2 A Guided Tour

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

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

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

Base and Core

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

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

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

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

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

2.1 OCaml as a Calculator

Our first step is to open `Base`:

```
# open Base;;
```

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

Now let's try a few simple numerical calculations:

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

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

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

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

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

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

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

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

The following examples, however, are not legal:

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

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

2.2 Functions and Type Inference

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

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

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

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

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

Note that in OCaml, function arguments are separated by spaces instead of by parentheses and commas, which is more like the UNIX shell than it is like traditional programming languages such as Python or Java.

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

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

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

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

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

The notation for the type-signature of a multiargument function may be a little surprising at first, but we'll explain where it comes from when we get to function currying in Chapter 3.2.2 (Multiargument Functions). For the moment, think of the arrows as separating different arguments of the function, with the type after the final arrow being the return value. Thus, `int -> int -> float` describes a function that takes two `int` arguments and returns a `float`.

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

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

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

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

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

2.2.1 Type Inference

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

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

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

1.. OCaml requires that both branches of an `if` expression have the same type, so the expression

    `if test first then first else 0`

    requires that `first` must be the same type as `0`, and so `first` must be of type `int`.

    Similarly, from

    `if test second then second else 0`

    we can infer that `second` has type `int`.
2.. `test` is passed `first` as an argument. Since `first` has type `int`, the input type of `test` must be `int`.
3.. `test first` is used as the condition in an `if` expression, so the return type of `test` must be `bool`.
4.. The fact that + returns `int` implies that the return value of `sum_if_true` must be int.

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

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

Here's an annotated version of `sum_if_true`:

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

In the above, we've marked every argument to the function with its type, with the final annotation indicating the type of the return value. Such type annotations can be placed on any expression in an OCaml program.

2.2.2 Inferring Generic Types

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

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

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

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

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

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

As well as this:

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

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

```
# first_if_true big_number "short" "loooooong";;
Line 1, characters 26-33:
Error: This expression has type string but an expression was expected
    of type
         int
```

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

Type Errors Versus Exceptions

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

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

```
# let add_potato x =
    x + "potato";;
Line 2, characters 9-17:
Error: This expression has type string but an expression was expected
    of type
         int
```

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

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

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

2.3 Tuples, Lists, Options, and Pattern Matching

2.3.1 Tuples

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

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

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

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

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

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

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

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

Pattern matching can also show up in function arguments. Here's a function for computing the distance between two points on the plane, where each point is represented as a pair of `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) =
    Float.sqrt ((x1 -. x2) **. 2. +. (y1 -. y2) **. 2.);;
val distance : float * float -> float * float -> float = <fun>
```

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

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

Operators in Base and the Stdlib

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

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

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

2.3.2 Lists

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

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

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

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

The List Module

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

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

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

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

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

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

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

We'll learn more about labeled arguments and why they're important in Chapter 3 (Variables and Functions).

Constructing Lists with ::

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

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

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

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

Semicolons Versus Commas

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

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

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

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

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

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

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

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

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

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

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

List Patterns Using Match

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

```
# let my_favorite_language (my_favorite :: the_rest) =
    my_favorite;;
Lines 1-2, characters 26-16:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
val my_favorite_language : 'a list -> 'a = <fun>
```

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

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

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

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

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

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

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

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

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

Recursive List Functions

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

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

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

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

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

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

This suggests a reasonable if not entirely accurate mental model for what OCaml is actually doing to evaluate a recursive function.

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

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

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

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

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

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

2.3.3 Options

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

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

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

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

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

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

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

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

In OCaml, however, missing values are explicit. A value of type `string * string` always contains two well-defined values of type `string`. If you want to allow, say, the first of those to be absent, then you need to change the type to `string option * string`. As we'll see in Chapter 8 (Error Handling), this explicitness allows the compiler to provide a great deal of help in making sure you're correctly handling the possibility of missing data.

2.4 Records and Variants

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

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

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

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

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

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

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

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

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

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

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

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

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

Now, imagine that you want to combine multiple objects of these types together as a description of a multi-object scene. You need some unified way of representing these objects together in a single type. *Variant* types let you do just that:

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

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

Here's how we might write a function for testing whether a point is in the interior of some element of a list of `scene_elements`. Note that there are two `let` bindings in a row without a double semicolon between them. That's because the double semicolon is required only to tell *utop* to process the input, not to separate two declarations

```
# let is_inside_scene_element point scene_element =
    let open Float.O in
    match scene_element with
    | Circle { center; radius } ->
      distance center point < radius
    | Rect { lower_left; width; height } ->
      point.x    > lower_left.x && point.x < lower_left.x + width
      && point.y > lower_left.y && point.y < lower_left.y + height
    | Segment _ -> false

  let is_inside_scene point scene =
    List.exists scene
      ~f:(fun el -> is_inside_scene_element point el);;
val is_inside_scene_element : point2d -> scene_element -> bool = <fun>
val is_inside_scene : point2d -> scene_element list -> bool = <fun>
# is_inside_scene {x=3.;y=7.}
  [ Circle {center = {x=4.;y= 4.}; radius = 0.5 } ];;
- : bool = false
# is_inside_scene {x=3.;y=7.}
  [ Circle {center = {x=4.;y= 4.}; radius = 5.0 } ];;
- : bool = true
```

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

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

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

Base and Polymorphic Comparison

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

2.5 Imperative Programming

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

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

2.5.1 Arrays

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

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

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

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

2.5.2 Mutable Record Fields

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

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

The fields in `running_sum` are designed to be easy to extend incrementally, and sufficient to compute means and standard deviations, as shown in the following example.

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

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

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

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

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

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

```
# let rsum = create ();;
val rsum : running_sum = {sum = 0.; sum_sq = 0.; samples = 0}
# List.iter [1.;3.;2.;-7.;4.;5.] ~f:(fun x -> update rsum x);;
- : unit = ()
# mean rsum;;
- : float = 1.33333333333333326
# stdev rsum;;
- : float = 3.94405318873307698
```

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

2.5.3 Refs

We can create a single mutable value by using a `ref`. The `ref` type comes predefined in the standard library, but there's nothing really special about it. It's just a record type with a single mutable field called `contents`:

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

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

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

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

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

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

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

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

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

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

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

Nesting lets with `let` and `in`

The definition of `sum` in the above examples was our first use of `let` to define a new variable within the body of a function. A `let` paired with an `in` can be used to introduce a new binding within any local scope, including a function body. The `in` marks the beginning of the scope within which the new variable can be used. Thus, we could write:

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

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

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

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

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

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

2.5.4 For and While Loops

OCaml also supports traditional imperative control-flow constructs like `for` and `while` loops. Here, for example, is some code for permuting an array that uses a `for` loop:

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

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

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

Here's an example run of this code:

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

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

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

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

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

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

2.6 A Complete Program

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

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

```
open Base
open Stdio

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

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

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

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

2.6.1 Compiling and Running

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

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

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

In this case, however, we just have one:

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

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

We can now invoke dune to build the executable.

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

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

```
$ ./_build/default/sum.exe
1
2
3
94.5
Total: 100.5
```

More work is needed to make a really usable command-line program, including a proper command-line parsing interface and better error handling, all of which is covered in Chapter 16 (Command-Line Parsing).

2.7 Where to Go from Here

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