CHAPTER 4

Understanding Functions

Functions are the main part of OCaml, which is not surprising because OCaml is a functional programming (FP) language. Functions in OCaml have signatures that are displayed in the OCaml toplevel and are used extensively in the module language.

For now, you'll focus on function signatures displayed in the toplevel. These function signatures enable you to know what parameters a function will accept and the return type of the function. They do not indicate whether a given function is recursive, however.

Creating Values and Functions

Functions are data items that take arguments and return a data item of a given type. A function can take and return any valid OCaml type, including other functions. (This capability to take and return functions will be discussed in more detail later in this chapter.) A *value* is a data item of a given type that contains some data. Values can be thought of as variables, although they are not really "variable" in the sense of being able to be changed.

Values are a label for a given collection of data. They are *immutable*—they cannot be changed once assigned. They can, however, be overridden. Values are created by using the `let` function and can be defined statically (for example, assigning a value to a string or number) or from the result of computation (for example, the result of a function).

If you go to the OCaml toplevel, you can assign some values by using the `let` keyword:

```
# let a = 5;;
val a : int = 5
# let b = 10;;
val b : int = 10
#
```

Note Keywords such as `let` are not really functions. Although they take arguments and return items, they do not perform computations. You can think of them as prepositions (where values are nouns and functions are verbs).

The preceding code assigned the value `a` to be `5` and the value `b` to be `10`. You can now use `a` and `b` in other functions.

```
# a + b;;
- : int = 15
#
```

You can define functions in the same way as values by using the `let` keyword. In the following code, the function `myfunc` is defined with `()` as the first argument. This is shorthand for a `unit` argument, which is very similar in concept to `void` (found in languages such as Java). This function performs some simple math and returns an integer:

```
# let myfunc () = 1 + 1;;
val myfunc : unit -> int = <fun>
```

Pay careful attention to the line after the definition. This line, which is called the *signature*, shows what type of arguments a given function takes (and how many) and the return type. It also shows that `myfunc` is a special value of the `fun` type, which indicates that `myfunc` is a function instead of a simple value.

If you call this function with the `unit` argument, you get the expected result, which is a value. The signature of the return tells you the value of the return and its type:

```
# myfunc ();;
- : int = 2
```

You can explicitly define a function by using the `fun` keyword:

```
# let myfunc = (fun () -> 1 + 1);;
val myfunc : unit -> int = <fun>
# myfunc ();;
- : int = 2
```

No matter how you define functions, their properties are the same.

Most functions are like these—prefix functions—because the default is to define functions as prefix functions. However, infix functions are easy to define. This chapter will cover both prefix and infix functions.

The previous example shows the difference between functions and values: functions must take parameters. The function signature of both definitions is the same. The function signature says that the function `myfunc` takes one parameter of type `unit` and returns an `int`. The special `()` is the `unit` type, which can serve as a placeholder for a value you have no intention of passing. This is a way to make functions that don't actually take any real parameters.

You can also have `let` bindings inside of your functions, which can be any allowed value. A common use for the bindings is to have values defined within your functions, which is done like so:

```
# let myfunc x y =
  let someval = x + y in
  Printf.printf "Hello internal value: %i\n" someval;;
val myfunc : int -> int -> unit = <fun>
# myfunc 10 20;;
Hello internal value: 30
- : unit = ()
#
```

This code creates the equivalent of temporary variables in a function. The garbage collection will handle their destruction, so you need only to define them. These definitions can be arbitrarily deep, although you might want to create new functions if your definitions get too complicated.

Functions Must Have One Return Type

A nonpolymorphic function can have only one return type in OCaml. Even polymorphic functions have only one return type, although this return type can be a polymorphic type. Polymorphism is enforced by the type inference engine, and it is very easy for a novice to try to create functions that do not return the type you think they do. Fortunately, the compiler will give you an error message if you attempt to do this (this is shown in the next example).

Function return types will be automatically figured out by the compiler. The compiler will find the most generic type a function can return and then use it. If your function is polymorphic, it will automatically be designated as such, and you cannot force a function to be polymorphic simply by defining it.

This automatic determination of return type can lead to some frustrating situations in which you and the compiler disagree with what a given function returns. As shown in the following example, if you try to create a function that does not have one return type, the compiler gives you an error. The error message the compiler gives can sometimes appear unhelpful (in this example, the error is that a string is returned if an exception is encountered).

```
# let errorprone x = try
   while !x < 10 do
      incr x
   done
   with _ -> "Failed";;
Characters 100-108:
     with _ -> "Failed";;
               ^^^^^^^^
This expression has type string but is here used with type unit
# let errorprone x = try
   while !x < 10 do
    incr x
   done
   with _ -> ();;
val errorprone : int ref -> unit = <fun>
#
```

Constraining Types in Function Calls

You can specify a function's parameter types for a couple of reasons (the most important reason is that you might have to). Although the compiler attempts to infer the types used in your functions, it does not always succeed, especially when mutable values are used and in certain other cases (this topic is discussed in more detail in Chapter 5).

Another reason to specify the type is to make the type clear. If you know that a given parameter will be a given type, you can define it and know that the compiler will give an error if it is used incorrectly.

In most cases, however, the explicit definition of a parameter type is not required. Although there are some cases in which constraining the type will yield performance benefits, it is not often done because the development benefit of having the compiler do the work is frequently greater. The compiler will still infer the type information and verify that it is correct, even if you have specified it. The compiler will give you errors if the inferred type is incompatible with the specified type:

```
# let mismatching (x:int) (y: float) = x + (int_of_string y);;
Characters 56-57:
  let mismatching (x:int) (y: float) = x + (int_of_string y);;
                                                          ^

This expression has type float but is here used with type string
#
```

Using Higher-Order Functions

OCaml has support for *higher-order functions (HOFs)*, which take other functions as arguments. OCaml can do this because functions are first-class types. For example, the following (somewhat contrived) example is a function that takes three arguments. The first argument is a function that takes two arguments; the other two arguments are then passed as arguments to the first argument. In this case, the example just uses the numeric comparison arguments to demonstrate. Because the > and < operators are infix operators, they must be enclosed in parentheses (if you don't do this, you get the shown error):

```
# let bigger f x y = f x y;;
val bigger : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# bigger (>) 10 20;;
- : bool = false
# bigger (<) 10 20;;
- : bool = true
# bigger < 10 20;;
Characters 9-11:
  bigger < 10 20;;
           ^^
This expression is not a function, it cannot be applied
#
```

Another example of a HOF is when some (but not all) of the arguments that a function needs are passed to it. This function is called a *curried function*.

```
# let add_one = (+) 1;;
val add_one : int -> int = <fun>
# add_one 10;;
- : int = 11
# (fun x y -> x * y);;
- : int -> int -> int = <fun>
```

You can define anonymous functions and use them in much the same way. An anonymous function that is assigned to a name is indistinguishable from a function defined any other way. The preceding code shows both a curried function and an anonymous function. Read on for more on their creation and use.

Using Lists

Lists are used to create some of the examples in this chapter. However, the syntax for dealing with lists will not be covered for a few chapters yet. To make the examples more understandable, here is a quick introduction to OCaml lists:

- List elements must be of the same type.
- Lists are defined by using square brackets, with elements separated by semicolons.
- Lists are indexed starting at 0 (not 1).
- Lists are a basic type and a very important data structure.
- List elements cannot be modified.
- Lists are of fixed length and cannot be resized.

```
# [1;2;3;4;5];;
- : int list = [1; 2; 3; 4; 5]
# List.sort compare [4;5;6;2;4;2;0];;
- : int list = [0; 2; 2; 4; 4; 5; 6]
# List.nth [0;1;2;3;4] 3;;
- : int = 3
# List.nth [0;1;2;3;4] 0;;
- : int = 0
```

Lists also have comparing/sorting functions. The `compare` function shown in the preceding code is a built-in function that can be used for sorting any valid OCaml type. Lists are covered in much greater depth later in the book, but this introduction should give you enough information to understand the examples in this chapter.

Anonymous Functions

Anonymous functions are very useful in OCaml. Sometimes referred to as *generic* functions, anonymous functions are functions that do not have a name so they are not assigned to any value. There are many reasons to use and create anonymous functions.

Many functions take functions as arguments, and anonymous functions are an easy way to pass those functions.

The `compare` function for sorting lists is a good example—it takes two arguments and returns an integer of `1` if the second value is less than the first. It returns `0` if the values are equal and `-1` if the second value is greater than the first.

You can pass your own `compare` function to other functions, such as the `List.sort` function, and have them sort based on your `compare` function. This example sorts a list by a reversing anonymous function and by the default `compare` function.

```
# List.sort (fun x y -> if x = y then
  -1
  else if x < y then
  1
  else
  0) [1;4;9;3;2;1];;
- : int list = [9; 4; 3; 2; 1; 1]
# List.sort compare [1;4;9;3;2;1];;
- : int list = [1;1;2;3;4;9]
```

Although anonymous functions are used a great deal in OCaml, they can sometimes hurt code readability. Functions are basic types in OCaml and (like other values in OCaml) also do not have to be assigned to a name. Unlike other values, unnamed or anonymous functions have many uses.

This property of functions is one of the differences between OCaml (and FP languages in general) and other kinds of programming languages. Functions can be passed to other functions and returned from functions. These passed and returned functions can also be anonymous functions. In this book, you will see anonymous functions used with the `scanf` commands in many chapters.

```
# Scanf.sscanf "hello world" "%s %s" (fun x y -> Printf.printf "%s %s\n" y x);;
world hello
- : unit = ()
```

The anonymous function used here reversed the strings. This function was created on-the-fly by using the `fun` keyword. The only difference between anonymous functions and named functions is that one has a name and the other does not. Because the naming of functions is a help to the programmer more than it is a technical requirement, you could write all your programs using only anonymous functions. Doing that is not recommended—for obvious reasons.

Putting the *Fun* in Functions

The `fun` keyword can be used to define functions that are assigned to a name and it can create them without assigning them to a name. Functions in OCaml must take an argument. This property is one of the attributes that separate functions from plain values. You do not have to use the `fun` keyword when defining anonymous functions if you are currying functions (which is discussed later in this chapter).

In an interactive session, you can see the results:

```
# (fun x -> 10);;
- : int -> int = <fun>
# (fun x -> 10 + x) 30;;
- : int = 40
# Printf.printf "%s\n";;
- : string -> unit = <fun>
# let funclist = [Printf.printf "%s\n"];;
val funclist : (string -> unit) list = [<fun>]
```

```
# (List.nth funclist 0) "hello world";;
hello world
- : unit = ()
#
```

Why Use Anonymous Functions?

You can use anonymous functions when you are creating functions that operate on functions. HOFs can be very useful for solving problems, especially when you want to build up a group of operations and results. They can also be used for complicated operations and lists of functions and/or callbacks.

List folding is an example of when using an anonymous function is very handy. (*Folding* is referred to as *reducing* in some languages.) Although the `map` function applies a given function to each element in a collection, the `fold` function takes two arguments—the first is the list element and the second is the result of the previous function call (and so on). The initial value is specified as an argument to the `fold` function. This value is also the return value if you have an empty list. In the following example, the + function is used (indicate to the compiler that it is a function by enclosing it in parentheses).

```
# List.fold_left (+) 0 [1;2;3;4;5];;
- : int = 15
```

This is equivalent to doing 0 + 1 + 2 + 3 + 4 + 5. You can curry this function and create a new function, `sum`, that does it all in a simple-to-call manner.

```
# let sum = List.fold_left (+) 0;;
val sum : int list -> int = <fun>
# sum [1;2;3;4;5;6];;
- : int = 21
#
```

Understanding Consequences of Functions As Data

As shown in the callback example, when the functions are registered, they are anonymized. Although they lose their name, you can find the name of a function that has been stored in a data structure or passed as an argument to another function using physical equality (the = operator). This doesn't change anything about the function. For example, the following example creates a simple function and stores it into a list. Once stored in the list, the function is not accessible from that list by its name. Physical equality also distinguishes the function from another function, even if that function is otherwise the same.

```
# let f x y = x + y;;
val f : int -> int -> int = <fun>
# let m = [f];;
val m : (int -> int -> int) list = [<fun>]
# (List.hd m) == f;;
- : bool = true
# let b = f;;
val b : int -> int -> int = <fun>
```

```
# b == f;;
- : bool = true
# let c x y = x + y;;
val c : int -> int -> int = <fun>
# c == f;;
- : bool = false
#
```

Although functions cannot be decomposed, you can compose functions however you like. Take care when you do this—function composition can yield results that are difficult to figure out. The compiler can do the composition if the code is syntactically valid and can sometimes present error messages that are equally difficult to understand. This is a situation in which the syntax (the structure) of the code might be mostly correct, but the semantics (the meaning) is almost certainly incorrect.

To illustrate, here is a simple test case. This case really doesn't do anything except highlight the ease in which you can get yourself in trouble with composition. You start with a simple function: `compose`:

```
# let compose m y = y m;;
val compose : 'a -> ('a -> 'b) -> 'b = <fun>
```

This function reorders a pair of functions that are passed to it and returns a third, newly composed function:

```
# compose (fun x -> x 3.14159) (fun m n o -> (m n) o);;
- : (float -> '_a -> '_b) -> '_a -> '_b = <fun>
# let b = compose (fun x -> x 3.14159) (fun m n o -> (m n) o);;
val b : (float -> '_a -> '_b) -> '_a -> '_b = <fun>
# b 3.123;;
Characters 2-7:
  b 3.123;;
    ^^^^^
This expression has type float but is here used with type float -> 'a -> 'b
# b (fun n m o -> o);;
- : '_a -> '_b -> '_b = <fun>
# (b (fun n m o -> o)) "hi" "there";;
- : string = "there"
#
```

The only way you know that a given value is actually a function is by paying careful attention to the signatures. You can also pay attention to the compiler errors, although doing so makes the process take much longer.

■ **Caution** If you find yourself often composing functions, you might want to review your functions and break them down. They might be too complicated.

Functions cannot be serialized via the `Marshal` module. This is a limitation of the `Marshal` module that is difficult to overcome because functions require abstract values (which also cannot be serialized). If you need to have anonymous functions serialized, you have to do it manually or convert the functions into a data structure (which is often much easier said than done).

Curried Functions

Currying is when you take a function that takes multiple arguments and turn it into a function that takes only one argument. The term *curried function* is often used to describe any situation in which you transform a function into a function that takes different arguments. You can do more than simply pass functions as parameters to other functions—you can assign a value from a function with some of its parameters passed to it and then use it as if it were a function with different arguments.

Looking back at the example given earlier in the chapter, here is a new function that takes only one argument from a function that originally took two arguments:

```
# let add_one = (+) 1;;
val add_one : int -> int = <fun
# add_one 10;;
- : int = 11
#
```

In this case, you fixed one of the two arguments (by setting it to `1`) in the function. The function is not evaluated until all its arguments are complete.

Why Curried Functions Are Important

By using curried functions, you can build up the arguments to your functions at runtime, so you do not have to resort to list processing to build a runtime-defined list of arguments.

Curried functions also enable you to refactor your code more effectively. Because you can easily redefine how function arguments appear to any given function, you can restructure your code accordingly.

Working with the Distance Type

You will be working with the nonpolymorphic version of the `distance` type created in the last chapter. You will start with conversions between the different variants and then go on to create a four-function distance calculator.

```
# type distance = Meter of int | Foot of int | Mile of int;;
type distance = Meter of int | Foot of int | Mile of int
```

Converting Between Kinds of Distances

Simple conversion functions enable you to convert to meters, feet, and miles (for clarity, this example uses the naming convention `_*` instead of the more standard `distance_of_*` convention):

```
# let to_meter x = match x with
  Foot n -> Meter ( n / 3)
  | Mile n -> Meter (n * 1600)
  | Meter n -> Meter n;;
val to_meter : distance -> distance = <fun>
# let to_foot x = match x with
  Mile n -> Foot (n * 5000)
  | Meter n -> Foot (n * 3)
  | Foot n -> Foot n;;

val to_foot : distance -> distance = <fun>
# let to_mile x = match x with
  Meter n -> Mile (n / 1600)
  | Foot n -> Mile (n / 5000)
  | Mile n -> Mile n;;
val to_mile : distance -> distance = <fun>
#
let meter_of_int x = Meter x;;
val meter_of_int : int -> distance = <fun>
# meter_of_int 10;;
- : distance = Meter 10
```

You might wonder whether you can just define an `int_of_distance` function, pull out the number, and be done with it. The reason for not doing this is type preservation. The distance example was created to show how type can prevent certain types of programming errors from occurring. By erasing the type of the `distance` (`Foot` or `Meter`), you can have a situation in your code in which distances get added that shouldn't be, which might cause a very subtle error in your output (or worse). If you were really working with these kinds of measurements, you might not even want a `distance` type; you might want a `Meter` type and a `Foot` type, and so on.

Because it is a safe language, OCaml gives you tools to avoid these kinds of errors right out of the gate. You should always think very hard before reverting to type erasure to solve a problem. You might be opening yourself up to future problems.

Creating a Four-Function Distance Calculator

You will define a default match for most of these functions, which will prevent a warning being generated by the compiler. The warning that would be generated from these definitions if you did not include a default match is the following: `Warning: this pattern-matching is not exhaustive. Here is an example of a value that is not matched: (Mile _|Foot _)`.

Although there is no operator overloading in OCaml, it is easy to define your own infix functions. There are no restrictions on defining infix functions, although you should not use (* or *) because they can cause confusion with the comment character in OCaml. In fact, you should always put spaces around any use of * to avoid confusion. Take care to not redefine any built-in functions. Although redefining is not prohibited, it is a bad practice and will confound anyone after you who must maintain the code.

The first function you will define is addition. You will use the infix syntax for all of these functions. These are examples of how HOFs can simplify code. The function names imply that they are strictly mathematical functions, but you can pass arbitrary functions as long as they take two arguments and return the correct type.

```
# let math_on_meter x y z = match x,y with
  Meter n, Meter m -> Meter (z n m)
  | _ -> raise Not_found ;;
val math_on_meter : distance -> distance -> (int -> int -> int) -> distance =
  <fun>
# let math_on_foot x y z = match x,y with
  Foot n,Foot m -> Foot (z n m)
  | _ -> raise Not_found;;
val math_on_foot : distance -> distance -> (int -> int -> int) -> distance =
  <fun>
# let math_on_mile x y z = match x,y with
  Mile n,Mile m -> Mile (z n m)
  | _ -> raise Not_found;;
val math_on_mile : distance -> distance -> (int -> int -> int) -> distance =  <fun>
#
```

Having defined infix functions, you can now clearly implement the math functions:

```
# let ( %+ ) x y = match x with
  Meter n -> math_on_meter x (to_meter y) ( + )
  | Foot n -> math_on_foot x (to_foot y) ( + )
  | Mile n -> math_on_mile x (to_mile y) ( + );;
val ( %+ ) : distance -> distance -> distance = <fun>
# let ( %- ) x y = match x with
  Meter n -> math_on_meter x (to_meter y) ( - )
  | Foot n -> math_on_foot x (to_foot y) ( - )
  | Mile n -> math_on_mile x (to_mile y) ( - );;
val ( %- ) : distance -> distance -> distance = <fun>
# let ( %* ) x y = match x with
  Meter n -> math_on_meter x (to_meter y) ( * )
  | Foot n -> math_on_foot x (to_foot y) ( * )
  | Mile n -> math_on_mile x (to_mile y) ( * );;
val ( %* ) : distance -> distance -> distance = <fun>
# let ( %/ ) x y = match x with
  Meter n -> math_on_meter x (to_meter y) ( / )
  | Foot n -> math_on_foot x (to_foot y) ( / )
  | Mile n -> math_on_mile x (to_mile y) ( / );;
val ( %/ ) : distance -> distance -> distance = <fun>
#
```

You see an example of these functions together here:

```
# (Meter 3) %* (Mile 1);;
- : distance = Meter 4800
# (Foot 12) %- (Foot 3);;
- : distance = Foot 9
#
```

Creating Recursive Functions

Recursion is very important in OCaml. In particular, *tail recursion* is significant because it can create enormous performance improvements in your code.

Anonymous functions cannot be recursive because you cannot call what has no name. We will talk about how you can do many other equally interesting things with anonymous functions in this chapter.

Recursion is used extensively in functions, data structures, and types. Also, because of performance gains that can be realized via tail recursion, recursive solutions are often chosen over iterative solutions. However, recursion does not automatically make programs perform better. In fact, recursive solutions that are not tail-recursive often perform poorly compared with other solutions.

The Fibonacci sequence is often solved via recursion:

```
# let rec fib n = if (n < 2) then
  1
  else
  (fib (n - 1)) + (fib (n - 2));;
val fib : int -> int = <fun>
# fib 6;;
- : int = 13
#
```

But when was the last time you needed to use the Fibonacci sequence? Recursion is used very often in OCaml code to solve a variety of problems and is often used to replace iterative loops. The following example presents two functions that explode strings into lists of chars and collapse lists of chars to strings:

```
let explode_string x =
  let strlen = String.length x in
  let rec es i acc =
    if (i < strlen) then
      es (i+1) (x.[i] :: acc)
    else
      List.rev acc
  in
  es 0 [];;
```

```
let collapse_string x =
  let buf = Buffer.create (List.length x) in
  let rec cs i = match i with
    [] -> Buffer.contents buf
  | h :: t -> Buffer.add_char buf h;cs t
  in
  cs x;;
```

These two functions also show the common practice of wrapping a recursive function in a nonrecursive one to keep the accumulator variable out of the function called by the programmer.

Why Do Recursive Functions Need a Special Designation?

Recursive functions are special in the OCaml world. You must tell the compiler that the function is recursive so the name of the function is available to itself (otherwise, you get an error). The following example shows that without defining the function as recursive, the compiler does not know the function exists when the function is called recursively:

```
# let wrong_recursive lst acc = match lst with
     [] -> acc
   | h :: t -> wrong_recursive t ((String.length h) :: acc);;
    Characters 75-90:
     | h :: t -> wrong_recursive t ((String.length h) :: acc);;
                 ^^^^^^^^^^^^^^^
Unbound value wrong_recursive
# let rec wrong_recursive lst acc = match lst with
     [] -> acc
   | h :: t -> wrong_recursive t ((String.length h) :: acc);;
    val wrong_recursive : string list -> int list -> int list = <fun>
#
```

There are also optimizations that the compiler can make on recursive functions. The best-known of these are tail-recursion optimizations, which allow for very fast constant stack operations.

Tail Recursion and Efficient Programming

When a function is called, the arguments it was called with remain on the execution stack until the function returns; then they are popped off. Because the memory available to any given machine is finite, there is a real limit to how many times a function might recur until the machine runs out of memory and OCaml throws a `Stack_overflow` exception.

Tail recursion is a type of recursive call in which there is no further computation required on the result of the call, so the values of the function arguments are no longer required and can be popped. The OCaml compiler can detect tail-recursive calls and optimize for them, making recursive calls run in a constant stack.

A function is tail-recursive if it meets two criteria. The first criterion is the easiest: the recursive call cannot be within a try/with block. The following example, therefore, cannot be tail-recursive:

```
# let rec scan_input scan_buf acc_buf = try
                Scanf.bscanf scan_buf "%c" (fun x -> Buffer.add_char acc_buf x);
                scan_input scan_buf acc_buf
                with End_of_file -> Buffer.contents acc_buf;;
val scan_input : Scanf.Scanning.scanbuf -> Buffer.t -> string = <fun>
#
```

The second criterion for tail recursion is that the returned value is the unmodified return value. The Fibonacci function defined previously is not tail-recursive because the return value requires recursive calculations. However, the `explode` and `collapse` functions are tail-recursive.

Doing More Pattern Matching

Although your functions do not have to do pattern matching, it is a powerful tool and is used widely within the world of OCaml code. So far, you have done pattern matching only on types, but you can pattern match on values as well.

Pattern matches must be static; that is, they must be known at compile time. You can, however, use guarded matches to provide for extended matching.

Pattern matching must be used to access data structures other than pairs, for example. You can define triplets and quadruplets on-the-fly and then access them by using pattern matching. These sequence types are enforced by the compiler, although they do not have a name. You can name these types if you want, although these kinds of structures are often anonymous. You can also use them as components in any aggregate type.

```
let myfunc x = match x with
  n,m,z -> (n+m,z+. 4.);;
val myfunc : int * int * float -> int * float = <fun>
# myfunc (1,2,3.);;
- : int * float = (3, 7.)
# myfunc 1;;
Characters 7-8:
  myfunc 1;;
         ^
This expression has type int but is here used with type int * int * float
#
```

The preceding function can also be written (and is more commonly written) this way:

```
# let myfunc (n,m,z) = (n+m,z+. 0.4);;
val myfunc : int * int * float -> int * float = <fun>
```

You can have a default match, which matches anything. The previous examples used the default match to throw an exception when a parameter is not the right kind of variant of a given type. That is not all you can do, however.

If you have a match that is not used, the compiler will tell you about it, too. For example, if you want to define a polymorphic function that is similar to the previous one but takes three integers, you could write this:

```
# let myfunc x = match x with
  n,m,z -> n+m+z
  | n,m,_ -> n+m;;
Characters 49-54:
Warning: this match case is unused.
    | n,m,_ -> n+m;;
      ^^^^^
val myfunc : int * int * int -> int = <fun>
```

This code does not work, however. If you want the third argument to be polymorphic, you have to use type constraints or not operate on that argument with type-specific operations. Polymorphism is denoted by `'a` in the function signature.

```
let myfunc x = match x with
  n,m,_ -> n+m;;
val myfunc : int * int * 'a -> int = <fun>
# let myfunc (n,m,z) = n+m;;
val myfunc : int * int * 'a -> int = <fun>
```

Lists can also be used in pattern matching. A very LISP-ish kind of function definition can be created by using recursion and pattern matching. For example, the following introduces the `::` operator, which splits a list into its head and tail. This function is like the LISP `car` and `cdr` folded into one operation:

```
# let rec lispy x acc = match x with
  [] -> acc
  | head :: tail -> lispy tail (acc + head);;
val lispy : int list -> int -> int = <fun>
# lispy [1;2;3;4;5] 0;;
- : int = 15
#
```

Understanding the Default Match

The *default match* is present if there are no matches defined. The compiler also gives a warning if your pattern matches are not exhaustive and if you have matches that will never hit.

Having a default match does not make your functions polymorphic. The compiler infers the most generic type the arguments can be and assigns them accordingly. Having a default match doesn't change the fact that a function can have only one return type.

Bindings Within Pattern Matches

You can pretty much do anything you would normally do in OCaml inside an inner `let` binding, but you need to always be aware of scoping issues. Like most languages, you should use parentheses to clearly show scope:

```
# let b x y = match x with
  0 -> (let q = x in match y with
                0 -> 1
              | _ -> q)
  | 1 -> y
  | _ -> x * y;;
val b : int -> int -> int = <fun>
# b 0 3;;
- : int = 0
# b 0 0;;
- : int = 1
#
```

Guarded Matches: A Return to the Distance Calculator

So, let's say you want a special addition function that adds only distances that are positive. You could rewrite the math functions or you could add another set of functions with guards.

Guards are ways to limit the allowed values in pattern matches. Guards are Boolean functions that create a match if and only if the function is `true`. Guards do have a performance penalty, and it is considered bad form to have a function with all pattern matches guarded.

Those items aside, guards are an excellent way to communicate allowed ranges in your functions. The following example shows that an exception is raised if the first parameter is less than or equal to `0`:

```
# let ( %%+ ) x y = match x with
  Foot n when n > 0 -> math_on_foot x (to_foot y) ( + )
  | Meter n when n > 0 -> math_on_meter x (to_meter y) ( + )
  | Mile n when n > 0  -> math_on_mile x (to_mile y) ( + )
  | _ -> raise (Invalid_argument "Not a distance type I have defined");;
val ( %%+ ) : distance -> distance -> distance = <fun>
# (Foot 0) %%+ (Mile 3);;
Exception: Invalid_argument "Not a distance type I have defined".
# (Foot 3) %%+(Mile 3);;
- : distance = Foot 15003
#
```

Writing code for both sides is left as an exercise for the reader.

Understanding Built-in Functions

We have not formally introduced many of the built-in functions in OCaml. Some are included where needed, but this book is not intended to be a full language primer.

You should consult the OCaml reference manual for documentation about the myriad built-in functions that exist. The OCaml standard library is quite large and contains many convenience functions you might be tempted to write.

Using Labels and Optional Arguments

In OCaml, most function parameters are *sequential*, which means they are used in the order in which they are supplied. However, you can change this order and provide optional arguments and default values by using *labels*.

Functions with labels have a slightly different syntax, as can be seen in the following examples. To designate code with labels, add the ~ flag to the parameters.

```
# let add_some_labeled ~x ~y = x + y;;
val add_some_labeled : x:int -> y:int -> int = <fun>
# add_some_labeled 10 20;;
- : int = 30
# add_some_labeled ~x:10 30;;
Characters 10-12:
  add_some_labeled ~x:10 30;;
            ^^
Expecting function has type y:int -> int
This argument cannot be applied without label
# add_some_labeled ~x:10 ~y:10;;
- : int = 20
#
```

You can call the function with or without labels, but you cannot mix them because the labels are not commutated. However, you can define one as an optional argument and the other as unlabeled, like so:

```
# let increment ?(by = 1) v = v + by;;
val increment : ?by:int -> int -> int = <fun>
# increment 10;;
- : int = 11
# increment ~by:30 10;;
- : int = 40
```

One of the problems with labels is that you cannot call those parameters without labels after they are labeled. There is also a performance penalty for optional arguments, but it is small. If the labels provide greatly improved readability, I recommend their use.

```
# increment 30 10;;
Characters 13-15:
  increment 30 10;;
               ^^
Expecting function has type ?by:int -> int
This argument cannot be applied without label
#
```

After the function has a labeled argument, it must be called using the label. This is one of the reasons why libraries that provide labeled variants often provide them in a separate version of the library.

Conclusion

You can now define a variety of functions and work with them. You also should have a basic understanding of the semantics of values and functions in OCaml. So armed, you can go on to the next chapter, in which you will build a simple database of parts using some of the types covered in the past few chapters.

Functional programming depends on functions. In other languages, developers sometimes feel it is important to limit the number of functions they create. Remember that functions are the building blocks of complex systems in FP. They should be like bricks: small and simple. It is the assembly of those bricks that is important; if you try to limit your bricks, you might not be able to create programs that are as flexible as you want.