binds. But sometimes this explicit yielding is impractical, since it may involve intrusive changes to an existing codebase. Another solution is to run the code in question in a separate thread. Async's `In_thread` module provides multiple facilities for doing just this, `In_thread.run` being the simplest. We can simply write:

```
# let def = In_thread.run (fun () -> List.range 1 10);;
val def : int list Deferred.t = <abstr>
# def;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9]
```

To cause `List.range 1 10` to be run on one of Async's worker threads. When the computation is complete, the result is placed in the deferred, where it can be used in the ordinary way from Async.



**Thread-safety and locking**

Once you start working with system threads, you'll need to be careful about locking your data-structures. Most OCaml data-structures do not have well-defined semantics when accessed concurrently by multiple threads. The issues you can run into range from runtime exceptions to corrupted data-structures to, in some rare cases, segfaults. That means you should always use mutexes when sharing data between different systems threads. Even data-structures that seem like they should be safe, like lazy values, can have undefined behavior when accessed from multiple threads.

There are two commonly available mutex packages for OCaml: the `Mutex` module that's part of the standard library, which is just a wrapper over OS-level mutexes, and `Nano_mutex`, a more efficient alternative that takes advantage of some of the locking done by the OCaml runtime to avoid needing to create an OS-level mutex much of the time. As a result, creating a `Nano_mutex.t` is 20x faster than creating a `Mutex.t`, and acquiring the mutex is about 40% faster.

Interoperability between Async and system threads can be quite tricky, so let's work through some of the issues. Consider the following function tries which schedules itself to wake up every `100ms`, and keeps a list of the delays after which it actually woke up, until the deferred it was passed becomes determined. We can use this to see how responsive Async is.

```
# let log_delays d =
    let start = Time.now () in
    let rec loop stamps =
      let delay = Time.diff (Time.now ()) start in
      match Deferred.peek d with
      | Some () -> return (delay :: stamps)
      | None ->
        after (sec 0.1)
        >>= fun () ->
        loop (delay :: stamps)
    in
    loop [] >>| List.rev
  ;;
```

If we feed this function a simple timeout, it works as you might expect.

```
# log_delays (after (sec 1.));;
- : Core.Span.t list =
[0s; 101.38ms; 206.134ms; 312.682ms; 413.978ms; 522.833ms; 626.277ms;
 727.612ms; 829.041ms; 930.538ms; 1.03186s]
```

Now, if instead of simply waiting a second, what if we have a busy loop running instead?

```
# let busy_loop n =
    let x = ref None in
```

```
    for i = 1 to n * 100_000 do x := Some i done
  ;;
# log_delays (Deferred.unit >>| fun () -> busy_loop 100);;
- : Core.Span.t list = [0.000953674ms; 2.35119s]
```

As you can see, instead of waking up ten times a second, `log_delays` is blocked out for nearly a second while `busy_loop` churns away.

If, on the other hand, we use `In_thread.run` to offload this to a different system thread, the behavior will be different.

```
# log_delays (In_thread.run (fun () -> busy_loop 100));;
- : Core.Span.t list = [0.000953674ms; 1.47171s; 2.23492s; 2.33602s]
```

Now `log_delays` does get a chance to run, but not nearly as often as it would like to. The reason for this is that that now that we're using system threads, we are at the mercy of the operating system in terms of when each thread gets scheduled. The behavior becomes very dependent on the OS and how you configure the priority of your processes within the OS itself.

Another tricky aspect of dealing with OCaml threads has to do with allocation. OCaml's threads only get a chance to give up the runtime lock when they interact with the allocator, so if there's a piece of code that doesn't allocate at all, then it will never allow any other OCaml thread to run. We can see this if we rewrite our busy-loop slightly.

```
# let noalloc_busy_loop n =
    let rec loop n =
      if n <= 0 then ()
      else loop (n-1)
    in
    loop (n * 100_000)
  ;;
val noalloc_busy_loop : int -> unit = <fun>
# log_delays (In_thread.run (fun () -> noalloc_busy_loop 500));;
- : Core.Span.t list =
[0.000953674ms; 977.943ms; 1.13111s; 2.30346s; 3.11876s; 4.44648s; 6.57483s]
```

Even though `noalloc_busy_loop` was running in a different thread, it didn't let `log_delays` run at all.

DANGER WILL ROBINSON. Why doesn't this work as I expect? Did the compiler change somehow?

Overall, combining Async and threads is quite tricky, but it can be done safely and easily if you follow the following hold:

- There is no shared state between the various threads involved.
- The computations executed by `In_thread.run` do not make any calls to the async library.

It is possible to safely use threads in ways that violate these constraints. In particular, foreign threads can acquire the Async lock using calls from the `Thread_safe` module in Async, and thereby run Async computations safely. This is a very flexible way of connecting threads to the Async world, but it's a complex use-case that is beyond the scope of this chapter.

# PART III

# The Runtime System

Part III is all about understanding the compiler toolchain and runtime system in OCaml. It's a remarkably simple system in comparison to other language runtimes (such as Java or the .NET CLR).

You'll need to read this to build very high performance systems that have to minimise resource usage or interface to C libraries. This is also where we talk about profiling and debugging techniques using tools such as GNU gdb.

2013-07-04
10:28:31

2013-07-04
10:28:31

CHAPTER 19

# Foreign Function Interface

OCaml has several options available to interact with non-OCaml code. The compiler can link to external system libraries via C code, and also produce standalone native object files that can be embedded within other non-OCaml applications.

The mechanism by which code in one programming language can invoke routines in another different programming language is called a *foreign function interface*. This chapter will teach you:

- how to call routines in C libraries directly from your OCaml code.
- build higher-level abstractions in OCaml from the low-level C bindings.
- work through some full examples for binding a terminal interface and UNIX date/time functions.

## The Ctypes library

The simplest foreign function interface in OCaml doesn't even require you to write any C code at all! The Ctypes library lets you define the C interface in pure OCaml, and the library then takes care of loading the C symbols and invoking the foreign function call.

Let's dive straight into a realistic example to show you how the library looks. We'll create a binding to the Ncurses terminal toolkit, as it's widely available on most systems and doesn't have any complex dependencies.

**Installing the Ctypes library**

You'll need to install the `libffi` (*https://github.com/atgreen/libffi*) library as a prerequisite to using Ctypes. It's a fairly popular library and should be available in your OS package manager.

A special note for Mac users: the version of `libffi` installed by default in MacOS X 10.8 is too old for some of the features that Ctypes needs. Use Homebrew to `brew install libffi` to get the latest version before installing the OCaml library.

Once that's done, Ctypes is available via OPAM as usual.

```
$ brew install libffi     # for MacOS X users
$ opam install ctypes
$ utop
# require "ctypes.foreign" ;;
```

You'll also need the Ncurses library for the first example. This comes pre-installed on many operating systems such as MacOS X. Debian Linux provides it as the `ncurses-dev` package.

## Example: a terminal interface

Ncurses is a library to help build terminal-independent text interfaces in a reasonably efficient way. It's used in console mail clients like Mutt and Pine, and console web browsers such as Lynx.

The full C interface is quite large and is explained in the online documentation (*http://www.gnu.org/software/ncurses/*). We'll just use the smaller excerpt that's shown below since we just want to demonstrate Ctypes in action.

```
// <ncurses.h>
typedef struct _win_st WINDOW;

WINDOW *initscr   (void);
WINDOW *newwin    (int, int, int, int);
void    endwin    (void);
void    refresh   (void);
void    wrefresh  (WINDOW *);
void    mvwaddstr (WINDOW *, int, int, char *);
```

The Ncurses functions either operate on the current pseudo-terminal or on a window that has been created via `newwin`. The `WINDOW` structure holds the internal library state and is considered abstract outside of Ncurses. Ncurses clients just need to store the pointer somewhere and pass it back to Ncurses library calls, which in turn dereference its contents.

Note that there are over 200 library calls in Ncurses, so we're only binding a select few for this example. The `initscr` and `newwin` create `WINDOW` pointers for the global and sub-windows respectively. The `mvwaddrstr` takes a window, x/y offsets and a string and

writes to the screen at that location. The terminal is only updated after `refresh` or `wrefresh` are called.

Ctypes provides an OCaml interface that lets you map these C functions to equivalent OCaml functions. The library takes care of converting OCaml function calls and arguments into the C calling convention, invoking the foreign call within the C library and finally returning the result as an OCaml value.

Let's begin by defining the basic values we need, starting with the `WINDOW` state pointer.

```
(* ncurses.ml 1/3 *)
open Ctypes

type window = unit ptr
let window : window typ = ptr void
```

We don't know the internal representation of the window pointer, so we treat it as a C void pointer. We'll improve on this later on in the chapter, but it's good enough for now. The second statement defines an OCaml value that represents the `WINDOW` C pointer. This value is used later in the Ctypes function definitions.

```
(* ncurses.ml 2/3 *)
open Foreign

let initscr =
  foreign "initscr" (void @-> returning window)
```

That's all we need to invoke our first function call to `initscr` to initalize the terminal. The `foreign` function accepts two parameters:

- the C function call name, which is looked up using the `dlsym` C standard library function.
- a value that defines the complete set of C function arguments and its return type. The `@->` operator adds an argument to the C parameter list and `returning` terminates the parameter list with the return type.

The remainder of the Ncurses binding simply expands on these definitions.

```
(* ncurses.ml 3/3 *)
let endwin =
  foreign "endwin" (void @-> returning void)

let refresh =
  foreign "refresh" (void @-> returning void)

let wrefresh =
  foreign "wrefresh" (window @-> returning void)

let newwin =
  foreign "newwin" (int @-> int @-> int @-> int @-> returning window)
```

```
let mvwaddch =
  foreign "mvwaddch" (window @-> int @-> int @-> char @-> returning void)

let addstr =
  foreign "addstr" (string @-> returning void)

let mvwaddstr =
  foreign "mvwaddstr" (window @-> int @-> int @-> string @-> returning void)

let box =
  foreign "box" (window @-> int @-> int @-> returning void)

let cbreak =
  foreign "cbreak" (void @-> returning void)
```

These definitions are all straightforward mappings from the C declarations in the Ncurses header file. The scalar C types such as `int` come pre-defined in Ctypes. The `string` in these definitions maps from OCaml strings (which have a specific length) onto C character buffers (whose length is defined by a terminating null character that immediately follows the string data).

The module signature for `ncurses.mli` looks much like a normal OCaml signature. You can infer it directly from the `ncurses.ml` by running:

```
$ ocamlfind ocamlc -i -package ctypes.foreign ncurses.ml
```

The OCaml signature can be customized to improve its safety for external callers by making some of its internals more abstract.

```
(* ncurses.mli *)
type window

val window    : window Ctypes.typ
val initscr   : unit   -> window
val endwin    : unit   -> unit
val refresh   : unit   -> unit
val wrefresh  : window -> unit
val newwin    : int    -> int -> int -> int -> window
val mvwaddch  : window -> int -> int -> char -> unit
val addstr    : string -> unit
val mvwaddstr : window -> int -> int -> string -> unit
val box       : window -> int -> int -> unit
val cbreak    : unit   -> unit
```

The `window` type is left abstract in the signature to ensure that window pointers can only be constructed via the `Ncurses.initscr` function. This prevents void pointers obtained from other sources from being mistakenly passed to an Ncurses library call.

Now compile a "hello world" terminal drawing program to tie this all together.

```
(* hello.ml *)
open Ncurses
```

```
let () =
  let main_window = initscr () in
  cbreak ();
  let small_window = newwin 10 10 5 5 in
  mvwaddstr main_window 1 2 "Hello";
  mvwaddstr small_window 2 2 "World";
  box small_window 0 0;
  refresh ();
  Unix.sleep 1;
  wrefresh small_window;
  Unix.sleep 5;
  endwin ()
```

The `hello` executable is compiled by linking against the `ctypes.foreign` OCamlfind package.

```
$ ocamlfind ocamlopt -linkpkg -package ctypes.foreign -cclib -lncurses \
    ncurses.mli ncurses.ml hello.ml -o hello
```

Running `./hello` should now display a Hello World in your terminal!

**Reliable dynamic linking of external libraries**

The command-line above includes `-cclib -lncurses` to make the OCaml compiler link the executable to the `ncurses` C library, which in turns makes the C symbols available to the program when it starts. You'll get an error when you run the binary if you omit that link directive.

```
$ ocamlfind ocamlopt -linkpkg -package ctypes -package unix \
  ncurses.mli ncurses.ml hello.ml -o hello_broken
$ ./hello_broken
Fatal error: exception Dl.DL_error("dlsym(RTLD_DEFAULT, initscr): symbol not found")
```

There's one additional twist on modern versions of GCC (as included in recent Ubuntus). The `--as-needed` flag is passed to the linker by default, which instructs it to only link libraries that actually contain symbols used by the program at build time. This must be disabled when using Ctypes 0.1, as it will drop any libraries that might be opened at runtime by the `foreign` function.

So on Ubuntu, you'll need to compile with this option disabled to ensure that the Ncurses library dependency isn't dropped at compile time.

```
$ ocamlfind ocamlopt -linkpkg -package ctypes.foreign \
    -cclib '-Wl,--no-as-needed -lncurses' \
    ncurses.mli ncurses.ml hello.ml -o hello
```

*Note to reviewers*: this will be fixed in a future Ctypes release, so is only needed when using Ctypes 0.1.

Ctypes wouldn't be very useful if it were limited to only defining simple C types of course. It provides full support for C pointer arithmetic, pointer conversions, reading and writing through pointers, using OCaml functions as function pointers to C code, as well as struct and union definitions.

We'll go over some of these features in more detail for the remainder of the chapter, using some POSIX date functions as running examples.

## Basic scalar C types

First, let's look at how to define basic scalar C types. Every C type is represented by an OCaml equivalent via the single type definition below.

```
(* Ctypes *)
type 'a typ
```

`Ctypes.typ` is the type of values that represents C types to OCaml. There are two types associated with each instance of `typ`:

- the C type used to store and pass values to the foreign library.
- the corresponding OCaml type. The `'a` type parameter contains the OCaml type such that a value of type `t typ` is used to read and write OCaml values of type `t`.

There are various other uses of `typ` values within Ctypes.

- constructing function types for binding native functions.
- constructing pointers for reading and writing locations in C-managed storage.
- describing the fields of arrays, structures and unions.

Here are the definitions for most of the standard C99 scalar types, including some platform-dependent ones.

```
(* Ctypes *)
val void  : unit typ
val char : char typ
val schar : int typ
val short : int typ
val int   : int typ
val long  : long typ
val llong  : llong typ
val nativeint : nativeint typ

val int8_t : int typ
val int16_t : int typ
val int32_t : int32 typ
val int64_t : int64 typ
val uchar : uchar typ
val uchar : uchar typ
val uint8_t : uint8 typ
```

```
val uint16_t : uint16 typ
val uint32_t : uint32 typ
val uint64_t : uint64 typ
val size_t : size_t typ
val ushort : ushort typ
val uint : uint typ
val ulong : ulong typ
val ullong : ullong typ

val float : float typ
val double : float typ
```

These values are all of type `'a typ`, where the value name (e.g. `void`) tells you the C type and the `'a` component (e.g. `unit`) is the OCaml representation of that C type. Most of the mappings are straightforward, but some of them need a bit more explanation.

- Void values appear in OCaml as the `unit` type. Using `void` in an argument or result type specification produces an OCaml function which accepts or returns unit. Dereferencing a pointer to `void` is an error, as in C, and will raise the `Incomplete Type` exception.
- The C `size_t` type is an alias for one of the unsigned integer types. The actual size and alignment requirements for `size_t` varies between platforms. Ctypes provides an OCaml `size_t` type that is aliased to the appropriate integer type.
- OCaml only supports double-precision floating point numbers, and so the C `float` and `double` functions both map onto the OCaml `float` type.

## Pointers and arrays

Pointers are at the heart of C, so they are necessarily part of Ctypes, which provides support for pointer arithmetic, pointer conversions, reading and writing through pointers, and passing and returning pointers to and from functions.

We've already seen a simple use of pointers in the Ncurses example. Let's start a new example by binding some POSIX functions. The `time` function returns the current calendar time, and has the following C signature:

```
time_t time(time_t *);
```

The first step is to open some of the Ctypes modules.

- The `Ctypes` module provides functions for describing C types in OCaml.
- The `PosixTypes` module includes some extra POSIX-specific types (such as `time_t`).
- The `Foreign` module exposes the `foreign` function that makes it possible to invoke C functions.

We can now create a binding to `time` directly from the top-level.

```
$ utop
```

```
# require "ctypes.foreign" ;;
# open Ctypes ;;
# open PosixTypes ;;
# open Foreign ;;
# let time = foreign "time" (ptr time_t @-> returning time_t) ;;
val time : time_t ptr -> time_t = <fun>
```

The `foreign` function is the main link between OCaml and C. It takes two arguments: the name of the C function to bind, and a value describing the type of the bound function. In the `time` binding, the function type specifies one argument of type `ptr time_t` and a return type of `time_t`.

We can now call `time` immediately in the same top-level. The argument is actually optional, so we'll just pass a null pointer that has been coerced into becoming a null pointer to `time_t`.

```
# let cur_time = time (from_voidp time_t null) ;;
val cur_time : time_t = <abstr>
```

Since we're going to call `time` a few times, let's create a wrapper function that passes the null pointer through.

```
# let time' () = time (from_voidp time_t null) ;;
val time' : unit -> time_t = <fun>
```

Since `time_t` is an abstract type, we can't actually do anything useful with it directly. We need to bind a second function to do anything useful with the return values from `time`. We'll use the standard C function `difftime`, which has the following signature:

```
double difftime(time_t, time_t);
```

A binding to `difftime` is sufficient to compare two `time_t` values.

```
# let difftime = foreign "difftime" (time_t @-> time_t @-> returning double) ;;
val difftime : time_t -> time_t -> float = <fun>
# let t1 = time' () in
  Unix.sleep 2;
  let t2 = time' () in
  difftime t2 t1 ;;
 - : float = 2.
```

## Allocating typed memory for pointers

Let's look at a slightly less trivial example where we pass a non-null pointer to a function. Continuing with the theme from earlier, we'll bind to the `ctime` function which converts a `time_t` value to a human-readable string. The C signature of `ctime` is as follows:

```
char *ctime(const time_t *timep);
```

The corresponding binding can be written in the top-level to add to our growing collection.

```
# let ctime = foreign "ctime" (ptr time_t @-> returning string) ;;
val ctime : time_t ptr -> string = <fun>
```

However, we can't just pass the result of `time` to `ctime`.

```
# ctime (time' ());;
Error: This expression has type time_t but an expression was expected
of type time_t ptr
```

This is because `ctime` needs a pointer to the `time_t` rather than passing it by value. We thus need to allocate some memory for the `time_t` and obtain its memory address.

```
# let t_ptr = allocate time_t (time' ()) ;;
val t_ptr : time_t ptr = <abstr>
```

The `allocate` function takes the type of the memory to be allocated and the initial value, and it returns a suitably-typed pointer. We can now call `ctime` passing the pointer as an argument:

```
# ctime t_ptr;;
- : string = "Sat Jun  8 12:20:42 2013\n"
```

## Using views to map complex values

While scalar types typically have a 1-1 representation, other C types require extra work to convert them into OCaml. Views create new C type descriptions that have special behaviour when used to read or write C values.

We've already used one view in the definition of `ctime` earlier. The `string` view wraps the C type `char *` (written in OCaml as `ptr char`), and converts between the C and OCaml string representations each time the value is written or read.

Here is the type signature of the `view` function.

```
(* Ctypes *)
val view : read:('a -> 'b) -> write:('b -> 'a) -> 'a typ -> 'b typ
```

Next, we have the conversion functions to convert between an OCaml `string` and a C character buffer.

```
val string_of_char_ptr : char ptr -> string
val char_ptr_of_string : string -> char ptr
```

The definition of `string` that uses views is quite simple and is written using these conversion functions.

```
let string =
  view
    ~read:string_of_char_ptr
    ~write:char_ptr_of_string
    (char ptr)
```

The type of this `string` function is a normal `typ`, with no external sign of the use of the view function.

```
val string : string typ
```

**OCaml strings versus C character buffers**

Although OCaml strings may look like C character buffers from an interface perspective, they're very different in terms of their memory representations.

OCaml strings are stored in the OCaml heap with a header that explicitly defines their length. C buffers are also fixed-length, but by convention a C string is terminated by a null (a `0` byte) character. The C string functions calculate their length by scanning the buffer until the first null character is encountered.

This means you need to be careful when passing OCaml strings to C buffers that don't contain any null values within the OCaml string, or else the C string will be rudely truncated.

# Structs and unions

The C constructs `struct` and `union` make it possible to build new types from existing types. Ctypes contains counterparts that work similarly.

## Defining a structure

Let's improve the timer function that we wrote earlier. The POSIX function `gettimeofday` retrieves the time with microsecond resolution. The signature of `gettimeofday` is as follows, including the structure definitions.

```
struct timeval {
  long tv_sec;
  long tv_usec;
};

int gettimeofday(struct timeval *, struct timezone *tv);
```

Using Ctypes, we can describe this type as follows in our top-level.

```
# type timeval;;
```

```
type timeval
# let timeval : timeval structure typ = structure "timeval" ;;
val timeval : timeval structure typ = <abstr>
```

The first command defines a new OCaml type `timeval` that we'll use to instantiate the OCaml version of the `struct`. Creating a new OCaml type to reflect the underlying C type in this way means that the structure we define will be distinct from other structures we define elsewhere, which helps to avoid getting them mixed up.

The second command calls `structure` to create a fresh structure type. At this point the structure type is incomplete: we can add fields but cannot yet use it in `foreign` calls or use it to create values.

## Adding fields to structures

The `timeval` structure definition still doesn't have any fields, so we need to add those next.

```
# let tv_sec  = timeval *:* long ;;
val tv_sec : (Signed.long, timeval structure) field = <abstr>
# let tv_usec = timeval *:* long ;;
val tv_usec : (Signed.long, timeval structure) field = <abstr>
# seal timeval ;;
- : unit = ()
```

The `*:*` operator appends a field to the structure, as shown with `tv_sec` and `tv_usec` above. Structure fields are typed accessors that are associated with a particular structure, and they correspond to the labels in C. Note that there's no explicit requirement that the OCaml variable names for a field are the same as the corresponding C struct label names, but it helps avoid confusion.

Every field addition mutates the structure variable and records a new size (the exact value of which depends on the type of the field that was just added). Once we `seal` the structure we will be able to create values using it, but adding fields to a sealed structure is an error.

## Incomplete structure definitions

Since `gettimeofday` needs a `struct timezone` pointer for its second argument, we also need to define a second structure type.

```
# type timezone ;;
type timezone
# let timezone : timezone structure typ = structure "timezone" ;;
val timezone : timezone structure typ = <abstr>
```

We don't ever need to create `struct timezone` values, so we can leave this struct as incomplete without adding any fields or sealing it. If you ever try to use it in a situation

where its concrete size needs to be known, the library will raise an `IncompleteType` exception.

We're finally ready to bind to `gettimeofday`.

```
# let gettimeofday = foreign "gettimeofday"
    (ptr timeval @-> ptr timezone @-> returning_checking_errno int) ;;
val gettimeofday : timeval structure ptr -> timezone structure ptr -> int = <fun>
```

There's one other new feature here: the `returning_checking_errno` function behaves like `returning`, except that it checks whether the bound C function modifies the C error flag. Changes to `errno` are mapped into OCaml exceptions and raise a `Unix.Unix_error` exception just as the standard library functions do.

As before we can create a wrapper to make `gettimeofday` easier to use. The functions `make`, `addr` and `getf` create a structure value, retrieve the address of a structure value, and retrieve the value of a field from a structure.

```
# let gettimeofday' () =
  let tv = make timeval in
  ignore(gettimeofday (addr tv) (from_voidp timezone null));
  let secs = Signed.Long.(to_int (getf tv tv_sec)) in
  let usecs = Signed.Long.(to_int (getf tv tv_usec)) in
  Pervasives.(float secs +. float usecs /. 1000000.0) ;;
val gettimeofday : timeval structure ptr -> timezone structure ptr -> int = <fun>

# gettimeofday' () ;;
- : float = 1370714234.606070
```

You need to be a little careful not to get all the open modules mixed up here. Both Pervasives and Ctypes define different `float` functions. The Ctypes module we opened up earlier overrides the Pervasives definition. As seen above though, you just need to locally open Pervasives again to bring the usual `float` function back in scope,

#### Recap: a time-printing command

We built up a lot of bindings in the earlier section, so let's recap them with a complete example that ties it together with a command-line frontend.

```
(* datetime.ml: display time in various formats *)
open Core.Std
open Ctypes
open PosixTypes
open Foreign

let time     = foreign "time" (ptr time_t @-> returning time_t)
let difftime = foreign "difftime" (time_t @-> time_t @-> returning double)
let ctime    = foreign "ctime" (ptr time_t @-> returning string)

type timeval
let timeval : timeval structure typ = structure "timeval"
```

```
let tv_sec   = timeval *:* long
let tv_usec  = timeval *:* long
let ()       = seal timeval

type timezone
let timezone : timezone structure typ = structure "timezone"

let gettimeofday = foreign "gettimeofday"
    (ptr timeval @-> ptr timezone @-> returning_checking_errno int)

let time' () = time (from_voidp time_t null)

let gettimeofday' () =
  let tv = make timeval in
  ignore(gettimeofday (addr tv) (from_voidp timezone null));
  let secs = Signed.Long.(to_int (getf tv tv_sec)) in
  let usecs = Signed.Long.(to_int (getf tv tv_usec)) in
  Pervasives.(float secs +. float usecs /. 1_000_000.)

let float_time () = printf "%f%!\n" (gettimeofday' ())

let ascii_time () =
  let t_ptr = allocate time_t (time' ()) in
  printf "%s%!" (ctime t_ptr)

let () =
  let open Command in
  basic ~summary:"Display the current time in various formats"
    Spec.(empty +> flag "-a" no_arg ~doc:" Human-readable output format")
    (fun human -> if human then ascii_time else float_time)
  |> Command.run
```

This can be compiled as usual with ocamlfind and ocamlopt.

```
$ ocamlfind ocamlopt -o datetime -package core -package ctypes.foreign \
  -thread -linkpkg datetime.ml
$ ./datetime -a
Sat Jun  8 19:28:59 2013
```

### Why do we need to use `returning`?

The alert reader may be curious why all these function definitions have to be terminated by `returning`.

```
val time: ptr time_t @-> returning time_t
val difftime: time_t @-> time_t @-> returning double
```

The `returning` function may appear superfluous here. Why couldn't we simply give the types as follows?

```
val time: ptr time_t @-> time_t
val difftime: time_t @-> time_t @-> double
```

The reason involves higher types and two differences between the way that functions are treated in OCaml and C. Functions are first-class values in OCaml, but not in C. For example, in C, it is possible to return a function pointer from a function, but not to return an actual function.

Secondly, OCaml functions are typically defined in a curried style. The signature of a two-argument function is written as follows:

```
val curried : int -> int -> int
```

but this really means

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

and the arguments can be supplied one at a time to create a closure. In contrast, C functions receive their arguments all at once. The equivalent C function type is the following:

```
int uncurried_C(int, int);
```

and the arguments must always be supplied together:

```
uncurried_C(3, 4);
```

A C function that's written in curried style looks very different:

```
/* A function that accepts an int, and returns a function pointer that
   accepts a second int and returns an int. */
typedef int (function_t)(int);
function_t *curried_C(int);

/* supply both arguments */
curried_C(3)(4);

/* supply one argument at a time */
function_t *f = curried_C(3); f(4);
```

The OCaml type of `uncurried_C` when bound by Ctypes is `int -> int -> int`: a two-argument function. The OCaml type of `curried_C` when bound by `ctypes` is `int -> (int -> int)`: a one-argument function that returns a one-argument function.

In OCaml, of course, these types are absolutely equivalent. Since the OCaml types are the same but the C semantics are quite different, we need some kind of marker to distinguish the cases. This is the purpose of `returning` in function definitions.

## Defining arrays

Arrays in C are contiguous blocks of the same value. Any of the basic types defined earlier can be allocated as blocks via the `Array` module.

```
module Array : sig
  type 'a t = 'a array

  val get : 'a t -> int -> 'a
  val set : 'a t -> int -> 'a -> unit
  val of_list : 'a typ -> 'a list -> 'a t
  val to_list : 'a t -> 'a list
  val length : 'a t -> int
  val start : 'a t -> 'a ptr
  val from_ptr : 'a ptr -> int -> 'a t
  val make : 'a typ -> ?initial:'a -> int -> 'a t
end
```

The array functions are similar to the standard library `Array` module except that they represent flat C arrays instead of OCaml ones.

The conversion between arrays and lists still requires copying the values, and can be expensive for large data structures. Notice that you can also convert an array into a `ptr` pointer to the head of buffer, which can be useful if you need to pass the pointer and size arguments separately to a C function.

Unions in C are named structures that can be mapped onto the same underlying memory. They are also fully supported in Ctypes, but we won't go into more detail here.

### Pointer operators for dereferencing and arithmetic

Ctypes defines a number of operators that let you manipulate pointers and arrays just as you would in C. The Ctypes equivalents do have the benefit of being more strongly typed, of course.

| Operator | Purpose |
|---|---|
| `!@ p` | Dereference the pointer p. |
| `p <-@ v` | Write the value v to the address p. |
| `p +@ n` | If p points to an array element, then compute the address of the nth next element. |
| `p -@ n` | If p points to an array element, then compute the address of the nth previous element. |

There are also other useful non-operator functions available (see the Ctypes documentation), for example for pointer differencing and comparison.

## Passing functions to C

It's also straightforward to pass OCaml function values to C. The C standard library function `qsort` has the following signature that requires a function pointer to use.

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

C programmers often use `typedef` to make type definitions involving function pointers easier to read. Using a typedef, the type of `qsort` looks a little more palatable.

```
typedef int(compare_t)(const void *, const void *);
void qsort(void *base, size_t nmemb, size_t size, compare_t *);
```

This also happens to be a close mapping to the corresponding Ctypes definition. Since type descriptions are regular values, we can just use `let` in place of `typedef` and end up with working OCaml bindings to `qsort`.

```
let compare_t = ptr void @-> ptr void @-> returning int

let qsort = foreign "qsort"
   (ptr void @-> size_t @-> size_t @-> funptr compare_t @-> returning void)
```

We only use `compare_t` once (in the `qsort` definition), so you can choose to inline it in the OCaml code if you prefer. The resulting `qsort` value is a higher-order function, as shown by its type.

```
val qsort: void ptr -> size_t -> size_t ->
           (void ptr -> void ptr -> int) -> unit
```

As before, let's define a wrapper function to make `qsort` easier to use. The second and third arguments to `qsort` specify the length (number of elements) of the array and the element size.

Arrays created using Ctypes have a richer runtime structure than C arrays, so we don't need to pass size information around. Furthermore, we can use OCaml polymorphism in place of the unsafe `void ptr` type.

## Example: a command-line quicksort

Below is a command-line tool that uses the `qsort` binding to sort all of the integers supplied on the standard input.

```
(* qsort.ml: quicksort integers from stdin *)
open Core.Std
open Ctypes
open PosixTypes
open Foreign

let compare_t = ptr void @-> ptr void @-> returning int

let qsort = foreign "qsort"
   (ptr void @-> size_t @-> size_t @-> funptr compare_t @->
    returning void)

let qsort' cmp arr =
  let open Unsigned.Size_t in
  let ty = Array.element_type arr in
```

```
  let len = of_int (Array.length arr) in
  let elsize = of_int (sizeof ty) in
  let start = to_voidp (Array.start arr) in
  let compare l r = cmp (!@ (from_voidp ty l)) (!@ (from_voidp ty r)) in
  qsort start len elsize compare;
  arr

let sort_stdin () =
  In_channel.input_lines stdin
  |> List.map ~f:int_of_string
  |> Array.of_list int
  |> qsort' Int.compare
  |> Array.to_list
  |> List.iter ~f:(fun a -> printf "%d\n" a)

let () =
  Command.basic ~summary:"Sort integers on standard input"
    Command.Spec.empty sort_stdin
  |> Command.run
```

Compile it in the usual way with ocamlfind, but also examine the inferred interface of the module.

```
$ ocamlfind ocamlopt -package core -package ctypes.foreign -thread -linkpkg \
    -o qsort qsort.ml
$ ./qsort
5
3
2
1
# press <Control-D> to end the standard input
1
2
3
5
$ ocamlfind ocamlopt -i -package core -package ctypes.foreign -thread qsort.ml
val compare_t : (unit Ctypes.ptr -> unit Ctypes.ptr -> int) Ctypes.fn
val qsort :
  unit Ctypes.ptr ->
  PosixTypes.size_t ->
  PosixTypes.size_t -> (unit Ctypes.ptr -> unit Ctypes.ptr -> int) -> unit
val qsort' : ('a -> 'a -> int) -> 'a Ctypes.array -> 'a Ctypes.array
val sort_stdin : unit -> unit
```

The `qsort'` wrapper function has a much more canonical OCaml interface than the raw binding. It accepts a comparator function and a Ctypes array, and returns the same Ctypes array. It's not strictly required that it returns the array since it modifies it in-place, but it makes it easier to chain the function using the `|>` operator (as `sort_stdin` does in the example).

Using `qsort'` to sort arrays is straightforward. Our example code reads the standard input as a list, converts it to a C array, passes it through qsort, and outputs the result to the standard output. Again, remember to not confuse the `Ctypes.Array` module with

the `Core.Std.Array` module: the former is in scope since we opened `Ctypes` at the start of the file.

**Lifetime of allocated Ctypes**

Values allocated via Ctypes (i.e. using `allocate`, `Array.make` and so on) will not be garbage-collected as long as they are reachable from OCaml values. The system memory they occupy is freed when they do become unreachable, via a finalizer function registered with the GC.

The definition of reachability for Ctypes values is a little different from conventional OCaml values though. The allocation functions return an OCaml-managed pointer to the value, and as long as some derivative pointer is still reachable by the GC, the value won't be collected.

"Derivative" means a pointer that's computed from the original pointer via arithmetic, so a reachable reference to an array element or a structure field protects the whole object from collection.

A corollary of the above rule is that pointers written into the C heap don't have any effect on reachability. For example, if you have a C-managed array of pointers to structs then you'll need some additional way of keeping the structs around to protect them from collection. You could achieve this via a global array of values on the OCaml side that would keep them live until they're no longer needed.

## Learning more about C bindings

The Ctypes distribution (*http://github.com/ocamllabs/ocaml-types*) contains a number of larger-scale examples, including:

- bindings to the POSIX `fts` API which demonstrates C callbacks more comprehensively.
- a more complete Ncurses binding than the example we opened the chapter with.
- a comprehensive test suite that covers the complete library, and can provide useful snippets for your own bindings.

This chapter hasn't really needed you to understand the innards of OCaml at all. Ctypes does its best to make function bindings easy, but the rest of this part will also fill you in about how interactions with OCaml memory layout and the garbage collector work.

**Production note**

This chapter contains significant contributions from Jeremy Yallop.

2013-07-04
10:28:31

# CHAPTER 20
# Memory Representation of Values

The FFI interface we described in Chapter 19 hides the precise details of how values are exchanged across C libraries and the OCaml runtime. There is a simple reason for this: using this interface directly is a delicate operation that requires understanding a few different moving parts before you can get it right. You first need to know the mapping between OCaml types and their runtime memory representation. You also need to ensure that your code is interfacing correctly with OCaml runtime's memory management.

However, knowledge of the OCaml internals is useful beyond just writing foreign function interfaces. As you build and maintain more complex OCaml applications, you'll need to interface with various external system tools that operate on compiled OCaml binaries. For example, profiling tools report output based on the runtime memory layout and debuggers execute binaries without any knowledge of the static OCaml types. To use these tools effectively, you'll need to do some translation between the OCaml and C worlds.

Luckily, the OCaml toolchain is very predictable. The compiler minimizes the amount of optimization magic that it performs, and relies instead on its straightforward execution model for good performance. With some experience, you can know rather precisely where a block of performance-critical OCaml code is spending its time.

**Why do OCaml types disappear at runtime?**

The OCaml compiler runs through several phases during the compilation process. The first phase is syntax checking, during which source files are parsed into Abstract Syntax Trees (ASTs). The next stage is a *type checking* pass over the AST. In a validly typed program, a function cannot be applied with an unexpected type. For example, the `print_endline` function must receive a single `string` argument, and an `int` will result in a type error.

Since OCaml verifies these properties at compile time, it doesn't need to keep track of as much information at runtime. Thus, later stages of the compiler can discard and simplify the type declarations to a much more minimal subset that's actually required to distinguish polymorphic values at runtime. This is a major performance win versus something like a Java or .NET method call, where the runtime must look up the concrete instance of the object and dispatch the method call. Those languages amortize some of the cost via "Just-in-Time" dynamic patching, but OCaml prefers runtime simplicity instead.

We'll explain this compilation pipeline in more detail in Chapter 22 and Chapter 23.

This chapter covers the precise mapping from OCaml types to runtime values and walks you through them via the toplevel. We'll cover how these values are managed by the runtime later on in Chapter 21.

## OCaml blocks and values

A running OCaml program uses blocks of memory (i.e. contiguous sequences of words in RAM) to represent values such as tuples, records, closures or arrays. An OCaml program implicitly allocates a block of memory when such a value is created.

```
# let x = { foo = 13; bar = 14 } ;;
```

An expression such as the record above requires a new block of memory with two words of available space. One word holds the `foo` field and the second word holds the `bar` field. The OCaml compiler translates such an expression into an explicit allocation for the block from OCaml's runtime system.

OCaml uses a uniform memory representation in which every OCaml variable is stored as a *value*. An OCaml value is a single memory word that is either an immediate integer or a pointer to some other memory. The OCaml runtime tracks all values so that it can free them when they are no longer needed. It thus needs to be able to distinguish between integer and pointer values, since it scans pointers to find further values but doesn't follow integers that don't point to anything meaningful beyond their immediate value.

2013-07-04
10:28:31

## Distinguishing integer and pointers at runtime

Wrapping primitives types (such as integers) inside another data structure that records extra metadata about the value is known as *boxing*. Values are boxed in order to make it easier for the garbage collector to do its job, but at the expense of an extra level of indirection to access the data within the boxed value.

OCaml values don't all have to be boxed at runtime. Instead, values use a single tag bit per word to distinguish integers and pointers at runtime. The value is an integer if the lowest bit of the block word is non-zero, and a pointer if the lowest bit of the block word is zero. Several OCaml types map onto this integer representation, including `bool`, `int`, the empty list, `unit`, and variants without constructors.

This representations means that integers are unboxed runtime values in OCaml so that they can be stored directly without having to allocate a wrapper block. They can be passed directly to other function calls in registers and are generally the cheapest and fastest values to use in OCaml.

A value is treated as a memory pointer if its lowest bit is zero. A pointer value can still be stored unmodified despite this, since pointers are guaranteed to be word-aligned (with the bottom bits always being zero).

The only problem that remains with this memory representation is distinguishing between pointers to OCaml values (which should be followed by the garbage collector) and pointers into the system heap to C values (which shouldn't be followed).

The mechanism for this is simple, since the runtime system keeps track of the heap blocks it has allocated for OCaml values. If the pointer is inside a heap chunk that is marked as being managed by the OCaml runtime, it is assumed to point to an OCaml value. If it points outside the OCaml runtime area, it is treated as an opaque C pointer to some other system resource.

**Some history about OCaml's word-aligned pointers**

The alert reader may be wondering how OCaml can guarantee that all of its pointers are word-aligned. In the old days when RISC chips such as Sparc, MIPS and Alpha were commonplace, unaligned memory accesses were forbidden by the instruction set architecture and would result in a CPU exception that terminated the program. Thus, all pointers were historically rounded off to the architecture word-size (usually 32- or 64-bits).

Modern CISC processors such as the Intel x86 do support unaligned memory accesses, but the chip still runs faster if accesses are word-aligned. OCaml therefore simply mandates that all pointers be word-aligned, which guarantees that the bottom few bits of any valid pointer will be zero. Setting the bottom bit to a non-zero value is a simple way to mark an integer, at the cost of losing that single bit of precision.

An even more alert reader will be wondering about the performance implications are for integer arithmetic using this tagged representation. Since the bottom bit is set, any operation on the integer has to shift the bottom bit right to recover the "native" value. The native code OCaml compiler generates efficient x86 assembly code in this case. It takes advantage of modern processor instructions to hide the extra shifts as much as possible. Addition and substraction are a single instruction, and multiplication is only a few more.

## Blocks and values

An OCaml *block* is the basic unit of allocation on the heap. A block consists of a one-word header (either 32- or 64-bits depending on the CPU architecture) followed by variable-length data that is either opaque bytes or an array of *fields*. The header has a multi-purpose tag byte that defines whether to interpret the subsequent data as opaque bytes or OCaml fields.

The garbage collector never inspects opaque bytes. If the tag indicates an array of OCaml fields are present, their contents are all treated as more valid OCaml values. The garbage collector always inspects fields and follows them as part of the collection process described earlier.

```
+-----------------------+---------+----------+----------+----------+----
| size of block in words |  color  | tag byte | value[0] | value[1] | ...
+-----------------------+---------+----------+----------+----------+----
 <-either 22 or 54 bits-> <-2 bit-> <--8 bit-->
```

The `size` field records the length of the block in memory words. This is 22 bits on 32-bit platforms, which is the reason why OCaml strings are limited to 16MB on that architecture. If you need bigger strings, either switch to a 64-bit host, or use the `Bigarray` module.

The 2-bit `color` field is used by the garbage collector to keep track of its state during mark-and-sweep collection. We'll come back to this field in Chapter 21. This tag isn't exposed to OCaml source code in any case.

A block's tag byte is multi-purpose, and indicates whether the data array represents opaque bytes or fields. If a block's tag is greater than or equal to `No_scan_tag` (251), then the block's data are all opaque bytes, and are not scanned by the collector. The most common such block is the `string` type, which we describe in more detail later in this chapter.

The exact representation of values inside a block depends on their static OCaml type. All OCaml types are distilled down into `values`, and summarised in the table below.

| OCaml Value | Representation |
|---|---|
| `int` or `char` | directly as a value, shifted left by 1 bit, with the least significant bit set to 1 |
| `unit`, `[]`, `false` | as OCaml `int` 0. |
| `true` | as OCaml `int` 1. |
| `Foo \| Bar` | as ascending OCaml `int`s, starting from 0. |
| `Foo \| Bar of int` | variants with parameters are boxed, while variants with no parameters are unboxed. |
| polymorphic variants | variable space usage depending on the number of parameters. |
| floating point number | as a block with a single field containing the double-precision float. |
| string | word-aligned byte arrays that are also directly compatible with C strings. |
| `[1; 2; 3]` | as `1::2::3::[]` where `[]` is an int, and `h::t` a block with tag 0 and two parameters. |
| tuples, records and arrays | an array of values. Arrays can be variable size, but structs and tuples are fixed size. |
| records or arrays, all float | special tag for unboxed arrays of floats, or records that only have `float` fields. |

## Integers, characters and other basic types

Many basic types are efficiently stored as unboxed integers at runtime. The native `int` type is the most obvious, although it drops a single bit of precision due to the tag bit. Other atomic types such as `unit` and empty list `[]` value are stored as constant integers. Boolean values have a value of `0` and `1` for `true` and `false` respectively.

These basic types such as empty lists and `unit` are very efficient to use since integers are never allocated on the heap. They can be passed directly in registers and not appear on the stack if you don't have too many parameters to your functions. Modern architectures such as `x86_64` have a lot of spare registers to further improve the efficiency of using unboxed integers.

# Tuples, records and arrays

```
+---------+----------+----------- - - - -
```

```
| header  | value[0] | value[1] | ....
+---------+----------+----------+- - - -
```

Tuples, records and arrays are all represented identically at runtime as a block with tag `0`. Tuples and records have constant sizes determined at compile-time, whereas arrays can be of variable length. While arrays are restricted to containing a single type of element in the OCaml type system, this is not required by the memory representation.

You can check the difference between a block and a direct integer yourself using the `Obj` module, which exposes the internal representation of values to OCaml code.

```
# Obj.is_block (Obj.repr (1,2,3)) ;;
- : bool = true
# Obj.is_block (Obj.repr 1) ;;
- : bool = false
```

The `Obj.repr` function retrieves the runtime representation of any OCaml value. `Obj.is_block` checks the bottom bit to determine if the value is a block header or an unboxed integer.

## Floating point numbers and arrays

Floating point numbers in OCaml are always stored as full double-precision values. Individual floating point values are stored as a block with a single field that contains the number. This block has the `Double_tag` set which signals to the collector that the floating point value is not to be scanned.

```
# Obj.tag (Obj.repr 1.0) ;;
- : int = 253
# Obj.double_tag ;;
- : int = 253
```

Since each floating-point value is boxed in a separate memory block, it can be inefficient to handle large arrays of floats in comparison to unboxed integers. OCaml therefore special-cases records or arrays that contain *only* `float` types. These are stored in a block that contains the floats packed directly in the data section, with the `Double_array_tag` set to signal to the collector that the contents are not OCaml values.

```
+---------+----------+---------- - - - -
| header  | float[0] | float[1] | ....
+---------+----------+----------+- - - - -
```

First, let's check that float arrays do in fact have a different tag number from normal floating point values.

```
# Obj.double_tag;;
- : int = 253
# Obj.double_array_tag;;
- : int = 254
```

This tells us that float arrays have a tag value of 254. Now let's test some sample values using the `Obj.tag` function to check that the allocated block has the expected runtime tag, and also use `Obj.double_field` to retrieve a float from within the block.

```
# open Obj ;;
# tag (repr [| 1.0; 2.0; 3.0 |]) ;;
- : int = 254
# tag (repr (1.0, 2.0, 3.0) ) ;;
- : int = 0
# double_field (repr [| 1.1; 2.2; 3.3 |] ) 1 ;;
- : float = 2.2
# double_field (repr 1.234) 0;;
- : float = 1.234
```

The first thing we tested above was that a float array has the correct unboxed float array tag value (254). However, the next line tests a tuple of floating point values instead, which are *not* optimized in the same way and have the normal tuple tag value (0).

Only records and arrays can have the float array optimization, and for records every single field must be a float.

## Variants and lists

Basic variant types with no extra parameters for any of their branches are simply stored as an OCaml integer, starting with `0` for the first option and in ascending order.

```
# open Obj ;;
# type t = Apple | Orange | Pear ;;
type t = Apple | Orange | Pear
# ((magic (repr Apple)) : int) ;;
- : int = 0
# ((magic (repr Pear)) : int) ;;
- : int = 2
# is_block (repr Apple) ;;
- : bool = false
```

`Obj.magic` unsafely forces a type cast between any two OCaml types; in this example the `int` type hint retrieves the runtime integer value. The `Obj.is_block` confirms that the value isn't a more complex block, but just an OCaml `int`.

Variants that have parameters arguments are a little more complex. They are stored as blocks, with the value *tags* ascending from 0 (counting from leftmost variants with parameters). The parameters are stored as words in the block.

```
# type t = Apple | Orange of int | Pear of string | Kiwi ;;
type t = Apple | Orange of int | Pear of string | Kiwi
# is_block (repr (Orange 1234)) ;;
- : bool = true
# tag (repr (Orange 1234)) ;;
- : int = 0
```

```
# tag (repr (Pear "xyz")) ;;
- : int = 1
# (magic (field (repr (Orange 1234)) 0) : int) ;;
- : int = 1234
(magic (field (repr (Pear "xyz")) 0) : string) ;;
- : string = "xyz"
```

In the above example, the `Apple` and `Kiwi` values are still stored as normal OCaml integers with values `0` and `1` respectively. The `Orange` and `Pear` values both have parameters, and are stored as blocks whose tags ascend from `0` (and so `Pear` has a tag of `1`, as the use of `Obj.tag` verifies). Finally, the parameters are fields which contain OCaml values within the block, and `Obj.field` can be used to retrieve them.

Lists are stored with a representation that is exactly the same as if the list was written as a variant type with `Head` and `Cons`. The empty list `[]` is an integer `0`, and subsequent blocks have tag `0` and two parameters: a block with the current value, and a pointer to the rest of the list.

**`Obj` module considered harmful**

The `Obj` module is an undocumented module that exposes the internals of the OCaml compiler and runtime. It is very useful for examining and understanding how your code will behave at runtime, but should *never* be used for production code unless you understand the implications. The module bypasses the OCaml type system, making memory corruption and segmentation faults possible.

Some theorem provers such as Coq do output code which uses `Obj` internally, but the external module signatures never expose it. Unless you too have a machine proof of correctness to accompany your use of `Obj`, stay away from it except for debugging!

Due to this encoding, there is a limit around 240 variants with parameters that applies to each type definition, but the only limit on the number of variants without parameters is the size of the native integer (either 31- or 63-bits). This limit arises because of the size of the tag byte, and that some of the high numbered tags are reserved.

## Polymorphic variants

Polymorphic variants are more flexible than normal variants when writing code, but are slightly less efficient at runtime. This is because there isn't as much static compile-time information available to optimise their memory layout.

A polymorphic variant without any parameters is stored as an unboxed integer and so only takes up one word of memory, just like a normal variant. This integer value is determined by applying a hash function to the *name* of the variant. The hash function isn't exposed directly by the compiler, but the `type_conv` library from Core provides an alternative implementation.

```
# #require "type_conv" ;;
# Pa_type_conv.hash_variant "Foo" ;;
- : int = 3505894
# (Obj.magic (Obj.repr `Foo) : int) ;;
- : int = 3505894
```

The hash function is designed to give the same results on 32-bit and 64-bit architectures, so the memory representation is stable across different CPUs and host types.

Polymorphic variants use more memory space than normal variants when parameters are included in the datatype constructors. Normal variants use the tag byte to encode the variant value and save the fields for the contents, but this single byte is insufficient to encode the hashed value for polymorphic variants. They must allocate a new block (with tag `0`) and store the value in there instead. Polymorphic variants with constructors thus use one word of memory more than normal variant constructors.

Another inefficiency over normal variants is when a polymorphic variant constructor has more than one parameter. Normal variants hold parameters as a single flat block with multiple fields for each entry, but polymorphic variants must adopt a more flexible uniform memory representation since they may be reused in a different context across compilation units. They allocate a tuple block for the parameters that is pointed to from the argument field of the variant. There are thus three additional words for such variants, along with an extra memory indirection due to the tuple.

The extra space usage is generally not significant in a typical application, and polymorphic variants offer a great deal more flexibility than normal variants. However, if you're writing code that demands high performance or must run within tight memory bounds, the runtime layout is at least very predictable. The OCaml compiler never switches memory representation due to optimization passes. This lets you predict the precise runtime layout by referring to these guidelines and your source code.

## String values

Strings are standard OCaml blocks with the header size defining the size of the string in machine words. The `String_tag` (252) is higher than the `No_scan_tag`, indicating that the contents of the block are opaque to the collector. The block contents are the contents of the string, with padding bytes to align the block on a word boundary.

```
+---------------+----------------+--------+-----------+
| header        | 'a' 'b' 'c' 'd' 'e' 'f' | '\0' '\1' |
+---------------+----------------+--------+-----------+
                L data                    L padding
```

On a 32-bit machine, the padding is calculated based on the modulo of the string length and word size to ensure the result is word-aligned. A 64-bit machine extends the potential padding up to 7 bytes instead of 3.

| String length mod 4 | Padding |
|---|---|
| 0 | 00 00 00 03 |
| 1 | 00 00 02 |
| 2 | 00 01 |
| 3 | 00 |

This string representation is a clever way to ensure that the contents are always zero-terminated by the padding word, and still compute its length efficiently without scanning the whole string. The following formula is used:

```
number_of_words_in_block * sizeof(word) - last_byte_of_block - 1
```

The guaranteed `NULL`-termination comes in handy when passing a string to C, but is not relied upon to compute the length from OCaml code. OCaml strings can thus contain `NULL` bytes at any point within the string.

Care should be taken that any C library functions that receive these buffers can also cope with arbitrary bytes within the buffer contents and are not expecting C strings. For instance, the C `memcopy` or `memmove` standard library functions can operate on arbitrary data, but `strlen` or `strcpy` both require a `NULL` terminated buffer and has no mechanism for encoding a `NULL` value within its contents.

## Custom heap blocks

OCaml supports *custom* heap blocks via a `Custom_tag` that let the runtime perform user-defined operations over OCaml values. A custom block lives in the OCaml heap like an ordinary block and can be of whatever size the user desires. The `Custom_tag` (255) is higher than `No_scan_tag` and so isn't scanned by the garbage collector.

The first word of the data within the custom block is a C pointer to a `struct` of custom operations. The custom block cannot have pointers to OCaml blocks and is opaque to the garbage collector.

```
struct custom_operations {
  char *identifier;
  void (*finalize)(value v);
  int (*compare)(value v1, value v2);
  intnat (*hash)(value v);
  void (*serialize)(value v,
                    /*out*/ uintnat * wsize_32 /*size in bytes*/,
                    /*out*/ uintnat * wsize_64 /*size in bytes*/);
  uintnat (*deserialize)(void * dst);
  int (*compare_ext)(value v1, value v2);
};
```

The custom operations specify how the runtime should perform polymorphic comparison, hashing and binary marshalling. They also optionally contain a *finalizer* that the runtime calls just before the block is garbage collected. This finalizer has nothing to do with ordinary OCaml finalizers (as created by `Gc.finalise` and explained in Chapter 21). They are instead used to call C cleanup functions such as `free`.

## Managing external memory with Bigarray

A common use of custom blocks is to manage external system memory directly from within OCaml. The Bigarray interface was originally intended to exchange data with Fortran code, and maps a block of system memory as a multi-dimensional array that can be accessed from OCaml. Bigarray operations work directly on the external memory without requiring it to be copied into the OCaml heap (which is a potentially expensive operation for large arrays).

Bigarray sees a lot of use beyond just scientific computing, and several Core libraries use it for general-purpose I/O:

- The `Iobuf` module maps I/O buffers as a 1-dimensional array of bytes. It provides a sliding window interface that lets consumer processes read from the buffer while it's being filled by producers. This lets OCaml use I/O buffers that have been externally allocated by the operating system without any extra data copying.
- The `Bigstring` module provides a `String`-like interface that uses `Bigarray` internally. The `Bigbuffer` collects these into extensible string buffers that can operate entirely on external system memory.

The Lacaml (*https://bitbucket.org/mmottl/lacaml*) library isn't part of Core, but provides the recommended interfaces to the widely used BLAS and LAPACK mathematical Fortran libraries. These allow developers to write high-performance numerical code for applications that require linear algebra. It supports large vectors and matrices, but with static typing safety of OCaml to make it easier to write safe algorithms.

2013-07-04
10:28:31

2013-07-04
10:28:31

CHAPTER 21

# Understanding the Garbage Collector

We've described the runtime format of individual OCaml variables earlier in Chapter 20. When you execute your program, OCaml manages the lifecycle of these variables by regularly scanning allocated values and freeing them when they're no longer needed. This in turn means that your applications don't need to manually implement memory management and greatly reduces the likelihood of memory leaks creeping into your code.

The OCaml runtime is a C library that provides routines that can be called from running OCaml programs. The runtime manages a *heap*, which is a collection of memory regions that it obtains from the operating system. The runtime uses this memory to hold *heap blocks* that it fills up with OCaml values in response to allocation requests by the OCaml program.

## Mark and sweep garbage collection

When there isn't enough memory available to satisfy an allocation request from the pool of allocated heap blocks, the runtime system invokes the *garbage collector* (or GC). An OCaml program can't explicitly free a value when it is done with it. Instead, the GC regularly determines which values are *live* and which values are *dead*, i.e. no longer in use. Dead values are collected and their memory made available for reuse by the application.

The garbage collector doesn't keep constant track of values as they are allocated and used. Instead, it regularly scans them by starting from a set of *root* values that the application always has access to (such as the stack). The GC maintains a directed graph in which heap blocks are nodes, and there is an edge from heap block `b1` to heap block `b2` if some field of `b1` points to `b2`. All blocks reachable from the roots by following edges in the graph must be retained, and unreachable blocks can be reused by the application.

The algorithm used by OCaml to perform this heap traversal is commonly known as *mark and sweep* garbage collection, and we'll explain it further now.

## Generational garbage collection

The usual OCaml programming style involves allocating many small variables that are used for a short period of time and then never accessed again. OCaml takes advantage of this fact to improve performance by using a *generational* garbage collector.

A generational GC maintains separate memory regions to hold blocks based on how long the blocks have been live. OCaml's heap is split in two such regions:

- a small fixed-size *minor heap* where most blocks are initially allocated.
- a larger variable-sized *major heap* for blocks that have been live longer.

A typical functional programming style means that young blocks tend to die young and old blocks tend to stay around for longer than young ones. This is often referred to as the *generational hypothesis*.

OCaml uses different memory layouts and garbage collection algorithms for the major and minor heaps to account for this generational difference. We'll explain how they differ in more detail next.

### The `Gc` module and `OCAMLRUNPARAM`

OCaml provides several mechanisms to query and alter the behaviour of the runtime system. The `Gc` module provides this functionality from within OCaml code, and we'll frequently refer to it in the rest of the chapter. As with several other standard library modules, Core alters the `Gc` interface from the standard OCaml library. We'll assume that you've opened `Core.Std` in our explanations.

You can also control the behaviour of OCaml programs by setting the `OCAMLRUNPARAM` environment variable before launching your application. This lets you set garbage collector parameters without recompiling, for example to benchmark the effects of different settings. The format of `OCAMLRUNPARAM` is documented in the OCaml manual (*http://caml.inria.fr/pub/docs/manual-ocaml/manual024.html*).

## The fast minor heap

The minor heap is where most of your short-lived values are held. It consists of one contiguous chunk of virtual memory containing a sequence of OCaml blocks. If there is space, allocating a new block is a fast constant-time operation that requires just a couple of CPU instructions.

To garbage collect the minor heap, OCaml uses *copying collection* to move all live blocks in the minor heap to the major heap. This takes work proportional to the number of live blocks in the minor heap, which is typically small according to the generational hypothesis.

## Allocating on the minor heap

The minor heap is a contiguous chunk of virtual memory that is usually a few megabytes in size so that it can be scanned quickly.

```
                <---- size ---->
base --- start ---------------- end
         limit      ptr <------
                         blocks
```

The runtime stores the minor heap in two pointers (`caml_young_start` and `caml_young_end`, but we will drop the `caml_young` prefix for brevity) that delimit the start and end of the heap region. The `base` is the memory address returned by the system `malloc`, and `start` is aligned against the next nearest word boundary from `base` to make it easier to store OCaml values.

In a fresh minor heap, the `limit` equals the `start` and the current `ptr` will equal the `end`. `ptr` decreases as blocks are allocated until it reaches `limit`, at which point a minor garbage collection is triggered.

Allocating a block in the minor heap just requires `ptr` to be decremented by the size of the block (including the header) and checking that it's not less than `limit`. If there isn't enough space left for the block without decrementing past the `limit`, a minor garbage collection is triggered. This is a very fast check (with no branching) on most CPU architectures.

You may wonder why `limit` is required at all, since it always seems to equal `start`. It's because the easiest way for the runtime to schedule a minor heap collection is by setting `limit` to equal `end`. The next allocation will never have enough space after this is done and will always trigger a garbage collection.

#### Setting the size of the minor heap

The minor heap size defaults to 8MB on 64-bit platforms, unless overridden by the `s=<words>` argument to `OCAMLRUNPARAM`. You can change it after the program has started by calling the `Gc.set` function.

```
# open Gc;;
# let c = Gc.get ();;
val c : Gc.control =
  {minor_heap_size = 262144; major_heap_increment = 126976;
   space_overhead = 80; verbose = 0; max_overhead = 500;
   stack_limit = 1048576; allocation_policy = 0}
# Gc.tune ~minor_heap_size:(262144 * 2) () ;;
- : unit = ()
```

Changing the GC size dynamically will trigger an immediate minor heap collection. Note that Core increases the default minor heap size from the standard OCaml installation quite significantly, and you'll want to reduce this if running in very memory-constrained environments.

2013-07-04
10:28:31

## The long-lived major heap

The major heap is where the bulk of the longer-lived and larger values in your program are stored. It consists of any number of non-contiguous chunks of virtual memory, each containing live blocks interspersed with regions of free memory. The runtime system maintains a free-list data structure that indexes all the free memory that it has allocated, and uses it to satisfy allocation requests for OCaml blocks.

The major heap is typically much larger than the minor heap and can scale to gigabytes in size. It is cleaned via a mark-and-sweep garbage collection algorithm that operates in several phases.

- The *mark* phase scans the block graph and marks all live blocks by setting a bit in the tag of the block header (known as the *color* tag).
- The *sweep* phase sequentially scans the heap chunks and identifies dead blocks that weren't marked earlier.
- The *compact* phase relocates live blocks into a freshly allocated heap to eliminate gaps in the free list. This prevents the fragmentation of heap blocks in long-running programs, and normally occurs much less frequently than the mark and sweep phases.

A major garbage collection must *stop the world* (that is, halt the application) to ensure that blocks can be moved around without this being observed by the live application. The mark-and-sweep phases run incrementally over slices of the heap to avoid pausing the application for long periods of time. Only the compaction phase touches all the memory in one go, and is a relatively rare operation.

### Allocating on the major heap

The major heap consists of a singly-linked list of contiguous memory chunks sorted in increasing order of virtual address. Each chunk is a single memory region allocated via *malloc(3)* and consists of a header and data area which contains OCaml heap chunks. A heap chunk header contains:

- the *malloc*'ed virtual address of the memory region containing the chunk.
- the size in bytes of the data area.
- an allocation size in bytes used during heap compaction to merge small blocks to defragment the heap.
- a link to the next heap chunk in the list.

Each chunk's data area starts on a page boundary and its size is a multiple of the page size (4KB). It contains a contiguous sequence of heap blocks which can be as small as one or two 4KB pages, but are usually allocated in 1MB chunks (or 512KB on 32-bit architectures).

#### Controlling major heap growth

The `Gc` module uses the `major_heap_increment` value to control the major heap growth. This defines the number of words to add to the major heap per expansion, and is the only memory allocation operation that the operating system observes from the OCaml runtime after initial startup (since the minor is fixed in size).

If you anticipate allocating some large OCaml values, then setting the heap increment to a larger value will let the operating system return a contiguous block of memory. This is preferable to lots of smaller heap chunks that may be spread across different regions of virtual memory, and require more housekeeping in the OCaml runtime to keep track of them.

```
# open Core.Std;;
# Gc.tune ~major_heap_increment:(1000448 * 4) ();;
```

Allocating an OCaml value on the major heap first checks the free list of blocks for a suitable region to place it. If there isn't enough room on the free list, the runtime expands the major heap by allocating a fresh heap chunk that will be large enough. That chunk is then added to the free list and the free list is checked again (and this time will definitely succeed).

Remember that most allocations to the major heap will go via the minor heap, and only be promoted if they are still used by the program after a minor collection. The one exception to this is for values larger than 256 words (that is, 2kB on 64-bit platforms). These will be allocated directly on the major heap since an allocation on the minor heap would likely trigger an immediate collection and copy it to the major heap anyway.

## Memory allocation strategies

The major heap does its best to manage memory allocation as efficiently as possible, and relies on heap compaction to ensure that memory stays contiguous and unfragmented. The default allocation policy normally works fine for most applications, but it's worth bearing in mind that there are other options too.

The free list of blocks is always checked first when allocating a new block in the major heap. The default free list search is called *next-fit allocation*, with an alternative *first-fit* algorithm also available.

### Next-fit allocation

Next-fit allocation keeps a pointer to the block in the free list that was most recently used to satisfy a request. When a new request comes in, the allocator searches from the next block until the end of the free list, and then from the beginning of the free list up to that block.

Next-fit allocation is the default allocation strategy. It's quite a cheap allocation mechanism since the same heap chunk can be reused across allocation requests until it runs out. This in turn means that there is good memory locality to use CPU caches better.

#### First-fit allocation

If your programs allocates values of many varied sizes, you may sometimes find that your free list becomes fragmented. In this situation, the GC is forced to perform an expensive compaction despite there being free chunks, since none of the chunks alone are big enough to satisfy the request.

First-fit allocation focusses on reducing memory fragmentation (and hence the number of compactions), but at the expense of slower memory allocation. Every allocation scans the free list from the beginning for a suitable free chunk, instead of reusing the most recent heap chunk as the next-fit allocator does.

For some workloads that need more real-time behaviour under load, the reduction in the frequency in heap compaction will outweigh the extra allocation cost.

#### Controlling the heap allocation policy

You can set the heap allocation policy via the `Gc.allocation_policy` field. A value of `0` (the default) sets it to next-fit, and `1` to the first-fit allocator.

The same behaviour can be controlled at runtime by setting `a=0` or `a=1` in `OCAMLRUNPARAM`.

## Marking and scanning the heap

The marking process can take a long time to run over the complete major heap, and has to pause the main application while it's active. It therefore runs incrementally by marking the heap in *slices*. Each value in the heap has a 2-bit *color* field in its header that is used to store information about whether the value has been marked, so that the GC can resume easily between slices.

| Tag Color | Block Status |
|---|---|
| blue | on the free list and not currently in use |
| white (during marking) | not reached yet, but possibly reachable |
| white (during sweeping) | unreachable and can be freed |
| gray | reachable, but its fields have not been scanned |
| black | reachable, and its fields have been scanned |

The color tags in the value headers store most of the state of the marking process, allowing it to be paused and resumed later. The GC and application alternate between

marking a slice of the major heap and actually getting on with executing the program logic. The OCaml runtime calculates a sensible value for the size of each major heap slice based on the rate of allocation and available memory (see below).

The marking process starts with a set of *root* values that are always live (such as the application stack). All values on the heap are initially marked as white values that are possibly reachable, but haven't been scanned yet. It recursively follows all the fields in the roots via a depth-first search, and pushes newly encountered white blocks onto an intermediate stack of *gray values* while it follows their fields. When a gray value's fields have all been followed it is popped off the stack and colored black.

This process is repeated until the gray value stack is empty and there are no further values to mark. There's one important edge case in this process, though. The gray value stack can only grow to a certain size, after which the GC can no longer recurse into intermediate values since it has nowhere to store them while it follows their fields. If this happens, the heap is marked as *impure* and a more expensive check is initiated once the existing gray values have been processed.

To mark an impure heap, the GC first marks it as pure and walks through the entire heap block-by-block in increasing order of memory address. If it finds a gray block, it adds it to the gray list and recursively marks it using the usual strategy for a pure heap. Once the scan of the complete heap is finished, the mark phase checks again whether the heap has again become impure, and repeats the scan until if it is. These full-heap scans will continue until a successful scan completes without overflowing the gray list.

#### Controlling major heap collections

You can trigger a single slice of the major GC via the `major_slice` call. This performs a minor collection first, and then a single slice. The size of the slice is normally automatically computed by the GC to an appropriate value, and returns this value so that you can modify it in future calls if necessary.

```
# open Core.Std;;
# Gc.major_slice 0 ;;
- : int = 232340
# Gc.full_major ();;
- : unit = ()
```

The `space_overhead` setting controls how aggressive the GC is about setting the slice size to a large size. This represents the proportion of memory used for live data that will be "wasted" because the GC doesn't immediately collect unreachable blocks. Core defaults this to `100` to reflect a typical system that isn't overly memory-constrained. Set this even higher if you have lots of memory, or lower to cause the GC to work harder and collect blocks faster at the expense of using more CPU time.

## Heap Compaction

After a certain number of major GC cycles have completed, the heap may begin to be fragmented due to values being deallocated out of order from how they were allocated. This makes it harder for the GC to find a contiguous block of memory for fresh allocations, which in turn would require the heap to be grown unnecessarily.

The heap compaction cycle avoids this by relocating all the values in the major heap into a fresh heap that places them all contiguously in memory again. A naive implementation of the algorithm would require extra memory to store the new heap, but OCaml performs the compaction in-place within the existing heap.

#### Controlling frequency of compactions

The `max_overhead` setting in the `Gc` module defines the connection between free memory and allocated memory after which compaction is activated.

A value of `0` triggers a compaction after every major garbage collection cycle, whereas the maximum value of `1000000` disables heap compaction completely. The default settings should be fine unless you have unusual allocation patterns that are causing a higher-than-usual rate of compactions.

```
# open Core.Std;;
# Gc.tune ~max_overhead:0 ();;
- : unit = ()
```

## Inter-generational pointers

One complexity of generational collection arises from the fact that minor heap sweeps are much more frequent than major heap collections. In order to know which blocks in the minor heap are live, the collector must track which minor-heap blocks are directly pointed to by major-heap blocks. Without this information, each minor collection would also require scanning the much larger major heap.

OCaml maintains a set of such *inter-generational pointers* to avoid this dependency between a major and minor heap collection. The compiler introduces a write barrier to update this so-called *remembered set* whenever a major-heap block is modified to point at a minor-heap block.

#### The mutable write barrier

The write barrier can have profound implications for the structure of your code. It's one of the reasons why using immutable data structures and allocating a fresh copy with changes can sometimes be faster than mutating a record in-place.

The OCaml compiler keeps track of any mutable types and adds a call to the runtime `caml_modify` function before making the change. This checks the location of target write and the value it's being changed to, and ensures that the remembered set is consistent. Although the write barrier is reasonably efficient, it can sometimes be slower than simply allocating a fresh value on the fast minor heap and doing some extra minor collections.

Let's see this for ourselves with a simple test program. You'll need to install the Core benchmarking suite via `opam install core_bench` before you compile this code.

```
(* barrier_bench.ml: benchmark mutable vs immutable writes *)
open Core.Std
open Core_bench.Std

type t1 = { mutable iters1: int; mutable count1: float }
type t2 = { iters2: int; count2: float }

let rec test_mutable t1 =
  match t1.iters1 with
  |0 -> ()
  |n ->
    t1.iters1 <- t1.iters1 - 1;
    t1.count1 <- t1.count1 +. 1.0;
    test_mutable t1

let rec test_immutable t2 =
  match t2.iters2 with
  |0 -> ()
  |n ->
    let iters2 = n - 1 in
    let count2 = t2.count2 +. 1.0 in
    test_immutable { iters2; count2 }

let () =
  let iters = 1000000 in
  let tests = [
    Bench.Test.create ~name:"mutable"
      (fun () -> test_mutable { iters1=iters; count1=0.0 });
    Bench.Test.create ~name:"immutable"
      (fun () -> test_immutable { iters2=iters; count2=0.0 })
  ] in
  Bench.make_command tests |> Command.run
```

This program defines a type `t1` that is mutable and `t2` that is immutable. The benchmark loop iterates over both fields and increments a counter. Compile and execute this with some extra options to show the amount of garbage collection occurring.

```
$ ocamlbuild -use-ocamlfind -package core -package core_bench -tag thread barrier_bench.native
$ ./barrier_bench.native name alloc
Estimated testing time 20s (change using -quota SECS).
┌──────────┬──────────┬──────────────┬─────────┬─────────┬──────────┬────────────┐
│ Name     │ Time (ns) │         Time 95ci │   Minor │   Major │ Promoted │ Percentage │
```

```
┌─────────────┬─────────────┬──────────────────────────┬──────────────┬──────────┬──────────────┬──────────┐
│ mutable     │ 7_954_262   │ 7_827_275-8_135_261      │ 2_000_004    │ -51.42   │    -51.42    │   100.00 │
│ immutable   │ 3_694_618   │ 3_396_611-4_037_053      │ 5_000_005    │ -28.43   │    -28.43    │    46.45 │
└─────────────┴─────────────┴──────────────────────────┴──────────────┴──────────┴──────────────┴──────────┘
```

There is a stark space/time tradeoff here. The mutable version takes significantly longer to complete than the immutable one, but allocates many fewer minor heap words than the immutable version. Minor allocation in OCaml is very fast and so it is often better to use immutable data structures in preference to the more conventional mutable versions. On the other hand, if you only rarely mutate a value, it can be faster to take the write barrier hit and not allocate at all.

The only way to know for sure is to benchmark your program under real-world scenarios using `Core_bench`, and experiment with the tradeoffs. The command-line benchmark binaries have a number of useful options that affect garbage collection behaviour.

```
$ ./barrier_bench.native -help
Benchmark for mutable, immutable

  barrier_bench.native [COLUMN ...]

Columns that can be specified are:
    name       - Name of the test.
    cycles     - Number of CPU cycles (RDTSC) taken.
    cycles-err - 95% confidence interval and R^2 error for cycles.
    ~cycles    - Cycles taken excluding major GC costs.
    time       - Number of nano secs taken.
    time-err   - 95% confidence interval and R^2 error for time.
    ~time      - Time (ns) taken excluding major GC costs.
    alloc      - Allocation of major, minor and promoted words.
    gc         - Show major and minor collections per 1000 runs.
    percentage - Relative execution time as a percentage.
    speedup    - Relative execution cost as a speedup.
    samples    - Number of samples collected for profiling.

R^2 error indicates how noisy the benchmark data is. A value of
1.0 means the amortized cost of benchmark is almost exactly predicated
and 0.0 means the reported values are not reliable at all.
Also see: http://en.wikipedia.org/wiki/Coefficient_of_determination

Major and Minor GC stats indicate how many collections happen per 1000
runs of the benchmarked function.

The following columns will be displayed by default:
    +name time percentage

To specify that a column should be displayed only if it has a non-trivial value,
prefix the column name with a '+'.

=== flags ===

  [-clear-columns]     Don't display default columns. Only show user specified
                       ones.
```

```
[-display STYLE]     Table style (short, tall, line or blank). Default short.
[-geometric SCALE]   Use geometric sampling. (default 1.01)
[-linear INCREMENT]  Use linear sampling to explore number of runs, example 1.
[-no-compactions]    Disable GC compactions.
[-quota SECS]        Time quota allowed per test (default 10s).
[-save]              Save benchmark data to <test name>.txt files.
[-stabilize-gc]      Stabilize GC between each sample capture.
[-v]                 High verbosity level.
[-width WIDTH]       width limit on column display (default 150).
[-build-info]        print info about this build and exit
[-version]           print the version of this build and exit
[-help]              print this help text and exit
                     (alias: -?)
```

The `-no-compactions` and `-stabilize-gc` options can help force a situation where your application has fragmented memory. This can simulate the behaviour of a long-running application without you having to actually wait that long to recreate the behaviour in a performance unit test.

## Attaching finalizer functions to values

OCaml's automatic memory management guarantees that a value will eventually be freed when it's no longer in use, either via the garbage collector sweeping it or the program terminating. It's sometimes useful to run extra code just before a value is freed by the garbage collector, for example to check that a file descriptor has been closed, or that a log message is recorded.

**What values can be finalized?**

Various values cannot have finalizers attached since they aren't heap-allocated. Some examples of values that are not heap-allocated are integers, constant constructors, booleans, the empty array, the empty list and the unit value. The exact list of what is heap-allocated or not is implementation-dependent, which is why Core provides the `Heap_block` module to explicitly check before attaching the finalizer.

Some constant values can be heap-allocated but never deallocated during the lifetime of the program, for example a list of integer constants. `Heap_block` explicitly checks to see if the value is in the major or minor heap, and rejects most constant values. Compiler optimisations may also duplicate some immutable values such as floating-point values in arrays. These may be finalised while another duplicate copy is being used by the program.

For this reason, attach finalizers only to values that you are explicitly sure are heap-allocated and aren't immutable. A common use is to attach them to file descriptors to ensure it is closed. However, the finalizer normally shouldn't be the primary way of closing the file descriptor, since it depends on the garbage collector running in order to collect the value. For a busy system, you can easily run out of a scarce resource such as file descriptors before the GC catches up.

Core provides a `Heap_block` module that dynamically checks if a given value is suitable for finalizing. This block is then passed to Async's `Gc.add_finalizer` function that schedules the finalizer safely with respect to all the other concurrent program threads.

Let's explore this with a small example that finalizes values of different types, some of which are heap-allocated and others which are compile-time constants.

```
(* finalizer.ml : explore finalizers for different types *)
open Core.Std
open Async.Std

let attach_finalizer n v =
  match Heap_block.create v with
  | None -> printf "%20s: FAIL\n%!" n
  | Some hb ->
      let final _ = printf "%20s: OK\n%!" n in
      Gc.add_finalizer hb final

type t = { foo: bool }

let () =
  let alloced_float = Unix.gettimeofday () in
  let alloced_bool = alloced_float > 0.0 in
  let alloced_string = String.create 4 in
  attach_finalizer "immediate int" 1;
  attach_finalizer "immediate float" 1.0;
```

```
attach_finalizer "immediate variant" (`Foo "hello");
attach_finalizer "immediate string" "hello world";
attach_finalizer "immediate record" { foo=false };
attach_finalizer "allocated float" alloced_float;
attach_finalizer "allocated bool" alloced_bool;
attach_finalizer "allocated variant" (`Foo alloced_bool);
attach_finalizer "allocated string" alloced_string;
attach_finalizer "allocated record" { foo=alloced_bool };
Gc.compact ();
never_returns (Scheduler.go ())
```

Building and running this should show the following output.

```
$ ocamlfind ocamlopt -package core -package async -thread \
  -o finalizer -linkpkg finalizer.ml
$ ./finalizer
       immediate int: FAIL
     immediate float: FAIL
   immediate variant: FAIL
    immediate string: FAIL
    immediate record: FAIL
      allocated bool: FAIL
    allocated record: OK
    allocated string: OK
   allocated variant: OK
     allocated float: OK
```

The GC calls the finalization functions in the order of the deallocation. If several values become unreachable during the same GC cycle, the finalisation functions will be called in the reverse order of the corresponding calls to `add_finalizer`. Each call to `add_finalizer` adds to the set of functions that are run when the value becomes unreachable. You can have many finalizers all pointing to the same heap block if you wish.

After a garbage collection determines that a heap block `b` is unreachable, it removes from the set of finalizers all the functions associated with `b`, and serially applies each of those functions to `b`. Thus, every finalizer function attached to `b` will run at most once. However, program termination will not cause all the finalizers to be run before the runtime exits.

The finalizer can use all features of OCaml, including assignments that make the value reachable again and thus prevent it from being garbage collected. It can also loop forever, which will cause other finalizers to be interleaved with it.

**Production note**

This chapter contains significant contributions from Stephen Weeks.

2013-07-04
10:28:31

2013-07-04
10:28:31

CHAPTER 22

# The Compiler Frontend: Parsing and Type Checking

Compiling source code into executable programs is a fairly complex process that involves quite a few tools -- preprocessors, compilers, runtime libraries, linkers and assemblers. It's important to understand how these fit together to help with your day-to-day workflow of developing, debugging and deploying applications.

OCaml has a strong emphasis on static type safety and rejects source code that doesn't meet its requirements as early as possible. The compiler does this by running the source code through a series of checks and transformations. Each stage performs its job (e.g. type checking, optimization or code generation) and discards some information from the previous stage. The final native code output is low-level assembly code that doesn't know anything about the OCaml modules or objects that the compiler started with.

You don't have to do all this manually, of course. The compiler frontends (`ocamlc` and `ocamlopt`) are invoked via the command-line and chain the stages together for you. Sometimes though, you'll need to dive into the toolchain to hunt down a bug or investigate a performance problem. This chapter explains the compiler pipeline in more depth so you understand how to harness the command-line tools effectively.

In this chapter, we'll cover the following topics:

- the compilation pipeline and what each stage represents.
- source preprocessing via Camlp4 and the intermediate forms.
- the type-checking process, including module resolution.

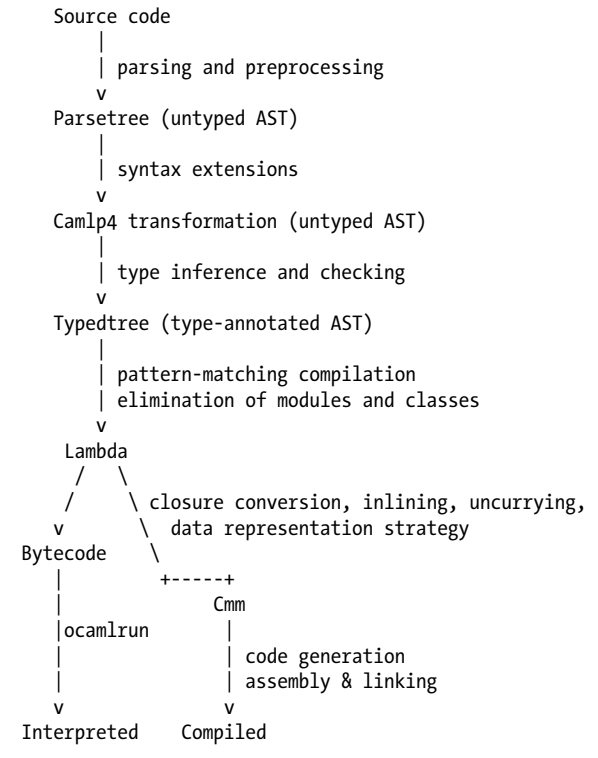
The details of the compilation process into executable code can be found next in Chapter 23.

## An overview of the toolchain

The OCaml tools accept textual source code as input, using filename extensions of `.ml` and `.mli` for modules and signatures respectively. We explained the basics of the build process earlier in Chapter 4, so we'll assume you've built a few OCaml programs already by this point.

Each source file represents a *compilation unit* that is built separately. The compiler generates intermediate files with different filename extensions to use as it advances through the compilation stages. The linker takes a collection of compiled units and produces a standalone executable or library archive that can be reused by other applications.

The overall compilation pipeline looks like this:




Notice that the pipeline branches towards the end. OCaml has multiple compiler frontends that reuse the early stages of compilation, but produce very different final outputs. The *bytecode interpreter* is portable and can even be transformed into Javascript. The *native code compiler* generates specialized executable binaries suitable for high-performance applications.

### Obtaining the compiler source code

Although it's not necessary to understand the examples, you may find it useful to have a copy of the OCaml source tree checked out while you read through this chapter. The source code is available from multiple places:

- Stable releases as zip and tar archives from the OCaml download site (*http://caml.inria.fr/download.en.html*).
- A Subversion anonymous mirror of the main development sources available on the development resources (*http://caml.inria.fr/ocaml/anonsvn.en.html*) page online.
- A Git mirror of the Subversion repository with all the history and development branches included, browsable online at Github (*https://github.com/ocaml/ocaml*).

The source tree is split up into sub-directories. The core compiler consists of:

- `config/`: configuration directives to tailor OCaml for your operating system and architecture.
- `bytecomp/` and `byterun/`: byte-code compiler and runtime, including the garbage collector.
- `asmcomp/` and `asmrun/`: native-code compiler and runtime. The native runtime symlinks many modules from the `byterun` directory to share code, most notably the garbage collector.
- `parsing/`: the OCaml lexer, parser and libraries for manipulating them.
- `typing/`: the static type checking implementation and type definitions.
- `camlp4/`: the source code macro preprocessor.
- `driver/`: command-line interfaces for the compiler tools.

There are a number of tools and scripts also built alongside the core compiler:

- `debugger/`: the interactive byte-code debugger.
- `toplevel/`: interactive top-level console.
- `emacs/`: a *caml-mode* for the Emacs editor.
- `stdlib/`: the compiler standard library, including the `Pervasives` module.
- `ocamlbuild/`: build system that automates common OCaml compilation modes.
- `otherlibs/`: optional libraries such as the Unix and graphics modules.
- `tools/`: command-line utilities such as `ocamldep` that are installed with the compiler.
- `testsuite/`: regression tests for the core compiler.

We'll go through each of the compilation stages now and explain how that'll be useful to you during day-to-day OCaml development.

# Parsing source code

When a source file is passed to the OCaml compiler, its first task is to parse the text into a more structured Abstract Syntax Tree (AST). The parsing logic is implemented in OCaml itself using the techniques described earlier in Chapter 16. The lexer and parser rules can be found in the `parsing` directory in the source distribution.

## Syntax errors

The OCaml parser's goal is to output a well-formed AST data structure to the next phase of compilation, and so it rejects any source code that doesn't match basic syntactic requirements. The compiler emits a *syntax error* in this situation, with a pointer to the filename and line and character number that's as close to the error as possible.

Here's an example syntax error that we obtain by performing a module assignment as a statement instead of as a let-binding.

```
(* broken_module.ml *)
let () =
  module MyString = String;
  ()
```

The above code results in a syntax error when compiled.

```
$ ocamlc -c broken_module.ml
File "broken_module.ml", line 3, characters 2-8:
Error: Syntax error
```

The correct version of this source code creates the `MyString` module correctly via a local open, and compiles successfully.

```
(* fixed_module.ml *)
let () =
  let module MyString = String in
  ()
```

The syntax error points to the line and character number of the first token that couldn't be parsed. In the broken example the `module` keyword isn't a valid token at that point in parsing, so the error location information is correct.

## Automatically indenting source code

Sadly, syntax errors do get more inaccurate sometimes depending on the nature of your mistake. Try to spot the deliberate error in the following function definitions.

```
(* follow_on_function.ml *)
let concat_and_print x y =
  let v = x ^ y in
```

```
  print_endline v;
  v;

let add_and_print x y =
  let v = x + y in
  print_endline (string_of_int v);
  v

let () =
  let _x = add_and_print 1 2 in
  let _y = concat_and_print "a" "b" in
  ()
```

When you compile this file you'll get a syntax error.

```
$ ocamlc -c follow_on_function.ml
File "follow_on_function.ml", line 12, characters 0-3:
Error: Syntax error
```

The line number in the error points to the end of the `add_and_print` function, but the actual error is at the end of the *first* function definition. There's an extra semicolon at the end of the first definition that causes the second definition to become part of the first `let` binding. This eventually results in a parsing error at the very end of the second function.

This class of bug (due to a single errant character) can be hard to spot in a large body of code. Luckily, there's a great tool in OPAM called ocp-indent that applies structured indenting rules to your source code on a line-by-line basis. This not only beautifies your code layout, but it also makes this syntax error much easier to locate.

Let's run our erronous file through ocp-indent and see how it processes it.

```
$ opam install ocp-indent
$ ocp-indent follow_on_function.ml
(* follow_on_function.ml *)
let concat_and_print x y =
  let v = x ^ y in
  print_endline v;
  v;

  let add_and_print x y =
    let v = x + y in
    print_endline (string_of_int v);
    v

let () =
  let _x = add_and_print 1 2 in
  let _y = concat_and_print "a" "b" in
  ()
```

The `add_and_print` definition has been indented as if it were part of the first `con cat_and_print` definition, and the errant semicolon is now much easier to spot. We just need to remove that semicolon and re-run ocp-indent to verify that the syntax is correct.

```
$ ocp-indent follow_on_function_fixed.ml
(* follow_on_function_fixed.ml *)
let concat_and_print x y =
  let v = x ^ y in
  print_endline v;
  v

let add_and_print x y =
  let v = x + y in
  print_endline (string_of_int v);
  v

let () =
  let _x = add_and_print 1 2 in
  let _y = concat_and_print "a" "b" in
  ()

$ ocamlc -i follow_on_function_fixed.ml
val concat_and_print : string -> string -> string
val add_and_print : int -> int -> int
```

The ocp-indent homepage (*https://github.com/OCamlPro/ocp-indent*) documents how to integrate it with your favourite editor. All the Core libraries are formatted using it to ensure consistency, and it's a good idea to do this before publishing your own source code online.

## Generating documentation from interfaces

Whitespace and source code comments are removed during parsing and aren't significant in determining the semantics of the program. However, other tools in the OCaml distribution can interpret comments for their own ends.

The OCamldoc tool uses specially formatted comments in the source code to generate documentation bundles. These comments are combined with the function definitions and signatures and output as structured documentation in a variety of formats. It can generate HTML pages, LaTeX and PDF documents, UNIX manual pages and even module dependency graphs that can be viewed using Graphviz (*http://www.graphviz.org*).

Here's a sample of some source code that's been annotated with OCamldoc comments.

```
(** example.ml: The first special comment of the file is the comment
    associated with the whole module. *)

(** Comment for exception My_exception. *)
exception My_exception of (int -> int) * int

(** Comment for type [weather]  *)
type weather =
| Rain of int (** The comment for construtor Rain *)
| Sun         (** The comment for constructor Sun *)
```

```
(** Find the current weather for a country
   @author Anil Madhavapeddy
   @param location The country to get the weather for.
*)
let what_is_the_weather_in location =
  match location with
  | `Cambridge  -> Rain 100
  | `New_york   -> Rain 20
  | `California -> Sun
```

The OCamldoc comments are distinguished by beginning with the double asterisk. There are formatting conventions for the contents of the comment to mark metadata. For instance, the `@tag` fields mark specific properties such as the author of that section of code.

Try compiling the HTML documentation and UNIX man pages by running `ocamldoc` over the source file.

```
$ mkdir -p html man/man3
$ ocamldoc -html -d html example.ml
$ ocamldoc -man -d man/man3 example.ml
$ man -M man Example
```

You should now have HTML files inside the `html/` directory and also be able to view the UNIX manual pages held in `man/man3`. There are quite a few comment formats and options to control the output for the various backends. Refer to the OCaml manual (*http://caml.inria.fr/pub/docs/manual-ocaml/manual029.html*) for the complete list.

#### Using custom OCamldoc generators

The default HTML output stylesheets from OCamldoc are pretty spartan and distinctly Web 1.0. The tool supports plugging in custom documentation generators, and there are several available that provide prettier or more detailed output.

- Argot (*http://argot.x9c.fr/*) is an enchanced HTML generator that supports code folding and searching by name or type definition.
- ocamldoc-generators (*https://gitorious.org/ocamldoc-generators/ocamldoc-generators*) add support for Bibtex references within comments and generating literate documentation that embeds the code alongside the comments.
- JSON output is available via a custom generator (*https://github.com/xen-org/ocamldoc-json*) in Xen.

## Preprocessing source code

One powerful feature in OCaml is a facility to extend the standard language grammar without having to modify the compiler. You can roughly think of it as a type-safe version to the `cpp` preprocessor used in C/C++ to control conditional compilation directives.

The OCaml distribution includes a system called Camlp4 for writing extensible parsers. This provides some OCaml libraries that are used to define grammars and also dynamically loadable syntax extensions of such grammars. Camlp4 modules register new language keywords and later transform these keywords (or indeed, any portion of the input program) into conventional OCaml code that can be understood by the rest of the compiler.

We've already seen several Core libraries that use Camlp4:

- `Fieldslib` generates first-class values that represent fields of a record.
- `Sexplib` to convert types to textual s-expressions.
- `Bin_prot` for efficient binary conversion and parsing.

These libraries all extend the language in quite a minimal way by adding a `with` keyword to type declarations to signify that extra code should be generated from that declaration. For example, here's a trivial use of Sexplib and Fieldslib.

```
(* type_conv_example.ml *)
open Sexplib.Std

type t = {
  foo: int;
  bar: string
} with sexp, fields
```

Compiling this code will normally give you a syntax error if you do so without Camlp4 since the `with` keyword isn't normally allowed after a type definition.

```
$ ocamlfind ocamlc -c type_conv_example.ml
File "type_conv_example.ml", line 7, characters 2-6:
Error: Syntax error
```

Now add in the syntax extension packages for `fieldslib` and `sexplib`, and everything will compile again.

```
$ ocamlfind ocamlc -c -syntax camlp4o -package sexplib.syntax \
  -package fieldslib.syntax type_conv_example.ml
```

We've specified a couple of additional flags here. The `-syntax` flag directs `ocamlfind` to add the `-pp` flag to the compiler command-line. This flag instructs the compiler to run the preprocessor during its parsing phase.

The `-package` flag imports other OCaml libraries. The `.syntax` suffix in the package name is a convention that indicates these libraries are preprocessors that should be run during parsing. The syntax extension modules are dynamically loaded into the `camlp4o` command which rewrites the input source code into conventional OCaml code that has no trace of the new keywords. The compiler then compiles this transformed code with no knowledge of the preprocessor's actions.

Both Fieldslib and Sexplib need this new `with` keyword, but they both can't register the same extension. Instead, a library called Type_conv provides the common extension framework for them to use. Type_conv registers the `with` grammar extension to Camlp4, and the OCamlfind packaging ensures that it's loaded before Variantslib or Sexplib.

The two extensions generate boilerplate OCaml code based on the type definition. This avoids the inevitable performance hit of doing the code generation dynamically. It also doesn't require a Just-In-Time (JIT) runtime that can be a source of unpredictable dynamic behaviour. Instead, all code is simply generated at compile-time via Camlp4.

The syntax extensions accept an input AST and output a modified one. If you're not familiar with the Camlp4 module in question, how do you figure out what changes it's made to your code? The obvious way is to read the documentation that accompanies the extension. Another approach is to use the top-level to explore the extension's behaviour or run Camlp4 manually yourself to see the transformation in action. We'll show you how to do both of these now.

## Using Camlp4 interactively

The `utop` top-level can run the phrases that you type through `camlp4` automatically. You should have at least these lines in your `~/.ocamlinit` file in your home directory (see Appendix A for more information).

```
#use "topfind"
#camlp4o
```

The first directive loads the `ocamlfind` top-level interface that lets you require `ocamlfind` packages (including all their dependent packages). The second directive instructs the top-level to filter all phrases via Camlp4. You can now run `utop` and load the syntax extensions in. We'll use the `comparelib` syntax extension for our experiments.

OCaml provides a built-in polymorphic comparison operator that inspects the runtime representation of two values to see if they're equal. As we noted in Chapter 13, the polymorphic comparison is less efficient than defining explicit comparison functions between values. However, it quickly become tedious to manually define comparison functions for complex type definitions.

Let's see how `comparelib` solves this problem by running it in `utop`.

```
# #require "comparelib.syntax" ;;

# type t = { foo: string; bar : t } ;;
type t = { foo : string; bar : t; }

# type t = { foo: string; bar: t } with compare ;;
type t = { foo : string; bar : t; }
val compare : t -> t -> int = <fun>
val compare_t : t -> t -> int = <fun>
```

The first definition of `t` is a standard OCaml phrase and results in the expected output. The second one includes the `with compare` directive. This is intercepted by `compare lib` and transformed into the original type definition with two new functions also incuded.

## Running Camlp4 from the command line

The top-level is a quick way to examine the signatures generated from the extensions, but how can we see what these new functions actually do? You can't do this from `utop` directly since it embeds the Camlp4 invocation as an automated part of its operation.

Let's turn to the command-line to obtain the result of the `comparelib` transformation instead. Create a file that contains the type declaration from earlier:

```
(* comparelib_test.ml *)
type t = {
  foo: string;
  bar: t
} with compare
```

We need to run the Camlp4 binary with the library paths to Comparelib and Type_conv. Let's use a small shell script to wrap this invocation.

```
#!/bin/sh
# camlp4_dump

OCAMLFIND="ocamlfind query -predicates syntax,preprocessor -r"
INCLUDE=`$OCAMLFIND -i-format comparelib.syntax`
ARCHIVES=`$OCAMLFIND -a-format comparelib.syntax`
camlp4o -printer o $INCLUDE $ARCHIVES $1
```

The script uses the `ocamlfind` package manager to list the include and library paths needed by `comparelib`. It then invokes the `camlp4o` preprocessor with these paths and outputs the resulting AST to the standard output.

```
$ sh camlp4_dump comparelib_test.ml
type t = { foo : string; bar : t }

let _ = fun (_ : t) -> ()
```

```
let rec compare : t -> t -> int =
  fun a__001_ b__002_ ->
    if Pervasives.( == ) a__001_ b__002_
    then 0
    else
      (let ret =
         (Pervasives.compare : string -> string -> int) a__001_.foo
           b__002_.foo
       in
         if Pervasives.( <> ) ret 0
         then ret
         else compare a__001_.bar b__002_.bar)

let _ = compare
let compare_t = compare
let _ = compare_t
```

The output contains the original type definition accompanied by some automatically generated code that implements an explicit comparison function for each field in the record. If you're using the extension in your compiler command-line, this generated code is then compiled as if you had typed it in yourself.

**A style note: wildcards in `let` bindings**

You may have noticed the `let _ = fun` construct in the auto-generated code above. The underscore in a `let` binding is just the same as a wildcard underscore in a pattern match, and tells the compiler to accept any return value and discard it immediately.

This is fine for mechanically generated code from Type_conv, but should be avoided in code that you write by hand. If it's a unit-returning expression, then write a `unit` binding explicitly instead. This will cause a type error if the expression changes type in the future (e.g. due to code refactoring).

```
let () = <expr>
```

If the expression has a different type, then write it explicitly.

```
let (_:some_type) = <expr>
```

```
let () = ignore (<expr> : some_type)
let () = don't_wait_for (<expr>)  (* if the expression returns a unit Deferred.t *)
```

The last one is used to ignore Async expressions that should run in the background rather than blocking in the current thread.

One other important reason for using wildcard matches is to bind a variable name to something that you want to use in future code, but don't want to use right away. This would normally generate an "unused value" compiler warning. These warnings are suppressed for any variable name that's prepended with an underscore.

```
let fn x y =
  let _z = x + y in
  ()
```

Although you don't use `_z` in your code, this will never generate an unused variable warning.

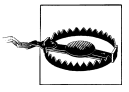
## Preprocessing module signatures

Another useful feature of `type_conv` is that it can generate module signatures too. Copy the earlier type definition into a `comparelib_test.mli` and rerun the Camlp4 dumper script.

```
$ sh camlp4_dump comparelib_test.mli
type t = { foo : string; bar : t }

val compare : t -> t -> int
```

The external signature generated by `comparelib` is much simpler than the actual code. Running Camlp4 directly on the original source code lets you see these all these transformations precisely.



**Don't overdo the syntax extensions**

Syntax extensions are a powerful extension mechanism that can completely alter your source code's layout and style. Core includes a very conservative set of extensions that take care to minimise the syntax changes. There are a number of third-party libraries that are much more ambitious -- some introduce whitespace-sensitive indentation while others build entirely new embedded languages using OCaml as a host language.

While it's tempting to compress all your boiler-plate code into Camlp4 extensions, it can make your source code much harder for other people to quickly read and understand. Core mainly focuses on type-driven code generation using the `type_conv` extension and doesn't fundamentally change the OCaml syntax.

Another thing to consider before deploying your own syntax extension is compatibility with other extensions. Two separate extensions can create a grammar clash that leads to odd syntax errors and hard-to-reproduce bugs. That's why most of Core's syntax extensions go through `type_conv`, which acts as a single point for extending the grammar via the `with` keyword.

### Further reading on Camlp4

We've deliberately only shown you how to use Camlp4 extensions here, and not how to build your own. The full details of building new extensions are fairly daunting and could be the subject of an entirely new book.

The best resources to get started are:

- the online Camlp4 wiki (*http://brion.inria.fr/gallium/index.php/Camlp4*).
- using OPAM to install existing Camlp4 extensions and inspecting their source code.
- a series of blog posts (*http://ambassadortothecomputers.blogspot.co.uk/p/reading-camlp4.html*) by Jake Donham describe the internals of Camlp4 and its syntax extension mechanism.

## Static type checking

After obtaining a valid abstract syntax tree, the compiler has to verify that the code obeys the rules of the OCaml type system. Code that is syntactically correct but misuses values is rejected with an explanation of the problem.

Although type checking is done in a single pass in OCaml, it actually consists of three distinct steps that happen simultaneously:

- an *automatic type inference* algorithm that calculates types for a module without requiring manual type annotations.

- a *module system* that combines software components with explicit knowledge of their type signatures.
- *explicit subtyping* checks for objects and polymorphic variants.

Automatic type inference lets you write succinct code for a particular task and have the compiler ensure that your use of variables is locally consistent.

Type inference doesn't scale to very large code bases that depend on separate compilation of files. A small change in one module may ripple through thousands of other files and libraries and require all of them to be recompiled. The module system solves this by providing the facility to combine and manipulate explicit type signatures for modules within a large project, and also to reuse them via functors and first-class modules.

Subtyping in OCaml objects is always an explicit operation (via the `:>` operator). This means that it doesn't complicate the core type inference engine and can be tested as a separate concern.

## Displaying inferred types from the compiler

We've already seen how you can explore type inference directly from the top-level. It's also possible to generate type signatures for an entire file by asking the compiler to do the work for you. Create a file with a single type definition and value.

```
(* typedef.ml *)
type t = Foo | Bar
let v = Foo
```

Now run the compiler with the `-i` flag to infer the type signature for that file. This runs the type checker but doesn't compile the code any further after displaying the interface to the standard output.

```
$ ocamlc -i typedef.ml
type t = Foo | Bar
val v : t
```

The output is the default signature for the module which represents the input file. It's often useful to redirect this output to an `mli` file to give you a starting signature to edit the external interface without having to type it all in by hand.

The compiler stores a compiled version of the interface as a `cmi` file. This interface is either obtained from compiling an `mli` signature file for a module, or by the inferred type if there is only an `ml` implementation present.

The compiler makes sure that your `ml` and `mli` files have compatible signatures. The type checker throws an immediate error if this isn't the case.

```
$ echo type t = Foo > test.ml
```

```
$ echo type t = Bar > test.mli
$ ocamlc -c test.mli test.ml
File "test.ml", line 1:
Error: The implementation test.ml does not match the interface test.cmi:
       Type declarations do not match:
         type t = Foo
       is not included in
         type t = Bar
       File "test.ml", line 1, characters 5-12: Actual declaration
       Their first fields have different names, Foo and Bar.
```

**Which comes first: the `ml` or the `mli`?**

There are two schools of thought on which order OCaml code should be written in. It's very easy to begin writing code by starting with an `ml` file and using the type inference to guide you as you build up your functions. The `mli` file can then be generated as described above, and the exported functions documented.

If you're writing code that spans multiple files, it's sometimes easier to start by writing all the `mli` signatures and checking that they type check against each other. Once the signatures are in place, you can write the implementations with the confidence that they'll all glue together correctly with no cyclic dependencies between the modules.

As with any such stylistic debate, you should experiment with which system works best for you. Everyone agrees on one thing though: no matter what order you write them, production code should always explicitly define an `mli` file for every `ml` file in the project.

Signature files provide a place to write succinct documentation and to abstract internal details that shouldn't be exported. Maintaining separate signature files also speeds up incremental compilation in larger code-bases, since recompiling a `mli` signature is much faster than a full compilation of the implementation to native code.

## Type inference

Type inference is the process of determining the appropriate types for expressions based on their use. It's a feature that's partially present in many other languages such as Haskell and Scala, but OCaml embeds it as a fundamental feature throughout the core language.

OCaml type inference is based on the Hindley-Milner algorithm, which is notable for its ability to infer the most general type for an expression without requiring any explicit type annotations. The algorithm can deduce multiple types for an expression, and has the notion of a *principal type* that is the most general choice from the possible inferences. Manual type annotations can specialize the type explicitly, but the automatic inference selects the most general type unless told otherwise.

OCaml does have some language extensions which strain the limits of principal type inference, but by and large most programs you write will never *require* annotations (although they sometimes help the compiler produce better error messages).

#### Adding type annotations to find errors

It's often said that the hardest part of writing OCaml code is getting past the type checker -- but once the code does compile, it works correctly the first time!

There are a couple of tricks to make it easier to quickly locate type errors in your code. The first is to introduce manual type annotations to narrow down the source of your error more accurately. These annotations shouldn't actually change your types and can be removed once your code is correct. However, they act as anchors to locate errors while you're still writing your code.

Manual type annotations are particulary useful if you use lots of polymorphic variants or objects. Type inference with row polymorphism can generate some very large signatures, and errors tend to propagate more widely than if you are using more explicitly typed variants or classes.

For instance, consider this broken example that expresses some simple algebraic operations over integers.

```
(* broken_poly.ml *)

let rec algebra =
  function
  | `Add (x,y) -> (algebra x) + (algebra y)
  | `Sub (x,y) -> (algebra x) - (algebra y)
  | `Mul (x,y) -> (algebra x) * (algebra y)
  | `Num x     -> x

let _ =
  algebra (
    `Add (
      (`Num 0),
      (`Sub (
          (`Num 1),
          (`Mul (
              (`Nu 3),(`Num 2)
            ))
        ))
    ))
```

There's a single character typo in the code so that it uses `Nu` instead of `Num`. The resulting type error is impressive.

```
$ ocamlc -c broken_poly.ml
File "broken_poly.ml", line 11, characters 10-154:
Error: This expression has type
         [> `Add of
```

```
            ([< `Add of 'a * 'a
              | `Mul of 'a * 'a
              | `Num of int
              | `Sub of 'a * 'a
              > `Num ]
             as 'a) *
            [> `Sub of 'a * [> `Mul of [> `Nu of int ] * [> `Num of int ] ] ] ]
       but an expression was expected of type 'a
       The second variant type does not allow tag(s) `Nu
```

The type error is perfectly accurate, but rather verbose and with a line number that doesn't point to the exact location of the incorrect variant name. The best the compiler can do is to point you in the general direction of the `algebra` function application.

This is because the type checker doesn't have enough information to match the inferred type of the `algebra` definition to its application a few lines down. It calculates types for both expressions separately, and when they don't match up, outputs the difference as best it can.

Let's see what happens with an explicit type annotation to help the compiler out.

```
(* broken_poly_with_annot.ml *)

type t = [
  | `Add of t * t
  | `Sub of t * t
  | `Mul of t * t
  | `Num of int
]

let rec algebra (x:t) =
  match x with
  | `Add (x,y) -> (algebra x) + (algebra y)
  | `Sub (x,y) -> (algebra x) - (algebra y)
  | `Mul (x,y) -> (algebra x) * (algebra y)
  | `Num x     -> x

let _ =
  algebra (
    `Add (
      (`Num 0),
      (`Sub (
          (`Num 1),
          (`Mul (
              (`Nu 3),(`Num 2)
            ))
        ))
    ))
```

This code contains exactly the same error as before, but we've added a closed type definition of the polymorphic variants, and a type annotation to the `algebra` definition. The compiler error we get is much more useful now.

```
$ ocamlc -i broken_poly_with_annot.ml
File "broken_poly_with_annot.ml", line 24, characters 14-21:
Error: This expression has type [> `Nu of int ]
       but an expression was expected of type t
       The second variant type does not allow tag(s) `Nu
```

This error points directly to the correct line number that contains the typo. Once you fix the problem, you can remove the manual annotations if you prefer more succinct code. You can also leave the annotations there of course, to help with future refactoring and debugging.

#### Enforcing principal typing

The compiler also has a stricter *principal type checking* mode that is activated via the `-principal` flag. This warns about risky uses of type information to ensure that the type inference has one principal result. A type is considered risky if the success or failure of type inference depends on the order in which sub-expressions are typed.

The principality check only affects a few language features:

- polymorphic methods for objects.
- permuting the order of labeled arguments in a function from their type definition.
- discarding optional labelled arguments.
- generalized algebraic data types (GADTs) present from OCaml 4.0 onwards.
- automatic disambiguation of record field and constructor names (since OCaml 4.1)

Here's an example of principality warnings when used with record disambiguation.

```
(* non_principal.ml *)
type s = { foo: int; bar: unit }
type t = { foo: int }

let f x =
  x.bar;
  x.foo
```

Inferring the signature with `-principal` will show you a new warning.

```
$ ocamlc -i -principal non_principal.ml
File "non_principal.ml", line 7, characters 4-7:
Warning 18: this type-based field disambiguation is not principal.
type s = { foo : int; bar : unit; }
type t = { foo : int; }
val f : s -> int
```

This example isn't principal since the inferred type for `x.foo` is guided by the inferred type of `x.bar`, whereas principal typing requires that each sub-expression's type can be calculated independently. If the `x.bar` use is removed from the definition of `f`, its argument would be of type `t` and not `type s`.

You can fix this either by permuting the order of the type declarations, or by adding an explicit type annotation.

```
(* principal.ml *)
type s = { foo: int; bar: unit }
type t = { foo: int }

let f (x:s) =
  x.bar;
  x.foo
```

There is now no ambiguity about the inferred types, since we've explicitly given the argument a type and the order of inference of the sub-expressions no longer matters.

```
$ ocamlc -i -principal principal.ml
type s = { foo : int; bar : unit; }
type t = { foo : int; }
val f : s -> int
```

Ideally, all code should systematically use `-principal`. It reduces variance in type inference and enforces the notion of a single known type. However, there are drawbacks to this mode: type inference is slower and the `cmi` files become larger. This is generally only a problem if you use objects extensively, which usually have larger type signature to cover all their methods.

As a result, the suggested approach is to only compile with `-principal` occasionally to check if your code is compliant. If compiling in principal mode works, it is guaranteed that the program will pass type checking in non-principal mode too.

Bear in mind that the `cmi` files generated in principal mode differ from the default mode. Try to ensure that you compile your whole project with it activated. Getting the files mixed up won't let you violate type safety, but can result in the type checker failing unexpectedly very occasionally. In this case, just recompile with a clean source tree.

## Modules and separate compilation

The OCaml module system enables smaller components to be reused effectively in large projects while still retaining all the benefits of static type safety. We covered the basics of using modules earlier in Chapter 4. The module language that operates over these signatures also extends to functors and first-class modules, described in Chapter 9 and Chapter 10 respectively.

This section discusses how the compiler implements them in more detail. Modules are essential for larger projects that consist of many source files (also known as *compilation units*). It's impractical to recompile every single source file when changing just one or two files, and the module system minimizes such recompilation while still encouraging code reuse.

### The mapping between files and modules

Individual compilation units provide a convenient way to break up a big module hierarchy into a collection of files. The relationship between files and modules can be explained directly in terms of the module system.

Create a file called `alice.ml` with the following contents.

```
(* alice.ml *)
let friends = [ Bob.name ]
```

and a corresponding signature file.

```
(* alice.mli *)
val friends : Bob.t list
```

These two files are exactly analogous to including the following code directly in another module that references `Alice`.

```
module Alice : sig
  val friends : Bob.t list
end = struct
  let friends = [ Bob.name ]
end
```

### Defining a module search path

In the example above, `Alice` also has a reference to another module `Bob`. For the overall type of `Alice` to be valid, the compiler also needs to check that the `Bob` module contains at least a `Bob.name` value and defines a `Bob.t` type.

The type checker resolves such module references into concrete structures and signatures in order to unify types across module boundaries. It does this by searching a list of directories for a compiled interface file matching that module's name. For example, it will look for `alice.cmi` and `bob.cmi` on the search path, and use the first ones it encounters as the interfaces for `Alice` and `Bob`.

The module search path is set by adding `-I` flags to the compiler command-line with the directory containing the `cmi` files as the argument. Manually specifying these flags gets complex when you have lots of libraries, and is the reason why the OCamlfind frontend to the compiler exists. OCamlfind automates the process of turning third-party package names and build descriptions into command-line flags that are passed to the compiler command-line.

By default, only the current directory and the OCaml standard library will be searched for `cmi` files. The `Pervasives` module from the standard library will also be opened by default in every compilation unit. The standard library location is obtained by running `ocamlc -where`, and can be overridden by setting the `CAMLLIB` environment variable.

Needless to say, don't override the default path unless you have a good reason to (such as setting up a cross-compilation environment).

### Inspecting compilation units with `ocamlobjinfo`

For separate compilation to be sound, we need to ensure that all the `cmi` files used to type-check a module are the same across compilation runs. If they vary, this raises the possibility of two modules checking different type signature for a common module with the same name. This in turn lets the program completely violate the static type system and can lead to memory corruption and crashes.

OCaml guards against this by recording a CRC checksum in every `cmi`. Let's examine our earlier `typedef.ml` more closely.

```
$ ocamlc -c typedef.ml
$ ocamlobjinfo typedef.cmi
File typedef.cmi
Unit name: Typedef
Interfaces imported:
    559f8708a08ddf66822f08be4e9c3372    Typedef
    65014ccc4d9329a2666360e6af2d7352    Pervasives
```

`ocamlobjinfo` examines the compiled interface and displays what other compilation units it depends on. In this case, we don't use any external modules other than `Pervasives`. Every module depends on `Pervasives` by default, unless you use the `-nopervasives` flag (this is an advanced use-case, and you shouldn't normally need it).

The long alphanumeric identifier beside each module name is a hash calculated from all the types and values exported from that compilation unit. It's used during type-checking and linking to ensure that all of the compilation units have been compiled consistently against each other. A difference in the hashes means that a compilation unit with the same module name may have conflicting type signatures in different modules. The compiler will reject such programs with an error similar to this:

```
File "foo.ml", line 1, characters 0-1:
Error: The files /home/build/bar.cmi
       and /usr/lib/ocaml/map.cmi make inconsistent assumptions
       over interface Map
```

This hash check is very conservative, but ensures that separate compilation remains type-safe all the way up to the final link phase. Your build system should ensure that you never see the error messages above, but if you do run into it, just clean out your intermediate files and recompile from scratch.

## Shorter module paths in type errors

Core uses the OCaml module system quite extensively to provide a complete replacement standard library. It collects these modules into a single `Std` module which provides

a single module that needs to be opened to import the replacement modules and functions.

There's one downside to this approach: type errors suddenly get much more verbose. We can see this if you run the vanilla OCaml top-level (not `utop`).

```
$ ocaml
# List.map print_endline "" ;;
Error: This expression has type string but an expression was expected of type
         string list
```

This type error without `Core.Std` has a straightforward type error. When we switch to Core, though, it gets more verbose.

```
# open Core.Std ;;
# List.map ~f:print_endline "" ;;
Error: This expression has type string but an expression was expected of type
         'a Core.Std.List.t = 'a list
```

The default `List` module in OCaml is overridden by `Core.Std.List`. The compiler does its best to show the type equivalence, but at the cost of a more verbose error message.

The compiler can remedy this via a so-called "short paths" heuristic. This causes the compiler to search all the type aliases for the shortest module path, and use that as the preferred output type. The option is activated by passing `-short-paths` to the compiler, and works on the top-level too.

```
$ ocaml -short-paths
# open Core.Std;;
# List.map ~f:print_endline "foo";;
Error: This expression has type string but an expression was expected of type
         'a list
```

The `utop` enhanced top-level activates short paths by default, which is why you've not had to do this before in our interactive examples. However, the compiler doesn't default to the short path heuristic since there are some situations where the type aliasing information is useful to know, and would be lost in the error if the shortest module path is always picked.

You'll need to choose for yourself if you prefer short paths or the default behaviour in your own projects, and pass the `-short-paths` flag to the compiler if you need it.

## The typed syntax tree

When the type checking process has successfully completed, it is combined with the AST to form a *typed abstract syntax tree*. This contains precise location information for every token in the input file, and decorates each token with concrete type information.

The compiler can output this as compiled `cmt` and `cmti` files that contain the typed AST for the implementation and signatures of a compilation unit. This is activated by passing the `-bin-annot` flag to the compiler.

The `cmt` files are particularly useful for IDE tools to match up OCaml source code at a specific location to the inferred or external types.

## Using ocp-index for auto-completion

One such command-line tool to display auto-completion information in your editor is `ocp-index`. Install it via OPAM as follows.

```
$ opam install ocp-index
$ ocp-index
```

Let's refer back to our Ncurses binding example from the beginning of Chapter 19. This module defined bindings for the Ncurses library. First, compile the interfaces with `-bin-annot` so that we can obtain the `cmt` and `cmti` files.

```
$ ocamlfind ocamlopt -bin-annot -c -package ctypes.foreign \
    ncurses.mli ncurses.ml
```

Next, run `ocp-index` in completion mode. You pass it a set of directories to search for `cmt` files in, and a fragment of text to autocomplete.

```
$ ocp-index complete -I . Ncur
Ncurses module

$ ocp-index complete -I . Ncurses.a
Ncurses.addstr val string -> unit
Ncurses.addch val char -> unit

$ ocp-index complete -I . Ncurses.
Ncurses.cbreak val unit -> unit
Ncurses.box val Ncurses.window -> int -> int -> unit
Ncurses.mvwaddstr val Ncurses.window -> int -> int -> string -> unit
Ncurses.mvwaddch val Ncurses.window -> int -> int -> char -> unit
Ncurses.addstr val string -> unit
Ncurses.addch val char -> unit
Ncurses.newwin val int -> int -> int -> int -> Ncurses.window
Ncurses.refresh val unit -> unit
Ncurses.endwin val unit -> unit
Ncurses.initscr val unit -> Ncurses.window
Ncurses.wrefresh val Ncurses.window -> unit
Ncurses.window val Ncurses.window Ctypes.typ
```

As you can imagine, autocompletion is invaluable on larger codebases. See the ocp-index (*https://github.com/ocamlpro/ocp-index*) homepage for more information on how to integrate it with your favourite editor.

## Examining the typed syntax tree directly

The compiler has a couple of advanced flags that can dump the raw output of the internal AST representation. You can't depend on these flags to give the same output across compiler revisions, but they are a useful learning tool.

We'll use our toy `typedef.ml` again.

```
(* typedef.ml *)
type t = Foo | Bar
let v = Foo
```

Let's first look at the untyped syntax tree that's generated from the parsing phase.

```
$ ocamlc -dparsetree typedef.ml
[
  structure_item (typedef.ml[1,0+0]..[1,0+18])
    Pstr_type [
      "t" (typedef.ml[1,0+5]..[1,0+6])
        type_declaration (typedef.ml[1,0+5]..[1,0+18])
          ptype_params = []
          ptype_cstrs = []
          ptype_kind =
            Ptype_variant
              [
                (typedef.ml[1,0+9]..[1,0+12])
                  "Foo" (typedef.ml[1,0+9]..[1,0+12])
                  [] None
                (typedef.ml[1,0+15]..[1,0+18])
                  "Bar" (typedef.ml[1,0+15]..[1,0+18])
                  [] None
              ]
          ptype_private = Public
          ptype_manifest = None
    ]
  structure_item (typedef.ml[2,19+0]..[2,19+11])
    Pstr_value Nonrec [
      <def>
        pattern (typedef.ml[2,19+4]..[2,19+5])
          Ppat_var "v" (typedef.ml[2,19+4]..[2,19+5])
        expression (typedef.ml[2,19+8]..[2,19+11])
          Pexp_construct "Foo" (typedef.ml[2,19+8]..[2,19+11])
          None false
    ]
]
```

This is rather a lot of output for a simple two-line program, but it shows just how much structure the OCaml parser generates even from a small source file.

Each portion of the AST is decorated with the precise location information (including the filename and character location of the token). This code hasn't been type checked yet, and so the raw tokens are all included.

The typed AST that is normally output as a compiled `cmt` file can be displayed in a more developer-readable form via the `-dtypedtree` option.

```
$ ocamlc -dtypedtree typedef.ml
[
  structure_item (typedef.ml[1,0+0]..typedef.ml[1,0+18])
    Pstr_type [
      t/1008
        type_declaration (typedef.ml[1,0+5]..typedef.ml[1,0+18])
          ptype_params = []
          ptype_cstrs = []
          ptype_kind =
            Ptype_variant
              [
                "Foo/1009" []
                "Bar/1010" []
              ]
          ptype_private = Public
          ptype_manifest = None
    ]
  structure_item (typedef.ml[2,19+0]..typedef.ml[2,19+11])
    Pstr_value Nonrec [
      <def>
        pattern (typedef.ml[2,19+4]..typedef.ml[2,19+5])
          Ppat_var "v/1011"
        expression (typedef.ml[2,19+8]..typedef.ml[2,19+11])
          Pexp_construct "Foo" [] false
    ]
]
```

The typed AST is more explicit than the untyped syntax tree. For instance, the type declaration has been given a unique name (`t/1008`), as has the `v` value (`v/1011`).

You'll rarely need to look at this raw output from the compiler unless you're building IDE tools such as `ocp-index`, or are hacking on extensions to the core compiler itself. However, it's useful to know that this intermediate form exists before we delve further into the code generation process next in Chapter 23.

2013-07-04
10:28:31

2013-07-04
10:28:31

CHAPTER 23

# The Compiler Backend: Byte-code and Native-code

Once OCaml has passed the type checking stage, it can stop emitting syntax and type errors and begin the process of compiling the well-formed modules into executable code.

In this chapter, we'll cover the following topics:

- the untyped intermediate lambda code where pattern matching is optimized.
- the bytecode `ocamlc` compiler and `ocamlrun` interpreter.
- the native code `ocamlopt` code generator, and debugging and profiling native code.

## The untyped lambda form

The first code generation phase eliminates all the static type information into a simpler intermediate *lambda form*. The lambda form discards higher-level constructs such as modules and objects and replaces them with simpler values such as records and function pointers. Pattern matches are also analyzed and compiled into highly optimized automata.

The lambda form is the key stage that discards the OCaml type information and maps the source code to the runtime memory model described in Chapter 20. This stage also performs some optimizations, most notably converting pattern match statements into more optimized but low-level statements.

### Pattern matching optimization

The compiler dumps the lambda form in an s-expression syntax if you add the `-dlambda` directive to the command-line. Let's use this to learn more about how the OCaml pattern matching engine works by building three different pattern matches and comparing their lambda forms.

Let's start by creating a straightforward exhaustive pattern match using normal variants.

```
(* pattern_monomorphic_exhaustive.ml *)
type t = | Alice | Bob | Charlie | David

let test v =
  match v with
  | Alice   -> 100
  | Bob     -> 101
  | Charlie -> 102
  | David   -> 103
```

The lambda output for this code looks like this.

```
$ ocamlc -dlambda -c pattern_monomorphic_exhaustive.ml
(setglobal Pattern_monomorphic_exhaustive!
  (let
    (test/1013
       (function v/1014
         (switch* v/1014
          case int 0: 100
          case int 1: 101
          case int 2: 102
          case int 3: 103)))
    (makeblock 0 test/1013)))
```

It's not important to understand every detail of this internal form, but some interesting points emerge from reading it.

- There are no mention of modules or types any more. Global values are created via `setglobal` and OCaml values are constructed by `makeblock`. The blocks are the runtime values you should remember from Chapter 20.
- The pattern match has turned into a switch case that jumps to the right case depending on the header tag of `v`. Recall that variants without parameters are stored in memory as integers in the order which they appear. The pattern matching engine knows this and has transformed the pattern into an efficient jump table.
- Values are addressed by a unique name that distinguished shadowed values by appending a number (e.g. `v/1014`). The type safety checks in the earlier phase ensure that these low-level accesses never violate runtime memory safety, so this layer doesn't do any dynamic checks. Unwise use of unsafe features such as the `Obj.magic` module can still easily induce crashes at this level.

The first pattern match is *exhaustive*, so there are no unknown match cases that the compiler needs to check for (e.g. a value greater than 3). What happens if we modify the code to use an incomplete pattern match instead?

```
(* pattern_monomorphic_incomplete.ml *)
type t = | Alice | Bob | Charlie | David
```

```
let test v =
  match v with
  | Alice   -> 100
  | Bob     -> 101
  | _       -> 102
```

The lambda output for this code is now quite different.

```
$ ocamlc -dlambda -c pattern_monomorphic_incomplete.ml
(setglobal Pattern_monomorphic_incomplete!
  (let
    (test/1013
       (function v/1014 (if (!= v/1014 1) (if (!= v/1014 0) 102 100) 101)))
    (makeblock 0 test/1013)))
```

The compiler has reverted to testing the value as a set of nested conditionals. The lambda code above first checks to see if the value is `Alice`, then if it's `Bob` and finally falls back to the default `102` return value for everything else.

Exhaustive pattern matching is thus a better coding style at several levels. It rewards you with more useful compile-time warnings when you modify type definitions *and* generates more efficient runtime code too.

Finally, let's look at the same code, but with polymorphic variants instead of normal variants.

```
(* pattern_polymorphic.ml *)
let test v =
  match v with
  | `Alice   -> 100
  | `Bob     -> 101
  | `Charlie -> 102
  | `David   -> 103
```

The lambda form for this reveals the most inefficient result yet.

```
$ ocamlc -dlambda -c pattern_polymorphic.ml
(setglobal Pattern_polymorphic!
  (let
    (test/1008
       (function v/1009
         (if (>= v/1009 482771474) (if (>= v/1009 884917024) 100 102)
           (if (>= v/1009 3306965) 101 103))))
    (makeblock 0 test/1008)))
```

We mentioned earlier in Chapter 6 that pattern matching over polymorphic variants is slightly less efficient, and it should be clearer why this is the case now. Polymorphic variants have a runtime value that's calculated by hashing the variant name, and so the compiler has to test each of these possible hash values in sequence.

## Benchmarking pattern matching

Let's benchmark these three pattern matching techniques to quantify their runtime costs more accurately. The `Core_bench` module runs the tests thousands of times and also calculates statistical variance of the results. You'll need to `opam install core_bench` to get the library.

```
(* pattern.ml: benchmark different pattern matching styles *)
open Core.Std
open Core_bench.Std

type t = | Alice | Bob | Charlie | David

let polymorphic_pattern () =
  let test v =
    match v with
    | `Alice   -> 100
    | `Bob     -> 101
    | `Charlie -> 102
    | `David   -> 103
  in
  List.iter ~f:(fun v -> ignore(test v))
    [`Alice; `Bob; `Charlie; `David]

let monomorphic_pattern_exhaustive () =
  let test v =
    match v with
    | Alice   -> 100
    | Bob     -> 101
    | Charlie -> 102
    | David   -> 103
  in
  List.iter ~f:(fun v -> ignore(test v))
    [ Alice; Bob; Charlie; David ]

 let monomorphic_pattern_incomplete () =
  let test v =
    match v with
    | Alice   -> 100
    | Bob     -> 101
    | _       -> 102
  in
  List.iter ~f:(fun v -> ignore(test v))
    [ Alice; Bob; Charlie; David ]

let tests = [
    "Polymorphic pattern", polymorphic_pattern;
    "Monomorphic incomplete pattern", monomorphic_pattern_incomplete;
    "Monomorphic exhaustive pattern", monomorphic_pattern_exhaustive
]

let () =
  List.map tests ~f:(fun (name,test) -> Bench.Test.create ~name test)
```

```
|> Bench.make_command
|> Command.run
```

Building and executing this example will run for around 30 seconds by default, and you'll see the results summarised in a neat table.

```
$ ocamlbuild -use-ocamlfind -package core -package core_bench -tag thread pattern.native
Estimated testing time 30s (change using -quota SECS).
┌──────────────────────────────┬────────────┬─────────────────┬────────────┐
│ Name                         │ Time (ns) │    Time 95ci │ Percentage │
├──────────────────────────────┼────────────┼─────────────────┼────────────┤
│ Polymorphic pattern          │      22.38 │ 22.34-22.43 │      100.00 │
│ Monomorphic incomplete pattern │      20.98 │ 20.95-21.02 │       93.77 │
│ Monomorphic exhaustive pattern │      19.53 │ 19.49-19.58 │       87.25 │
└──────────────────────────────┴────────────┴─────────────────┴────────────┘
```

These results confirm our earlier performance hypothesis obtained from inspecting the lambda code. The shortest running time comes from the exhaustive pattern match and polymorphic variant pattern matching is the slowest. There isn't a hugely significant difference in these examples, but you can use the same techniques to peer into the innards of your own source code and narrow down any performance hotspots.

The lambda form is primarily a stepping stone to the bytecode executable format that we'll cover next. It's often easier to look at the textual output from this stage than to wade through the native assembly code from compiled executables.

**Learning more about pattern matching compilation**

Pattern matching is an important part of OCaml programming. You'll often encounter deeply nested pattern matches over complex data structures in real code. A good paper that describes the fundamental algorithms implemented in OCaml is "Optimizing pattern matching" (*http://dl.acm.org/citation.cfm?id=507641*) by Fabrice Le Fessant and Luc Maranget.

The paper describes the backtracking algorithm used in classical pattern matching compilation, and also several OCaml-specific optimizations such as the use of exhaustiveness information and control flow optimizations via static exceptions.

It's not essential that you understand all of this just to use pattern matching of course, but it'll give you insight as to why pattern matching is such a lightweight language construct to use in OCaml code.

## Generating portable bytecode

After the lambda form has been generated, we are very close to having executable code. The OCaml tool-chain branches into two separate compilers at this point. We'll describe the bytecode compiler first, which consists of two pieces:

- `ocamlc` compiles files into a bytecode that is a close mapping to the lambda form.
- `ocamlrun` is a portable interpreter that executes the bytecode.

The big advantage of using bytecode is simplicity, portability and compilation speed. The mapping from the lambda form to bytecode is straightforward, and this results in predictable (but slow) execution speed.

The interpreter uses the OCaml stack and an accumulator to store values. It only has seven registers in total: the program counter, stack pointer, accumulator, exception and argument pointers, and environment and global data.

You can display the bytecode instructions in textual form via `-dinstr`. Try this on one of our earlier pattern matching examples.

```
$ ocamlc -dinstr pattern_monomorphic_exhaustive.ml
    branch L2
L1: acc 0
    switch 6 5 4 3/
L6: const 100
    return 1
L5: const 101
    return 1
L4: const 102
    return 1
L3: const 103
    return 1
L2: closure L1, 0
    push
    acc 0
    makeblock 1, 0
    pop 1
    setglobal Pattern_monomorphic_exhaustive!
```

The bytecode above has been simplified from the lambda form into a set of simple instructions that are executed in serial by the interpreter.

There are around 140 instructions in total, but most are just minor variants of commonly encountered operations (e.g. function application at a specific arity). You can find full details online (*http://cadmium.x9c.fr/distrib/caml-instructions.pdf*).

**Where did the bytecode instruction set come from?**

The bytecode interpreter is much slower than compiled native code, but is still remarkably performant for an interpreter without a JIT compiler. Its efficiency can be traced back to Xavier Leroy's ground-breaking work in 1990 on "The ZINC experiment: An Economical Implementation of the ML Language" (*http://hal.inria.fr/docs/00/07/00/49/PS/RT-0117.ps*).

This paper laid the theoretical basis for the implementation of an instruction set for a strictly evaluated functional language such as OCaml. The bytecode interpreter in modern OCaml is still based on the ZINC model. The native code compiler uses a different model since it uses CPU registers for function calls instead of always passing arguments on the stack as the bytecode interpreter does.

Understanding the reasoning behind the different implementations of the bytecode interpreter and the native compiler is a very useful exercise for any budding language hacker.

## Compiling and linking bytecode

The `ocamlc` command compiles individual `ml` files into bytecode files that have a `cmo` extension. The compiled bytecode files are matched with the associated `cmi` interface which contains the type signature exported to other compilation units.

A typical OCaml library consists of multiple source files, and hence multiple `cmo` files that all need to be passed as command-line arguments to use the library from other code. The compiler can combine these multiple files into a more convenient single archive file by using the `-a` flag. Bytecode archives are denoted by the `cma` extension.

The individual objects in the library are linked as regular `cmo` files in the order specified when the library file was built. If an object file within the library isn't referenced elsewhere in the program, then it isn't included in the final binary unless the `-linkall` flag forces its inclusion. This behaviour is analogous to how C handles object files and archives (`.o` and `.a` respectively).

The bytecode files are then linked together with the OCaml standard library to produce an executable program. The order in which `.cmo` arguments are presented on the command line defines the order in which compilation units are initialized at runtime. Remember that OCaml has no single `main` function like C, so this link order is more important than in C programs.

## Executing bytecode

The bytecode runtime comprises three parts: the bytecode interpreter, garbage collector, and a set of C functions that implement the primitive operations. The bytecode contains instructions to call these C functions when required.

The OCaml linker produces bytecode that targets the standard OCaml runtime by default, and so needs to know about any C functions that are referenced from other libraries that aren't loaded by default.

Information about these extra libraries can be specified while linking a bytecode archive.

```
$ ocamlc -a -o mylib.cma a.cmo b.cmo -dllib -lmylib
```

The `dllib` flag embeds the arguments in the archive file. Any subsequent packages linking this archive will also include the extra C linking directive. This in turn lets the interpreter dynamically load the external library symbols when it executes the bytecode.

You can also generate a complete standalone executable that bundles the `ocamlrun` interpreter with the bytecode in a single binary. This is known as a *custom runtime* mode and is built as follows.

```
$ ocamlc -a -o mylib.cma -custom a.cmo b.cmo -cclib -lmylib
```

The custom mode is the most similar mode to native code compilation, as both generate standalone executables. There are quite a few other options available for compiling bytecode (notably with shared libraries or building custom runtimes). Full details can be found in the manual (*http://caml.inria.fr/pub/docs/manual-ocaml/manual022.html*).

## Embedding OCaml bytecode in C

A consequence of using the bytecode compiler is that the final link phase must be performed by `ocamlc`. However, you might sometimes want to embed your OCaml code inside an existing C application. OCaml also supports this mode of operation via the `-output-obj` directive.

This mode causes `ocamlc` to output a C object file that containing the bytecode for the OCaml part of the program, as well as a `caml_startup` function. All of the OCaml modules are linked into this object file as bytecode, just as they would be for an executable.

This object file can then be linked with C code using the standard C compiler, and only needs the bytecode runtime library (which is installed as `libcamlrun.a`). Creating an executable just requires you to link the runtime library with the bytecode object file. Here's an example to show how it all fits together.

Create two OCaml source files that contain a single print line.

```
$ cat embed_me1.ml
let () = print_endline "hello embedded world 1"
$ cat embed_me2.ml
let () = print_endline "hello embedded world 2"
```

Next, create a C file which will be your main entry point.

```
/* main.c */
#include <stdio.h>
#include <caml/alloc.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/callback.h>

int
main (int argc, char **argv)
{
  puts("Before calling OCaml");
  caml_startup (argv);
  puts("After calling OCaml");
  return 0;
}
```

Now compile the OCaml files into a standalone object file.

```
$ ocamlc -output-obj -o embed_out.o embed_me1.ml embed_me2.ml
```

After this point, you no longer need the OCaml compiler, as `embed_out.o` has all of the OCaml code compiled and linked into a single object file. Compile an output binary using gcc to test this out.

```
$ gcc -Wall -I `ocamlc -where` -L `ocamlc -where` -lcamlrun -ltermcap \
  -o final_out embed_out.o main.c
$ ./final_out
Before calling OCaml
hello embedded world 1
hello embedded world 2
After calling OCaml
```

You can inspect the commands that `ocamlc` is invoking by adding `-verbose` to the command line to help figure out the GCC command-line if you get stuck. You can even obtain the C source code to the `-output-obj` result by specifying a `.c` output file extension instead of the `.o` we used earlier.

```
$ ocamlc -output-obj -o embed_out.c embed_me1.ml embed_me2.ml
$ cat embed_out.c
```

Embedding OCaml code like this lets you write OCaml that interfaces with any environment that works with a C compiler. You can even cross back from the C code into OCaml by using the `Callback` module to register named entry points in the OCaml code. This is explained in detail in the interfacing with C (*http://caml.inria.fr/pub/docs/manual-ocaml/manual033.html#toc149*) section of the OCaml manual.

## Compiling fast native code

The native code compiler is ultimately the tool that most production OCaml code goes through. It compiles the lambda form into fast native code executables, with cross-module inlining and additional optimization passes that the bytecode interpreter doesn't perform. Care is taken to ensure compatibility with the bytecode runtime, so the same code should run identically when compiled with either toolchain.

The `ocamlopt` command is the frontend to the native code compiler, and has a very similar interface to `ocamlc`. It also accepts `ml` and `mli` files, but compiles them to:

- A `.o` file containing native object code.
- A `.cmx` file containing extra information for linking and cross-module optimization.
- A `.cmi` compiled interface file that is the same as the bytecode compiler.

When the compiler links modules together into an executable, it uses the contents of the `cmx` files to perform cross-module inlining across compilation units. This can be a significant speedup for standard library functions that are frequently used outside of their module.

Collections of `.cmx` and `.o` files can also be be linked into a `.cmxa` archive by passing the `-a` flag to the compiler. However, unlike the bytecode version, you must keep the individual `cmx` files in the compiler search path so that they are available for cross-module inlining. If you don't do this, the compilation will still succeed, but you will have missed out on an important optimization and have slower binaries.

### Inspecting assembly output

The native code compiler generates assembly language that is then passed to the system assembler for compiling into object files. You can get `ocamlopt` to output the assembly by passing the `-S` flag to the compiler command-line.

The assembly code is highly architecture specific, so the discussion below assumes an Intel or AMD 64-bit platform. We've generated the example code using `-inline 20` and `-nodynlink` since it's best to generate assembly code with the full optimizations that the compiler supports. Even though these optimizations make the code a bit harder to read, it will give you a more accurate picture of what executes on the CPU. Don't forget that you can use the lambda code from earlier to get a slightly higher level picture of the code if you get lost in the more verbose assembly.

#### The impact of polymorphic comparison

We warned you earlier in Chapter 13 that using polymorphic comparison is both convenient and perilous. Let's look at precisely what the difference is at the assembly language level now.

First create a comparison function where we've explicitly annotated the types, so the compiler knows that only integers are being compared.

```
(* compare_mono.ml *)
let cmp (a:int) (b:int) =
  if a > b then a else b
```

Now compile this into assembly and read the resulting `compare_mono.S` file.

```
$ ocamlopt -inline 20 -nodynlink -S compare_mono.ml
$ cat compare_mono.S
```

If you've never seen assembly language before then the contents may be rather scary. While you'll need to learn x86 assembly to fully understand it, we'll try to give you some basic instructions to spot patterns in this section. The excerpt of the implementation of the `cmp` function can be found below.

```
_camlCompare_mono__cmp_1008:
    .cfi_startproc
.L101:
    cmpq    %rbx, %rax
    jle .L100
    ret
    .align  2
.L100:
    movq    %rbx, %rax
    ret
    .cfi_endproc
```

The `_camlCompare_mono__cmp_1008` is an assembly label that has been computed from the module name (`Compare_mono`) and the function name (`cmp_1008`). The numeric suffix for the function name comes straight from the lambda form (which you can inspect using `-dlambda`, but in this case isn't necessary).

The arguments to `cmp` are passed in the `%rbx` and `%rax` registers, and compared using the `jle` "jump if less than or equal" instruction. This requires both the arguments to be immediate integers to work. Now let's see what happens if our OCaml code omits the type annotations and is a polymorphic comparison instead.

```
(* compare_poly.ml *)
let cmp a b =
  if a > b then a else b
```

Compiling this code with `-S` results in a significantly more complex assembly output for the same function.

```
_camlCompare_poly__cmp_1008:
        .cfi_startproc
        subq    $24, %rsp
        .cfi_adjust_cfa_offset  24
```

```
.L101:
        movq    %rax, 8(%rsp)
        movq    %rbx, 0(%rsp)
        movq    %rax, %rdi
        movq    %rbx, %rsi
        leaq    _caml_greaterthan(%rip), %rax
        call    _caml_c_call
.L102:
        leaq    _caml_young_ptr(%rip), %r11
        movq    (%r11), %r15
        cmpq    $1, %rax
        je      .L100
        movq    8(%rsp), %rax
        addq    $24, %rsp
        .cfi_adjust_cfa_offset  -24
        ret
        .cfi_adjust_cfa_offset  24
        .align  2
.L100:
        movq    0(%rsp), %rax
        addq    $24, %rsp
        .cfi_adjust_cfa_offset  -24
        ret
        .cfi_adjust_cfa_offset  24
        .cfi_endproc
```

The `.cfi` directives are assembler hints that contain Call Frame Information that lets the GNU debugger provide more sensible backtraces, and have no effect on runtime performance. Notice that the rest of the implementation is no longer a simple register comparison. Instead, the arguments are pushed on the stack (the `%rsp` register) and a C function call is invoked by placing a pointer to `caml_greaterthan` in `%rax` and jumping to `caml_c_call`.

OCaml on 64-bit Intel architectures caches the location of the minor heap in the `%r11` register since it's so frequently referenced in OCaml functions. This register isn't guaranteed to be preserved when calling into C code (which can clobber `%r11` for its own purposes), and so `%r11` is restored after returning from the `caml_greaterthan` call. Finally the return value of the comparison is popped from the stack and returned.

**Reading the implementation of the C primitives**

If you have a copy of the OCaml source tree handy, it's worth reading through the definition of `caml_greaterthan()`. The built-in primitives for polymorphic comparison can be found in `caml/byterun/compare.c`.

The key function is `compare_val()`, which directly examines the runtime representation of two OCaml values to decide which is greater. This requires the header tag to be examined, and recursive structures must be tested step-by-step.

Avoiding running all of this code is why you should try to write explicit comparison functions in OCaml instead.

### Benchmarking polymorphic comparison

You don't have to fully understand the intricacies of assembly language to see that this polymorphic comparison is much heavier than the simple monomorphic integer comparison from earlier. Let's confirm this hypothesis again by writing a quick `Core_bench` test with both functions.

```
$ cat bench_poly_and_mono.ml
open Core.Std
open Core_bench.Std

let polymorphic_compare () =
  let cmp a b = if a > b then a else b in
  for i = 0 to 1000 do
    ignore(cmp 0 i)
  done

let monomorphic_compare () =
  let cmp (a:int) (b:int) =
    if a > b then a else b in
  for i = 0 to 1000 do
    ignore(cmp 0 i)
  done

let tests = [
    "Polymorphic comparison", polymorphic_compare;
    "Monomorphic comparison", monomorphic_compare ]

let () =
  List.map tests ~f:(fun (name,test) -> Bench.Test.create ~name test)
  |> Bench.make_command
  |> Command.run
```

Running this shows quite a significant runtime difference between the two.

```
$ ./bench_poly_and_mono.native
Estimated testing time 20s (change using -quota SECS).
┌──────────────────────────┬──────────┬─────────────────┬────────────┐
│ Name                     │ Time (ns) │      Time 95ci │ Percentage │
├──────────────────────────┼──────────┼─────────────────┼────────────┤
│ Polymorphic comparison │     10_087 │ 10_080-10_096 │      100.00 │
│ Monomorphic comparison │     585.51 │ 584.60-586.57 │        5.80 │
└──────────────────────────┴──────────┴─────────────────┴────────────┘
```

We see that the polymorphic comparison is close to 20 times slower! These results shouldn't be taken too seriously as this is a very narrow test, which like all such microbenchmarks aren't representative of more complex codebases. However, if you're building numerical code that runs many iterations in a tight inner loop, it's worth manually peering at the produced assembly code to see if you can hand-optimize it.

## Debugging native code binaries

The native code compiler builds executables that can be debugged using conventional system debuggers such as GNU `gdb`. You need to compile your libraries with the `-g` option to add the debug information to the output, just as you need to with C compilers.

Extra debugging information is inserted into the output assembly when the library is compiled in debug mode. These include the CFI stubs you will have noticed in the profiling output earlier (`.cfi_start_proc` and `.cfi_end_proc` to delimit an OCaml function call, for example).

### Understanding name mangling

So how do you refer to OCaml functions into an interactive debugger like `gdb`? The first thing you need to know is how function names compile down into C symbols; a procedure generally called *name mangling*.

Each OCaml source file is compiled into a native object file that must export a unique set of symbols to comply with the C binary interface. This means that any OCaml values that may be used by another compilation unit need to be mapped into a symbol name. This mapping has to account for OCaml language features such as nested modules, anonymous functions and variable names that shadow each other.

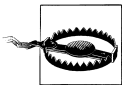
The conversion follows some straightforward rules for named variables and functions:

- The symbol is prefixed by `caml` and the local module name, with dots replaced by underscores.
- This is followed by a double __ suffix and the variable name.
- The variable name is also suffixed by a _ and a number. This is the result of the lambda compilation that replaces each variable name with a unique value within the module. You can determine this number by examining the `-dlambda` output from `ocamlopt`.

Anonymous functions are hard to predict without inspecting intermediate compiler output. If you need to debug them it's usually easier to modify the source code to let-bind the anonymous function to a variable name.

### Interactive breakpoints with the GNU debugger

Let's see name mangling in action with some interactive debugging in the GNU `gdb` debugger.



**Beware gdb on MacOS X**

The examples here assume that you are running `gdb` on either Linux or FreeBSD. MacOS X does have `gdb` installed, but it's a rather quirky experience that doesn't reliably interpret the debugging information contained in the native binaries. This can result in function names showing up as raw symbols such as `.L101` instead of their more human-readable form.

For OCaml 4.1, we'd recommend you do native code debugging on an alternate platform such as Linux, or manually look at the assembly code output to map the symbol names onto their precise OCaml functions.

Let's write a mutually recursive function that selects alternating values from a list. This isn't tail recursive and so our stack size will grow as we single-step through the execution.

```
(* alternate_list.ml : select every other value from an input list *)
open Core.Std

let rec take =
  function
  |[] -> []
  |hd::tl -> hd :: (skip tl)
and skip =
  function
  |[] -> []
  |hd::tl -> take tl

let () =
  take [1;2;3;4;5;6;7;8;9]
  |> List.map ~f:string_of_int
  |> String.concat ~sep:","
  |> print_endline
```

Compile and run this with debugging symbols. You should see the following output:

```
$ ocamlfind ocamlopt -g -package core -thread -linkpkg -o alternate alternate_list.ml
$ ./alternate
1,3,5,7,9
```

Now we can run this interactively within `gdb`.

```
$ gdb ./alternate
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
```

```
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/avsm/alternate...done.
(gdb)
```

The `gdb` prompt lets you enter debug directives. Let's set the program to break just before the first call to `take`.

```
(gdb) break camlAlternate_list__take_69242
Breakpoint 1 at 0x5658d0: file alternate_list.ml, line 5.
```

We used the C symbol name by following the name mangling rules defined earlier. A convenient way to figure out the full name is by tab-completion. Just type in a portion of the name and press the `<tab>` key to see a list of possible completions.

Once you've set the breakpoint, start the program executing.

```
(gdb) run
Starting program: /home/avsm/alternate
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, camlAlternate_list__take_69242 () at alternate_list.ml:5
4     function
```

The binary has run until the first take invocation and stopped, waiting for further instructions. GDB has lots of features, so let's continue the program and check the stacktrace after a couple of recursions.

```
(gdb) cont
Continuing.

Breakpoint 1, camlAlternate_list__take_69242 () at alternate_list.ml:5
4     function
(gdb) cont
Continuing.

Breakpoint 1, camlAlternate_list__take_69242 () at alternate_list.ml:5
4     function
(gdb) bt
#0  camlAlternate_list__take_69242 () at alternate_list.ml:4
#1  0x00000000005658e7 in camlAlternate_list__take_69242 () at alternate_list.ml:6
#2  0x00000000005658e7 in camlAlternate_list__take_69242 () at alternate_list.ml:6
#3  0x00000000005659f7 in camlAlternate_list__entry () at alternate_list.ml:14
#4  0x0000000000560029 in caml_program ()
#5  0x000000000080984a in caml_start_program ()
#6  0x00000000008099a0 in ?? ()
#7  0x0000000000000000 in ?? ()
(gdb) clear camlAlternate_list__take_69242
Deleted breakpoint 1
(gdb) cont
Continuing.
1,3,5,7,9
[Inferior 1 (process 3546) exited normally]
```

The `cont` command resumes execution after a breakpoint has paused it, `bt` displays a stack backtrace, and `clear` deletes the breakpoint so that the application can execute until completion. GDB has a host of other features we won't cover here, but you view more guidelines via Mark Shinwell's talk on "Real-world debugging in OCaml" (*http://www.youtube.com/watch?v=NF2WpWnB-nk*<).

One very useful feature of OCaml native code is that C and OCaml both share the same stack. This means that GDB backtraces can give you a combined view of what's going on in your program *and* runtime library. This includes any calls to C libraries or even callbacks into OCaml from the C layer if you're in an embedded environment.

## Profiling native code

The recording and analysis of where your application spends its execution time is known as *performance profiling*. OCaml native code binaries can be profiled just like any other C binary, by using the name mangling described earlier to map between OCaml variable names and the profiler output.

Most profiling tools benefit from having some instrumentation included in the binary. OCaml supports two such tools:

- GNU Gprof to measure execution time and call graphs.
- The Perf (*https://perf.wiki.kernel.org/*) profiling framework in modern versions of Linux.

### Gprof

Gprof produces an execution profile of an OCaml program by recording a call graph of which functions call each other, and recording the time these calls take during the program execution.

Getting precise information out of Gprof requires passing the `-p` flag to the native code compiler when compiling *and* linking the binary. This generates extra code that records profile information to a file called `gmon.out` when the program is executed. This profile information can then be examined using Gprof.

### Perf

Perf is a more modern alternative to Gprof that doesn't require you to instrument the binary. Instead, it uses hardware counters and debug information within the binary to record information accurately.

Run Perf on a compiled binary to record information first. We'll use our write barrier benchmark from earlier which measures memory allocation versus in-place modification.

```
$ perf record -g ./barrier.native
```

```
Estimated testing time 20s (change using -quota SECS).
┌──────────┬──────────┬──────────────────────┬────────────┐
│ Name     │ Time (ns)│           Time 95ci  │ Percentage │
├──────────┼──────────┼──────────────────────┼────────────┤
│ mutable  │ 7_306_219│ 7_250_234-7_372_469  │      96.83 │
│ immutable│ 7_545_126│ 7_537_837-7_551_193  │     100.00 │
└──────────┴──────────┴──────────────────────┴────────────┘
[ perf record: Woken up 11 times to write data ]
[ perf record: Captured and wrote 2.722 MB perf.data (~118926 samples) ]
```

When this completes, you can interactively explore the results.

```
$ perf report -g
+  48.86%  barrier.native  barrier.native     [.] camlBarrier__test_immutable_69282
+  30.22%  barrier.native  barrier.native     [.] camlBarrier__test_mutable_69279
+  20.22%  barrier.native  barrier.native     [.] caml_modify
```

This trace broadly reflects the results of the benchmark itself. The mutable benchmark consists of the combination of the call to `test_mutable` and the `caml_modify` write barrier function in the runtime. This adds up to slightly over half the execution time of the application.

Perf has a growing collection of other commands that let you archive these runs and compare them against each other. You can read more on the homepage (*http://perf.wiki.kernel.org*).

### Using the frame-pointer to get more accurate traces

Although Perf doesn't require adding in explicit probes to the binary, it does need to understand how to unwind function calls so that the kernel can accurately record the function backtrace for every event.

OCaml stack frames are too complex for Perf to understand directly, and so it needs the compiler to fall back to using the same conventions as C for function calls. On 64-bit Intel systems, this means that a special register known as the *frame pointer* is used to record function call history.

Using the frame pointer in this fashion means a slowdown (typically around 3-5%) since it's no longer available for general-purpose use. OCaml 4.1 thus makes the frame pointer an optional feature that can be used to improve the resolution of Perf traces.

OPAM provides a compiler switch that compiles OCaml with the frame pointer activated.

```
$ opam switch 4.01.0dev+fp
```

Using the frame pointer changes the OCaml calling convention, but OPAM takes care of recompiling all your libraries with the new interface. You can read more about this on the OCamlPro blog (*http://www.ocamlpro.com/blog/2012/08/08/profile-native-code.html*).

## Embedding native code in C

The native code compiler normally links a complete executable, but can also output a standalone native object file just as the bytecode compiler can. This object file has no further dependencies on OCaml except for the runtime library.

The native code runtime is a different library from the bytecode one and is installed as `libasmrun.a` in the OCaml standard library directory.

Try this custom linking by using the same source files from the bytecode embedding example earlier in this chapter.

```
$ ocamlopt -output-obj -o embed_native.o embed_me1.ml embed_me2.ml
$ gcc -Wall -I `ocamlc -where` -L `ocamlc -where` -lasmrun -ltermcap \
  -o final_out_native embed_native.o main.c
./final_out_native
Before calling OCaml
hello embedded world 1
hello embedded world 2
After calling OCaml
```

The `embed_native.o` is a standalone object file that has no further references to OCaml code beyond the runtime library, just as with the bytecode runtime.

#### Activating the debug runtime

Despite your best efforts, it is easy to introduce a bug into some components such as C bindings that cause heap invariants to be violated. OCaml includes a `libasmrund.a` variant of the runtime library that is compiled with extra debugging checks that perform extra memory integrity checks during every garbage collection cycle. Running these extra checks will abort the program nearer the point of corruption and help isolate the bug in the C code.

To use the debug library, just link your program with the `-runtime-variant d` flag.

```
$ ocamlopt -runtime-variant d -verbose -o hello hello.ml hello_stubs.c
$ ./hello
### OCaml runtime: debug mode ###
Initial minor heap size: 2048k bytes
Initial major heap size: 992k bytes
Initial space overhead: 80%
Initial max overhead: 500%
Initial heap increment: 992k bytes
Initial allocation policy: 0
Hello OCaml World!
```

If you get an error that `libasmrund.a` is not found, then this is probably because you're using OCaml 4.00 and not 4.01. It's only installed by default in the very latest version, which you should be using via the `4.01.0dev+trunk` OPAM switch.

## Summarising the file extensions

We've seen how the compiler uses intermediate files to store various stages of the compilation toolchain. Here's a cheat sheet of all them in one place.

Here are the intermediate files generated by `ocamlc`:

| Extension | Purpose |
|---|---|
| .ml | Source files for compilation unit module implementations. |
| .mli | Source files for compilation unit module interfaces. If missing, generated from the `.ml` file. |
| .cmi | Compiled module interface from a corresponding `.mli` source file. |
| .cmo | Compiled bytecode object file of the module implementation. |
| .cma | Library of bytecode object files packed into a single file. |
| .o | C source files are compiled into native object files by the system `cc`. |
| .cmt | Typed abstract syntax tree for module implementations. |
| .cmi | Typed abstract syntax tree for module interfaces. |
| .annot | Old-style annotation file for displaying typed, superseded by `cmt` files. |

The native code compiler generates some additional files.

| Extension | Purpose |
|---|---|
| .o | Compiled native object file of the module implementation. |
| .cmx | Contains extra information for linking and cross-module optimization of the object file. |
| .cmxa and .a | Library of `cmx` and `o` units, stored in the `cmxa` and `a` files respectively. These files are always needed together. |
| .S | Assembly language output if `-S` is specified. |

2013-07-04
10:28:31

APPENDIX A

# Installation

The easiest way to use OCaml is via the binary packages available for many operating systems. For day-to-day code development however, it's much easier to use a source-code manager that lets you modify individual libraries and automatically recompile all the dependencies.

An important difference between OCaml and scripting languages such as Python or Ruby is the static type safety that means that you can't just mix-and-match compiled libraries. Interfaces are checked when libraries are compiled, so when an interface is changed, all the dependent libraries must also be recompiled. Source-based package managers automate this process for you and make development life much easier.

To work through Real World OCaml, you'll need three major components installed:

- The OCaml compiler itself.
- The OPAM source package manager, through which we'll install several extra libraries.
- The `utop` interactive toplevel, a modern interactive toplevel with command history and tab completion.

Let's get started with how to install OCaml on various operating systems, and we'll get OPAM and `utop` running after that.

## Getting OCaml

The OCaml compiler is available as a binary distribution on many operating systems. This is the simplest and preferred installation route, but we'll also describe how to do a manual installation as a last resort.

### Mac OS X

The Homebrew (*http://github.com/mxcl/homebrew*) package manager has an OCaml installer, which is usually updated pretty quickly to the latest stable release. Make sure

that you have the latest XCode (and Command Line Tools for XCode) installed from the App Store before starting the OCaml installation.

```
$ brew install ocaml
$ brew install pcre
```

The Perl-compatible Regular Expression library (PCRE) is used by the Core suite. It's not strictly needed to use OCaml, but is a commonly used library that we're installing now to save time later.

Another popular package manager on Mac OS X is MacPorts (*http://macports.org*), which also has an OCaml port. As with Homebrew, make sure you have XCode installed and have followed the rest of the MacPorts installation instructions, and then type in:

```
$ sudo port install ocaml
$ sudo port install ocaml-pcre
```

## Debian Linux

On Debian Linux, you should install OCaml via binary packages. You'll need at least OCaml version 3.12.1 to bootstrap OPAM, which means using Debian Wheezy or greater. Don't worry about getting the absolute latest version of the compiler, as you just need one new enough to compile the OPAM package manager, after which you'll use OPAM to manage your compiler installation.

```
$ sudo apt-get install ocaml ocaml-native-compilers camlp4-extra
$ sudo apt-get install git libpcre3-dev curl build-essential m4
```

Notice that we've installed a few more packages than just the OCaml compiler here. The second command line installs enough system packages to let you build your own OCaml packages. You may find that some OCaml libraries require more system libraries (for example, `libssl-dev`), but we'll highlight these in the book when we introduce the library.

## Fedora and Red Hat

OCaml has been included in the basic distribution since Fedora 8. To install the latest compiler, just run:

```
# yum install ocaml
# yum install pcre-devel
```

The PCRE package is used by Core and is just included here for convenience later.

## Arch Linux

Arch Linux provides OCaml 4.00.1 (or later) in the standard repositories, so the easiest method of installation is using `pacman`:

```
$ pacman -Sy ocaml
```

## Windows

Windows is not currently supported by the examples in Real World OCaml, although it is being worked on. Until that's ready, we recommend using a virtual machine running Debian Linux on your local machine.

## Building from source

To install OCaml from source code, first make sure that you have a C compilation environment (usually either `gcc` or `llvm` installed).

```
$ curl -OL https://github.com/ocaml/ocaml/archive/trunk.tar.gz
$ tar -zxvf trunk.tar.gz
$ cd ocaml-trunk
$ ./configure
$ make world world.opt
$ sudo make install
```

The final step requires administrator privilege to install in your system directory. You can also install it in your home directory by passing the `prefix` option to the configuration script:

```
$ ./configure -prefix $HOME/my-ocaml
```

Once the installation is completed into this custom location, you will need to add `$HOME/my-ocaml/bin` to your `PATH`, normally by editing the `~/.bash_profile` file. You shouldn't really to do this unless you have special reasons, so try to install binary packages before trying a source installation.

**Note to reviewers**

We instruct you install the unreleased trunk version of OCaml in these instructions, as we take advantage of some recent additions to the language that simplify explanations in the book. The 4.01 release will happen before the book is released, but you may run into "bleeding edge" bugs with the trunk release. Leave a comment here if you do and we'll address them.

# Getting OPAM

OPAM manages multiple simultaneous OCaml compiler and library installations, tracks library versions across upgrades, and recompiles dependencies automatically if they get out of date. It's used throughout Real World OCaml as the mechanism to retrieve and use third-party libraries.

Before installing OPAM, make sure that you have the OCaml compiler installed as described above. Once installed, the entire OPAM database is held in your home directory (normally `$HOME/.opam`). If something goes wrong, just delete this `.opam` directory and start over from a clean slate. If youre using a beta version of OPAM, please upgrade it to at least version 1.0.0 or greater before proceeding.

## Mac OS X

Source installation of OPAM will take a minute or so on a modern machine. There is a Homebrew package for the latest OPAM:

```
$ brew update
$ brew install opam
```

And on MacPorts, install it like this:

```
$ sudo port install opam
```

## Debian Linux

OPAM has recently been packaged for Debian and will soon be part of the unstable distribution. If you're on an earlier stable distribution such as `wheezy`, you can either compile from source, or cherry-pick just the OPAM binary package from `unstable` by:

```
# apt-get update
# apt-get -t unstable install opam
```

**Note to reviewers**

The binary packages for OPAM are not yet available as of the 17th June 2013, but the package is in the `NEW` queue. It should be available by the the time the book is released, and these instructions will be updated accordingly.

## Fedora and Red Hat

There is currently no RPM available for Fedora or Red Hat, so please install OPAM via the source code instructions for the moment.

### Arch Linux

OPAM is available in the Arch User Repository (AUR) in two packages. You'll need both `ocaml` and the `base-devel` packages installed first:

- `opam` contains the most recent stable release, and is the recommended package.
- `opam-git` builds the package from the latest upstream source, and should only be used if you are looking for a specific bleeding-edge feature.

Run these commands to install the stable OPAM package:

```
$ sudo pacman -Sy base-devel
$ wget https://aur.archlinux.org/packages/op/opam/opam.tar.gz
$ tar -xvf opam.tar.gz && cd opam
$ makepkg
$ sudo pacman -U opam-_version_.pkg.tar.gz
```

### Source Installation

If the binary packages aren't available for your system, you'll need to install the latest OPAM release from source. The distribution only requires the OCaml compiler to be installed, so this should be straightforward. Download the latest version from the homepage (*https://github.com/OCamlPro/opam/tags*).

```
$ curl -OL https://github.com/OCamlPro/opam/archive/latest.tar.gz
$ tar -zxvf latest.tar.gz
$ cd opam-latest
$ ./configure && make
$ sudo make install
```

**Note to reviewers**

The OPAM instructions will be simplified when integrated upstream into Debian and Fedora, which is ongoing. Until then, we're leaving source-code installation instructions here. Please leave a comment with any amended instructions you encounter.

## Configuring the OPAM package manager

The entire OPAM package database is held in the `.opam` directory in your home directory, including compiler installations. On Linux and Mac OS X, this will be the `~/.opam` directory. You shouldn't switch to an admin user to install packages as nothing will be installed outside of this directory. If you run into problems, just delete the whole `~/.opam` directory and follow the installations instructions from the `opam init` stage again.

Let's begin by initialising the OPAM package database. This will require an active Internet connection, and ask you a few interactive questions at the end. It's safe to answer yes to these unless you want to manually control the configuration steps yourself as an advanced user.

```
$ opam init
<...>
=-=-=-= Configuring OPAM =-=-=-=
Do you want to update your configuration to use OPAM ? [Y/n] y
[1/4] Do you want to update your shell configuration file ? [default: ~/.profile] y
[2/4] Do you want to update your ~/.ocamlinit ? [Y/n] y
[3/4] Do you want to install the auto-complete scripts ? [Y/n] y
[4/4] Do you want to install the `opam-switch-eval` script ? [Y/n] y
User configuration:
  ~/.ocamlinit is already up-to-date.
  ~/.profile is already up-to-date.
Gloabal configuration:
  Updating <root>/opam-init/init.sh
    auto-completion : [true]
    opam-switch-eval: [true]
  Updating <root>/opam-init/init.zsh
    auto-completion : [true]
    opam-switch-eval: [true]
  Updating <root>/opam-init/init.csh
    auto-completion : [true]
    opam-switch-eval: [true]
```

You only need to run this command once, and it will create the `~/.opam` directory and sync with the latest package list from the online OPAM database.

When the `init` command finishes, you'll see some instructions about environment variables. OPAM never installs files into your system directories (which would require administrator privileges). Instead, it puts them into your home directory by default, and can output a set of shell commands which configures your shell with the right `PATH` variables so that packages will just work. This requires just one command:

```
$ eval `opam config -env`
```

This evaluates the results of running `opam config env` in your current shell, and sets the variables so that subsequent commands will use them. This only works with your current shell, and it can be automated for all future shells by adding the line to your login scripts. On Mac OS X or Debian, this is usually the `~/.bash_profile` file if you're using the default shell. If you've switched to another shell, it might be `~/.zshrc` instead. OPAM isn't unusual in this approach; the SSH `ssh-agent` also works similarly, so if you're having any problems just hunt around in your configuration scripts to see how that's being invoked.

If you answered `yes` to the auto-complete scripts question during `opam init`, this should have all been set up for you. You can verify this worked by listing the available packages:

```
$ opam list
```

**Note to reviewers**

OPAM 1.0.0 places the login commands into your `~/.profile` directory, which isn't always executed if your shell is `bash`. This has been fixed in subsequent versions, but for now you'll need to manually copy the contents of `~/.profile` over to `~/.bash_profile` via:

```
$ cat ~/.profile >> ~/.bash_profile
```

The most important package we need to install is Core, which is the replacement standard library that all of the examples in this book use. Before doing this, let's make sure you have exactly the right compiler version you need. We've made some minor modifications to the way the OCaml compiler displays type signatures, and the next command will install a patched `4.01.0` compiler with this functionality enabled.

```
$ opam switch 4.01.0dev+trunk
```

This step will take about 5-10 minutes on a modern machine, and will download and install (within the `~/.opam` directory) a custom OCaml compiler. OPAM supports multiple such installations, and you'll find this very useful if you ever decide to hack on the internals of the compiler itself, or you want to experiment with the latest release without sacrificing your current installation. You only need to install this compiler once, and future updates will be much faster as they only recompile libraries within the compiler installation.

The new compiler will be installed into `~/.opam/4.01.0dev+trunk` and any libraries you install for it will be tracked separately from your system installation. You can have any number of compilers installed simultaneously, but only one can be active at any time. Browse through the available compilers by running `opam switch list`.

Finally, we're ready to install the Core libraries. Run this:

```
$ opam install core core_extended async
```

This will take about five or ten minutes to build, and will install a series of packages. OPAM figures out the dependencies you need automatically, but the three packages that really matter are:

- `core` is the main, well-supported Core distribution from Jane Street.
- `core_extended` contains a number of experimental, but useful, extension libraries that are under review for inclusion in Core. We use some of these in places, but much less than Core itself.

- `async` is the network programming library that we use in Part II to communicate with other hosts. You can skip this for the initial installation until you get to Part II, if you prefer.

## Editing Environment

There's one last tool you need before getting started on the examples. The default `ocaml` command gives us an interactive command-line to experiment with code without compiling it. However, it's quite a spartan experience and so we use a more modern alternative.

```
$ opam install utop
```

The `utop` package is an interactive command-line interface to OCaml that has tab-completion, persistent history and integration with Emacs so that you can run it within your editing environment.

Remember from earlier that OPAM never installs files directly into your system directories, and this applies to `utop` too. You'll find the binary in `~/.opam/4.01.0dev+trunk/bin`. However, just typing in `utop` from your shell should just work, due to the `opam config env` step that configures your shell. Don't forget to automate this as described earlier, as it makes life much easier when developing OCaml code!

## Command Line

The `utop` tool provides a convenient interactive toplevel, with full command history, command macros and module name completion. When you first run `utop`, you'll find yourself at an interactive prompt with a bar at the bottom of the screen. The bottom bar dynamically updates as you write text, and contains the possible names of modules or variables that are valid at that point in the phrase you are entering. You can press the `<tab>` key to complete the phrase with the first choice.

The `~/.ocamlinit` file in your home directory initialises `utop` with common libraries and syntax extensions so you don't need to type them in every time. Now that you have Core installed, you should update it to load it every time you start `utop`, by adding this to it:

```
#use "topfind"
#camlp4o
#require "core.top"
#require "core.syntax"
```

You can also optionally add some more useful libraries that are used in the book, such as `Core_extended` and `Async`. You can also open the Core module by default if that's all you ever use the top-level for. Just append these lines to the `.ocamlinit` file if that's what you prefer.

```
#require "core_extended"
#require "async"
open Core.Std
```

When you run `utop` with these initialization rules, it should start up with Core opened and ready to use.

## Editors

**Note to reviewers**

The instructions for editor setup are still being compiled. If you have a relevant tip or HOWTO, then we'd *really* appreciate you leaving a note here with a pointer or direct instructions.

### Emacs

TODO: Emacs users have tuareg and Typerex (*http://www.typerex.org/*).

To use `utop` directly in Emacs, add the following line to your `~/.emacs` file:

```
(autoload 'utop "utop" "Toplevel for OCaml" t)
```

You also need to make the `utop.el` file available to your Emacs installation. The OPAM version of `utop` installs it into the `~/.opam` hierarchy, for example in `~/.opam/system/share/emacs/site-lisp/utop.el`. You may need to replace `system` with your current compiler switch, such as `4.01.0dev+trunk`.

Once this successfully loads in Emacs, you can run utop by executing the command `utop` in Emacs. There are more details instructions at the utop homepage (*https://github.com/diml/utop#integration-with-emacs*).

### Vim

TODO: Vim users can use the built-in style, and ocaml-annot (*http://github.com/avsm/ocaml-annot*) may also be useful.

### Eclipse

Eclipse is a popular IDE usually used for Java development. The OCaml Development Tools (ODT) project provides equivalent IDE features for editing and compiling OCaml code, such as automatic compilation and name completion.

ODT is distributed as a set of plugins for the Eclipse IDE environment from the homepage (*http://ocamldt.free.fr*). You just have to copy these plugins into your Eclipse distribution in order to access the new OCaml facilities.

2013-07-04
10:28:31

# APPENDIX B
# Packaging

**Note to reviewers**

This chapter hasn't been completed yet. Since this content can change rather fast, we may place this information online rather than directly in the final book.

## The OCamlfind frontend

## Distributing applications with OPAM

2013-07-04
10:28:31