2013-05-28
15:22:23

Real World OCaml

Jason Hickey, Anil Madhavapeddy, and Yaron Minsky

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

2013-05-28
15:22:23

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

Copyright © 2010 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: Mike Loukides
Production Editor:
Copyeditor:
Proofreader: FIX ME!

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

March 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

[?]

1369768942

2013-05-28
15:22:23

Table of Contents

Preface ..... **xi**

Prologue ..... **xv**

Part I. Language Concepts

1. A Guided Tour **3**
OCaml as a calculator 4
Functions and type Inference 5
Type inference 6
Inferring generic types 7
Tuples, Lists, Options and Pattern Matching 9
Tuples 9
Lists 10
Options 15
Records and Variants 17
Imperative programming 19
Arrays 19
Mutable record fields 19
Refs 21
For and while loops 22
A complete program 23
Compiling and running 23
Where to go from here 24

2. Variables and Functions **25**
Variables 25
Pattern matching and `let` 28
Functions 28
Anonymous Functions 29

2013-05-28
15:22:23

Multi-argument functions 30
Recursive functions 32
Prefix and Infix operators 33
Declaring functions with `function` 36
Labeled Arguments 37
Optional arguments 39

3. Lists and Patterns 45
List Basics 45
Using patterns to extract data from a list 46
Limitations (and blessings) of pattern matching 48
Performance 48
Detecting errors 49
Using the `List` module effectively 50
More useful list functions 53
Tail recursion 55
More concise and faster patterns 57

4. Files, Modules and Programs 63
Single File Programs 63
Multi-file programs and modules 66
Signatures and Abstract Types 67
Concrete types in signatures 70
Nested modules 71
Opening modules 73
Including modules 74
Common errors with modules 76
Type mismatches 76
Missing definitions 77
Type definition mismatches 77
Cyclic dependencies 78

5. Records 79
Patterns and exhaustiveness 81
Field punning 82
Reusing field names 84
Functional updates 86
Mutable fields 88
First-class fields 88

6. Variants 93
Combining records and variants 96
Variants and recursive data structures 99

2013-05-28
15:22:23

Polymorphic variants 102
Example: Terminal colors redux 104
When to use polymorphic variants 108

7. Error Handling 111
Error-aware return types 111
Encoding errors with `Result` 112
`Error` and `Or_error` 113
`bind` and other error-handling idioms 114
Exceptions 116
Helper functions for throwing exceptions 118
Exception handlers 119
Cleaning up in the presence of exceptions 119
Catching specific exceptions 120
Backtraces 121
From exceptions to error-aware types and back again 122

8. Imperative Programming 125
Example: Imperative dictionaries 125
Primitive mutable data 129
Array-like data 129
Mutable record and object fields and ref cells 130
Foreign functions 131
`for` and `while` loops 131
Example: Doubly-linked lists 132
Modifying the list 134
Iteration functions 135
Laziness and other benign effects 136
Memoization and dynamic programming 138
Input and Output 145
Terminal I/O 145
Formatted output with `printf` 146
File I/O 149
Order of evaluation 150
Side-effects and weak polymorphism 152
The Value Restriction 153
Partial application and the value restriction 154
Relaxing the value restriction 155

9. Functors 159
A trivial example 159
A bigger example: computing with intervals 161
Making the functor abstract 163

2013-05-28
15:22:23

Sharing constraints 164
Destructive substitution 166
Using multiple interfaces 167
Extending modules 169

10. Object Oriented Programming 173
When to use objects 174
OCaml objects 174
Object Polymorphism 175
Classes 177
An Example: Cryptokit 178
Class parameters and polymorphism 181
Object types 183
Immutable objects 185
Class types 186

11. Object Subtyping and Inheritance 189
Subtyping 189
Using more precise types to address subtyping problems 191
Using elided types to address subtyping problems 192
Narrowing 192
Binary methods 194
Private methods 196
Virtual classes and methods 198
Multiple inheritance 200
How names are resolved 200
Mixins 202

Part II. Tools and Techniques

12. Maps and Hashtables 207
A map example 208
Creating maps with comparators 209
The polymorphic comparator 211
Sets 212

13. Command Line Parsing 215
Basic command line parsing 215
Using flags to label the command line 219
Grouping sub-commands together 221
Advanced control over parsing 224
Composing specification fragments together 225

2013-05-28
15:22:23

Prompting for interactive input 226
Adding labelled arguments to callbacks 228
Command-line auto-completion with `bash` 229
Generating completion fragments from Command 230
Installing the completion fragment 230

14. Handling JSON data ... 233
JSON Basics 233
Parsing JSON with Yojson 234
Selecting values from JSON structures 237
Constructing JSON values 240
Using non-standard JSON extensions 242
Automatically mapping JSON to OCaml types 243
Example: Querying Github organization information 246

15. Parsing with OCamllex and Menhir ... 249
Defining a JSON parser with menhir 250
Token declarations 251
Specifying the grammar rules 252
Defining a lexer with ocamllex 255
Let-definitions for regular expressions 255
Lexing rules 256
Recursive rules 258
Bringing it all together 259

16. Data Serialization with S-Expressions ... 261
The Sexp format 263
Sexp converters 264
Preserving invariants 266
Getting good error messages 266
Sexp-conversion directives 268
`sexp_opaque` 268
`sexp_list` 268
`sexp_option` 269
Specifying defaults 269

17. Concurrent Programming with Async ... 273
Async basics 274
Ivars and upon 276
Examples: an echo server 278
Improving the echo server 281
Example: searching definitions with DuckDuckGo 283
URI handling 284

2013-05-28
15:22:23

Parsing JSON strings 284
Executing an HTTP client query 285
Exception handling 287
Monitors 289
Example: Handling exceptions with DuckDuckGo 291
Timeouts, Cancellation and Choices 293

18. Fast Binary Serialization 297
Defining a message broker 300
Bigstring 302
Fieldslib 302

19. First class modules 305
Another trivial example 306
Standard vs. first-class modules 307
A more complete example -- containers 309
Imperative containers 310
Pure functional containers 311
Dynamically choosing a module 313
Example: A service bundle 314

Part III. The Runtime System

20. Foreign Function Interface 321
The `ctypes` foreign function library 321
Example: an ncurses terminal interface 321
Defining basic C formats from OCaml 325
Arrays, structures and unions 326
Example: binding UNIX date functions 327
Callbacks between C and OCaml 329

21. Memory Representation of Values 331
The garbage collector 332
The mark and sweep GC strategy 333
Generational garbage collection 333
The fast minor heap 333
The long-lived major heap 334
The representation of OCaml values 334
Distinguishing integer and pointers at runtime 335
Blocks and values 336
Integers, characters and other basic types 337
Tuples, records and arrays 338

2013-05-28
15:22:23

Floating point numbers and arrays 338
Variants and lists 339
Polymorphic variants 340
String values 341
Custom heap blocks 342

22. The Compilation Pipeline 345
An overview of the toolchain 346
Obtaining the source code 347
Parsing source code 347
Formatting source code with `ocp-indent` 348
Generating documentation via `ocamldoc` 349
Preprocessing with `camlp4` 351
Inspecting `camlp4` output 352
The type checking phase 355
Using `ocamlobjinfo` to inspect compilation units 356
Examining the typed syntax tree 357
The untyped lambda form 358
Portable bytecode with `ocamlc` and `ocamlrun` 359
Compiling and linking OCaml bytecode 360
Native code generation 362
Building debuggable libraries 363

23. Using the direct C interface 365
A "Hello World" C binding 365
Converting from OCaml values in C 367
Constructing OCaml values from C 368

24. Understanding the Garbage Collector 373
Runtime Memory Management 373
Allocating on the minor heap 374
Allocating on the major heap 374
The major heap free list 375
Memory allocation strategies 375
Inter-generational pointers 377
How garbage collection works 379
Collecting the minor heap 379
Collecting the major heap 380
Marking the major heap 381
Sweeping unused blocks from the major heap 381
Compaction and defragmenting the major heap 381

2013-05-28
15:22:23

25. Parsing Binary Protocols with Bigarray 383
Bigarrays for external memory blocks 383

A. Installation 385

B. Packaging 395

2013-05-28
15:22:23

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

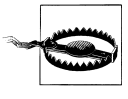
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`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

2013-05-28
15:22:23

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.

2013-05-28
15:22:23

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*

2013-05-28
15:22:23

2013-05-28
15:22:23

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 run-time errors, such as Java or Scala interfaces or variable type declarations in C#, Ada and Pascal.
- *Generics* to enable abstractions to be constructed across different datatypes, available as C++ templates or 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 support these ideas well 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 the 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 Garrique 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 language, and there is fast native code support for x86_64, ARM, PowerPC and Sparc64,

making OCaml a good choice for systems where resource usage, predictability and performance matters.

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. The standard library was actually developed for use within the compiler itself, and by design covers only a small subset of the functionality you expect for more general-purpose use.

In the world of open-source software, nothing stops alternative libraries from being written to supplement the compiler standard library, and this 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 it was designed from the start with an eye towards being a general-purpose standard library, and has very broad applicability. Core is also engineered with correctness, reliability and performance very much in mind.

Core is also 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.

If you've learnt some OCaml before, this book may surprise you with some differences from your past experience. The Core standard library 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.

The OCaml Platform

Core is comprehensive and effective standard library, but there's a lot more out there than Core. 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 over the standard library 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.

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 re-use 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 remains completely 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, with very little compiler magic involved. We'll explain some of these tricks to you as we go through the book and gradually introduce more complex concepts.

What to expect

Real World OCaml is split into three parts and appendices:

- Part I covers the basic 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 complete examples. 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 ex-

amples 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` and `gprof`. Contributing your code back to the community is also important (if only to get bug fixes from other people!), and this part also explains 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).

2013-05-28
15:22:23

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

2013-05-28
15:22:23

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 in 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, do 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, create a file called `~/.ocamlinit` in your home directory:

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

```
#thread
#require "core.top"
```

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` and return. 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 both the type of the result and the result itself.
- Function arguments are separated by spaces, instead of by parenthesis and commas, which is more like the UNIX shell than 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 in within the 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`.

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`).

Functions and type Inference

The `let` syntax can also be used for creating functions.

```
# 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 bind one to a variable using the same `let` keyword used for binding a variable to a simple value such as an integer.

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`.)

```
# 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.

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 30. 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 other related variables, 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 return the same type, so the expression `if test x then x else 0` requires that `x` must be the same type as `0`, which is `int`. By the same logic we can conclude that `y` has type `int`.
- `test` is passed `x` as an argument. Since `x` has type `int`, the input type of `test` must be `int`.
- `test x` is used as the condition in an `if` statement, so the return type of `test` must be `bool`.
- The fact that `+` returns an 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 never 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>
```

we see that rather than choose a single concrete type, OCaml has introduced a *type variable* `'a` to express that the type is generic. 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")
```

(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 on 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. Here, for example, 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.

In this example, the function `String.length` is passed under the *labeled argument* `~f`. Labels allow you to specify function 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"]
```

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.

```
# [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 comes because the compiler can't be certain that the pattern match won't lead to a runtime error. Indeed, the warning gives an example of a list, (`[]`, the empty list) that doesn't match the provided pattern. Indeed, 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 destuttering a list, *i.e.*, 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 `let`s 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`. An anonymous function is a function that is defined but not named, in this case, using the `fun` keyword. Anonymous 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.

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 modifying state as they go.

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 hashtables, 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.

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 your code is acting by side-effect.

A new and somewhat 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 a function like `update` that operates by side effect rather than by returning a value, and for the argument to a function like `create` that doesn't require any information to be passed into it in order to run. This is similar to the role that `void` plays in languages like C and Java.

Here's an example of `create` and `update` in action.

```
# let rsum = create ();;
val rsum : running_sum = {sum = 0.; sum_sq = 0.; samples = 0}
# List.iter [1.;3.;2.;-7.;4.;5.] ~f:(fun x -> update rsum x);;
- : unit = ()
# mean rsum;;
- : float = 1.33333333333333326
```

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

Refs

We can declare a single mutable value by using a `ref`, which is a record type with a single mutable field that is defined in the standard library.

```
# 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 refs 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
```

The definition of all this is quite straightforward. Here is the complete implementation of the `ref` type. 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.

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

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

Here, `!` and `:=` are infix operators that we're defining, where the parenthetical syntax marks them as such.

Even though a `ref` is just another record type, it's notable 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 ar =
    for i = 0 to Array.length ar - 2 do
       (* pick a j that is after i and before the end of the list *)
       let j = i + 1 + Random.int (Array.length ar - i - 1) in
       (* Swap i and j *)
       let tmp = ar.(i) in
       ar.(i) <- ar.(j);
       ar.(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 first non-negative position in an array. Note that `while` (like `for`) is also a keyword.

```
# let find_first_negative_entry ar =
     let pos = ref 0 in
     while !pos < Array.length ar && ar.(!pos) >= 0 do
       pos := !pos + 1
     done;
     if !pos = Array.length ar then None else Some !pos
  ;;
           val find_first_negative_entry : int Core.Std.Array.t -> 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
```

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-along 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`.

```
(* 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.

```
max $ ./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 13.

Where to go from here

That's it for our guided tour! There are plenty of features left to touch upon and lots of details to explain, but the hope is that this has given you enough of a feel for the language that you have a sense as to what to expect, and will be comfortable reading examples in the rest of the book.

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, and so we'll spend some time digging into the details so you can see how OCaml's variables and functions differ from what you may have encountered in other languages.

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.

```
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 toplevel, 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 "Imperative programming" on page 19, there are mutable values in OCaml, but no mutable variables.

Why don't variables vary?

One source of confusion for people new to functional langauges is the fact that variables are typically 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 9, 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>
```

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 f -> f 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 argument of a function as a variable being bound to the value passed by the caller. Indeed, the following two expressions are nearly equivalent:

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

This connection is important, and will come up more when programming in a monadic style, as we'll see in Chapter 17.

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 computation. 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 famous 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. This doesn't change the meaning of the signature, but it makes it easier to see how the currying fits in.

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

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 arms 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 "Imperative programming" on page 19, 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 actually the combination of two patterns; `[]`, 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 there-

fore 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 parenthesis 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 any identifier that is a sequence 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)
```

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
!..., ?..., ~...	Unary prefix
., .(, .[	
function application, constructor, assert, lazy	Left associative
-, -.	Unary 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.

Here's an example of a very useful operator that's defined in Core, following these rules. Here's the definition:

```
# 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. Normally, `List.iter` takes two arguments: a function to be called on each element of the list, and the list to iterate over. We can call `List.iter` with all its arguments:

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

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

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

It is this later form that we're using in the |> 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 ~com`

pare: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, we've written functions where the arguments are specified 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. Labels are marked by a leading tilde, and a label (followed by a colon) are put in front of the variable to be labeled.

```
# 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 hashtable where the first argument is the initial size of the table, and the second argument is a flag which, when true, indicates that the hashtable will reduce its size when the hashtable 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.

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>
```

2013-05-28
15:22:23

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 built up lists without parenthesis. The empty list `[]` is used to terminate a list.

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 | *---->||
+---+---+   +---+---+   +---+---+
```

The `::` operator 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 statment. 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. Moreover, the function clearly does the wrong thing, filtering out all elements of the list rather than just those equal to the provided value.

```
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>
# 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 run-time 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 x =
  match x with
  | 0 -> 1
  | 1 -> 2
  | 2 -> 3
  | _ -> x + 1

let plus_one_slow x =
  if      x = 0 then 1
  else if x = 1 then 2
  else if x = 2 then 3
  else x + 1
```

If you benchmark these, you'll see that `plus_one_slow` is about 35% slower than `plus_one`, and the advantage gets larger as the number of cases increases.

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_slow l =
```

```
  if List.is_empty l then 0
  else List.hd_exn l + sum_slow (List.tl_exn l)
;;
```

In this case, the match-based implementation is 70% 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 typically more efficient than the alternatives you might code by hand. One notable exception is matches over strings, which are in fact tested sequentially. But most of the time, using 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, takin 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 a string 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.

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 following expression uses `List.filter_map` to produce 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 ofa 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"]
```

In the match statement above, you may notice that we for the first time used an underscore in a pattern match. You use an underscore when you want to indicate that the pattern doesn't depend on some sub-component of the data structure, but that you don't want to name it is an explicit variable.

Another feature of OCaml's pattern language that we encounter here is *or-patterns*, which allow you to have multiple sub-patterns 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.

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`.

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

`@` is just a synonym for `List.append`. In addition, there is `List.concat`, for concatenating a list of lists.

`ocaml # 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
  ;;
# all_files ".";;
- : 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>
utop[41]> 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 implemenation 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 resuse the caller's stack frame.

Tail recursion are important for more than just lists. Ordinary (non-tail) recursive calls are reasonable when the 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 (xref)(#recursive-list-functions): 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. At the same time, we'll use the more concise `[_]` pattern to match a list with a single element, rather than `_ :: []`.

```
# 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 ex-

pression. 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 operator is smaller than, equal to, or greater than the second, respectively.

You might wonder how you could build function 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, they will fail at runtime if they encounter functions:

```
# (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 they 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 12.

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 to boot is not tail recursive. For real work, you should probably just use the`List.count` function from `Core` as follows:

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

2013-05-28
15:22:23

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 *)
[("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) -> compare y x)
  |> (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 order. 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` is required to do the final linking in of packages for building a runnable executable, and `-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, and debugging support for 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 now run the our program 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 very 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 the `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 it's 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. Note that the module name is capitalized even if the file is not.

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

type t
```

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

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.

Here's a rewrite of `counter.ml` to match this interface.

```
(* 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 11, characters 20-22:
Error: This expression has type 'a list
       but an expression was expected of type Counter.t
```

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 12.

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));;
```

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. The following two lines added to `freq.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 identifier 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.

```
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 fairly obvious bug, and indeed, the compiler will refuse to compile it, spitting out the following error.

```
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.
```

We can also combine this with the use of the include statement to add some extra functionality to such a module. Thus, we could have rewritten the definition of `Hostname` above as follows to add a function `Hostname.mine` that returns the hostname of the present machine.

```
module Hostname : sig
  include ID
  val mine : unit -> t
end = struct
  include String_id
  let mine = Unix.gethostname
end
```

Opening modules

One useful primitive in OCaml's module language is the `open` statement. We've seen that already in the `open Core.Std` that has been at the top of our source files.

We've used OCaml's `open` statement many times already in the `open

So far, we've been referring to values and types within a module by using the module name as an explicit qualifier. *e.g.*, we 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 that kind of explicit qualification. This is what the `open` statement is for.

We've already seen the `open` statement in use in the `open Core.Std` statements at the top of each source file. Opening a module adds its contents to the environment that the compiler looks in for finding identifiers. Here's a trivial example that gives you a sense of how this works.

```
# 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 harder it is to look at an identifier and figure out where it's defined.

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 `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 of 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 of 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 circular 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 32), you need to define them using `let rec` rather than ordinary `let`.

The same is true at the module level. By default, circular dependencies between modules are not allowed, and indeed, circular 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 circular 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" ]
```

2013-05-28
15:22:23

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;
      os_release : 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";
      os_release = sh "uname -r";
      cpu_arch   = sh "uname -p";
      timestamp  = Time.now ();
    };;
val my_host : host_info =
  {hostname = "yevaud.local"; os_name = "Darwin"; os_release = "12.3.0";
   cpu_arch = "i386"; timestamp = 2013-04-13 06:39:17.806527}
```

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 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;
                            os_release = r; cpu_arch = c;
                            timestamp = ts;
                          } =
      sprintf "%s (%s %s / %s, on %s)" h os r c (Time.to_sec_string ts);;
  val host_info_to_string : host_info -> string = <fun>
# host_info_to_string my_host;;
- : string = "yevaud.local (Darwin 12.3.0 / i386, on 2013-04-13 06:39:17)"
```

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_version`, as shown below.

```
# type host_info =
    { hostname   : string;
      os_name    : string;
      os_release : string;
      cpu_arch   : string;
      os_version : 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_version`, 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";;
# let host_info_to_string { hostname = h; os_name = os;
                            os_release = r; cpu_arch = c;
                            timestamp = ts;
                          } =
    sprintf "%s (%s %s / %s, on %s)" h os r c (Time.to_sec_string ts);;
Characters 24-183:
val host_info_to_string : host_info -> string = <fun>
Warning 9: the following labels are not bound in this record pattern:
os_version
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;
                            os_release = r; cpu_arch = c;
                            timestamp = ts; _
                          } =
    sprintf "%s (%s %s / %s, on %s)" h os r 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; os_release; cpu_arch } =
     sprintf "%s (%s %s / %s)" hostname os_name os_release cpu_arch;;
  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 os_release = sh "uname -r" in
```

```
    let cpu_arch   = sh "uname -p" in
    let os_version = sh "Uname -v" in
    let timestamp  = Time.now () in
    { hostname; os_name; os_release; cpu_arch; os_version; timestamp };;
val my_host : host_info =
  {hostname = "yevaud.local"; os_name = "Darwin"; os_release = "12.3.0";
   cpu_arch = "i386";
   os_version =
    "Darwin Kernel Version 12.3.0: Sun Jan  6 22:37:10 PST 2013; root:xnu-2050.22.13~1/RELEASE_X86_64";
   timestamp = 2013-04-13 06:49:57.771755}
```

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 ~os_release ~cpu_arch ~os_version =
    { os_name; os_release; cpu_arch; os_version;
      hostname = String.lowercase hostname;
      timestamp = Time.now () };;
val create_host_info :
  hostname:string ->
  os_name:string ->
  os_release:string ->
  cpu_arch:string -> os_version:string -> timestamp:Time.t -> 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 ~os_release:os_release
    ~cpu_arch:cpu_arch ~os_version:os_version =
    { os_name = os_name; os_release = os_release;
     cpu_arch = cpu_arch; os_version = os_version;
      hostname = String.lowercase hostname;
      timestamp = Time.now () };;
val create_host_info :
  hostname:string ->
  os_name:string ->
  os_release:string ->
  cpu_arch:string -> os_version:string -> timestamp:Time.t -> 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 = <funambulate>
```

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 heartbeat-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 19.

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 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 a field but not 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 combiation 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 guarntee 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.

2013-05-28
15:22:23

2013-05-28
15:22:24

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.

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 tag 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 being avoiding the catch-all case, and instead being explicit about the tags that are ignored.

Combining records and variants

Records and variants are most effective when used in concert. Consider again the type `Log_entry.t` from Chapter 5:

```
# 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 `blang` (short for "Boolean language") with one tag for each kind of expression we want to support.

```
# type 'a blang =
  | Base  of 'a
  | Const of bool
  | And   of 'a blang list
  | Or    of 'a blang list
  | Not   of 'a blang
  ;;
```

Note that the definition of the type `blang` is recursive, meaning that a `blang` may contain other `blang`s. Also, `blang` 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 `blang` 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 blang = <fun>
# And [ Or [ test To "doligez"; test CC "doligez" ];
        test Subject "runtime";
      ]
  ;;
- : mail_predicate blang =
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 `blang`s.

```
# let rec eval blang base_eval =
    (* a shortcut, so we don't need to repeatedly pass [base_eval]
       explicitly to [eval] *)
```

```
    let eval' blang = eval blang base_eval in
    match blang with
    | Base  base   -> base_eval base
    | Const bool   -> bool
    | And   blangs -> List.for_all blangs ~f:eval'
    | Or    blangs -> List.exists  blangs ~f:eval'
    | Not   blang  -> not (eval' blang)
  ;;
val eval : 'a blang  -> ('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 function applies some basic simplification rules, most of which are driven by the presence of constants.

```
# let rec simplify = function
    | Base _ | Const _ as x -> x
    | And blangs ->
      let blangs =
        List.map ~f:simplify blangs
        |> List.filter ~f:(fun x -> x <> Const true)
      in
      if List.is_empty blangs then Const true
      else if List.exists blangs ~f:(fun x -> x = Const false)
      then Const false
      else And blangs
    | Or blangs ->
      let blangs =
        List.map ~f:simplify blangs
        |> List.filter ~f:(fun x -> x <> Const false)
      in
      if List.is_empty blangs then Const false
      else if List.exists blangs ~f:(fun x -> x = Const true)
      then Const true
      else Or blangs
    | Not blang ->
      match simplify blang with
      | Const bool -> Const (not bool)
      | blang -> Not blang
  ;;
val simplify : 'a blang -> 'a blang = <fun>
```

One thing to notice about the above code is that it uses a catch-all case in the very last line within the `Not` case. It's generally better to be explicit about the cases you're ignoring. Indeed, if we change this snippet of code to be more explicit:

```
    | Not blang ->
      match simplify blang with
```

```
    | Const bool -> Const (not bool)
    | (And _ | Or _ | Base _ | Not _) -> Not blang
```

it's easy to see that we've missed an important case: double-negation.

```
  | Not blang ->
    match simplify blang with
    | Const b -> Const (not b)
    | Not blang -> blang
    | (And _ | Or _ | Base _ ) -> Not blang
```

This example is more than a toy. There's a module very much in this spirit in Core called `Blang`, and it gets a lot of practical use in a variety of applications.

More generally, using variants to build recursive data structures is a common technique, and shows up everywhere from designing little languages to building efficient 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 implementation 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 10, subtyping brings in a lot of complexity, and most of the time, that's complexity you want to avoid.

2013-05-28
15:22:24

2013-05-28
15:22:24

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";;
- : Core.Std.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);;
  - : Core.Std.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 library 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;;
- : Core.Std.Error.t =
"Something failed a long time ago: (1969-12-31 19:00:00.000000)"
```

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 16, 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.

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 parenthesis, 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 17.

```
# 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").
```

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 with `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 16.

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'tbe 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 shoudl 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 point of exceptions is to give useful debugging information. But at first glance, OCaml's exceptions can be less than informative. 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 pretty uninformative error:

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

The example in question is short enough that it's quite easy to see where the error came from. But in a complex program, simply knowing which exception was thrown is usually not enough information to figure out what went wrong.

We can get more information from OCaml if we turn on stack backtraces. A backtrace is essentially a summary of the stack of calls that were executed to get to the point where the exception was thrown. Backtraces can be enabled by setting the `OCAMLRUNPARAM` environment variable as shown.

```
exn $ export OCAMLRUNPARAM=b=1
exn $ ./exn
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.

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 Core.Std.Option.t = None
# find ["a",1; "b",2] "b";;
- : int Core.Std.Option.t = 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").
```

2013-05-28
15:22:24

2013-05-28
15:22:24

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.

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`.

While `Hashtbl.hash` can be used with any type, it won't necessarily succeed for all values. `Hashtbl.hash` will throw an exception if it encounters a value it can't handle, like a function or a value from a C libraries that lives outside the OCaml heap.

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, and a call to `List.iter` to walk over the list of 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 eqivalent 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
```

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 21. 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 10, 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.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.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 an 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-visisble 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 hashtable (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 not 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 17. 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 63) 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 Core.Std.List.t -> 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 oppose 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 10, 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>
```

2013-05-28
15:22:24

2013-05-28
15:22:24

CHAPTER 9
Functors

Up until now, we've seen OCaml's module system play an important but limited role. In particular, we've seen them as a mechanism for organizing code into units with specified interfaces. But modules can do much more than that, acting as a powerful toolset 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*. Sometimes, there is some functionality that you want to build in a standard way for different types, in each case based on a some piece of type-specific logic. 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

We'll start by considering the simplest possible example: a functor for incrementing an integer.

More precisely, we'll create a functor that takes a module containing a single integer variable `x`, and returns a new module with `x` incremented by one.

```
# module type X_int = sig val x : int end;;
module type X_int = sig val x : int end
# 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 about functors is that they're considerably more heavyweight syntactically than ordinary functions. For one thing, functors require explicit (module) type annotations, which ordinary functions do not. Here, we've specified the module type `X_int` for both the input and output of the functor. Technically, only the type on the input is mandatory, although in practice, one often specifies both.

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 a the contents of an `ml` file can 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 x_string : string end
# module Four = Increment(Three_and_more);;
module Four : sig val x : int end
```

A bigger example: computing with intervals

Let's now consider a more realistic example of how to use functors: a library for computing with intervals. This library will be functorized over the type of the endpoints of the intervals and the ordering of those endpoints.

First we'll define a module type that captures the information we'll need about the endpoint type. 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.

```
# module Make_interval(Endpoint : Comparable) = struct

    type t = | Interval of Endpoint.t * Endpoint.t
             | Empty

    let create low high =
      if Endpoint.compare low high > 0 then Empty
      else Interval (low,high)

    let is_empty = function
      | Empty -> true
      | Interval _ -> false

    let contains t x =
      match t with
      | Empty -> false
      | Interval (l,h) ->
        Endpoint.compare x l >= 0 && Endpoint.compare x h <= 0

    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, 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
```

When we choose our interfaces that are aligned with the standards of our libraries, we don't need to construct a custom module to feed to the functor. In this case, for example, 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. Standardized signatures are generally good practice, both because they makes functors easier to use, and because they make the 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 int 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 apply an interface to the output of the `Make_interval`. 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 represent the type of the endpoints of the interval. 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 make the implementation 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 unfortunately, it's 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 sharing constraint on a module type 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`. 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
```

And we can also use it in the context of a functor, where the right-hand side of the sharing constraint is an element of the functor argument. Thus, we expose an equality between a type in the output of the functor (in this case, the type `endpoint`) and a type in its input (`Endpoint.t`).

```
# 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 now do things like construct intervals again.

```
# 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`.

Here's an example of what we get if we use destructive substitution to specialize the `Interval_intf` interface to integer intervals.

```
# 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 mention of n `endpoint`, all occurrences of that type having 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, and we didn't need to define the `endpoint` type alias in the body of the module. If we instantiate this module, we'll see that it works properly: we can construct new intervals, but `t` is abstract, and so we can't directly access the constructors and violate the invariants of the data structure.

```
# 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
```

Using multiple interfaces

Another feature that we might want for our interval module is the ability to serialize the type, in particular, by converting to s-expressions. If we simply invoke the `sexplib` macros by adding `with sexp` to the definition of `t`, though, we'll get an error:

```
# 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. Note the use of destructive substitution to combine multiple signatures together. This is important because it stops the `type t`'s from the different signatures from interfering with each other.

Also note that we have been careful to override the sexp-converter here to ensure that the data structures invariants are still maintained when reading in from an s-expression.

```
# module type Interval_intf_with_sexp = sig
   type t
   include Interval_intf with type t := t
   include Sexpable      with type t := t
  end;;
# 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

      let create 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

      ....

     end ;;
module Make_interval :
  functor
    (Endpoint : sig
          type t
          val compare : t -> t -> int
          val sexp_of_t : t -> Sexplib.Sexp.t
          val t_of_sexp : Sexplib.Sexp.t -> 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 sexp_of_t : t -> Sexplib.Sexp.t
      val t_of_sexp : Sexplib.Sexp.t -> t
    end
```

And now, we can use that sexp-converter in the ordinary way:

```
# module Int = Make_interval(Int) ;;
# Int_interval.sexp_of_t (Int_interval.create 3 4);;
- : Sexplib.Sexp.t = (Interval 3 4)
# Int_interval.sexp_of_t (Int_interval.create 4 3);;
- : Sexplib.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

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

type 'a t

val empty : 'a t

val enqueue : 'a t -> 'a -> 'a t

(** [dequeue q] returns None if the [q] is empty *)
val dequeue : 'a t -> ('a * 'a t) option

val fold : 'a t -> init:'acc -> f:('acc -> 'a -> 'acc) -> 'acc
```

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 first list, and can efficiently dequeue from the out 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 (l1,l2) x = (x :: l1,l2)

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 =
  List.fold ~init:(List.fold ~init ~f out_list) ~f
    (List.rev in_list)
```

One problem with our `Fqueue` is that the interface is quite skeletal. There are lots of useful helper functions that one might want that aren't there. For example, for lists we have `List.iter` which runs a function on each node; and a `List.find` that finds the first element on the list that matches a given predicate. Such helper functions come up for pretty much every container type, and implementing them over and over is a bit of 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`.

```
open Core.Std

module T = struct
  type 'a t = 'a list * 'a list

  let empty = [],[]

  let enqueue (l1,l2) x =
    (x :: l1,l2)

  let rec dequeue (in_list,out_list) =
    match out_list with
    | hd :: tl -> Some (hd, (in_list,tl))
    | [] -> dequeue ([], List.rev in_list)

  let fold (in_list,out_list) ~init ~f =
    List.fold ~init:(List.fold ~init ~f out_list) ~f
      (List.rev in_list)
end
include T
include Foldable.Extend(T)
```

This pattern comes up quite a bit in Core, and is used to for a variety of purposes.

- Adding comparison-based data structures like maps and sets, based on the `Comparable` interface.
- Adding hash-based data structures like hash sets and hash heaps.
- Support for so-called monadic libraries, like the ones discussed in Chapter 7 and Chapter 17. Here, the functor is used to provide a collection of standard helper functions based on the core `bind` and `return` operators.

2013-05-28
15:22:24

CHAPTER 10
Object Oriented Programming

We've already seen several tools that OCaml provides for organizing programs, particularly first-class modules. In addition, OCaml also supports object-oriented programming. There are objects, classes, and their associated types. Objects are good for encapsulation and abstraction, and classes are good for code re-use.

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 four 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 publically-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 re-used to produce a new kind of object. This new definition can override some behaviour, but also share code with its parent.

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.

When to use objects

You might wonder when to use objects in OCaml, which has a multitude of alternative mechanisms to express the same concept. First-class modules are more expressive (a module can include types, while classes and objects cannot). Modules, functors, and algebraic data types 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.

Modules already provide these features in some form, but the main focus of classes is on code re-use through inheritance and late binding of methods. This is a critical property of classes: the methods that implement an object are determined when the object is instantiated, a form of *dynamic* binding. In the meantime, while classes are being defined, it is possible (and necessary) to refer to methods without knowing statically how they will be implemented.

In contrast, modules use static (lexical) scoping. 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 dynamic binding is a big win, for example if you have many similar variations in the implementation of a concept. 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.

In this chapter, we'll introduce you to the basics of object definition and use in OCaml, and then demonstrate their use with an example using Cryptokit. We'll return to the more advanced areas of object use later on in the book in Chapter 11.

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 subtyping and inheritance in OCaml. In a language like Java, a class name is also used as the type of objects created by instantiating it, and the subtyping rule corresponds to inheritance. For example, if we implement a class `Stack` in Java by inheriting from a class `Deque`, we would be allowed to pass a stack anywhere a deque is expected (this is a silly example of course, practitioners will point out that we shouldn't do it).

OCaml is entirely different. Classes are used to construct objects and support inheritance, including non-subtyping inheritance. 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 p =
  object
    val mutable x = 0
    method get = x
    method set i = x <- i
  end;;
val p : < get : int; set : int -> unit > = <obj>
```

The object has an integer value `x`, a method `get` that returns x, and a method `set` that updates the value of `x`.

The object type is enclosed in angle brackets `< ... >`, containing just the types of the methods. Fields, like `x`, 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.

```
# p#get;
- : int = 0
# p#set 17;;
- : unit = ()
# p#get;;
- : int = 17
```

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 make i =
  object
    val mutable x = i
    method get = x
    method set y = x <- y
  end;;
val make : 'a -> < get : 'a; set : 'a -> unit > = <fun>
# let p = make 5;;
val p : < get : int; set : int -> unit > = <obj>
# p#get;;
- : int = 5
```

Note that the types of the function `make` and the returned object now use the polymorphic type `'a`. When make is invoked on a concrete value `5`, we get the same object type as before, with type `int` for the value.

Object Polymorphism

Functions can also take object arguments. Let's construct a new object `average` that returns the average of any two objects with a `get` method.

```
# let average p1 p2 =
  object
    method get = (p1#get + p2#get) / 2
```

```
  end;;
val average :
  < get : int; .. > ->
  < get : int; .. > ->
  < get : int > = <fun>
```

There's some new syntax in the type that's been inferred for `average` here. The parameters have the object type `< get : int; .. >`. The `..` are ellipsis, standing for any other methods. The type `< get : int; .. >` specifies an object that must have at least a `get` method, and possibly some others as well.

We can use the `average` using the normal object invocation syntax:

```
# let p1 = make 5;;
# let p2 = make 15;;
# let a = average p1 p2;;
# a#get;;
- : int = 10
# p2#set 25;;
# a#get;;
- : int = 15
```

The potential extra parameters defined by the object are carefully tracked by the OCaml type checker. If we manually try and constrain the exact type `< get : int >` for an object with more methods, type inference will fail.

```
# let (p : < get : int >) = make 5;;
Error: This expression has type < get : int; set : int -> unit >
       but an expression was expected of type < get : int >
       The second object type has no method set
```

Elisions are polymorphic

The `..` in an 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 point = < get:int; .. >;;
```

```
Error: A type variable is unbound in this type declaration.
In type < get : int; .. > 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. The error message suggests a solution, which is to add the `as 'a` type constraint.

```
# type 'a point = < get:int; .. > as 'a;;
type 'a point = 'a constraint 'a = < get : int; .. >
```

In other words, the type `'a point` is equal to `'a`, where `'a = < get : int; .. >`. That may seem like an odd way to say it, and in fact, this type definition is not really an abbreviation because `'a` refers to the entire type.

An object of type `< get:int; .. >` can be any object with a method `get:int`, it doesn't matter how it is implemented. So far, we've constructed two objects with that type; the function `make` constructed one, and so did `average`. When the method `#get` is invoked, the actual method that is run is determined by the object.

```
# let print_point p = Printf.printf "Point: %d\n" p#get;;
val print_point : < get : int; .. > -> unit = <fun>
# print_point (make 5);;
Point: 5
# print_point (average (make 5) (make 15));;
Point: 10
```

Classes

Programming with objects directly is great for encapsulation, but one of the main goals of object-oriented programming is code re-use 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.

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)
```

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, possibily sending messages to other objects.

Indeed, in many programs, this makes sense, but it is by no means required. Let's define an object-oriented version of lists similar to the imperative list above. We'll implement it with a regular list type `'a list`, and insertion will be to the beginning of the list instead of to the end.

```
class ['a] flist =
object (self : 'self)
```

```
    val elements : 'a list = []

    method is_empty = elements = []

    method insert x : 'self = {< elements = x :: elements >}

    method iterator =
      (new flist_iterator elements :> 'a iterator)

    method iter (f : 'a -> unit) = List.iter f elements

    method fold : 'b. ('b -> 'a -> 'b) -> 'b -> 'b =
      (fun f x -> List.fold_left f x elements)
end;;
```

A key part of the implementation is the definition of the method `insert`. The expression `{< ... >}` produces a copy of the current object, with the same type, and the specified fields updated. In other words, the `new_fst new_x` method produces a copy of the object, with `x` replaced by `new_x`. The original object is not modified, and the value of `y` is also unaffected.

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.

We use the same object type `iterator` for iterators, but implement it differently.

```
class ['a] flist_iterator l =
object
  val mutable elements : 'a list = l

  method has_value = l <> []

  method get =
    match l with
      h :: _ -> h
    | [] -> raise (Invalid_argument "list is empty")

  method next =
    match l with
      _ :: l -> elements <- l
    | [] -> raise (Invalid_argument "list is empty")
end;;
```

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`.

CHAPTER 11

Object Subtyping and Inheritance

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`.

To explore this, let's define some simple classes for geometric shapes. The generic type `shape` has a method to compute the area, and a `square` is a specific kind of shape.

```
type shape = < area : float >;;

class square w =
object (self : 'self)
  method area = self#width *. self#width
  method width = w
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 new_square x : shape = new square x;;
Characters 27-39:
  let new_square x : shape = new square x;;
                             ^^^^^^^^^^^^
Error: This expression has type square but an expression was expected of type shape
The second object type has no method width
# let new_square x : shape = (new square x :> shape);;
val new_square : float -> shape = <fun>
```

What are the rules for subtyping? In general, object subtyping has two general forms, called *width* and *depth* 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).

The subtyping rules are purely technical, they have no relation to object semantics. We can define a class `rectangle` that has all of the methods of a `square`, so it is a subtype of square and can be used wherever a `square` is expected.

```
# class rectangle h w =
  object (self : 'self)
     inherit square w
     method area = self#width *. self#height
     method height = h
  end;;
# let square_rectangle h w : square = (new rectangle h w :> square);;
val square_rectangle : float -> float -> square = <fun>
```

This may seem absurd, but this concept is expressible in all object-oriented languages. The contradiction is semantic -- we know that in the real world, not all rectangles are squares; but in the programming world, rectangles have all of the features of squares (according to our definition), so they can be used just like squares. Suffice it to say that it is usually better to avoid such apparent contradictions.

Next, let's take a seemingly tiny step forward, and start building collections of shapes. It is easy enough to define a `slist` of squares.

```
# let squares =
     let l = SList.make () in
     l#insert (new square 1.0);
     l#insert (new square 2.0);
     l;;
val squares : square slist = <obj>
```

We can also define a function to calculate the total area of a list of shapes. There is no reason to restrict this to squares, it should work for any list of shapes with type `shape slist`. The problem is that doing so raises some serious typing questions -- can a `square slist` be passed to a function that expects a `shape slist`? If we try it, the compiler produces a verbose error message.

```
# let total_area (l : shape slist) : float =
     let total = ref 0.0 in
     let it = l#iterator in
     while it#has_value do
        total := !total +. it#get#area;
        it#next
     done;
     !total;;
val total_area : shape slist -> float = <fun>
# total_area squares;;
Characters 11-18:
  total_area squares;;
             ^^^^^^^
Error: This expression has type
         square slist =
           < insert : square -> unit; is_empty : bool;
```

```
      iterator : square iterator >
but an expression was expected of type
  shape slist =
    < insert : shape -> unit; is_empty : bool;
      iterator : shape iterator >
Type square = < area : float; width : float >
is not compatible with type shape = < area : float >
The second object type has no method width
```

It might seem tempting to give up at this point, especially because the subtyping is not even true -- the type `square slist` is not a subtype of `shape slist`. The problem is with the `insert` method. For `shape slist`, the `insert` method takes an arbitrary `shape` and inserts it into the list. So if we could coerce a `square slist` to a `shape slist`, then it would be possible to insert an arbitrary shape into the list, which would be an error.

Using more precise types to address subtyping problems

Still, the `total_area` function should be fine, in principle. It doesn't call `insert`, 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 mutating the list. We define a type `readonly_shape_slist` and confirm that we can coerce the list of squares.

```
# type readonly_shape_slist = < iterator : shape iterator >;;
type readonly_shape_slist = < iterator : shape iterator >
# (squares :> readonly_shape_slist);;
- : readonly_shape_slist = <obj>
# let total_area (l : readonly_shape_slist) : float = ...;;
val total_area : readonly_shape_slist -> float = <fun>
#   total_area (squares :> readonly_shape_slist);;
- : float = 5.
```

Why does this work, why is a `square slist` a subtype of `readonly_shape_slist`. The reasoning is in two steps. First, the easy part is width subtyping: we can drop the other methods to see that `square slist` is a subtype of `< iterator : square iterator >`. The next step is to use *depth* subtyping, which, in its general form, says that an object type `< m : t1 >` is a subtype of a type `< m : t2>` iff `t1` is a subtype of `t2`. In other words, instead of reasoning about the number of methods in a type (the width), the number of methods is fixed, and we look within the method types themselves (the "depth").

In this particular case, depth subtyping on the `iterator` method requires that `square iterator` be a subtype of `shape iterator`. Expanding the type definition for the type `iterator`, we again invoke depth subtyping, and we need to show that the type `< get : square >` is a subtype of `<get : shape >`, which follows because `square` is a subtype of `shape`.

This reasoning may seem fairly long and 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, the coercion would simply not be possible. For example, in C++, a STL type `slist<T>` is invariant in `T`, it is simply not possible to use

slist<square> where slist<shape> is expected (at least safely). The situation is similar in Java, although Java supports 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.

Using elided types to address subtyping problems

Before we move to the next topic, there is one more thing to address. The typing we gave above, using readonly_shape_slist, requires that the caller perform an explicit coercion before calling the total_area function. We would like to give a better type that avoids the coercion.

A solution is to use an elided type. Instead of shape, we can use the elided type < area : float; .. >. In fact, once we do this, it also becomes possible to use the slist type.

```
# let total_area (l : < area : float; .. > slist) : float = ...;;
val total_area : < area : float; .. > slist -> float = <fun>
# total_area squares;;
- : float = 5.
```

This works, and it removes the need for explicit coercions. This type is still fairly simple, but it does have the drawback that the programmer needs to remember that the types < area : float; ..> and shape are related.

OCaml supports an abbreviation in this case, but it works only for classes, not object types. The type expression # classname is an abbreviation for an elided type containing all of the methods in the named class, and more. Since shape is an object type, we can't write #shape. However, if a class definition is available, this abbreviation can be useful. The following definition is exactly equivalent to the preceeding.

```
# class cshape = object method area = 0.0 end;;
class cshape : object method area : float end
# let total_area (l : #cshape list) : float = ...;;
val total_area : #cshape slist -> float = <fun>
# total_area squares;;
- : float = 5.
```

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 slist, 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_bar_bell = 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_bar_bell = 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.

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 re-use. 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;;
```

PART II

Tools and Techniques

2013-05-28
15:22:24

2013-05-28
15:22:24

CHAPTER 12
Maps and Hashtables

Lots of programming problems require dealing with data organized as key/value pairs. Maybe the simplest way of representing sets of key/value pairs 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 *hashtables*. A map is an immutable tree-based data structure where most operations take time logarithmic in the size of the map, whereas a hashtable 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.

A map example

Let's start by considering an example of how one might use a map in practice. In Chapter 4, we showed a module `Counter` for keeping counts on a set of strings. Here's the interface.

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

type t

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

The intended behavior here is straightforward. `Counter.empty` represents an empty collection of counts; `touch` increments the count of the specified string by 1; and `to_list` returns the list of non-zero string/count pairs.

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 this have access to a string comparison function. Such a `Map` sub-module is included in any module that satisfies the `Comparable` interface from Core.

Creating maps with comparators

The specialized `Map` sub-module is convenient, but you don't necessarily need to use it to create 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.

Note that 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 used to build the map. 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)]
```

If we look at the type of `Map.symmetric_diff`, we'll see that it requires that the two maps it compares have the same comparator type, in addition to the same key type and value type. Each comparator has a fresh abstract type, so the type of a comparator identifies it 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>
```

We could create a new comparator for an existing type using the `Make` functor found in the `Comparator` module. This functor 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.)

```
# 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
```

Both `Reverse.comparator` and `String.comparator` can be used to create string maps.

```
# 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>
```

And by using `Map.min_elt`, which returns the key and value for the smallest key in the map, we can see that these two maps indeed do use different comparison fucntions.

```
# Map.min_elt ord_map;;
- : (string * int) option = Some ("foo", 0)
# Map.min_elt rev_map;;
- : (string * int) option = Some ("snoo", 3)
```

Since the algorithm behind `Map.symmetric_diff` depends on both maps using the same comparison function, we shouldn't be able to run this function jointly on these two maps. Indeed, if we try, the compiler will reject it as a type 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
```

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 this with `Map.to_tree`, which returns the tree type that the map is built on top of, which doesn't include the comparator directly.

```
# let ord_tree = Map.to_tree ord_map;;
val ord_tree : (string, int, String.comparator) Map.Tree.t = <abstr>
```

Note that although it doesn't physically include the comparator, it still includes the comparator in its type.

If we want to look something up in a tree, we need to provide the comparator explicitly.

```
# Map.Tree.find ~comparator:String.comparator ord_tree "snoo";;
- : int option = Some 3
```

The algorithm of `Map.find` depends on the fact that you're using the same comparator when looking a value up as you were when you stored it. Accordingly, 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 integration of comparators into the type of maps allows the type system to catch what would otherwise be fairly subtle errors.

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 polymorphic comparison function, which was discussed in Chapter 3. This comparator is found in the `Map.Poly` module. Thus, we can write:

```
# Map.of_alist_exn 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
```

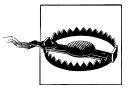
This is rejected for good reason: there's no guarantee that the comparator associated with a given type will order things in the same way that polymorphic compare does.

Sets

Sometimes, instead of keeping track of a set of key/value pairs, you just want a datatype for keeping track of a set set of keys. You could just build this on top of the map type, by representing a set of values by a map where the type of the data is just `unit`. A more idiomatic (and more efficient) solution is to use Core's set type, which is similar in design and spirit to the map type. 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 in map, sets, also support other operations you would expect for sets, including functions for computing unions, intersections and set differences, as well as testing for whether one set is a subset of another.



The perils of polymorphic compare

Polymorphic compare is highly convenient, but it has serious downsides as well, and should be used with care. The problem with polymorphic compare is that it has a fixed algorithm for comparing values of any type, and that that algorithm can sometimes yield surprising results.

To understand what's wrong with polymorphic comparison, you need to consider how it works. Polymorphic compare is a structural comparison, operating 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 behaves as you would expect for most OCaml types. 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 comparator. And it provides a reasonable total comparison function for most OCaml types, excluding only things like closures and values from outside of the OCaml heap.

But sometimes, a structural comparison is not what you want. Maps and 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 trees will be different, and so 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 of 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 all but the smallest applications.

2013-05-28
15:22:24

CHAPTER 13
Command Line Parsing

Many of the OCaml programs that you'll write will end up as binaries that will be run directly 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 simplifies this by letting you declare all your command-line options in one place, using the `Command` library. This then takes care of parsing the arguments, generating help text and provides interactive auto-completion to the user of the library.

The `Command` library also scales as you add more features to your programs. It's simple to use for small applications, but provides a sophisticated subcommand mode that groups related commands together as the application grows more options. You may also be familiar with this command-line style from the Git or Mercurial version control systems.

This chapter demonstrates how to use `Command` to extend the cryptographic utility from Chapter 10 and builds a simple equivalents to the `md5` and `shasum` utilities. It also demonstrates how *functional combinators* can be used to declare complex data structures in a type-safe and elegant way.

Basic command line parsing

We'll begin by cloning the `md5` binary that is present on most Linux distributions and Mac OS X. It reads in the contents of a file, applies the MD5 one-way hash function to the data, and outputs an ASCII hex representation of the result.

```
(* 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 file () -> do_hash file)

let () = Command.run ~version:"1.0" ~build_info:"RWO" command
```

You can compile this file the usual way with `ocamlfind`, but passing an additional `cryptokit` package. You may need to 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.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` value declares how to invoke `do_hash` by parsing the command-line arguments. When you compile this program and run it, you can query the version information simply by:

```
$ ./md5 -version
1.0
$ ./md5 -build-info
RWO
```

The `version` and `build_info` optional arguments to `Command.run` let you specify versions in the command-line help. 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 outputs a help screen that informs you that a required argument `filename` is missing. Supplying the argument to the command results in `do_hash` being called, and the MD5 output being displayed to the standard output.

```
$ ./md5 ./md5
59562f5e4f790d16f1b2a095cd5de844
```

So how does all this work? Most of the interesting logic lies in how the specifications are defined. 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.

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 chains more parameters via the `+>` combinator. Our example uses the `anon` function to define a single anonymous parameter. Anonymous parameters are assigned a string name that is used in help text, and an OCaml type that they are parsed into from the raw command-line string. The example `filename` argument above is extracted from the command-line and kept as an OCaml `string`.

This specification is then bundled together with the callback functions using `Command.basic`. For our `md5` example, we have:

```
Command.basic
  ~summary:"Generate an MD5 hash of the input data"
  ~readme:(fun () -> "More detailed information")
  Command.Spec.(
    empty
    +> anon ("filename" %: string)
  )
  (fun file () -> do_hash file)
```

The `basic` function takes a few more arguments in addition to the specification. The `summary` is a one-line description to go at the top of the command help screen, while `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 final argument is the callback function where all the actual work happens after the command-line parsing is complete. This function will be applied with the parsed command-line arguments, and should perform the actual work. 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 function also needs an extra `unit` argument after `file`. This is simply so that the command specifications can work even when they are empty (`Command.Spec.empty`). 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.

You aren't just limited to parsing command lines as strings of course. `Command.Spec` 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

Using flags to label the command line

You aren't just limited to anonymous arguments on the command-line, and you'll find that flags are useful once your program has more options. 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 mimic the Linux version. A `-s` flag specifies the string to be hashed directly on the command-line, and a `-t` runs a benchmarking 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 example specification uses the `flag` command now. 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 errros 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`	`TODO`
CentOS	`yum`	`TODO`
Mac OS X	Homebrew	`bash-completion`
Mac OS X	MacPorts	`TODO`
FreeBSD	Ports System	`/usr/ports TODO`
OpenBSD	`pkg_add`	`TODO`

Once you have bash completion 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 those, then you can continue on, but 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.

2013-05-28
15:22:24

2013-05-28
15:22:24

CHAPTER 14

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. Sometimes you need to match someone else's data format (such as XML), sometimes you need a highly efficient format, and sometimes you just want something that is easy for humans to read and 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.

```
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:

```
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_channel : ?buf:Bi_outbuf.t -> ?fname:string -> ?lnum:int -> in_channel -> json
(* Read a JSON value from a channel. See [from_string] for the meaning of the
   optional arguments. *)

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:

```
val from_string : string -> json
```

```
val from_file : string -> json
val from_channel : in_channel -> json
```

The `in_channel` constructor is from the original OCaml standard library, and its use is considered deprecated when using the Core standard library. This leaves us with two ways of parsing the JSON: either from a string or from a file on a filesystem. The next example shows both in action, assuming the JSON record is stored in a file called `book.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")
  print_endline (if phys_equal json1 json2 then "FAIL" else "OK")
```

You can build this by writing a `_tags` file to define the package dependencies, and then running `ocamlbuild`.

```
$ cat _tags
true: package(core,yojson)
true: thread, debug, annot

$ ocamlbuild -use-ocamlfind read_json.native
$ ./read_json.native
OK
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 17) or a database. Finally, the example checks that the two input mechanisms actually resulted in the same OCaml data structure.

The difference between = and ==, and phys_equal in Core

If you come from a C/C++ background, you will probably reflexively use == to test two values for equality. In OCaml, == tests for *physical* equality, and = 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 construc-

ted separately, will not match using this operator. In the JSON example, the `json1` and `json2` values are not identical and so would fail the physical equality test.

The = structural equality operator recursively inspects each field in the two values and tests them individually for equality. In the JSON parsing example, every field will be traversed and checked, and they will check out as equal. 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! In this situation, you must use the physical equality operator, or write a custom comparison function that breaks the recursion.

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:

```
# 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>
```

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.

```
(* 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.

```
$ ocamlbuild -use-ocamlfind 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 program 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:

```
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.

```
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, such as:

```
val member : string -> json -> json
val index : int -> json -> json
val to_string : json -> string
val to_int : json -> int
val filter_string : json list -> string list
```

We'll go through each of these uses one-by-one. Core provides the `|>` pipe-forward which can chain combinators together, and the example code uses this to select and convert values out of the JSON structure. Let's examine some of them in more detail:

```
let open Yojson.Basic.Util in
let title = json |> member "title" |> to_string in
```

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.

```
let tags = json |> member "tags" |> to_list |> filter_string in
let pages = json |> member "pages" |> to_int in
```

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`.

```
let is_online = json |> member "is_online" |> to_bool_option in
let is_translated = json |> member "is_translated" |> to_bool_option in
```

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`.

```
let authors = json |> member "authors" |> to_list in
let names = List.map authors ~f:(fun json -> member "name" json |> to_string) in
```

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:

```
let names =
  json
  |> member "authors"
  |> to_list
  |> List.map ~f:(fun json -> member "name" json |> to_string)
```

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 programer.

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 lay the output out in the JSON format.

```
# let x = `Assoc [ ("key", `String "value") ] ;;
val x : [> `Assoc of (string * [> `String of string ]) list ] =
  `Assoc [("key", `String "value")]
```

In the example above, we've constructed a value `x` that represents a simple JSON object. We haven't actually defined the type of `x` explicitly here, as we're relying on the magic of polymorphic variants to make this all work. The OCaml type system infers a type for `x` based on how you construct the value. In this case only the `Assoc` and `String`

variants are used, and the inferred type only contains these fields without knowledge of the other possible variants that you haven't used yet.

```
# Yojson.Basic.pretty_to_string ;;
- : ?std:bool -> Yojson.Basic.json -> string = <fun>
```

`pretty_to_string` has a more explicit signature that wants an argument of type `Yojson.Basic.json`. When `x` is applied to `pretty_to_string`, the inferred type of `x` is statically checked against the structure of the `json` type to ensure that they're compatible.

```
# Yojson.Basic.pretty_to_string x ;;
- : string = "{ \"key\": \"value\" }"

# Yojson.Basic.pretty_to_channel stdout x ;;
{ "key": "value" }
- : unit = ()
```

In this case, there are no problems. Our `x` value has an inferred type that is a valid subtype of `json`, and the function application just works without us ever having to explicitly specify a type for `x`. 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:

```
# let x = `Assoc ("key", `String "value");;
val x : [> `Assoc of string * [> `String of string ] ] =
  `Assoc ("key", `String "value")

# Yojson.Basic.pretty_to_string x;;
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 isn't *wrong* as such, but can be inconvenient to wade through for larger values. An easy way to narrow down this sort of type error is to add explicit type annotations as a compiler hint about your intentions:

```
# let (x:Yojson.Basic.json) = `Assoc ("key", `String "value");;
Error: This expression has type 'a * 'b
       but an expression was expected of type
         (string * Yojson.Basic.json) list
```

In this case, we've marked the `x` as being of type `Yojson.Basic.json`, and the compiler immediately 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 subtype across module boundaries, but the error messages can be more confusing. However, a bit of careful manual type annotation is all it takes to make tracking down such issues much easier.

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 make the data representation more expressive without having to refer to the original OCaml types. You can always cast 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 ccompiles 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:

```
$ 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.

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. It is based on a pseudo-standard web protocol known as OAuth, and is used to authorize users for GitHub services.

```
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;
}
```

ATD specifications are deliberately similar to OCaml type definitions. Each field can include extra annotations to customise the parsing code for a particular backend. For example, the GitHub `scope` field above is defined as a variant type, but with the actual JSON values being defined explicitly (as lower-case versions).

The ATD spec can be compiled to a number of OCaml targets. Let's run the compiler twice, to generate some OCaml type definitions, and a JSON serialiser.

```
$ atdgen -t github.atd
$ atdgen -j github.atd
```

This 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. The signature looks like this:

```
type scope = [
  | `User | `Public_repo | `Repo | `Repo_status
  | `Delete_repo | `Gist
]

type app = {
  app_name (*atd name *): string;
  app_url (*atd url *): string
}

type authorization_request = {
  auth_req_scopes (*atd scopes *): scope list;
```

```
  auth_req_note (*atd 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:

```
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. *)

val authorization_response_of_string :
  string -> authorization_response
```

This is pretty convenient! We've 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`: work in standard JSON mode, and never print non-standard JSON extensions.
- `-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 you can investigate which 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. Look at the online API documentation (*http://developer.github.com/v3/orgs/*) for Github to see what the JSON schema looks like retrieving the organization information. Then create an ATD file that covers the fields we need. Any extra fields present in the response will be ignored by the ATD parser.

```
(* 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 program that uses this will fetch the JSON and output a one-line summary. You'll also need the `curl` tool installed on your system to fetch the HTTP web pages, as our example below calls `curl` via the `Core_extended.Shell` interface.

```
(* 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
```

Finally, write a short shell script to generate the OCaml `Github_org` parsers via `atdgen`, and build the OCaml command-line interface.

```
$ cat _tags
true: package(core,core_extended,yojson,atdgen)
true: thread, debug, annot

$ cat buildgh.sh
#!/bin/sh

atdgen -t github_org.atd
atdgen -j github_org.atd
ocamlbuild -use-ocamlfind github_org_info.native

$ ./buildgh.sh
```

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.

```
$ curl https://api.github.com/orgs/janestreet
{
  "login": "janestreet",
  "id": 3384712,
  "url": "https://api.github.com/orgs/janestreet",
  "public_repos": 31,
  "public_gists": 0,
  "followers": 0,
  "following": 0,
  "html_url": "https://github.com/janestreet",
  "created_at": "2013-01-25T19:35:43Z",
  "updated_at": "2013-05-23T14:03:06Z",
  "type": "Organization"
}
$ ./github_org_info.native mirage
Mirage account (131943) with 32 public repos
$ ./github_org_info.native janestreet
??? (3384712) with 31 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 17.

2013-05-28
15:22:24

2013-05-28
15:22:24

CHAPTER 15
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 precise 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 = [
| `Object of (string * value) list
| `Array of value array
| `String of string
| `Int of int
| `Float of float
| `True
| `False
| `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*.

```
`Object
  ["title", `String "Cities";
   "cities", `Array
     [|`Object ["name", `String "Chicago"; "zips", `Array [|Int 60601|]];
       `Object ["name", `String "New York"; "zips", `Array [|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 grammer 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 remaning 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
    { `Object obj }
  | LEFT_BRACK; vl = array_values; RIGHT_BRACK
    { `Array vl }
  | s = STRING
    { `String s }
  | i = INT
    { `Int i }
  | x = FLOAT
    { `Float x }
  | TRUE
    { `True }
  | FALSE
    { `False }
  | NULL
    { `Null }
  ;
```

We can read it like this, "A JSON `value` is either an object bracketed with curly braces, or an array bracketed with square braces. or a string, integer, float, etc. In each of the

productions, 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
    { `Object 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 `Object 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 will 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
    { `Object obj }
  | LEFT_BRACK; vl = array_values; RIGHT_BRACK
    { `Array vl }
  | s = STRING
    { `String s }
  | i = INT
    { `Int i }
  | x = FLOAT
    { `Float x }
  | TRUE
    { `True }
  | FALSE
    { `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 }
  ;

array_values: /* empty */
    { [||] }
  | vl = rev_values
    { Array.of_list (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 vetical 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 definion 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 the 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: "Hel
```

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
```

CHAPTER 16
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 paranthetical 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. 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-exressions and a syntax extension for generating such conversion functions. With that synax 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 parenthesis 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 it 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-expressoin. 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 paramters 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 to do this, as follows.

```
# type http_server_config = {
    web_root: string;
    port: int with default(80);
    addr: string with default("localhost");
  } 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>
# 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 that 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 optiona, then you should still be able to parse older version of your config.

2013-05-28
15:22:24

2013-05-28
15:22:24

CHAPTER 17

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-along 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 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 parenthesis around the function on the right-hand side of the bind, and we've 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 114.

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.

`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_lines every time a client connects. *)
let run () =
  let buffer = String.create (16 * 1024) in
  let host_and_port =
    Tcp.Server.create
      ~on_handler_error:`Raise
      (Tcp.on_port 8765)
      (fun _addr r w -> 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 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 13. 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 14
- `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 14.

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 is returned, 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 ne 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 wrods 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 run-time error. When programming in OCaml, some of these errors will show up explicitly in a function's return type, and of them will show up as exceptions. We covered exception handling in OCaml in “Exceptions” on page 116, 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 nontheless 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 make a change that can cause it to fail more predictably, 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 mis-specified.

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 119.

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."
```

CHAPTER 18
Fast Binary Serialization

Now that we've learned the basics of working with Async, let's walk through a small but non-trivial application: a message broker which provides clients with a simple pub/sub API that lets them publish and subscribe to streams of values associated with a given topic.

All of this will require a serialization format for the messages themselves. S-expressions, which we encountered in Chapter 16, are a good serialization format when you need something machine-parseable as well as human readable and editable. But Sexplib's s-expressions are not particularly performant for a couple of reasons:

- s-expression serialization goes through an intermediate type, `Sexp.t`, which must be allocated and is then typically thrown away, putting non-trivial pressure on the garbage collector.
- parsing and printing to strings in an ASCII format can be expensive for types like `int`s, `float`s and `Time.t`s where some real computation needs to be done to produce or parse the ASCII representation.

Bin-prot is a library and syntax extension that addresses these issues by providing efficient serialization in a compact binary format. You can enable Bin-prot in your top-level by typing the following:

```
# #require "bin_prot.syntax";;
```

The syntax extension is triggered on a given type by writing `with bin_io` to the end of the type definition. Thus, we can write:

```
# module M = struct
    type t = { number: int;
               text: string;
               variant : [`Whatever of float | `Nothing ];
             }
    with bin_io
  end;;
module M :
```

```
sig
  type t = {
    number : int;
    text : string;
    variant : [ `Nothing | `Whatever of float ];
  }
  val bin_size_t : t -> int
  val bin_write_t_ :
    Bin_prot.Unsafe_common.sptr ->
    Bin_prot.Unsafe_common.eptr -> t -> Bin_prot.Unsafe_common.sptr
  val bin_write_t : Bin_prot.Common.buf -> pos:int -> t -> int
  val bin_writer_t : t Bin_prot.Type_class.writer0
  val bin_read_t__ : 'a -> 'b -> 'c -> 'd
  val bin_read_t_ :
    Bin_prot.Unsafe_common.sptr_ptr -> Bin_prot.Unsafe_common.eptr -> t
  val bin_read_t : Bin_prot.Common.buf -> pos_ref:int ref -> t
  val bin_reader_t : t Bin_prot.Type_class.reader0
  val bin_t : t Bin_prot.Type_class.t0
end
```

The details of the generated values are not particularly important, but they give you the functionality needed to serialize and deserialize binary messages efficiently.

Clients can either publish values under a topic, or subscribe to the stream of values associated with a given topic. The server will maintain a cache of the last value published under any given topic, so that a subscriber immediately receives the most recently published value under said topic. To make it easier to see what's going on, we'll also implement a query for dumping the current state of the server.

We'll use Async's `Rpc` module for implementing that client/server protocol. The following module specifies the specific message types we'll use, as well as the RPCs that will be used for communicating with the server.

First, we'll start with the basic types.

```
(* file: protocol.ml *)
open Core.Std
open Async.Std

module Username : Identifiable = String
module Topic    : Identifiable = String

module Message = struct
  type t = { text: string;
             topic: Topic.t;
             from: Username.t;
             time: Time.t;
           }
  with sexp, bin_io
end
```

`Username.t` and `Topic.t` are just abstract types that are implemented as strings. The `Message.t` type contains the basic information associated with a message, including the text of the message, who it's from, the topic, and the time it was sent.

Note that the declaration of `Message.t` is followed by the annotation `with sexp, bin_io`. We've seen `with sexp` before in Chapter 16, but `bin_io` is new. S-expressions are a convenient serialization format, but like any human-readable serialization format,

Now we can move on to declaring the `Rpc` protocol we'll use. The `Rpc` module actually supports two different kinds of RPC protocols: an ordinary RPC, represented by an `Rpc.Rpc.t`, is a simple back-and-forth style of communication: the client sends a message, and the server sends a response. In the following, we use `Rpc.Rpc.create` to declare the `Rpc` interface.

```
let publish_rpc = Rpc.Rpc.create
  ~name:"publish"
  ~version:0
  ~bin_query:Message.bin_t
  ~bin_response:Unit.bin_t
```

Note that we declare a name for the RPC and a version number. The name and the version number are used together to identify which RPC is being sent, with the version number allowing the minting of multiple revisions of the RPC, potentially with different types and behavior.

The argument `bin_query` and `bin_response` are used

```ocaml let subscribe_rpc = Rpc.Pipe_rpc.create ~name:"subscribe" ~version:0 ~bin_query:Topic.bin_t ~bin_response:Message.bin_t ~bin_error:String.bin_t

module Dump = struct type single = { topic : Topic.t; message : Message.t; num_subscribers: int; } with sexp,bin_io type t = single list with sexp,bin_io end

let dump_rpc = Rpc.Rpc.create ~name:"dump" ~version:0 ~bin_query:Unit.bin_t ~bin_response:Dump.bin_t ``` # Fast Binary Serialization with bin_prot

S-expressions are a good serialization format when you need something machine-parseable as well as human readable and editable. But Sexplib's s-expressions are not particularly performant for a couple of reasons:

- s-expression serialization goes through an intermediate type, `Sexp.t`, which must be allocated and is then typically thrown away, putting non-trivial pressure on the garbage collector.
- parsing and printing to strings in an ASCII format can be expensive for types like `int`s, `float`s and `Time.t`s where some real computation needs to be done to produce or parse the ASCII representation.

`Bin_prot` is a library that addresses these issues by providing fast serialization in a compact binary format. We'll also introduce the Core `Bigstring` library for handling large binary strings efficiently during this chapter.
```

Using `bin_prot` in the toplevel

The `bin_prot` syntax extension isn't activated by default in the toplevel, but is easily available if you add this to your `~/.ocamlinit` file. You can also just type this in directly into `utop` (with `;;` to finish the line) instead.

```
#require "bin_prot.syntax"
```

The extension is activated by putting `with bin_io` after the type declaration. This looks a bit unsightly in the toplevel because of all the definitions that are generated. We'll elide those definitions in the book, but you can see them for yourself in the toplevel.

Defining a message broker

Here's a small complete example of a program that can read and write values using `bin_io`. Here, the serialization is of types that might be used as part of a message-queue, where each message has a topic, some content, and a source, which is in turn a hostname and a port.

```
open Core.Std

module Message = struct
  module Source = struct
    type t = { hostname: string;
               port: int;
             }
    with bin_io
  end

  type t = { topic: string;
             content: string;
             source: Source.t;
           }
  with bin_io
end
```

You can can combine multiple syntax generators in the same type declaration by comma-separating them, so you could generate both formats via `with bin_io,sexp` above.

Next we need to define how to marshal and unmarshal these messages. The interface is a little more complex than for s-expressions since we don't just want to serialise from the normal OCaml `string`, but also to the `bigstring` type. We'll explain what this is in more detail shortly, but for now think of it as a more efficient alternative for large binary data.

```
let binable =
  (module Message : Binable.S with type t = Message.t)
```

```
let save_message outc msg =
  let s = Binable.to_bigstring binable msg in
  let len = Bigstring.length s in
  Out_channel.output_binary_int outc len;
  Bigstring.really_output outc s
```

The `binable` value above captures all the auto-generated `bin_io` functions into a first-class module of type `Binable.S`. This module has the low-level reader and writer functions which we don't want to have to manually construct.

The `save_message` is then responsible for writing the binary content out to a `bigstring`. It first invokes the `Binable.to_bigstring` on a `Message.t` value to retrieve a marshalled string. It then determines the length of this string, and writes out the length and the string to the output channel.

The `Binable` interface in Core is pretty simple: `type 'a m = (module Binable.S with type t = 'a) val of_bigstring : 'a m -> bigstring -> 'a val to_bigstring : ? prefix_with_length:bool -> 'a m -> 'a -> bigstring val of_string : 'a m -> string -> 'a val to_string : 'a m -> 'a -> string`

Since the `Binable.S` module values are generated for you automatically, the only functions you'll need to regularly use are the conversion functions above.

Reading back the binary value we've just defined is quite similar. We read in the length field, read that much data into a `bigstring`, and convert it to our type using `Binable.of_bigstring`.

```
let load_message inc =
  match In_channel.input_binary_int inc with
  | None -> failwith "length missing from header"
  | Some len ->
    let buf = Bigstring.create len in
    Bigstring.really_input ~pos:0 ~len inc buf;
    Binable.of_bigstring binable buf
```

The code to generate and read and write these messages now just uses the static `Message.t` type, with no need to worry about the marshalling mechanism.

```
(* Generate some example messages *)
let example content =
  let source =
    { Message.Source.
      hostname = "ocaml.org"; port = 2322 }
  in
  { Message.
    topic = "rwo-example"; content; source; }

(* write out three messages... *)
let write_messages () =
  let outc = Out_channel.create "tmp.bin" in
  List.iter ~f:(save_message outc) [
```

```
    example "a wonderful";
    example "trio";
    example "of messages";
  ];
  Out_channel.close outc

(* ... and read them back in *)
let read_messages () =
  let inc = In_channel.create "tmp.bin" in
  for i = 1 to 3 do
    let msg = load_message inc in
    printf "msg %d: %s\n" i msg.Message.content
  done

let () =
  write_messages (); read_messages ()
```

Bigstring

We earlier mentioned that `bigstring` is a more efficient version of `string`. Understanding the difference requires some understanding of how OCaml allocates values. TODO.

Fieldslib

TODO: out of place

One common idiom when using records is to provide field accessor functions for a particular record.

```
type t = { topic: string;
           content: string;
           source: Source.t;
         }

let topic   t = t.topic
let content t = t.content
let source  t = t.source
```

Similarly, sometimes you simultaneously want an accessor to a field of a record and a textual representation of the name of that field. This might come up if you were validating a field and needed the string representation to generate an error message, or if you wanted to scaffold a form in a GUI automatically based on the fields of a record. Fieldslib provides a module `Field` for this purpose. Here's some code for creating `Field.t`'s for all the fields of our type `t`.

```
# module Fields = struct
    let topic =
      { Field.
        name   = "topic";
        setter = None;
```

```
        getter = (fun t -> t.topic);
        fset   = (fun t topic -> { t with topic });
      }
    let content =
      { Field.
        name   = "content";
        setter = None;
        getter = (fun t -> t.content);
        fset   = (fun t content -> { t with content });
      }
    let source =
      { Field.
        name   = "source";
        setter = None;
        getter = (fun t -> t.source);
        fset   = (fun t source -> { t with source });
      }
  end ;;
module Fields :
  sig
    val topic : (t, string list) Core.Std.Field.t
    val content : (t, string) Core.Std.Field.t
    val source : (t, Source.t) Core.Std.Field.t
  end
```

There are several syntax extensions distributed with Core, including:

- **Sexplib**: provides serialization for s-expressions.
- **Bin_prot**: provides serialization to an efficient binary format.
- **Fieldslib**: generates first-class values that represent fields of a record, as well as accessor functions and setters for mutable record fields.
- **Variantslib**: like Fieldslib for variants, producing first-class variants and other helper functions for interacting with variant types.
- **Pa_compare**: generates efficient, type-specialized comparison functions.
- **Pa_typehash**: generates a hash value for a type definition, *i.e.*, an integer that is highly unlikely to be the same for two distinct types.

We'll discuss each of these syntax extensions in detail, starting with Sexplib.

2013-05-28
15:22:24

CHAPTER 19
First class modules

OCaml provides several mechanisms for organizing your programs, including modules and functors, files and compilation units, and classes and objects. Files and compilation units (`.ml` and `.mli` files) are really just a simplified module system. Classes and objects are a different form of organization altogether (as we'll see in Chapter 10. Yet, in each of these cases, there is a clear separation between types and values -- values cannot contain types, and types cannot contain values. And since modules can contain types, modules can't be values.

(yminsky: Instead of saying that ml and mli files are a simplified module system, maybe say that they "provide a simple way of creating modules and interfaces", or some such? It's not like there's a simplified module system floating around)

(yminsky: consider dropping "Yet" in the above.)

Next, we'll relax this restriction with *first-class modules*. "First-class" means that modules can be passed around as ordinary values that can be created from and converted back to regular modules. This is a relatively recent addition to the OCaml language, and while it might seem trivial to say, it has profound consequences on the language. First-class modules are strictly more expressive than any other organization mechanism, including classes and objects. Once you use first-class modules, you'll never want to go back.

(yminsky: I wouldn't say they're strictly more expressive. For example, they don't give you a way of expressing sub typing relationships effectively, which objects do.)

This is not say that first-class modules should be used indiscriminately. When you pass modules as values, the reason is to support dynamic behavior, and this can have a negative impact on understandability. As we proceed, we'll compare first-class modules to other techniques, and suggest alternatives when it seems appropriate.

_(jyh: Original text You can think of OCaml as being broken up into two sub-language: 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 definition 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. As we'll see, letting modules into the core language makes it possible to use more flexible and dynamic module-oriented designs.)_

Another trivial example

Much as we did with functors, we'll start out with an utterly trivial example, to allow us to show the basic mechanics of first class modules with a minimum of fuss.

A first-class module is created by packaging up a module with a signature that it satisfies. The following defines a simple signature and a module that matches it.

```
# module type X_int = sig val x : int end;;
module type X_int = sig val x : int end
# module Three : X_int = struct let x = 3 end;;
module Three : X_int
# Three.x;;
- : int = 3
```

We can then create a first-class module using the `module` keyword.

```
# let three = (module Three : X_int);;
val three : (module X_int) = <module>
```

Note that the type of the first-class module, `(module X_int)`, is based on the name of the signature that we used in constructing it.

To get at the contents of `three`, we need to unpack it into a module again, which we can do using the `val` keyword.

```
# module New_three = (val three : X_int) ;;
module New_three : X_int
# New_three.x;;
- : int = 3
```

Using these conversions as building blocks, we can create tools for working with first-class modules in a natural way. The following shows the definition of two function, `to_int`, which converts a `(module X_int)` into an `int`. And `plus`, which adds 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 start operating on our `(module X_int)`'s in a more natural style, taking full 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
```

Of course, all we've really done with this example is come up with a more cumbersome way of working with integers. Let's see what happens when with work with more complex abstract types.

Standard vs. first-class modules

(yminsky: I'm not in solve with the example. It feels in some sense too artificial, and that aside, when you get to the end of the example, you haven't really gotten any juice of first-class modules)

(yminsky: using "standard" in quotes seems a little awkward. Maybe just drop the quotes, and talk about standard or ordinary modules directly?)

Let's compare the style of "standard" modules to first-class modules, using a simple library of abstract geometric shapes. In a "standard" module definition, we would define the shapes using abstract data types, where there is a type `t` that defines the actual representation, and the module would include functions that operate on the values of type `t`. In the following code, the module type `Shape` defines the type of generic shape, and the modules `Rectangle` and `Line` implement some concrete shapes.

```
module type Shape = sig
  type t
  val area : t -> int
  val position : t -> int * int
end

module Rectangle = struct
   type t = { width : int; height : int; x : int; y : int }
   let make ~x ~y ~width ~height =
      { width = width; height = height; x = x; y = y }
   let area { width = width; height = height } = width * height
   let position { x = x; y = y } = (x, y)
end

module Line = struct
```

```
   type t = { dx : int; dy : int; x : int; y : int }
   let make ~x ~y ~dx ~dy = { dx = dx; dy = dy; x = x; y = y }
   let area _ = 0
   let position { x = x; y = y } = (x, y)
end
```

Next, if we want to define a generic shape that is either a rectangle or a line, we would probably use a variant type. The following module `Shapes` is entirely boilerplate. We define the variant type, then functions to perform a dynamic dispatch based on the type of object.

```
module Shapes = struct
   type t = [ `Rect of Rectangle.t | `Line of Line.t ]
   let make_rectangle = Rectangle.make
   let make_line = Line.make
   let area = function
      `Rect r -> Rectangle.area r
    | `Line l -> Line.area l
   let position = function
      `Rect r -> Rectangle.position r
    | `Line l -> Line.position l
end;;
```

In fact, confronted with this boilerplate, we would probably choose not use modules at all, but simply define a single module with a variant type and the code for all of the shapes. This isn't to say that separate code for separate shapes is bad, it just means that the language doesn't support it well (at least with standard modules).

With first-class modules, the situation changes, but we have to dispense with the representation type altogether. For immutable shapes, the implementation is now trivial.

```
# module type Shape = sig
    val area : int
    val position : int * int
  end;;
module type Shape = sig val area : int val position : int * int end
# let make_rectangle ~x ~y ~width ~height =
   let module Rectangle = struct
      let area = width * height
      let position = (x, y)
   end in
   (module Rectangle : Shape);;
val make_rectangle :
  x:int -> y:int -> width:int -> height:int -> (module Shape) = <fun>
# let make_line ~x ~y ~dx ~dy =
   let module Line = struct
      let area = 0
      let position = (x, y)
   end in
   (module Line : Shape);;
val make_line : x:int -> y:int -> dx:'a -> dy:'b -> (module Shape) = <fun>
```

For mutable shapes, it isn't much different, but we have to include the state as values in the module implementations. For this, we'll define a representation type `t` in the module implementation, and for rectangles, a value `rect` of that type. The code for lines is similar.

```
# module type Shape = sig
     val area : unit -> int
     val position : unit -> int * int
     val moveby : dx:int -> dy:int -> unit
     val enlargeby : size:int -> unit
  end;;
module type Shape = ...
# let make_rectangle ~x ~y ~width ~height =
    let module Rectangle = struct
      type t = { mutable x : int; mutable y : int;
                 mutable width : int; mutable height : int }
      let rect = { x = x; y = y; width = width; height = height }
      let area () = rect.width * rect.height
      let position () = (rect.x, rect.y)
      let moveby ~dx ~dy =
         rect.x <- rect.x + dx;
         rect.y <- rect.y + dy
      let enlargeby ~size =
         rect.width <- rect.width * size;
         rect.height <- rect.height * size
    end in
    (module Rectangle : Shape);;
val make_rectangle :
  x:int -> y:int -> width:int -> height:int -> (module Shape) = <fun>
```

A more complete example -- containers

So far, we haven't done anything that really needs modules. The type `Shape` could just as well be specified as a record type `type shape = { area : int; position : int * int; ... }`.

To explore the topic more fully, let's implement a system of dynamic containers. OCaml already provides a set of standard containers like `List`, `Set`, `Hashtbl`, etc., but these types have to be selected statically. If a function expects a value of type `Set.Make(ElementType).t`, then you have to pass it a set of exactly that type. What we would like is a kind of container where the container implementation is chosen by the caller. We define an abstract *interface*, as a module type, then define one or more concrete module implementations.

Let's start by defining an abstract container interface. It contains some elements of type `elt`, and functions to examine and iterate through the contents. For convenience, we also define a normal type `'a container` to represent containers with elements of type `'a`.

```
module type Container = sig
```

```
   type elt
   val empty : unit -> bool
   val iter : (elt -> unit) -> unit
   val fold : ('a -> elt -> 'a) -> 'a -> 'a
end;;

type 'a container = (module Container with type elt = 'a)
```

Imperative containers

For imperative containers, will also want functions to mutate the contents by adding or removing elements. For example, a stack can be implemented as a module `Stack` that includes all the functions in the generic `Container` module, as well as functions to push and pop elements.

```
module type Stack = sig
   include Container
   val push : elt -> unit
   val pop : unit -> elt
end;;

type 'a stack = (module Stack with type elt = 'a)
```

Now that the types are defined, the next step is to define a concrete container implementation. For this simple example, we'll use a list to represent a stack. The function `make_list_stack` constructs module implementation using a `let module` construction, then returns the result.

```
# let make_list_stack (type element) () : element stack =
    let module ListStack = struct
      type elt = element
      let contents = ref []
      let empty () = !contents = []
      let iter f = List.iter f !contents
      let fold f x = List.fold_left f x !contents
      let push x = contents := x :: !contents
      let pop () =
         match !contents with
            x :: rest -> contents := rest; x
          | [] -> raise (Invalid_argument "stack is empty")
    end in
    (module ListStack : Stack with type elt = element);;
val make_list_stack : unit -> 'a stack = <fun>
```

Note the use of the explicit type parameter `element`. This is required because the use of a type variable in the module definition (like `type elt = 'a`) would be rejected by the compiler. The construction and use of the stack is straightforward.

```
# let demo (s : int stack) =
    let module S = (val s) in
    S.push 5;
```

```
    S.push 17;
    S.iter (fun i -> Printf.printf "Element: %d\n" i);;
val demo : int stack -> unit = <fun>
# demo (make_list_stack ());;
Element: 17
Element: 5
- : unit = ()
```

The `demo` function is entirely oblivious to the implementation of the stack. Instead of passing a module implementation based on lists, we could pass a different implementation based on arrays.

We could go on to define other containers, sets, dictionaries, queues, etc. but the implementations would be similar to what we have seen. Instead, let's look at functional data structures, which require a little more work to express.

Pure functional containers

Imperative data structures have simpler types that functional ones because the return type of imperative functions is just `unit`. When we look at pure functional data structures, we immediately run into a problem with type recursion.

```
# module type Container = sig
    type elt
    val empty : bool
    val iter : (elt -> unit) -> unit
    val fold : ('a -> elt -> 'a) -> 'a -> 'a
    val add : elt -> (module Container)
  end;;
Characters 160-178:
     val add : elt -> (module Container)
                      ^^^^^^^^^^^^^^^^^^
Error: Unbound module type Container
```

The problem here is that module type definitions are not recursive -- we can't use the type being defined in its own definition.

Recursive modules provide a solution, but it requires a "trick", where we define a module that is equal to itself. This module contains only type definitions, and the only purpose of the outer recursive module is to allow the recursion in the definition. While we're at it, let's include a `map` function with the usual semantics.

```
module rec Container : sig
   module type T = sig
      type elt
      val empty : bool
      val iter : (elt -> unit) -> unit
      val fold : ('a -> elt -> 'a) -> 'a -> 'a
      val map : (elt -> 'a) -> 'a Container.t
      val add : elt -> elt Container.t
   end
```

```
    type 'a t = (module Container.T with type elt = 'a)
end = Container;;
```

There are several ways to write this model, but this definition is convenient because it defines both a module type `Container.T` and a value type `'a Container.t`. The outer recursive module `Container` allows the module type `T` to refer to the value type `t` and *vice versa*. Note that the module `Container` is defined as itself (as `Container`).

With this first technicality out of the way, the next one is how to construct values of type `Container.t`. In the imperative version of the stack, we used a function `make_list_stack`. We want to do the same here, but the function definition must be both recursive and polymorphic.

```
# let make_stack () =
    let rec make : 'a. 'a list -> 'a Container.t = fun
      (type element) (contents : element list) ->
      let module NewList = struct
         type elt = element
         let empty = contents = []
         let iter f = List.iter f contents
         let fold f x = List.fold_left f x contents
         let map f = make (List.map f contents)
         let add x = make (x :: contents)
      end in
      (module NewList : Container.T with type elt = element)
   in
   make [];;
val make_stack : unit -> 'a Container.t = <fun>
```

The recursion here is particularly important. The functions `map` and `add` return new collections, so they call the function `make` recursively. The explicit polymorphic type `make : 'a. 'a list -> 'a Container.t` means that the function `make` is properly polymorphic, so that the `map` function is polymorphic.

Now that the construction is done, the usage is similar to the imperative case, except that now the data structure is functional.

```
# let demo (s : int Container.t) =
    let module S = (val s) in
    let module S = (val (S.add 5)) in
    let module S = (val (S.add 17)) in
    S.iter (fun i -> Printf.printf "Int Element: %d\n" i);
    let s = S.map (fun i -> float_of_int i +. 0.1) in
    let module S = (val s) in
    S.iter (fun x -> Printf.printf "Float Element: %f\n" x);
    s;;
val demo : int Container.t -> float Container.t = <fun>
# demo (make_stack ());;
Int Element: 17
Int Element: 5
Float Element: 17.100000
```

```
Float Element: 5.100000
- : unit = ()
```

The syntactic load here is pretty high, requiring a `let module` expression to name every intermediate value. First-class modules are fairly new to the language, and this is likely to change, but in the meantime the syntactic load can be pretty daunting.

Let's look a some other more typical examples, where dynamic module selection is more localized.

_(jyh: This is a rough draft, I'm not sure about the ordering and the topics, yet. Switching back to Ron's text now.)

Dynamically choosing a module

Perhaps the simplest thing you can do with first-class modules that you can't do without them is to pick the implementation of a module at runtime.

Consider an application that does I/O multiplexing using a system call like `select` to determine which file descriptors are ready to use. There are in fact multiple APIs you might want to use, including `select` itself, `epoll`, and `libev`, where different multiplexers make somewhat different performance and portability trade-offs. You could support all of these in one application by defining a single module, let's call it `Mutliplexer`, whose implementation is chosen at run-time based on an environment variable.

To do this, you'd first need an interface `S` that all of the different multiplexer implementations would need to match, and then an implementation of each multiplexer.

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

(* An interface the OS-specific functionality *)
module type S = sig ... end

(* The implementations of each individual multiplexer *)
module Select : S = struct ... end
module Epoll  : S = struct ... end
module Libev  : S = struct ... end
```

We can choose the first-class module that we want based on looking up an environment variable.

```
let multiplexer =
  match Sys.getenv "MULTIPLEXER" with
  | None
  | Some "select" -> (module Select : S)
  | Some "epoll"  -> (module Epoll : S)
  | Some "libev"  -> (module Libev : S)
  | Some other -> failwithf "Unknown multiplexer: %s" other ()
```

Finally, we can convert the resulting first-class module back to an ordinary module, and then include that so it becomes part of the body of our module.

```
(* The final, dynamically chosen, implementation *)
include (val multiplexer : S)
```

Example: A service bundle

This section describes the design of a library for bundling together multiple services, where a service is a piece of code that exports a query interface. A service bundle combines together multiple individual services under a single query interface that works by dispatching incoming queries to the appropriate underlying service.

The following is a first attempt at an interface for our `Service` module, which contains both a module type `S`, which is the interface that a service should meet, as well as a `Bundle` module which is for combining multiple services.

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

open Core.Std

(** The module type for a service. *)
module type S = sig
  type t
  val name           : string
  val create         : unit -> t
  val handle_request : t -> Sexp.t -> Sexp.t Or_error.t
end

(** Bundles multiple services together *)
module Bundle : sig
  type t
  val create : (module S) list -> t
  val handle_request : t -> Sexp.t -> Sexp.t Or_error.t
  val service_names  : t -> string list
end
```

Here, a service has a state, represented by the type `t`, a name by which the service can be referenced, a function `create` for instantiating a service, and a function by which a service can actually handle a request. Here, requests and responses are delivered as s-expressions. At the `Bundle` level, the s-expression of a request is expected to be formatted as follows:

```
(<service-name> <body>)
```

where `<service_name>` is the service that should handle the request, and `<body>` is the body of the request.

Now let's look at how to implement `Service`. The core datastructure of `Bundle` is a hashtable of request handlers, one per service. Each request handler is a function of type `(Sexp.t -> Sexp.t Or_error.t)`. These request handlers really stand in for the underlying service, with the particular state of the service in question being hidden inside of the request handler.

The first part of `service.ml` is just the preliminaries: the definition of the module type `S`, and the definition of the type `Bundle.t`.

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

open Core.Std

module type S = sig
  type t
  val name           : string
  val create         : unit -> t
  val handle_request : t -> Sexp.t -> Sexp.t Or_error.t
end

module Bundle = struct
  type t = { handlers: (Sexp.t -> Sexp.t Or_error.t) String.Table.t; }
```

The next thing we need is a function for creating a `Bundle.t`. This `create` function builds a table to hold the request handlers, and then iterates through the services, unpacking each module, constructing the request handler, and then putting that request handler in the table.

```
  (** Creates a handler given a list of services *)
  let create services =
    let handlers = String.Table.create () in
    List.iter services ~f:(fun service_m ->
      let module Service = (val service_m : S) in
      let service = Service.create () in
      if Hashtbl.mem handlers Service.name then
        failwith ("Attempt to register duplicate handler for "^Service.name);
      Hashtbl.replace handlers ~key:Service.name
        ~data:(fun sexp -> Service.handle_request service sexp)
    );
    {handlers}
```

Note that the `Service.t` that is created is referenced by the corresponding request handler, so that it is effectively hidden behind the function in the `handlers` table.

Now we can write the function for the bundle to handle requests. The handler will examine the s-expression to determine the body of the query and the name of the service to dispatch to. It then looks up the handler calls it to generate the response.

```
  let handle_request t sexp =
    match sexp with
    | Sexp.List [Sexp.Atom name;query] ->
```

```
      begin match Hashtbl.find t.handlers name with
      | None -> Or_error.error_string ("Unknown service: "^name)
      | Some handler ->
        try handler query
        with exn -> Error (Error.of_exn exn)
      end
    | _ -> Or_error.error_string "Malformed query"
```

Last of all, we define a function for looking up the names of the available services.

```
  let service_names t = Hashtbl.keys t.handlers

end
```

To see this system in action, we need to define some services, create the corresponding bundle, and then hook that bundle up to some kind of client. For simplicity, we'll build a simple command-line interface. There are two functions below: `handle_one`, which handles a single interaction; and `handle_loop`, which creates the bundle and then runs `handle_one` in a loop.

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

open Core.Std

(** Handles a single request coming from stdin *)
let handle_one bundle =
  printf ">>> %!"; (* prompt *)
  match In_channel.input_line stdin with
  | None -> `Stop (* terminate on end-of-stream, so Ctrl-D will exit *)
  | Some line ->
    let line = String.strip line in (* drop leading and trailing whitespace *)
    if line = "" then `Continue
    else match Or_error.try_with (fun () -> Sexp.of_string line) with
    | Error err ->
      eprintf "Couldn't parse query: %s\n%!" (Error.to_string_hum err);
      `Continue
    | Ok query_sexp ->
      let resp = Service.Bundle.handle_request bundle query_sexp in
      Sexp.output_hum stdout (<:sexp_of<Sexp.t Or_error.t>> resp);
      Out_channel.newline stdout;
      `Continue

let handle_loop services =
  let bundle = Service.Bundle.create services in
  let rec loop () =
    match handle_one bundle with
    | `Stop -> ()
    | `Continue -> loop ()
  in
  loop ()
```

Now we'll create a couple of toy services. One service is a counter that can be updated by query; and the other service lists a directory. The last line then kicks off the shell with the services we've defined.

```
module Counter : Service.S = struct
  type t = int ref

  let name = "update-counter"
  let create () = ref 0

  let handle_request t sexp =
    match Or_error.try_with (fun () -> int_of_sexp sexp) with
    | Error _ as err -> err
    | Ok x ->
      t := !t + x;
      Ok (sexp_of_int !t)
end

module List_dir : Service.S = struct
  type t = unit

  let name = "ls"
  let create () = ()

  let handle_request () sexp =
    match Or_error.try_with (fun () -> string_of_sexp sexp) with
    | Error _ as err -> err
    | Ok dir -> Ok (Array.sexp_of_t String.sexp_of_t (Sys.readdir dir))
end

let () =
  handle_loop [(module List_dir : Service.S); (module Counter : Service.S)]
```

And now we can go ahead and start up the client.

```
$ ./service_client.byte
>>> (update-counter 1)
(Ok 1)
>>> (update-counter 10)
(Ok 11)
>>> (ls .)
(Ok
 (_build _tags service.ml service.mli service.mli~ service.ml~
  service_client.byte service_client.ml service_client.ml~))
>>>
```

Now, let's consider what happens to the design when we want to make the interface of a service a bit more realistic. In particular, right now services are created without any configuration. Let's add a config type to each service, and change the interface of `Bundle` so that services can be registered along with their configs. At the same time, we'll change the `Bundle` API to allow services to be changed dynamically, rather than just added at creation time.

2013-05-28
15:22:24

PART III

The Runtime System

2013-05-28
15:22:24

CHAPTER 20
Foreign Function Interface

OCaml has several options available to interact with non-OCaml code. The compiler toolchain can link to external system libraries and also produce standalone native object code that can be embedded within other non-OCaml libraries or applications.

The `ctypes` foreign function library

The simplest foreign function interface in OCaml doesn't even require you to write any C code. The `ctypes` library lets you describe the C interface in pure OCaml, and the library takes care of finding the C symbols and invoking the function call with the appropriate arguments.

Let's dive straight into an example to show you how the library looks. We'll use 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

TODO `ctypes` is not yet available on OPAM, but will be soon. For now, install it manually:

```
$ git clone git://github.com/ocamllabs/ocaml-ctypes
$ cd ocaml-ctypes
$ make && make install
```

It will then be available via the `ctypes` ocamlfind package. You will also need the `ncurses` library for the first example. It is pre-installed on MacOS X and Debian Linux includes it as the `ncurses-dev` package.

Example: an ncurses terminal interface

Ncurses is a library to 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 documentation (*http://www.gnu.org/software/ncurses/*) explains the full C interface, but here's an excerpt that we need for a basic binding to OCaml. The full header file can usually be found in `/usr/include/ncurses.h` on MacOS X or Linux.

```
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` library calls either work on the current pseudo-terminal, or on a window that has been created via the library. The `WINDOW` typdef represents this external `ncurses` library state. The specific contents of the structure don't matter; OCaml code just needs to store the pointer and pass it back to `ncurses` library calls that then dereference its contents.

There are two library calls that create `WINDOW` pointers. The `initscr` function initialises the library and returns the global window, and `newwin` allows further windows to be created. The `WINDOW` pointer can also be passed to terminal drawing functions such as `mvwaddrstr` (there are over 200 library calls in `ncurses`, so we are just binding a select few for this example). The terminal is updated when `refresh` or `wrefresh` are called. All other drawing calls just manipulate library data structures without actually changing the screen layout.

The `ctypes` library provides an OCaml interface that lets you declare these C functions as OCaml values. The library takes care of converting the OCaml arguments into the C calling convention, invoking the foreign call within the `ncurses` library, and finally returning the result as an OCaml value.

```
(* ncurses.ml 1/3 *)
open Ctypes.Ffi.C
open Type

type window = unit ptr
let window = ptr void
```

We first define a `window` type to represent the C `WINDOW` pointer. The `unit ptr` type is equivalent to a `void *` pointer (we'll constrain the signature later on to avoid mixing up different pointer types). We also need a value representing this `window` type to pass to the `ctypes` library, The `window` value is built using the `Ctypes.Ffi.C.ptr` function. The next step is to use this value to build a foreign function call to `initscr`.

```
(* ncurses.ml 2/3 *)
let initscr =
  foreign "initscr" (void @-> (returning window))
```

The `foreign` function takes two parameters: the C function call name, and a value that defines the C function arguments and return type. This definition can contain any of the C types (including function pointers), and is built using functions defined in `Ctypes.Ffi.C`.

Basic C types such as `void` are defined as values in `Ffi.Types.C`, and we have previously defined `window` as a `void ptr`. The `@->` operator adds an argument to the C parameter list and the `returning` function terminates the parameter list and declares the return value. The remainder of the `Ncurses` implementation expands on these definitions for the other library functions.

```
(* 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 headers from earlier in the chapter. They use the basic C types defined in `FFi.Types.C` such as `void` or `int`. The `string` value maps from OCaml strings (which have a specific length) onto C character buffers (whose length is defined by a null characters).

The module signature for `ncurses.mli` looks much like a normal OCaml signature. You can infer it from `ncurses.ml` by running:

```
$ ocamlfind ocamlc -i -package ctypes ncurses.ml
```

We've tweaked the automatic signature to make the `type window` abstract, and the result is below:

```
type window

val window    : window Ffi.C.Type.t
val initscr   : unit   -> window
val endwin    : unit   -> unit
val refresh   : unit   -> unit
val wrefresh  : window -> unit
val newwin    : int    -> int -> int -> int -> window
val addch     : char   -> unit
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 to external users so that it can only be constructed via the `Ncurses.initscr` function. This interface is now safe to use externally, since window pointers cannot be mixed up with other `void` pointers (e.g. those obtained by other libraries). Here's what a "hello world" that uses the library looks like:

```
(* 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 ()
```

This code can be compiled by linking against the `ctypes` and `unix` ocamlfind packages.

```
$ ocamlfind ocamlopt -linkpkg -package ctypes -package unix \
  -cclib -lncurses ncurses.mli ncurses.ml hello.ml -o hello
```

Running `./hello` should now display a Hello World in your terminal! The compilation line above includes `-cclib -lncurses` to make the OCaml compiler link the output to the `ncurses` C library, which in turns makes the C symbols available to the program when it starts. You should 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")
```

Defining basic C formats from OCaml

`Ctypes` provides an `Ffi.C` module that lets you describe not only basic C types, but also more complex structures and unions. It defines abstract OCaml types for these within the module.

```
(* Ctypes.Ffi.C *)
type 'a typ        (** Basic C type  *)
type 'a ptr        (** C pointer *)
type 'a array      (** C array of 'a values *)
type 'a structure  (** C `struct` *)
type 'a union      (** C `union` *)
type 'a abstract   (** Abstract C pointer *)
```

The module also defines constructors for the familiar C basic types. These constructors build a value of `Ctype.FFi.C.typ` that represents that basic C type.

```
(* Ctypes.Ffi.C 1/3 *)
module Type : sig
  type 'a t = 'a typ

  val void  : unit t
  val char : char t
  val schar : int t
  val float : float t
  val double : float t
  val short : int t
  val int   : int t
  val long  : long t
  val llong  : llong t
  val nativeint : nativeint t
  val int8_t : int t
  val int16_t : int t
  val int32_t : int32 t
  val int64_t : int64 t
...
```

These functions return an `'a typ` where the `'a` component is the OCaml representation of the C type. For example, OCaml only supports double-precision floating point numbers and so the C `float` and `double` functions both map to the OCaml `float` type.

The `Ffi.Unsigned` and `Ffi.Signed` modules provide optimized implementations of C types such as `llong` (for `long long` 64-bit values) or `int32_t` (for signed 32-bit values). The module also defines some more advanced C types that aren't straightforward mappings to and from OCaml.

```
(* Ctypes.Ffi.C 2/3 *)
  val string : string t
  val abstract : size:int -> alignment:int -> 'a abstract t
  val array : int -> 'a t -> 'a array t
  val ptr : 'a t -> 'a ptr t
```

Strings in C are null-terminated character arrays, while OCaml strings have a fixed-length specified in the value header. The `string` function creates a safe mapping between these two representations by copying the data to and from OCaml strings and C character buffers.

Arrays and pointers can be built from basic types by using the corresponding `array` and `ptr` functions. The `abstract` function accepts size and alignment requirements and ensures that these are satisfied when this type is used in a function call. Notice that the result types of these functions all share the same `Ffi.Type.C.t` type as the basic C type definitions, which means that they can all be used interchangeably.

The next step is to group collections of C types into function definitions, which are represented by type `'a Ffi.Type.f`.

```
(* Ctypes.Ffi.C 3/3 *)
  type 'a f
  val ( @-> ) : 'a t -> 'b f -> ('a -> 'b) f
  val returning : 'a t -> 'a f
  val funptr : ('a -> 'b) f -> ('a -> 'b) t
```

Sequences of `'a typ` values are constructed by using the `@->` and `returning` functions. You can even exchange function pointers between OCaml and C by wrapping the OCaml callback using `funptr`. The library takes care of the garbage collector interface to ensure that the OCaml value isn't moved around while the C library is holding a reference to the value. We'll come back to an example of using `funptr` later in the chapter.

Arrays, structures and unions

Arrays in C are contiguous blocks of the same value. Any of the basic types defined earlier can be allocated as blocks via the `Ffi.C.Type.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.

Structures in C can contain a mixture of types, and, like OCaml records, their order is significant. The `Ffi.C.Type.Struct` module defines combinators to make this definition as easy basic types were. Let's look at an with an example by binding some time-related UNIX functions that use C structures in their interface.

Example: binding UNIX date functions

The UNIX standard C library defines several useful time and date functions in `<time.h>` (usually found in `/usr/include` on a Linux or MacOS X system). The `local time` function has the following signature and return value:

```
/* /usr/include/time.h */

struct tm {
  int     tm_sec;         /* seconds after the minute [0-60] */
  int     tm_min;         /* minutes after the hour [0-59] */
  int     tm_hour;        /* hours since midnight [0-23] */
  int     tm_mday;        /* day of the month [1-31] */
  int     tm_mon;         /* months since January [0-11] */
  int     tm_year;        /* years since 1900 */
  int     tm_wday;        /* days since Sunday [0-6] */
  int     tm_yday;        /* days since January 1 [0-365] */
  int     tm_isdst;       /* Daylight Savings Time flag */
};

time_t time(time_t *);
struct tm *localtime(const time_t *);
```

This example is more complicated than ncurses for a couple of reasons. We need to allocate some external memory to store a `time_t` value, and pass that memory into the `time` library call to obtain the current timezone. This `time_t` value is passed to the `localtime` library call, which then returns a pointer to the `struct tm`.

The `time_t` and many other standard POSIX types are already provided by the `Ffi.Pos ixTypes` module. Let's start by defining the OCaml mapping to `struct tm`:

```
open Ffi.C
open Type
open PosixTypes
open Struct
```

```
type tm
let tm = structure "tm"
let tm_sec   = tm *:* int (* seconds *)
let tm_min   = tm *:* int (* minutes *)
let tm_hour  = tm *:* int (* hours *)
let tm_mday  = tm *:* int (* day of the month *)
let tm_mon   = tm *:* int (* month *)
let tm_year  = tm *:* int (* year *)
let tm_wday  = tm *:* int (* day of the week *)
let tm_yday  = tm *:* int (* day in the year *)
let tm_isdst = tm *:* int (* daylight saving time *)
let () = seals (tm : tm structure typ)
```

This is a fairly mechanical translation from the C structure by using the magic of the `*:*` combinator provided by the `Ffi.C.Struct` module. The structure is initialised in the `tm` variable via the `structure` allocator. The fields of the structure are then added in sequence. Each new field mutates the `tm` structure to append its name and offset. The structure is finalized via `seals` when all the fields have been added, and the structure can now be used.

The OCaml definitions of `time` and `localtime` are now straightforward calls to `foreign`, just like our earlier `ncurses` example.

```
let time = foreign "time" (ptr time_t @-> syscall time_t)
let asctime = foreign "asctime" (ptr tm @-> returning string)
let localtime = foreign "localtime" (ptr time_t @-> returning (ptr tm))
```

The OCaml signature for this definition looks like this:

```
open Ffi.C
type tm
val tm : tm structure typ
val tm_sec : (int, tm) Struct.field
val tm_min : (int, tm) Struct.field
val tm_hour : (int, tm) Struct.field
val tm_mday : (int, tm) Struct.field
val tm_mon : (int, tm) Struct.field
val tm_year : (int, tm) Struct.field
val tm_wday : (int, tm) Struct.field
val tm_yday : (int, tm) Struct.field
val tm_isdst : (int, tm) Struct.field

val time : PosixTypes.time_t ptr -> PosixTypes.time_t
val localtime : PosixTypes.time_t ptr -> tm structure ptr
val asctime : PosixTypes.time_t ptr -> string
```

Some of the FFI types are still exposed in this signature due to the manual memory interface required by the C libraries. The OCaml `time` and `localtime` can be used by allocating external memory and constructing values of type `time_t ptr`.

```
let () =
  let timep = Ptr.allocate ~count:1 time_t in
```

```
let time = time timep in
let tm = localtime timep in
print_endline (asctime tm)
```

The `Ptr.allocate` function allocates memory via `malloc` and creates an OCaml value to point to this external memory buffer. This OCaml value (`timep` in the example) has a finalizer function which frees the external memory when it is garbage collected. The `timep` pointer is passed into the `time` library call, which modifies it in-place. The same pointer is subsequently passed to `localtime`, whose return `tm` structure is converted into an OCaml string via the `asctime` function. The garbage collector is free to free `timep` during the next collection cycle.

Unions in C are a collection of named structures that can be mapped onto the same memory. They are also supported in the `ctypes` library via the `Ffi.C.Union` module, although we won't go into more detail here.

Callbacks between C and OCaml

TODO: the fts(3) interface.

2013-05-28
15:22:24

2013-05-28
15:22:24

CHAPTER 21
Memory Representation of Values

Much of the static type information contained within an OCaml program is checked and discarded at compilation time, leaving a much simpler *runtime* representation for values. You'll need to understand the runtime memory format in order to write programs that are efficient and compact, and to to interpret the output of profiling tools.

In this chapter, you'll learn:

- The architecture of the OCaml garbage collector.
- The layout of OCaml values as blocks within the major and minor heaps.
- The representation of all the OCaml types in runtime blocks.

This chapter is primarily informational, but it helps to know all this to understand the compilation process in Chapter 22, handling external memory in Chapter 25, or tracing and profiling your programs in Chapter 24.

You might also want to interface with the OCaml C runtime directly instead of using the simpler `ctypes` library described earlier in Chapter 20. This could be for performance reasons or a more specialised embedded or kernel execution environment.

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 talk more about the compilation process in Chapter 22.

Let's start by explaining the garbage collector architecture, and then move onto the memory layout of OCaml values.

The garbage collector

A running OCaml program uses blocks of memory (i.e. contiguous sequences of words in RAM) to represent many of the values that it deals with 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.

The OCaml runtime is a C library that provides a collection of routines that can be called by running OCaml programs. The runtime manages a *heap*, which is a collection of memory regions it obtains from the operating system using *malloc(3)*. The OCaml runtime uses these memory regions to hold *heap blocks*, which it fills up in response to allocation requests by the OCaml program.

The mark and sweep GC strategy

When there isn't enough memory available to satisfy an allocation request from the already-allocated heap blocks, the runtime system invokes the *garbage collector* (or GC). An OCaml program does not explicitly free a heap block when it is done with it. The GC determines which heap blocks are "live" and which heap blocks are "dead", i.e. no longer in use. Dead blocks are collected and their memory made available for re-use by the application.

The garbage collector does not keep constant track of blocks as they are allocated and used. Instead, it regularly scans blocks by starting from a set of *roots*, which are values that the application always has access to (such as the stack). The GC maintains a directed graph in which heap blocks are nodes. 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. This strategy is commonly known as "mark and sweep" collection.

Generational garbage collection

The typical OCaml programming style typically involves allocating many small blocks of memory 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 keeps separate memory regions to hold blocks based on how long the blocks have been live. OCaml's heap is split in two:

- a small, fixed-size *minor heap* where most most blocks are initially allocated
- a large, variable-sized *major heap* for blocks that have been live longer or are larger than 4KB.

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 the two mechanisms next.

The fast minor heap

The minor heap is 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 in which the pointer to the end of the heap is incremented by the desired size.

To garbage collect the minor heap, OCaml uses *copying collection* to copy 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.

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, the minor collection would require scanning the much larger major heap.

OCaml maintains a set of such *inter-generational pointers*, and the compiler introduces a write barrier to update this set whenever a major-heap block is modified to point at a minor-heap block. We'll talk more about the implications of the write barrier in Chapter 24.

The long-lived major heap

The major heap 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. This list is used to satisfy allocation requests for OCaml blocks. The major heap is typically much larger than the minor heap, and is cleaned via a mark-and-sweep garbage collection algorithm.

- The *mark* phase traverses 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 moves live blocks to eliminate the gaps of free memory into a freshly allocated heap. This prevents heap blocks fragmenting in long-lived programs.

A major heap garbage collection must *stop the world* (that is, halt the application) in order to ensure that blocks can be safely moved around without this move being observed by the live application. The mark-and-sweep phases run incrementally over slices of memory 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.

The `Gc` module lets you control all these parameters from your application. Although the defaults are usually sensible, understanding them and tuning them is an important concern we will discuss later in Chapter 24.

The representation of OCaml values

We've described the overall memory layout, but not what the contents of each block contains. Every OCaml variable points to a `value` at runtime. A value is a single memory

word that is either an integer or a pointer. The OCaml runtime needs to understand the difference between the two so that it can follow pointers to look for more values, but not integers (which directly hold the value and don't point to anything meaningful).

Distinguishing integer and pointers at runtime

Values use a single tag bit the word to distinguish integers and pointers at runtime. The value is an integer if the lowest bit of the block word is non-zero. Several OCaml types map onto this integer representation, including `bool`, `int`, the empty list, `unit`, and variants without constructors.

Integers are unboxed runtime values in OCaml, which means that they can be stored directly without having to allocate a wrapper block that will take up more memory. They can also be passed directly to other function calls in registers, and are generally the cheapest and fastest values to use in OCaml.

The value is treated as a memory pointer if the lowest bit of the `value` is zero. A pointer value is stored unmodified since pointers are guaranteed to be word-aligned with the bottom bits always being zero. The next problem is distinguishing between pointers to OCaml values (which should be followed by the garbage collector) and C pointers (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 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) 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 interprete 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 |  col  | 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 described in Chapter 25.

The 2-bit `color` field is used by the garbage collector to keep track of its state during mark-and-sweep collection. This makes incremental collection possible, and isn't exposed directly to OCaml programs.

Tag Color	Block Status
blue	on the free list and not currently in use
white	not reached yet, but possibly reachable
gray	reachable, but its fields have not been scanned
black	reachable, and its fields have been scanned

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 more below.

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 archi-

tectures as 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) = Obj.double_tag ;;
- : 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] | ....
+---------+----------+----------+- - - - -
```

You can test this for yourself using the `Obj.tag` function to check that the allocated block has the expected runtime tag, and `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
# Obj.double_field (Obj.repr 1.234) 0;;
- : float = 1.234
```

Notice that float tuples are *not* optimized in the same way as float records or arrays, and so they have the usual tuple tag value of `0`. Only records and arrays can have the array optimization, and only if every single field is 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.

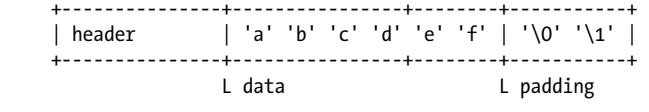
Polymorphic variants use more memory space than normal variants when parameters are included in the datatype constructors. Normal variants use the tag byte to encode the variant value and save the fields for the contents, but this single byte is insufficient to encode the hashed value for polymorphic 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 re-used 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.



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

String length mod 4	Padding
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 cope with arbitrary `NULL` values within the buffer 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 isn't scanned by the garbage collector. This means that it cannot contain any OCaml values, but is useful to track pointers into external system memory.

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 TODO. They are instead used to call C cleanup functions such as `free`.

A common use of custom blocks is to manage external system memory directly from within OCaml, via the `Bigarray` module. We'll cover this later on in Chapter 25.

2013-05-28
15:22:24

CHAPTER 22
The Compilation Pipeline

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 how to understand how these fit together to help with your day-to-day workflow of developing, debugging, maintaining and deploying applications.

OCaml has a strong emphasis on static type safety and tries to reject source code that doesn't meet its requirements as early as possible. The compiler toolchain implements these checks as a series of transformations that start with the source code. Each stage checks some properties and discards information from the previous stage, until the final output is low-level assembly code that no longer knows anything about OCaml modules or objects.

TODO note about how this different from JVM/NET w type erasure?

Much of this complexity is hidden away via the OCaml build system. However, you'll sometime need to dive into the toolchain to hunt down a bug or investigate a performance regression. In these cases, you can manually run a stage and examine the output more closely.

It's even possible to compile OCaml to run efficiently on environments such as Javascript or the Java Virtual Machine. These are third-party tools that extend the core compilation toolchain, and so it helps to understand how everything fits together before starting to use them.

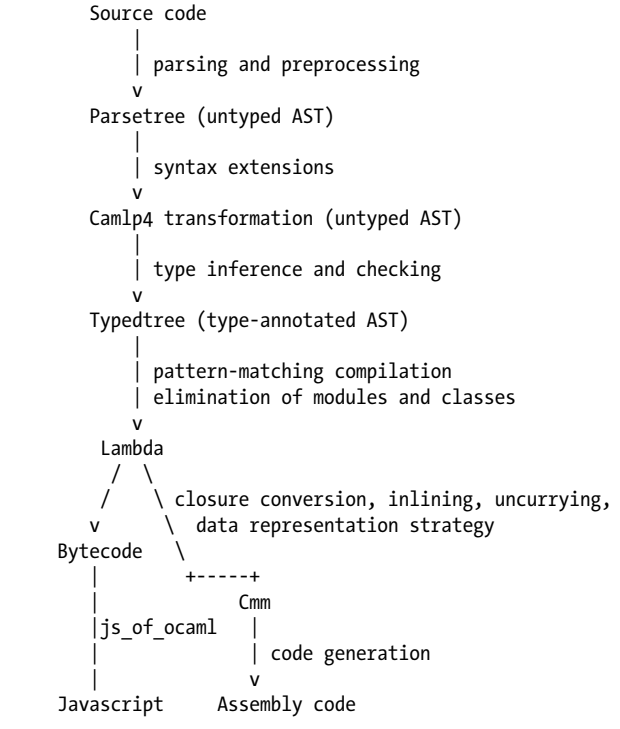
In this chapter, you'll learn:

- the compilation pipeline and what each stage represents.
- source preprocessing via `camlp4` and the intermediate forms.
- the bytecode `ocamlc` compiler and `ocamlrun` interpreter.
- the native code `ocamlopt` code generator.

An overview of the toolchain

The OCaml tools accept textual source code as input with filename extensions of `.ml` and `.mli` for modules and signatures respectively. 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. Finally, the compiler linker gathers together a collection of compiled compilation units and produces a standalone executable or library archive that can be re-used by other applications.

The overall compilation pipeline looks like this:



Notice that the pipeline branches towards the end. This is because OCaml has multiple compiler frontends that re-use the early stages of compilation, but produce very different final outputs. The bytecode interpreter is portable and lightweight (and can be transformed into Javascript), whereas the native code compiler generates specialized native code binaries suitable for high-performance applications.

Obtaining the 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*).

Once you've checked out a copy of the source tree, it has a few sub-directories (TODO: describe depending on the contents below).

We'll now go go through the compilation stages and explain how the tools behind them will be useful to you during OCaml development.

Parsing source code

The first thing the compiler does is parse the input source files into a more structured data type. The compiler lexes the source text into a stream of tokens and parses them into an Abstract Syntax Tree (AST) data structure. This parser is implemented in OCaml itself, using the techniques described earlier in Chapter 15. The lexer and parser rules can be found in the `ocaml/parsing/` directory in the source distribution.

Here's a syntax error that we obtain by using the `module` keyword incorrectly inside a `let` binding.

```
(* broken_module.ml *)
let _ =
  module MyString = String;
  ()
```

The fixed version of this locally opens the `module` correctly.

```
(* fixed_module.ml *)
let _ =
  let module MyString = String in
  ()
```

Parsing immediately eliminates code which doesn't match basic syntactic requirements, so the first example will result in a syntax error when we run the OCaml compiler on it.

```
$ ocamlc -c broken_module.ml
File "broken_module.ml", line 3, characters 2-8:
```

```
Error: Syntax error
$ ocamlc -c fixed_module.ml
```

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 shouldn't be encountered at that point in the `let` binding, so the error location information is correct.

Formatting source code with `ocp-indent`

Sadly, syntax errors do get more inaccurate sometimes, particularly in large codebases. 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 _ = add_and_print 1 2 in
  let _ = 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 syntax error in 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 available in OPAM called `ocp-indent` that applies structured indenting rules to your source code. This not only beautifies your code layout, but it also works with partially written code and makes this syntax error much more obvious.

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 _ = add_and_print 1 2 in
  let _ = 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. If we remove that semicolon, then the result is exactly what we expect.

```
$ 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 _ = add_and_print 1 2 in
  let _ = 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*) has details on how to integrate it with your favourite editor. All the Core libraries are formatted by `ocp-indent`, so it's a good idea to match this style when you publish your own open-source project online.

Generating documentation via `ocamldoc`

Whitespace and source code comments are removed during parsing and aren't significant in determining the semantics of the program for compilation purposes. However, other tools in the OCaml distribution can interpret them for their own ends.

The `ocamldoc` tool looks for 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. `ocaml`

doc support generating 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-style 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 asterix, with some text formatting conventions within the comment to mark metadata (most notably, the @tag fields to mark specific properties such as the author or argument description).

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 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, so 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.
- The custom generators available here (*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 `odoc_json` (TODO: avsm, pull out of Xen).

Preprocessing with `camlp4`

One powerful feature in OCaml is the facility to extend the language grammar via the `camlp4` macro tool. Camlp4 modules can register new keywords with the lexer 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 examples of using `camlp4` within Core:

- `Fieldslib` to 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 all extend the language with a single `with` keyword that extends a `type` declaration with additional directives that cause code to be generated. 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`.

```
$ ocamlfind ocamlc -c type_conv_example.ml
File "type_conv_example.ml", line 7, characters 2-6:
Error: Syntax error
```

Now add in the ocamlfind syntax extension packages for `fieldslib` and `sexplib`, and everything compiles 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 adds third-party packages as normal, but the `.syntax` suffix in the package name is a convention that indicate that these are `camlp4` preprocessors instead of normal libraries. These modules are dynamically loaded into `camlp4o` which then parses and rewrites the input source code into conventional OCaml code that has no trace of the new keywords. The compiler continues on to compile this transformed code with no knowledge of the preprocessor's actions.

Both `fieldslib` and `sexplib` need this `with` keyword, so there is a common preprocessor library called `type_conv` that provides the extension framework for them to use. Type_conv defines the `with` grammar extension and `ocamlfind` ensures that it's loaded before `variantslib` or `sexplib`.

The preprocessor extensions can now generate boiler-plate OCaml code based on the type definition. This approach avoids the inevitable performance hit of doing this generation dynamically, but also doesn't require a complex Just-In-Time (JIT) runtime that is a source of unpredictable dynamic behaviour. Instead, all code is generated at compilation time via `camlp4`.

Inspecting `camlp4` output

All `camlp4` modules accept an input AST and output a modified one. If you're not familiar with the `camlp4` module in question, then how do you figure out what changes it's made to your code? The first obvious way is to read the documentation that accompanies the extension. Another more direct way is to use the top-level to explore the extension's behaviour, or to run `camlp4` manually yourself to see the transformation in action.

Using `camlp4` in the interactive top-level

The `utop` top-level can run phrase that you type in through `camlp4` automatically. To get this to work, you should have at least these lines in your `~/.ocamlinit` file in your home directory (the contents of this file are automatically executed every time you run `utop`, so you can run them by hand too).

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

The first directive loads the extra `ocamlfind` top-level commands, which let you load `ocamlfind` packages directly (including all their dependent packages). The second directive instructs the top-level to filter all future phrases via `camlp4`.

You can now run `utop` and load the syntax extension you want to experiment with. We'll show you the `comparelib` syntax extension from Core. OCaml provides a polymorphic comparison operator that inspects the runtime representation of two values to see if they are equal. As we noted in Chapter 12, this is not as efficient or as safe as defining explicit comparison functions between values.

Let's see how `comparelib` solves this problem 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 type 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 `comparelib` and turned into two new functions that are generated from the type into the `compare` and `compare_t` functions.

Running `camlp4` directly on the source code

While the top-level is quick and useful to figure out the function signatures generated from the extensions, how can we see what these functions actually do? You can't do this from `utop` directly, since it embeds the `camlp4` compilation as an automated part of its operation.

Let's turn to the command-line to inspect 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
```

Now create a shell script that runs `camlp4` directly.

```
#!/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
```

This shell script uses the `ocamlfind` package manager to list the include and library paths needed by `comparelib`. The final command invokes the `camlp4o` preprocessor directly and outputs the resulting AST to standard output as textual source code.

```
$ 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 code 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.

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.

```
$ ./camlp4_dump.sh test_comparelib.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 very powerful extension mechanism that can completely change your source code's layout and style. Core includes a very conservative set of extensions that minimise the syntax changes. There are a number of third-party libraries that perform much more wide-sweeping changes, such as introducing whitespace-sensitive indentation or even building entirely new languages.

While it's tempting to compress all your boiler-plate code into `camlp4` extensions, it can make production source code much harder for other people to read and review. 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 syntax extensions. Two separate extensions create a grammar clash can lead to 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.

We've only shown you how to use `camlp4` extensions here. The full details of building new extensions with `camlp4` are fairly daunting and could be the subject of an entirely new book. The best resources to start with this are the online Camlp4 wiki (*http://brion.inria.fr/gallium/index.php/Camlp4*) and also by searching for existing extensions using OPAM and seeing how they implement their particular grammar extensions. A series of blog posts (*http://ambassadortothecomputers.blogspot.co.uk/p/reading-camlp4.html*) are also a useful guide to the internals of extensions.

The type checking phase

After obtaining a valid parsed AST (with or without `camlp4`) the compiler must then check that the code obeys the rules of the static type system. Code that is syntactically correct but misuses values is rejected with an explanation of the problem at this stage. We're not going to delve into the details of how type-checking works here (the rest of the book covers that), but rather how it fits in with the rest of the compilation process.

Assuming that the source code is validly typed, the original AST is transformed into a *typed AST*. This has the same broad structure of the untyped AST, but syntactic phrases are replaced with typed variants instead.

You can explore the type checking process very simply. Create a file with a single type definition and value.

```
(* typedef.ml *)
type t = Foo | Bar
let v = Foo
```

Now run the compiler on this file to *infer* a default type for the compilation unit. This will run the type checking process on the compilation unit you specify.

```
$ ocamlc -i typedef.ml
type t = Foo | Bar
val v : t
```

The type checker has run and provided a default signature for the module. 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.

Using `ocamlobjinfo` to inspect compilation units

Recall from Chapter 4 that `mli` files are optional. If you don't supply an explicit signature, then the inferred output from the module implementation is used instead. The compiled version of the signature is stored in a filename ending with `cmi`.

```
$ ocamlc -c typedef.ml
$ ocamlobjinfo typedef.cmi
File typedef.cmi
Unit name: Typedef
Interfaces imported:
    559f8708a08ddf66822f08be4e9c3372    Typedef
    65014ccc4d9329a2666360e6af2d7352    Pervasives
```

The `ocamlobjinfo` command 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 need it in normal use).

The long alphanumeric identifier beside each module name is a hash calculated from all the types and values exported from that compilation unit. This is used during type-checking and linking to check 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. This violates static type safety if these conflicting instances are ever linked together, as every well-typed value has precisely one type in OCaml.

This hash check ensures that separate compilation remains type safe all the way up to the final link phase. Each source file is type checked separately, but the hash of the signature of any external modules is recorded with the compiled signature.

Examining the typed syntax tree

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.

First, let's look at the untyped AST from our `typedef.ml`.

```
$ 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 also reveals a lot about how the compiler works. Each portion of the tree 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. After type checking, the structure is much simpler.

```
$ ocamlc -dtypedtree typedef.m
[
  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 never need to use this information in day-to-day development, but it's always instructive to examine how the type checker folds in the source code into a more compact form like this.

The untyped lambda form

Once OCaml gets past the typed AST, it eliminates all the static type information into a simpler intermediate *lambda form*. This discards all the modules and objects, replacing them direct references to values such as records and function pointers instead. Pattern matches are compiled into automata that are highly optimized for the particular type involved.

It's possible to examine all this behaviour via another intermediate output from the compiler. Create a new `pattern.ml` file alongside the previous `typedef.ml`.

```
(* pattern.ml *)
open Typedef
let _ =
  match v with
  | Foo -> "foo"
  | Bar -> "bar"
```

The lambda form is the first representation that discards the OCaml type information and begins to look like the runtime memory model from Chapter 21, and should be quite familiar to Lisp aficionados. To see it for `pattern.ml`, compile as usual but add the `-dlambda` directive.

```
$ ocamlc -dlambda -c pattern.ml
```

```
(setglobal Pattern!
  (seq
    (let (match/1008 (field 0 (global Typedef!)))
      (if (!= match/1008 0) "bar" "foo"))
    (makeblock 0)))
```

It's not important to understand every detail of this internal form, but some interesting points are:

- There are no mention of modules or types any more, and have been replaced by references to global values instead.
- The pattern match has turned into an integer comparison by checking 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 a single integer comparison. If `v` has a tag of `0`, the function returns `"foo"`, and otherwise returns `"bar"`.
- Values are addressed directly by an index and `field` (`v` got assigned to `1008` during type checking). The type safety checks earlier have ensured that these fields will always exist, and so the lambda form doesn't do any dynamic checks. However, unwise use of unsafe language features such as `Obj.magic` module can easily induce crashes at this level.

The lambda form is primarily a stepping-stone to the bytecode engine that we cover next. However, it's often easier to look at the textual output here than wade through native assembly code from compiled executables.

Portable bytecode with `ocamlc` and `ocamlrun`

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 the `ocamlc` bytecode compiler first, which consists of two pieces:

- `ocamlc` compiles files into a simple bytecode that is a close mapping to the lambda form.
- `ocamlrun` is a portable interpreter that executes the bytecode.

`ocamlrun` is an interpreter that uses the OCaml stack and an accumulator to store values, and only has seven registers in total (the program counter, stack pointer, accumulator, exception and argument pointers, and environment and global data). It implements around 140 opcodes which form the OCaml program. The full details of the opcodes are available online (*http://cadmium.x9c.fr/distrib/caml-instructions.pdf*).

The big advantage of using `ocamlc` 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.

Compiling and linking OCaml bytecode

`ocamlc` compiles individual `ml` files into bytecode `cmo` files. These are 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 (recall that OCaml has no single `main` function like C does).

A typical OCaml library consists of multiple modules (and hence multiple `cmo` files). `ocamlc` can combine these into a single `cma` bytecode archive by using the `-a` flag. The objects in the library are linked as regular `cmo` files in the order specified when the library file was built. However, if an object file within the library isn't referenced elsewhere in the program, it is not linked unless the `-linkall` flag forces it to be included. This behaviour is analogous to how C handles object files and archives (`.o` and `.a` respectively).

The bytecode runtime comprises three main parts: the bytecode interpreter, garbage collector, and a set of C functions that implement the primitive operations. Bytecode instructions are provided to call these C functions. The OCaml linker produces bytecode for the standard runtime system by default, and any custom C functions in your code (e.g. from C bindings) require the runtime to dynamically load a shared library.

This can be specified as follows:

```
$ ocamlc -a -o mylib.cma a.cmo b.cmo -dllib -lmylib
```

The `dllib` flag embeds the arguments in the archive file. Any subsequent packages including this archive will also include the extra linking directive. This lets the `ocaml run` runtime locate the extra 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 the "custom" runtime mode, and can be run by:

```
$ ocamlc -a -o mylib.cma -custom a.cmo b.cmo -cclib -lmylib
```

The custom mode is the most similar 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

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. The object file can then be linked with other C code using the standard C compiler, or even turned in a standalone C shared library.

The bytecode runtime library is installed as `libcamlrun` in the standard OCaml directory (obtained by `ocamlc -where`). Creating an executable just requires you to link the runtime library with the bytecode object file. Here's a quick 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
```

Once inconvenience with `gcc` is that you need to specify the location of the OCaml library directory. The OCaml compiler can actually handle C object and source files, and it adds the `-I` and `-L` directives for you. You can compile the previous object file with `ocamlc` to try this.

```
$ ocamlc -o final_out2 embed_out.o main.c
$ ./final_out2
Before calling OCaml
hello embedded world 1
hello embedded world 2
After calling OCaml
```

You can also verify the system commands that `ocamlc` is invoking by adding `-verbose` to the command line. You can even obtain the 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 code 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.

Native code generation

The native code compiler is ultimately the tool that production OCaml code goes through. It compiles the lambda form into fast native code executables, with cross-module inlining and code optimization passes that the bytecode interpreter doesn't perform. However, care is taken to ensure compatibility with the bytecode runtime, and 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 indi-

vidual `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.

Building debuggable libraries

The native code compiler builds executables that can be debugged using conventional system debuggers such as GNU `gdb`. You'll 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.

TODO add example of gdb breakpoint use

Profiling native code libraries

TODOs

Embedding native code in libraries

The native code compiler also supports `output-obj` operation just like the bytecode compiler. The native code runtime is called `libasmrun.a`, and should be linked instead of the bytecode `libcamlrun.a`.

Try this out using the same files from the bytecode embedding example earlier.

```
$ 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 C bindings that cause heap invariants to be violated. OCaml includes a variant of the runtime library `libasmrun.a` that is compiled with debugging symbols. This is available as `libasmrund.a` and includes extra memory integrity checks upon every garbage collection cycle. Running these often will abort the program near the point of corruption and helps isolate the bug.

To use this debug library, just link with `-runtime-variant d` set:

```
$ 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.

CHAPTER 23
Using the direct C interface

You should now understand the runtime structure of the OCaml heap and how the compilation pipeline fits together. This lets us move onto explaining how to interface OCaml and C code directly using the direct interface it provides to the OCaml heap.

OCaml defines an `external` keyword that maps OCaml functions to a C symbol. When the function is invoked from OCaml, the C function will be called with the OCaml function arguments using their native `value` representation. This corresponds to the memory layout for OCaml values described earlier.

A "Hello World" C binding

Let's define a simple "Hello World" C binding to see how this works. First create a `hello.ml` that contains the external declaration:

```
external hello_world: unit -> unit = "caml_hello_world"
let _ = hello_world ()
```

If you try to compile this module to an executable now, you should receive a linker error:

```
$ ocamlopt -o hello hello.ml
Undefined symbols for architecture x86_64:
  "_caml_hello_world", referenced from:
      .L100 in hello.o
      _camlHello in hello.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
File "caml_startup", line 1:
Error: Error during linking
```

This is the system linker telling you that there is a missing `caml_hello_world` symbol. We need to provide a C file that will implement this function and make it available to the linker before it creates a standalone executable. The OCaml compiler uses file extensions to determine how to compile each file. When it sees a `.c` extension, it passes it to the system C compiler and appends an include directory containing the OCaml

runtime header files. You can find these runtime header files by running `ocamlc -where` and looking under the `caml/` subdirectory.

`mlvalues.h` is the basic header file that all OCaml-C bindings need. It is also shared by the garbage collector, and defines a few important typedefs early on that should be familiar after the earlier explanation about the memory representation of OCaml values:

```
typedef intnat value;

#define Is_long(x)   (((x) & 1) != 0)
#define Is_block(x)  (((x) & 1) == 0)

#define Val_unit Val_int(0)
```

The `value` typedef is a memory word that can either be an integer if `Is_long` is true, or a heap block if `Is_block` is true. All of the arguments passed to the C bindings will be of type `value`, since this is sufficient to represent any valid OCaml value in memory. Let's look at the external declaration for `hello_world` again:

```
external hello_world: unit -> unit = "caml_hello_world"
```

This external function has a single argument of type `unit`, which is represented as an integer of value 0 in memory. Our C function definition of `caml_hello_world` must therefore accept a single `value` parameter and return a `value`. Let's create the `hello_stubs.c` file now that implements this:

```
#include <stdio.h>
#include <caml/mlvalues.h>

CAMLprim value
caml_hello_world(value v_unit)
{
  printf("Hello OCaml World from C!\n");
  return Val_unit;
}
```

You can now recompile the `hello` binary with this additional C file included in the compiler command-line, and it should succeed:

```
$ ocamlopt -o hello hello.ml hello_stubs.c
$ ./hello
Hello OCaml World from C!
```

You must be *very* careful that the value you return from the C function corresponds exactly to the memory representation of the types you declared earlier in the `external` declaration of the ML file, or else heap carnage and corruption will ensure.

Converting from OCaml values in C

The earlier hello world example is rather basic and only uses `unit` types. Let's extend the signature to take a couple of `int` arguments instead of a single `unit` so that we can send more useful data between OCaml and C:

```
external add_numbers: int -> int -> int = "caml_add_numbers"
let () = Printf.printf "From OCaml: %d\n" (add_numbers 10 15)
```

The `add_numbers` external function now takes two arguments, and returns an integer instead of a simple `unit`. The updated C stub looks like this:

```
#include <stdio.h>
#include <caml/mlvalues.h>

CAMLprim value
caml_add_numbers(value v_arg1, value v_arg2)
{
  int v1 = Int_val(v_arg1);
  int v2 = Int_val(v_arg2);
  printf("From C:     %d + %d\n", v1, v2);
  return Val_int(v1+v2);
}
```

OCaml passes the integers to `caml_add_numbers` as `value` types, so the binding uses the `Int_val` macro to convert them into local stack variables. The `Int_val` macro converts a `value` to an integer by removing the tag bit. The C integers are then added together and the result `value` is constructed by applying the `Val_int` macro, which takes a C integer and tags it into becoming an OCaml `value`. When you compile and run this version of the code, you should see this output:

```
$ ./hello
From C:     10 + 15
From OCaml: 25
```

You should keep an eye out for warnings from your compiler which often indicate that you've forgotten to correctly convert a `value`. For example, try a broken version of the previous binding:

```
#include <stdio.h>
#include <caml/mlvalues.h>

CAMLprim value
caml_add_numbers(value v_arg1, value v_arg2)
{
  printf("From C:     %d + %d\n", v_arg1, v_arg2);
  return Val_int(v_arg1 + v_arg2);
}
```

The compiler will now complain of invalid format strings when you compile this:

```
$ ocamlopt -o hello -ccopt -Wall hello_stubs.c hello.ml
hello_stubs.c: In function caml_add_numbers:
hello_stubs.c:7: warning: format %d expects type int, but argument 2 has type value
hello_stubs.c:7: warning: format %d expects type int, but argument 3 has type value
hello_stubs.c:7: warning: format %d expects type int, but argument 2 has type value
hello_stubs.c:7: warning: format %d expects type int, but argument 3 has type value
$ ./hello
From C:     21 + 31
From OCaml: 26
```

Notice that both the input and output integers are incorrect in the output, since they still have their tag bits set when being added in the C code. Don't depend on the good graces of your C compiler to always spot such errors though. It's also invalid to add two `value`s together without converting them into native C type, but the C compiler can't warn about this and the result is silently incorrect. It's good practise to immediately convert arguments to local C stack variables as early as possible, so you don't get the types mixed up deep into the C function.

OCaml provides macros to convert to and from all the basic OCaml runtime values and C types, of the form `to_from`. For example `Val_long` means "Value from long", and `Long_val` means "Long from value". The table below summarises the macros to extract various C types from OCaml `values` for 64-bit architectures. Note that OCaml doesn't support a single-precision float, so these are always double-precision.

TODO: buggy markdown below in table rendering

Macro	OCaml Type	C type
Long_val	int`	long`
Int_val	int`	int`
Unsigned_long_val	int`	unsigned long`
Unsigned_int_val	int`	unsigned int`
Bool_val	bool`	int`
Double_val	float`	double`
String_val	string`	string`
Nativeint_val	Nativeint.t`	long`
Int32_val	Int32.t`	long`
Int64_val	Int64.t`	unsigned long`

Constructing OCaml values from C

Building OCaml values to return from C is a little more involved, since we must ensure that any OCaml allocations aren't immediately cleaned up by the garbage collector before they have been registered as live values. Luckily, OCaml provides some more

macros to make this easier to enforce. Let's extend our earlier example to return a tuple of integers instead of just the result.

```
external add_numbers: int -> int -> int * int * int32 = "caml_add_numbers"
let () =
  let (l,r,v) = add_numbers 10 15 in
  Printf.printf "From OCaml: %d+%d=%ld\n" l r v
```

The `add_numbers` external now returns a more complex data type that requires allocating an OCaml tuple from within the C binding, storing the results within the tuple, and returning that tuple. The contents of the tuple are also a mix of immediate values (the two first `int` fields in the tuple) and the last boxed `int32` that also needs to be allocated on the OCaml heap.

```
#include <stdio.h>
#include <caml/memory.h>
#include <caml/alloc.h>

CAMLprim value
caml_add_numbers(value v_arg1, value v_arg2)
{
  CAMLparam2(v_arg1, v_arg2);
  CAMLlocal1(v_res);
  int v1 = Int_val(v_arg1);
  int v2 = Int_val(v_arg2);
  printf("From C:     %d+%d=%d\n", v1, v2, (v1+v2));
  v_res = caml_alloc_tuple(3);
  Store_field(v_res, 0, Val_int(v1));
  Store_field(v_res, 1, Val_int(v2));
  Store_field(v_res, 2, caml_copy_int32(v1 + v2));
  CAMLreturn(v_res);
}
```

The stub now uses several new macros that are all defined in `caml/memory.h`. The `CAMLparam` macro registers its parameters as *local roots* with the garbage collector. This ensures that if the garbage collector is triggered during the C binding, it will not relocate or free the value we just allocated. We also need to allocate a tuple to store the result. This is declared with the `CAMLlocal` macro, and returned via `CAMLreturn`. For many simple bindings, you can just follow the simple rule of replacing the C `return` with `CAMLreturn` and starting every function with `CAMLparam` and `CAMLlocal`, and not have to worry about the garbage collector.

FFI rule: Use `CAMLparam` and `CAMLreturn` for OCaml values

A function that has parameters or local variables of type value must begin with a call to one of the `CAMLparam` macros and return with `CAMLreturn`, `CAMLreturn0`, or `CAMLreturnT`. Local variables of type `value` must be declared with one of the `CAMLlocal` macros. Arrays of values are declared with `CAMLlocalN`. These macros must be used at the beginning of the function, not in a nested block.

The tuple is then allocated via `caml_alloc_tuple`, with the number of fields representing the size of the tuple. This must *immediately* be followed by the `Store_field` macros to set the newly allocated tuple to sensible values. In our example, we assign the first two fields to the local integers. The `int32` requires another allocation, and `caml_copy_int32` is used which copies a C `int32` into the correspondig OCaml `value`. Once the tuple has been set, the function returns via `CAMLreturn`, which frees up the local roots that we registered at the beginning of the function.

FFI rule: Use `Store_field` to assign to tuples, records and arrays

Assignments to the fields of structured blocks must be done with the `Store_field` macro (for normal blocks) or `Store_double_field` macro (for arrays and records of floating-point numbers). Other assignments must not use `Store_field` nor `Store_double_field`.

The `caml/alloc.h` header file lists all of the functions that allocate OCaml values.

Function name	Argument	OCaml return type
`caml_alloc_tuple (<len>)`	Number of fields	Tuple or record
`caml_alloc_string (<len>)`	Size in bytes	`string`
`caml_copy_string (char *)`	Pointer to a C string	`string`
`caml_copy_string_array (char **)`	Pointer to array of C strings	`string array`
`caml_copy_double (double)`	C `double` value	`double`
`caml_copy_int32 (int32)`	32-bit C integer	`int32`
`caml_copy_int64 (int64);`	64-bit C integer	`int64`
`caml_copy_nativeint (intnat);`	native C integer size (`long`)	`Nativeint.t`

Tuples and records both have the same memory representation, with the same tag value (`0`). This means that you can change how you map these fields into OCaml, without having to modify the C bindings. For example, our earlier integer addition example can also look like this:

```
type t = {
  v1: int;
  v2: int;
  res: int32;
}

external add_numbers: int -> int -> t = "caml_add_numbers"

let () =
  let v = add_numbers 10 15 in
  Printf.printf "From OCaml: %d+%d=%ld\n" v.v1 v.v2 v.res
```

Records are never rearranged in memory, so the fields will appear in the same order they are declared.

Faster bindings for zero-allocation functions

TODO Talk about "alloc" and "float" qualifiers to `external`

2013-05-28
15:22:25

CHAPTER 24
Understanding the Garbage Collector

(*avsm*: this chapter is still being chopped and changed)

Runtime Memory Management

The OCaml runtime divides the address space into memory pages of 4KB each (this is configurable by recompiling the runtime). At any given time, every page that is in use is used for a single purpose: major heap, minor heap, static data or code. The runtime guarantees this by always allocating slightly more memory than requested so that that it can choose align the memory it will actually use at the beginning of a 4KB page.

The runtime maintains a *page table* that allows it to determine the status of any virtual memory address in the operating system process. The status defines whether that address is a page in use by the OCaml runtime, and if so, which of the four purposes it is being used for.

Since the virtual memory space can be very large and sparsely used (especially on a 64-bit CPU), the page table is implemented as a hash table in which keys are page-aligned addresses and values are a single byte. The hash table is represented as an array of words, with each word being a key-value pair. The key-value pair is the bitwise `or` of the virtual address of the start of the page (which has zeros for its lower 12-bits due to being aligned to 4KB), and the lower 8 bits are used for the value. To look up an address, one masks out the lower 12-bits of the memory address, compute a multiplicative hash to get a table index, and then compares against the address (i.e. the key) at that index. Linear probing is used to resolve collisions.

The byte value stored is a bitwise `or` of the following status bits:

Page table status	Value	Meaning
`In_heap`	1	in the major heap
`In_young`	2	in the minor heap
`In_static_data`	4	in the statically allocated data segment

Page table status	Value	Meaning
`In_code_area`	8	in the statically allocated code segment

The page table starts with a size aiming to be between 25% and 50% full of entries, and is automatically doubled in size if it becomes half full. It is never shrunk.

Allocating on the minor heap

The minor heap is a contiguous chunk of virtual memory. Its size is set on program startup and decided by the `OCAMLRUNPARAM` environment variable (*avsm*: xref), and then only changed later by calls to `Gc.set`. The default size is 256k.

The range of memory usable for allocation goes from the `caml_young_start` to `caml_young_end` C variables managed by the runtime.

```
                <---- size ---->
base --- start ---------------- end
         limit      ptr <------
                         blocks
```

In a fresh minor heap, the `limit` will equal the `start`, and the current `ptr` will equal the `end`. As blocks are allocated, `caml_young_ptr` will decrease until it reaches `caml_young_limit`, at which point a minor garbage collection is triggered. To allocate a block in the minor heap, we decrement `caml_young_ptr` by the size of the block (including the header), and then set the the header to a valid value. If there isn't enough space left for the block without decrementing past the `limit`, a minor collection is triggered.

To force a minor gc to occur, one can set the `caml_young_limit` to equal `caml_young_end`, which causes signal handlers to be run and to "urge" the runtime (*avsm*: elaborate on this urging business, and how to set young from within OCaml via `Gc.??`).

Allocating on the major heap

The major heap is a singly linked list of contiguous memory chunks, sorted in increasing order of virtual address. Each chunk is a single memory chunk allocated via *malloc(3)* and consists of a header and a data area which contains OCaml blocks. A pointer to a heap chunk points to the start of the data area, and access to the header is done by a negative offset from this pointer. A chunk header has:

- the address of the memory that the chunk is in, as allocated by *malloc(3)*. It is needed when the chunk is freed.
- 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.

The chunk's data area always starts on a page boundary, and its size is a multiple of the page size (4KB). It contains a contiguous sequence of heap blocks. These can be as small as one or two 4KB pages, but are usually allocated in 1MB chunks (or 512KB on 32-bit architectures). You can modify these defaults by editing `Heap_chunk_def` in `byte run/config.h` and recompiling the runtime. (*avsm*: talk about modifying the defaults in a separate callout, as there are quite a few variables which can be tweaked)

Allocating a block on the major heap first checks the free list of blocks (see below). If there isn't enough room on the free list, the runtime expands the major heap with a fresh block that will be large enough. That block is then added to the free list, and the free list is checked again (and this time will definitely succeed).

The major heap free list

The free space in the major heap's chunks is organized as a singly linked list of OCaml blocks, ordered by increasing virtual address. The runtime has a pointer to the first block in the free list. A free list block is at least two words: a header followed by a pointer to the next free-list block. The header specifies the length of the block just as with a normal block. (*avsm*: I'm not sure that this is quite true. It seems from `free list.c` that the freelist blocks are normal OCaml blocks, with the first data entry being the next pointer. when detached, they become normal ocaml blocks)

As soon as the runtime finds a free block that is larger than the request, there are three possibilities:

- If the free block is exactly the right size, it is unlinked from the free list and returned as the requested block.
- If the free block is one word too large, it is unlinked from the free list, and the first word is given a special header recognizable to the collector as an unused word, while the rest of the block is returned as the requested block.
- If the free block is two or more words larger than the requested block, it remains in the free list, with its length shortened, and the end of the free block is returned for the requested block. Since the allocated block is right-justified within the free block, the linking of the free list doesn't need to be changed at all as the block that remains in the free list is the original one.

Memory allocation strategies

Allocating a new block in the major heap always looks in the free list. There are two allocation policies: first fit and next fit (the default).

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.

First-fit allocation

First-fit allocation focusses on reducing memory fragmentation, at the expense of slower block allocation. For some workloads, the reduction in the frequency in heap compaction will outweigh the extra allocation cost. (*avsm*: example?)

The runtime maintains an ordered array of freelist chunks, called the `flp` array. Imagine a function mapping a block's index in the free list to its size. The flp array pointers are to the high points of this graph. That is, if you walk the free list between `flp[i]` and `flp[i+1]`, you will come across blocks that have sizes at most the size of `flp[i]`. Furthermore this sequence of smaller-than-flp[i] blocks cannot be extended, which is equivalent to saying `size(flp[i+1]) > size(flp[i])`.

When allocating, we first check the flp-array. If `flp[i]` is not big enough for our new block, then we may as well skip to `flp[i+1]`, because everything in the free list before then will also be too small.

If there's nothing big enough in the `flp` array, we extend it by walking the free list starting at the *last* pointer in the `flp`-array, say `flp[N]`. We extend the `flp` array along the way, so that at each block, if this block is bigger than the current last thing in `flp` (which is equivalent to saying this is the biggest block we've ever seen, since the blocks pointed to by the $_{\text{flp}}$ array are increasing in size), we add it to the end of $_{\text{flp}}$. We stop this walk when we come across a block big enough to house our desired new block.

There's also the case when the `flp` array has its ceiling size of `FLP_MAX` (default 100). Then we just start at the end of the `flp` array and walk until we find something big enough. This is known in the as a slow first-fit search, since this linear walk may take a long time.

If we did manage to find something suitable in the `flp` array, say at index `i`, we need to update `flp`. This update is rather complex, and the reason why first-fit allocation is slower than next-fit. We walk through the free list between `flp[i-1]` and `flp[i]` and record every high point we come across. Say we find `j` such points. We move the upper portion of `flp` (from `flp[i+1]` to the end) to the right by `j` places and insert each new high point into the array. There is a further corner case when adding in `j` new high points would make `flp` bigger than `FLP_MAX`.

(*avsm*: this really needs a diagram)

Which allocation policy should I use?

(*avsm*: 0 is the next-fit policy, which is quite fast but can result in fragmentation. 1 is the first-fit policy, which can be slower in some cases but can be better for programs with fragmentation problems.)

Inter-generational pointers

Most incremental generational garbage collectors have to keep careful track of values pointing from old generations to younger ones. The OCaml runtime is no exception, and maintains a set of addresses in the major heap that may point into the minor heap. These addresses are *not* OCaml pointers, and just literal memory addresses. The runtime ensures that it never relocates values in the major heap unless this "remembered" set is empty. The set is maintained as a dynamically resized array of pointers, which is itself maintained via a collection of pointers known as the `caml_ref_table`.

```
struct caml_ref_table {
  value **base;
  value **end;
  value **threshold;
  value **ptr;
  value **limit;
  asize_t size;
  asize_t reserve;
};
```

The relationships of the pointers are as follows:

```
                 limit
  base <= ptr    threshold          end
    |---------------|---------------|
          size            reserve
```

An address is added to `caml_ref_table` when all of these conditions are satisfied:

- a field in a block in the major heap is mutated
- the field previously did not point to the minor heap
- the field is being changed to point into the minor heap

In that case the entry is added at `caml_ref_table.ptr`, which is then incremented. If `ptr` is already at `limit`, the table is doubled in size before adding the address.

The same address can occur in `caml_ref_table` multiple times if a block field is mutated repeatedly and alternated between pointing at the minor heap and the major heap. The field in `caml_ref_table` also may not always point into the minor heap (if it was changed after being added), since fields are never removed. The entire table is cleared as part of the minor collection process.

The write barrier

The write barrier is one of the reasons why using immutable data structures can sometimes be faster than mutable records. The OCaml compiler keeps track of any mutable types and adds a call to `caml_modify` before making the change. The `caml_modify` checks that the remembered set is consistent, which, although reasonably efficient, can be slower than simply allocating a fresh value on the fast minor heap.

Let's see this for ourselves with a simple test program:

```
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 }

open Printf
let time name fn arg =
  Gc.compact ();
  let w1 = Gc.((stat ()).minor_collections) in
  let t1 = Unix.gettimeofday () in
  fn arg;
  let w2 = Gc.((stat ()).minor_collections) in
  let t2 = Unix.gettimeofday () in
  printf "%s: %.4fs (%d minor collections)\n" name (t2 -. t1) (w2 - w1)

let _ =
  let iters = 1000000000 in
  time "mutable" test_mutable { iters1=iters; count1=0.0 };
  time "immutable" test_immutable { iters2=iters; count2=0.0 }
```

This program defines a type `t1` that is mutable, and `t2` that is immutable. The main loop iterates over both fields and runs a simple counter. It measures two things: the wallclock time that all the iterations take, and the number of minor garbage collections that occurred during the test. The results are very hardware-dependent, but on a modern Intel i7 processor they look something like this:

```
mutable: 8.6923s (7629 minor collections)
immutable: 2.6186s (19073 minor collections)
```

Notice the space/time tradeoff here. The mutable version runs almost 4 times slower than the immutable one, but has significantly fewer garbage collection cycles. Minor collections in OCaml are very fast, and so it is often acceptable to use immutable data structures in preference to the more conventional mutable versions. On the other hand, if you only rarely mutable a value, it can be faster to take the write barrier hit and not allocate at all.

(*avsm*: it would be really nice to use a benchmark suite here and shorten the example. Investigate the options and edit this section)

(*avsm*: need to mention when a value is allocated directly into the major heap somewhere)

How garbage collection works

Collecting the minor heap

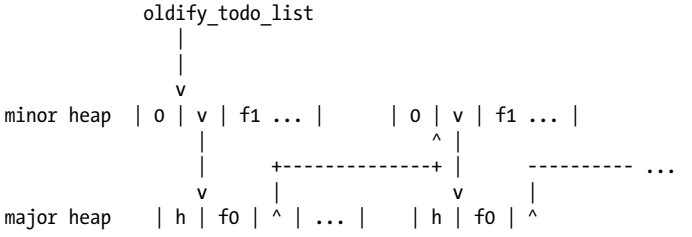
For those familiar with garbage collection terminology, here is OCaml's minor colection in one sentence. OCaml's minor collection uses copying collection with forwarding pointers, and does a depth-first traversal of the block graph using a stack represented as a linked list threaded through blocks that need to be scanned.

The goal of minor collection is to empty the minor heap by moving to the major heap every block in the minor heap that might be used in the future, and updating each pointer to a moved block to the new version of the block. A block is *live* if it is reachable by starting at some *root* pointer into a block in the minor heap, and then following pointers in blocks. There are many different kinds of roots:

- OCaml stack(s)
- C stack(s), identified by `BeginRoots` or `CAMLparam` in C code (*avsm*: xref C bindings chapter)
- Global roots
- Finalized values (*avsm*: ?)
- Intergenerational pointers in the `caml_ref_table` (*avsm*: xref above?)

Moving a block between heaps is traditionally called *forwarding*. The OCaml runtime code uses that term as well as the term *oldify*, which is useful to understand when profiling hotspots in your code. The minor collector first visits all roots and forwards them if they point to a block in the minor heap. When a block is forwarded, the collector sets the tag of the original block to a special `Forward_tag` (250), and the first field of the original block to point to the new block. Then, if the collector ever encounters a pointer to the original block again, it can simply update the pointer directly into the forwarded block.

Because a forwarded block might itself contain pointers, it must at some point be scanned to see if those pointers point to blocks in the minor heap, so that those blocks can also be forwarded. The collector maintains a linked list (called the `oldify_todo_list`) of forwarded objects that it still needs to scan. That linked list looks like:



Each value on the `oldify_todo_list` is marked as forwarded, and the first word points to the new block in the major heap. That new version contains the actual value header, the real first field of the value, and a link (pointer) to the next value on the oldify_todo_list, or NULL at the end of the list. Clearly this approach won't work if an value has only one field, since there will be no second field to store the link in. Values with exactly one field are never put on the `oldify_todo_list`; instead, the collector immediately traverses them, essentially making a tail call in the depth-first search.

Values that are known from the tag in their header to not contain pointers are simply forwarded and completely copied, and never placed on the `oldify_todo_list`. These tags are all greater than `No_scan_tag` and include strings and float arrays.

(*avsm*: note from sweeks to investigate: There is a hack for objects whose tag is `Forward_tag` that does some kind of path compression, or at least removal of one link, but I'm not sure what's going on.)

(*avsm*: I dont think we've introduced weak references yet, so this needs rearranging) At the end of the depth-first search in minor collection, the collector scans the `weak-ref` table, and clears any weak references that are still pointing into the minor heap. The collector then empties the `weak-ref` table and the minor heap.

Collecting the major heap

The major heap collections operates incrementally, as the amount of memory being tracked is a lot larger than the minor heap. The major collector can be in any of a number of phases:

- `Phase_idle`
- `Phase_mark`
  - `Subphase_main`: main marking phase
  - `Subphase_weak1`: clear weak pointers
  - `Subphase_weak2`: remove dead weak arrays, observe finalized values

— `Subphase_final`: initialise for the sweep phase

- `Phase_sweep`

Marking the major heap

Marking maintains an array of gray blocks, `gray_vals`. It uses as them as a stack, pushing on a white block that is then colored gray, and popping off a gray block when it is scanned and colored black. The `gray_vals` array is allocated via *malloc(3)*, and there is a pointer, `gray_vals_cur`, to the next open spot in the array.

The `gray_vals` array initially has 2048 elements. `gray_vals_cur` starts at `gray_vals`, and increases until it reachs `gray_vals_end`, at which point the `gray_vals` array is doubled, as long as its size (in bytes) is less than 1/2^10th of the heap size (`caml_stat_heap_size`). When the gray vals is of its maximum allowed size, it isn't grown any further, and the heap is marked as impure (`heap_is_pure=0`), and last half of `gray_vals` is ignored (by setting `gray_vals_cur` back to the middle of the `gray_vals` array.

If the marking is able to complete using just the gray list, it will. Otherwise, once the gray list is emptied, the mark phase will observe that the heap is impure and initiate a backup approach to marking. In this approach it marks the heap as pure and then 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 does a DFS marking using the gray list as a stack in the usual way. Once the scan of the complete heap is finished, the mark phase checks again whether the heap has again become impure, and if so initiates another scan. These full-heap scans will continue until a successful scan completes without overflowing the gray list.

(*avsm*: I need to clarify this more, possibly a diagram too. It's not really clear what the implications of an impure heap are atm)

Sweeping unused blocks from the major heap

Compaction and defragmenting the major heap

2013-05-28
15:22:25

2013-05-28
15:22:25

CHAPTER 25

Parsing Binary Protocols with Bigarray

Bigarrays for external memory blocks

An OCaml bigarray is a useful custom block provided as standard to manipulate memory blocks outside the OCaml heap. It has `Custom_tag` in the header, and the first word points to the `custom_operations` struct for bigarrays. Following this is a `caml_ba_array` struct.

```
struct caml_ba_array {
  void * data;                 /* Pointer to raw data */
  intnat num_dims;             /* Number of dimensions */
  intnat flags;                /* Kind of element array + memory layout + allocation status */
  struct caml_ba_proxy * proxy; /* The proxy for sub-arrays, or NULL */
  intnat dim[]  /*[num_dims]*/; /* Size in each dimension */
};
```

The `data` is usually a pointer to a `malloc`'ed chunk of memory, which the custom finalizer operation `free`'s when the block is free. The `flags` field encodes three values, located in the bits as specified by three masks:

```
CAML_BA_KIND_MASK = 0xFF     /* Mask for kind in flags field */
CAML_BA_LAYOUT_MASK = 0x100  /* Mask for layout in flags field */
CAML_BA_MANAGED_MASK = 0x600 /* Mask for "managed" bits in flags field */
```

The `CAML_BA_KIND_MASK` bits hold a value of the `caml_ba_kind` enum that identifies the kind of value in the bigarray `data`.

```
enum caml_ba_kind {
  CAML_BA_FLOAT32,             /* Single-precision floats */
  CAML_BA_FLOAT64,             /* Double-precision floats */
  CAML_BA_SINT8,               /* Signed 8-bit integers */
  CAML_BA_UINT8,               /* Unsigned 8-bit integers */
  CAML_BA_SINT16,              /* Signed 16-bit integers */
  CAML_BA_UINT16,              /* Unsigned 16-bit integers */
  CAML_BA_INT32,               /* Signed 32-bit integers */
  CAML_BA_INT64,               /* Signed 64-bit integers */
```

```
  CAML_BA_CAML_INT,           /* OCaml-style integers (signed 31 or 63 bits) */
  CAML_BA_NATIVE_INT,         /* Platform-native long integers (32 or 64 bits) */
  CAML_BA_COMPLEX32,          /* Single-precision complex */
  CAML_BA_COMPLEX64,          /* Double-precision complex */
}
```

The `CAML_BA_LAYOUT_MASK` bit says whether multi-dimensional arrays are layed out C or Fortran style.

```
enum caml_ba_layout {
  CAML_BA_C_LAYOUT = 0,           /* Row major, indices start at 0 */
  CAML_BA_FORTRAN_LAYOUT = 0x100, /* Column major, indices start at 1 */
};
```

The `CAML_BA_MANAGED_MASK` bits hold a value of the `caml_ba_managed` enum that identifies whether OCaml is responsible for freeing the `data` or some other code is.

```
enum caml_ba_managed {
  CAML_BA_EXTERNAL = 0,        /* Data is not allocated by OCaml */
  CAML_BA_MANAGED = 0x200,     /* Data is allocated by OCaml */
  CAML_BA_MAPPED_FILE = 0x400, /* Data is a memory mapped file */
};
```

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

There are experimental binary packages available for Debian Wheezy/amd64. You should be able to use these on 64-bit Ubuntu and other derivative distributions such as Linux Mint also. Just add the following line to your `/etc/apt/sources.list`:

```
deb http://www.recoil.org/~avsm/ wheezy main
```

When this is done, update your packages and install OPAM. You can ignore the warning about unsigned packages, which will disappear when OPAM is upstreamed into Debian mainline.

```
# apt-get update
# apt-get install opam
```

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_extended"
#require "async"
open Core.Std
```

When you run `utop` with this initialization file, it should start up with Core opened and ready to use.

Editors

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.

2013-05-28
15:22:25

APPENDIX B
Packaging

The OCaml toolchain is structured much like a C compiler, with several tools that generate intermediate files and finally link against a runtime. The final outputs don't have to be just executables. Many people embed OCaml code as object files that are called from other applications, or even compile it to Javascript and other esoteric targets. Let's start by covering some of the standard OCaml tools, and then move on to some of the higher level methods for packaging and publishing your code online.

The OCaml toolchain

There are two distinct compilers for OCaml code included in the standard distribution. The first outputs bytecode that is interpreted at runtime, and the second generates fast, efficient native code directly. Both of these share the front-end type-checking logic, and only diverge when it comes to code generation.

The `ocamlc` bytecode compiler

The simplest code generator is the `ocamlc` compiler, which outputs bytecode that is interpreted via the `ocamlrun` runtime. The OCaml bytecode virtual machine is a stack machine (much like the Java Virtual Machine), with the exception of a single register that stores the most recent result. This provides a simple runtime model that is easy to implement or embed within other systems, but executes rather slowly due to being interpreted.

Here are some of 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.

Extension	Purpose
.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`.

To obtain a bytecode executable, you need to compile a set of `cmo` object files, and then link them into an executable

The `ocamlopt` native code compiler

Extension	Purpose
.cmi	Compiled module interface from a corresponding `.mli` source file. (*avsm*: this is not compatible with the ocamlc version iirc)
.o	Compiled native object file of the module implementation.
.cmx	Contains extra information for linking and cross-module optimization of the object file.
.cmxa/.a	Library of `cmx` and `o` units, stored in the `cmxa` and `a` files respectively.

The `ocaml` toplevel loop

The Findlib compiler frontend

Packaging applications with OASIS

ocamlbuild

Distributing applications with OPAM