CHAPTER 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. This should give a sense of what OCaml can do, without going into too much detail about any particular topic.

We'll present this guided tour using the OCaml toplevel, an interactive shell that lets you type in expressions and evaluate them interactively. When you get to the point of running real programs, you'll want to leave the toplevel behind, but it's a great tool for getting to know the language.

You should have a working toplevel as you go through this chapter, so you can try out the examples as you go. There is a zero-configuration browser-based toplevel that you can use for this, which you can find here:

```
http://realworldocaml.org/core-top
```

Or you can install OCaml and Core on your computer directly. Instructions for this are found in Appendix {???}.

OCaml as a calculator

Let's spin up the toplevel and open the `Core.Std` module, which gives us access to Core's libraries, and then try out a few simple numerical calculations.

```
$ rlwrap ocaml
        Objective Caml version 3.12.1

# open Core.Std;;
# 3 + 4;;
- : int = 7
# 8 / 3;;
- : int = 2
# 3.5 +. 6.;;
- : float = 9.5
```

```
# sqrt 9.;;
- : float = 3.
```

This looks a lot what you'd expect from any language, but there are a few differences 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 pecularity of the toplevel that is not required in compiled code.
- After evaluating an expression, the toplevel spits out both the type of the result and the result itself.
- Function application in OCaml is syntactically unusual, in that function arguments are written out separated by spaces, rather than being demarcated by parentheses and commas.
- OCaml carefully distinguishes between `float`, the type for floating point numbers and `int`. The types have different literals (`6.` instead of `6`) and different infix operators (`+.` instead of `+`), and OCaml doesn't do any automated casting between the types. This can be a bit of a nuisance, but it has its benefits, since it prevents some classes of bugs that arise from confusion between the semantics of `int` and `float`.

We can also create variables to name the value of a given expression, using the `let` syntax.

```
# 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, in addition to its type and value.

The above examples are of top-level variables. We can introduce a *local* variable that exists only for the purpose of evaluating a single expression using `let` and `in`:

```
# let z = 3 in z + z;;
- : int = 6
# z;;
Characters 0-1:
  z;;
  ^
Error: Unbound value z
```

Note that `z` is a valid variable in the scope of the expression `z + z`, but that it doesn't exist thereafter.

We can also define multiple local variables using nested `let`/`in` expressions.

```
# let x = 3 in
  let y = 4 in
  x * y ;;
- : int = 12
```

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 (square 2);;
- : int = 16
```

Now that we're creating more interesting values, 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:

```
# let abs_diff x y =
    abs (x - y) ;;
val abs_diff : int -> int -> int = <fun>
```

and even functions that take other functions as arguments:

```
# let abs_change f x =
    abs_diff (f x) x ;;
val abs_change : (int -> int) -> int -> int = <fun>
# abs_change square 10;;
- : int = 90
```

This notation for multi-argument functions may be a little surprising at first, but we'll explain where it comes from when we get to function currying in Chapter 3. 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 of the function. Thus,

```
int -> int -> int
```

describes a function that takes two `int` arguments and returns an `int`, while

```
(int -> int) -> int -> int
```

describes a function of two arguments where the first argument is itself a function.

As the types we encounter get more complicated, you might ask yourself how OCaml is able to determine these types, given that we didn't write down any explicit type information. It turns out that OCaml is able to determine the type of a new expression using a technique called *type-inference*, by which it infers the type of a new expression based on what it already knows about the types of other related variables. For example, in `abs_change` above, the fact that `abs_diff` is already known to take two integer arguments lets the compiler infer that `x` is an `int` and that `f` returns an `int`.

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 to tell you what the type of `x` and `y` are, which 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`.

This genericity means that we can write:

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

And we can also write:

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

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";;
Characters 25-30:
  first_if_true big_number "short" "loooooong";;
                           ^^^^^^^
Error: This expression has type string but
    an expression was expected of type int
```

In this example, `big_number` requires that `'a` be of type `int`, whereas `"short"` and `"loooooong"` require that `'a` be of type `string`, and they can't all be right at the same time. This kind of genericity is called *parametric polymorphism*, and is very similar to generics in C# and Java.

2012-06-26
22:58:57

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 top-level somewhat obscures the difference between run-time and compile time errors, but that difference is still there. Generally, type errors, like this one:

```
# 3 + "potato";;
Characters 4-12:
  3 + "potato";;
      ^^^^^^^^
Error: This expression has type string but an expression was expected of type
         int
```

are compile-time errors, whereas an error that can't be caught by the type system, like division by zero, leads to a runtime exception.

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

One important distinction is that type errors will stop you whether or not the offending code is ever actually executed. Thus, you get an error from typing in this code:

```
# if 3 < 4 then 0 else 3 + "potato";;
Characters 25-33:
  if 3 < 4 then 0 else 3 + "potato";;
                           ^^^^^^^^
Error: This expression has type string but an expression was expected of type
         int
```

but this code works fine.

```
# if 3 < 4 then 0 else 3 / 0;;
- : int = 0
```

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. You can create a tuple by joining values together with a comma:

```
# let tup = (3,"three");;
val tup : int * string = (3, "three")
```

The type `int * string` corresponds to the set of pairs of `int`s and `string`s. For the mathematically inclined, the `*` character is used because the space of all 2-tuples of type `t * s` corresponds to the Cartesian product of `t` and `s`.

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

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

Here, the `(x,y)` on the left-hand side of the `let` is the pattern. This pattern lets us mint the new variables `x` and `y`, each bound to different components of the value being matched. Note that the same syntax is used both for constructing and for pattern-matching on tuples.

Here's an example of how you might use pattern matching in practice: a function for computing the distance between two points on the plane, where each point is represented as a pair of `float`s.

```
# let distance p1 p2 =
    let (x1,y1) = p1 in
    let (x2,y2) = p2 in
    sqrt ((x1 -. x2) ** 2. +. (y1 -. y2) ** 2)
;;
val distance : float * float -> float * float -> float = <fun>
```

We can make this code more concise by doing the pattern matching on the arguments to the function directly.

```
# let distance (x1,y1) (x2,y2) =
    sqrt ((x1 -. x2) ** 2. +. sqr (y1 -. y2) ** 2.)
;;
```

This is just a first taste of pattern matching. We'll see that pattern matching shows up in many contexts in OCaml, and turns out to be a surprisingly powerful and pervasive tool.

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
```

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 extended list, not changing the list we started with, as we can see.

```
# 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]
```

Thus, `::` and `[]`, which are examples of what are called *type-constructors*, are the basic building-blocks for lists.

Basic list patterns

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

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

By pattern matching using `::`, we've broken off the first element of `languages` from the rest of the list. If you know Lisp or Scheme, what we've done is the equivalent of using `car` and `cdr` to break down a list.

If you tried the above example in the toplevel, you probably noticed that we omitted a warning generated by the compiler. Here's the full output:

```
# let (my_favorite :: the_rest) = languages ;;
Characters 5-28:
  let (my_favorite :: the_rest) = languages ;;
       ^^^^^^^^^^^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val my_favorite : string = "OCaml"
val the_rest : string list = ["Perl"; "C"]
```

The warning comes because the compiler can't be certain that the pattern match won't lead to a runtime error, and the warnings gives an example of the problem, the empty list, `[]`. Indeed, if we try to use such a pattern-match on the empty list:

```
# let (my_favorite :: the_rest) = [];;
Characters 5-28:
  let (my_favorite :: the_rest) = [];;
       ^^^^^^^^^^^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
Exception: (Match_failure "" 1 5).
```

we get a runtime error in addition to the compilation warning.

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. Here's an example:

```
# 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"
```

Recursive list functions

If we combine pattern matching with a recursive function call, we can do things like define a function for summing the elements of a list.

```
# let rec sum l =
    match l with
    | [] -> 0
    | hd :: tl -> hd + sum tl
  ;;
val sum : int list -> int
# sum [1;2;3;4;5];;
- : int = 15
```

We had to add the `rec` keyword in the definition of `sum` to allow `sum` to refer to itself. 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)
  ;;
```

Actually, the code above has a problem. If you type it into the top-level, 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 is warning you that we've missed something, in particular that our code doesn't handle one-element lists. That's easy enough to fix 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 in the above, we 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`.

The `List` module

So far, we've built up all of our list functions using pattern matching and recursion. But in practice, this isn't usually necessary. Instead, you'll mostly use the rich collection of utility functions contained in Core's `List` module. For example:

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

Here, we use the dot-notation to reference elements of the `List` and `String` module. `List.map` in particular is a function that takes a list and a function for transforming elements of that list (under the label `~f`. For now, you can ignore that bit of syntax, and we'll learn more about it in chapter {{{FUNCTIONS}}}), and returns to us a new list with the transformed elements.

Options

Another common data structure in OCaml is the `option`. An `option` is used to express that a value that 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>
```

`Some` and `None` are type 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 get a value out 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_string time) message ;;
val print_log_entry : Time.t option -> string -> unit
```

Here, we again use a match statement for handling the two possible states of an option. It's worth noting that we don't necessarily need to use an explicit `match` statement in this case. We can instead use some built in functions from the `Option` module, which, like the `List` module for lists, is a place where you can find a large collection of useful functions for working with options.

In this case, we can rewrite `print_log_entry` using `Option.value`, which either returns the content of an option if the option is `Some`, or a default value if the option is `None`.

```
# let print_log_entry maybe_time message =
    let time = Option.value ~default:(Time.now ()) maybe_time in
    printf "%s: %s\n" (Time.to_string time) message ;;
```

Options are important because they are the standard way in OCaml to encode a value that might not be there. Values in OCaml are non-nullable, so if you have a function that takes an argument of type `string`, then the compiler guarantees that, if the code compiles successfully, then at run-time, that function will only be called with awell-defined value of type `string`. This is different from most other languages, including

Java and C#, where objects are by default nullable, and whose type systems do little to defend from null-pointer exceptions at runtime.

Records and Variants

So far, we've only looked at data structures that were pre-defined 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>
```

We can write the pattern match even more tersely, using what's called *field punning*. In particular, when the name of the field and the name of the variable coincide, we don't have to write them both down. Thus, the 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 of these objects together, say 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 `Scene`, 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 =
     let point_is_inside_scene_element scene_element =
       is_inside_scene_element point scene_element
     in
     List.for_all scene ~f:point_is_inside_scene_element;;
val is_inside_shapes : point2d -> scene_element list -> bool = <fun>
```

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).

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. 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 datastructures 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 datastructure in OCaml is the array. 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|]
```

In the above, the `.(i)` syntax is used for referencing the element of an array, and the `<-` syntax is used for modifying an element of the array.

Arrays in OCaml are very similar to arrays in other languages like C: they are fixed width, 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. OCaml uses three words per element of a list, but only one per element of an array.

Mutable record fields

The array is an important mutable datastructure, 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 datastructure 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, for stdev *)
     mutable samples: int; } ;;
```

Here are some functions for computing means and standard deviations based on the `running_sum`.

```
# 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>
```

And finally, we can define functions for creating and modifying a `running_sum`. Note the use of single semi-colons to express a sequence of operations. When we were operating purely functionally, this wasn't necessary, but you start needing it when your code is acting by side-effect.

```
# 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>
```

A new and somewhat odd type has cropped up in this example: `unit`. What makes `unit` unusual is that there is only one value of type `unit`, which is written `()`. Because `unit` has only one inhabitant, a value of type `unit` can't 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.

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 = 1.61015297179882655
```

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 = ()
# incr x ;;       (* increment, i.e., x := !x + 1 *)
- : unit = ()
# !x ;;
- : int = 2
```

A ref is really just an example of a mutable record, but in practice, it's the standard way of dealing with a single mutable value in a computation.

For and while loops

Along with mutable data structures, OCaml gives you constructs like `while` and `for` loops for interacting with them. Here, for example, is a piece of imperative code for permuting an array. Here, we use the `Random` module as our source of randomness. (`Random` starts out with a deterministic seed, but you can call `Random.self_init` to get a new random seed chosen.)

```
# 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>
```

Note that the semi-colon after the first array assignment doesn't terminate the scope of the let-binding.

```
# 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|]
```

Where to go from here

That's it for our guided tour! There are plenty of features left to touch upon - we haven't shown you how to read from a file, as just one example - but the hope is that this has given you enough of a feel for the language that you have your bearings, and will be comfortable reading examples in the rest of the book.