1 Prologue

1.1 Why OCaml?

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

We wrote this book because we believe in the importance of programming languages, and that OCaml in particular is an important language to learn. Both of us have been using OCaml in our academic and professional lives for over 20 years, and in that time we've come to see it as a powerful tool for building complex software systems. This book aims to make this tool available to a wider audience, by providing a clear guide to what you need to know to use OCaml effectively in the real world.

What makes OCaml special is that it occupies a sweet spot in the space of programming language designs. It provides a combination of efficiency, expressiveness and practicality that is matched by no other language. That is in large part because OCaml is an elegant combination of a set of language features that have been developed over the last 60 years. These include:

- *Garbage collection* for automatic memory management, now a common feature of modern, high-level languages.
- *First-class functions* that can be passed around like ordinary values, as seen in JavaScript, Common Lisp, and C#.
- *Static type-checking* to increase performance and reduce the number of runtime errors, as found in Java and C#.
- *Parametric polymorphism*, which enables the construction of abstractions that work across different data types, similar to generics in Java, Rust, and C# and templates in C++.
- Good support for *immutable programming*, *i.e.*, programming without making destructive updates to data structures. This is present in traditional functional languages like Scheme, and is also commonly found in everything from distributed, big-data frameworks to user-interface toolkits.
- *Type inference*, so you don't need to annotate every variable in your program with its type. Instead, types are inferred based on how a value is used. Available in a

limited form in C# with implicitly typed local variables, and in C++11 with its `auto` keyword.

- *Algebraic data types* and *pattern matching* to define and manipulate complex data structures. Available in Scala, Rust, and F#.

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

Among this worthy set of languages, OCaml stands apart because it manages to provide a great deal of power while remaining highly pragmatic. The compiler has a straightforward compilation strategy that produces performant code without requiring heavy optimization and without the complexities of dynamic just-in-time (JIT) compilation. This, along with OCaml's strict evaluation model, makes runtime behavior easy to predict. The garbage collector is *incremental*, letting you avoid large garbage collection (GC)-related pauses, and *precise*, meaning it will collect all unreferenced data (unlike many reference-counting collectors), and the runtime is simple and highly portable.

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

1.1.1 A Brief History

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

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

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

Xavier Leroy continued extending Caml Light with new features, which resulted

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

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

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

1.1.2 The `Base` Standard Library

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

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

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

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

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

1.1.3 The OCaml Platform

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

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

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

1.2 About This Book

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

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

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

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

1.2.1 What to Expect

Real World OCaml is split into three parts:

- Part I covers the language itself, opening with a guided tour designed to provide a quick sketch of the language. Don't expect to understand everything in the tour; it's meant to give you a taste of many different aspects of the language, but the ideas covered there will be explained in more depth in the chapters that follow.

  After covering the core language, Part I then moves onto more advanced features like modules, functors, and objects, which may take some time to digest. Understanding these concepts is important, though. These ideas will put you in good stead even beyond OCaml when switching to other modern languages, many of which have drawn inspiration from ML.
- Part II builds on the basics by working through useful tools and techniques for addressing common practical applications, from command-line parsing to asynchronous network programming. Along the way, you'll see how some of the concepts from Part I are glued together into real libraries and tools that combine different features of the language to good effect.
- Part III discusses OCaml's runtime system and compiler toolchain. It is remarkably simple when compared to some other language implementations (such as Java's or .NET's CLR). Reading this part will enable you to build very-high-performance systems, or to interface with C libraries. This is also where we talk about profiling and debugging techniques using tools such as GNU `gdb`.

1.2.2 Installation Instructions

Real World OCaml uses some tools that we've developed while writing this book. Some of these resulted in improvements to the OCaml compiler, which means that you will need to ensure that you have an up-to-date development environment (using the 4.13.1 version of the compiler). The installation process is largely automated through the opam package manager. Instructions on how to set it up and what packages to install can be found at the installation page[3].

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

This book is not intended as a reference manual. We aim to teach you about the language as well as the libraries, tools, and techniques that will help you be a more effective OCaml programmer. But it's no replacement for API documentation or the OCaml manual and man pages. You can find documentation for all of the libraries and tools referenced in the book online[4].

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

1.2.3 Code Examples

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

The full text of the book, along with all of the example code is available online at https://github.com/realworldocaml/book [5].

1.3 Contributors

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

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

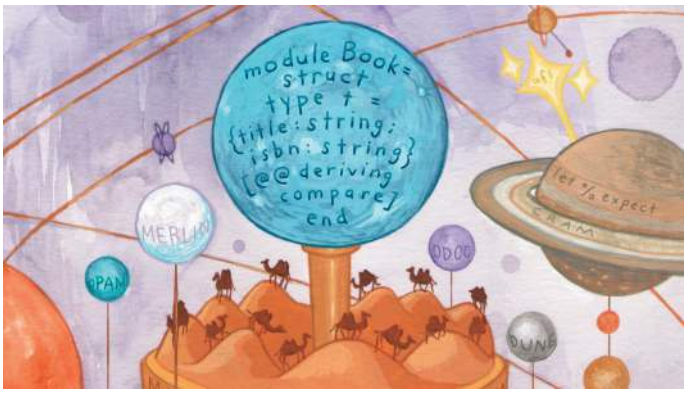
[5] `https://github.com/realworldocaml/book`

Part I

Language Concepts

Part I covers the language itself, opening with a guided tour designed to provide a quick sketch of the language. Don't expect to understand everything in the tour; it's meant to give you a taste of many different aspects of the language, but the ideas covered there will be explained in more depth in the chapters that follow.

After covering the core language, Part I then moves onto more advanced features like modules, functors, and objects, which may take some time to digest. Understanding these concepts is important, though. These ideas will put you in good stead even beyond OCaml when switching to other modern languages, many of which have drawn inspiration from ML.



2 A Guided Tour

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

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

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

Base and Core

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

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

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

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

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

2.1 OCaml as a Calculator

Our first step is to open `Base`:

```
# open Base;;
```

By opening `Base`, we make the definitions it contains available without having to reference `Base` explicitly. This is required for many of the examples in the tour and in the remainder of the book.

Now let's try a few simple numerical calculations:

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

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

- We needed to type `;;` in order to tell the toplevel that it should evaluate an expression. This is a peculiarity of the toplevel that is not required in standalone programs (though it is sometimes helpful to include `;;` to improve OCaml's error reporting, by making it more explicit where a given top-level declaration was intended to end).
- After evaluating an expression, the toplevel first prints the type of the result, and then prints the result itself.
- OCaml allows you to place underscores in the middle of numeric literals to improve readability. Note that underscores can be placed anywhere within a number, not just every three digits.
- OCaml carefully distinguishes between `float`, the type for floating-point numbers, and `int`, the type for integers. The types have different literals (`6.` instead of `6`) and different infix operators (`+.` instead of `+`), and OCaml doesn't automatically cast between these types. This can be a bit of a nuisance, but it has its benefits, since it prevents some kinds of bugs that arise in other languages due to unexpected differences between the behavior of `int` and `float`. For example, in many languages, `1 / 3` is zero, but `1.0 /. 3.0` is a third. OCaml requires you to be explicit about which operation you're using.

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

```
# let x = 3 + 4;;
val x : int = 7
# let y = x + x;;
val y : int = 14
```

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

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

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

The following examples, however, are not legal:

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

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

2.2 Functions and Type Inference

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

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

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

Now that we're creating more interesting values like functions, the types have gotten more interesting too. `int -> int` is a function type, in this case indicating a function that takes an `int` and returns an `int`. We can also write functions that take multiple arguments. (Reminder: Don't forget `open Base`, or these examples won't work!)