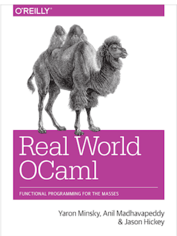

Buy in print and eBook.

Table of Contents

Chapter 2. Variables and Functions

Variables and functions are fundamental ideas that show up in virtually all programming languages. OCaml has a different take on these concepts than most languages you're likely to have encountered, so this chapter will cover OCaml's approach to variables and functions in some detail, starting with the basics of how to define a variable, and ending with the intricacies of functions with labeled and optional arguments.0 comments

Don't be discouraged if you find yourself overwhelmed by some of the details, especially toward the end of the chapter. The concepts here are important, but if they don't connect for you on your first read, you should return to this chapter after you've gotten a better sense for the rest of the language.0 comments

VARIABLES

At its simplest, a variable is an identifier whose meaning is bound to a particular value. In OCaml these bindings are often introduced using the `let` keyword. We can type a so-called *top-level* `let` binding with the following syntax. Note that variable names must start with a lowercase letter or an underscore:0 comments

```
let <variable> = <expr>
```

Syntax ∗ variables-and-functions/let.syntax ∗ all code

As we'll see when we get to the module system in Chapter 4, *Files, Modules, and Programs*, this same syntax is used for `let` bindings at the top level of a module.0 comments

Every variable binding has a *scope*, which is the portion of the code that can refer to that binding. When using **utop**, the scope of a top-level `let` binding is everything that follows it in the session. When it shows up in a module, the scope is the remainder of that module.0 comments

Here's a simple example:0 comments

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

OCaml Utop ∗ variables-and-functions/main.topscript ∗ all code

`let` can also be used to create a variable binding whose scope is limited to a particular expression, using the following syntax:0 comments

```
let <variable> = <expr1> in <expr2>
```

Syntax ∗ variables-and-functions/let_in.syntax ∗ all code

This first evaluates *`expr1`* and then evaluates *`expr2`* with *`variable`* bound to whatever value was produced by the evaluation of *`expr1`*. Here's how it looks in practice:0 comments

```
# let languages = "OCaml,Perl,C++,C";;
 val languages : string = "OCaml,Perl,C++,C"
# let dashed_languages =
    let language_list = String.split languages ~on:',' in
    String.concat ~sep:"-" language_list
  ;;
 val dashed_languages : string = "OCaml-Perl-C++-C"
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 1) ∗ all code

Note that the scope of `language_list` is just the expression `String.concat ~sep:"-" language_list` and is not available at the toplevel, as we can see if we try to access it now:0 comments

```
# language_list;;
 Characters -1-13:
 Error: Unbound value language_list
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 2) ∗ all code

A `let` binding in an inner scope can *shadow*, or hide, the definition from an outer scope. So, for example, we could have written the `dashed_languages` example as follows:0 comments

```
# let languages = "OCaml,Perl,C++,C";;
 val languages : string = "OCaml,Perl,C++,C"
# let dashed_languages =
     let languages = String.split languages ~on:',' in
     String.concat ~sep:"-" languages
  ;;
 val dashed_languages : string = "OCaml-Perl-C++-C"
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 3) ∗ all code

This time, in the inner scope we called the list of strings `languages` instead of `language_list`, thus hiding the original definition of `languages`. But once the definition of `dashed_languages` is complete, the inner scope has closed and the original definition of languages reappears:0 comments

```
# languages;;
 - : string = "OCaml,Perl,C++,C"
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 4) ∗ all code

One common idiom is to use a series of nested `let`/`in` expressions to build up the components of a larger computation. Thus, we might write:0 comments

```
# let area_of_ring inner_radius outer_radius =
     let pi = acos (-1.) in
     let area_of_circle r = pi *. r *. r in
     area_of_circle outer_radius -. area_of_circle inner_radius
  ;;
 val area_of_ring : float -> float -> float = <fun>
# area_of_ring 1. 3.;;
 - : float = 25.1327412287
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 5) ∗ all code

It's important not to confuse a sequence of `let` bindings with the modification of a mutable variable. For example, consider how `area_of_ring` would work if we had instead written this purposefully confusing bit of code:0 comments

```
# let area_of_ring inner_radius outer_radius =
     let pi = acos (-1.) in
     let area_of_circle r = pi *. r *. r in
     let pi = 0. in
     area_of_circle outer_radius -. area_of_circle inner_radius
  ;;


 Characters 126-128:
 Warning 26: unused variable pi.val area_of_ring : float -> float -> float = <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 6) ∗ all code

Here, we redefined `pi` to be zero after the definition of `area_of_circle`. You might think that this would mean that the result of the computation would now be zero, but in fact, the behavior of the function is unchanged. That's because the original definition of `pi` wasn't changed; it was just shadowed, which means that any subsequent reference to `pi` would see the new definition of `pi` as `0`, but earlier references would be unchanged. But there is no later use of `pi`, so the binding of `pi` to `0.` made no difference. This explains the warning produced by the toplevel telling us that there is an unused definition of `pi`.0 comments

In OCaml, `let` bindings are immutable. There are many kinds of mutable values in OCaml, which we'll discuss in Chapter 8, *Imperative Programming*, but there are no mutable variables.0 comments

Why Don't Variables Vary?

One source of confusion for people new to OCaml is the fact that variables are immutable. This seems pretty surprising even on linguistic terms. Isn't the whole point of a variable that it can vary?0 comments

The answer to this is that variables in OCaml (and generally in functional languages) are really more like variables in an equation than a variable in an imperative language. If you think about the mathematical identity `x(y + z) = xy + xz`, there's no notion of mutating the variables `x`, `y`, and `z`. They vary in the sense that you can instantiate this equation with different numbers for those variables, and it still holds.0 comments

The same is true in a functional language. A function can be applied to different inputs, and thus its variables will take on different values, even without mutation.0 comments

Pattern Matching and let

Another useful feature of `let` bindings is that they support the use of *patterns* on the lefthand side. Consider the following code, which uses `List.unzip`, a function for converting a list of pairs into a pair of lists:0 comments

```
# let (ints,strings) = List.unzip [(1,"one"); (2,"two"); (3,"three")];;
 val ints : int list = [1; 2; 3]
 val strings : string list = ["one"; "two"; "three"]
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 7) ∗ all code

Here, `(ints,strings)` is a pattern, and the `let` binding assigns values to both of the identifiers that show up in that pattern. A pattern is essentially a description of the shape of a data structure, where some components are identifiers to be bound. As we saw in the section called "Tuples, Lists, Options, and Pattern Matching", OCaml has patterns for a variety of different data types.0 comments

Using a pattern in a `let` binding makes the most sense for a pattern that is *irrefutable, i.e.*, where any value of the type in question is guaranteed to match the pattern. Tuple and record patterns are irrefutable, but list patterns are not. Consider the following code that implements a function for upper casing the first element of a comma-separated list:0 comments

```
# let upcase_first_entry line =
    let (first :: rest) = String.split ~on:',' line in
    String.concat ~sep:"," (String.uppercase first :: rest)
  ;;


 Characters 40-53:
 Warning 8: this pattern-matching is not exhaustive.
 Here is an example of a value that is not matched:
 []val upcase_first_entry : string -> string = <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 8) ∗ all code

This case can't really come up in practice, because `String.split` always returns a list with at least one element. But the compiler doesn't know this, and so it emits the warning. It's generally better to use a `match` statement to handle such cases explicitly:0 comments

```
# let upcase_first_entry line =
    match String.split ~on:',' line with
    | [] -> assert false (* String.split returns at least one element *)
    | first :: rest -> String.concat ~sep:"," (String.uppercase first :: rest)
  ;;
 val upcase_first_entry : string -> string = <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 9) ∗ all code

Note that this is our first use of `assert`, which is useful for marking cases that should be impossible. We'll discuss `assert` in more detail in Chapter 7, *Error Handling*.0 comments

FUNCTIONS

Given that OCaml is a functional language, it's no surprise that functions are important and pervasive. Indeed, functions have come up in almost every example we've done so far. This section will go into more depth, explaining the details of how OCaml's functions work. As you'll see, functions in OCaml differ in a variety of ways from what you'll find in most mainstream languages.0 comments

Anonymous Functions

We'll start by looking at the most basic style of function declaration in OCaml: the *anonymous function*. An anonymous function is a function that is declared without being named. These can be declared using the `fun` keyword, as shown here:0 comments

```
# (fun x -> x + 1);;
 - : int -> int = <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 10) ∗ all code

Anonymous functions operate in much the same way as named functions. For example, we can apply an anonymous function to an argument:0 comments

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

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 11) ∗ all code

Or pass it to another function. Passing functions to iteration functions like `List.map` is probably the most common use case for anonymous functions:0 comments

```
# List.map ~f:(fun x -> x + 1) [1;2;3];;
 - : int list = [2; 3; 4]
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 12) ∗ all code

You can even stuff them into a data structure:0 comments

```
# let increments = [ (fun x -> x + 1); (fun x -> x + 2) ] ;;
 val increments : (int -> int) list = [<fun>; <fun>]
# List.map ~f:(fun g -> g 5) increments;;
 - : int list = [6; 7]
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 13) ∗ all code

It's worth stopping for a moment to puzzle this example out, since this kind of higher-order use of functions can be a bit obscure at first. Notice that `(fun g -> g 5)` is a function that takes a function as an argument, and then applies that function to the number `5`. The invocation of `List.map` applies `(fun g -> g 5)` to the elements of the `increments` list (which are themselves functions) and returns the list containing the results of these function applications.0 comments

The key thing to understand is that functions are ordinary values in OCaml, and you can do everything with them that you'd do with an ordinary value, including passing them to and returning them from other functions and storing them in data structures. We even name functions in the same way that we name other values, by using a `let` binding:0 comments

```
# let plusone = (fun x -> x + 1);;
 val plusone : int -> int = <fun>
# plusone 3;;
 - : int = 4
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 14) ∗ all code

Defining named functions is so common that there is some syntactic sugar for it. Thus, the following definition of `plusone` is equivalent to the previous definition:0 comments

```
# let plusone x = x + 1;;
 val plusone : int -> int = <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 15) ∗ all code

This is the most common and convenient way to declare a function, but syntactic niceties aside, the two styles of function definition are equivalent.0 comments

let and fun

Functions and `let` bindings have a lot to do with each other. In some sense, you can think of the parameter of a function as a variable being bound to the value passed by the caller. Indeed, the following two expressions are nearly equivalent:0 comments

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

```
- : int = 8
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 16) ∗ all code

This connection is important, and will come up more when programming in a monadic style, as we'll see in Chapter 18, *Concurrent Programming with Async*.0 comments

Multiargument functions

OCaml of course also supports multiargument functions, such as:0 comments

```
# let abs_diff x y = abs (x - y);;
 val abs_diff : int -> int -> int = <fun>
# abs_diff 3 4;;
 - : int = 1
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 17) ∗ all code

You may find the type signature of `abs_diff` with all of its arrows a little hard to parse. To understand what's going on, let's rewrite `abs_diff` in an equivalent form, using the `fun` keyword:0 comments

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

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 18) ∗ all code

This rewrite makes it explicit that `abs_diff` is actually a function of one argument that returns another function of one argument, which itself returns the final result. Because the functions are nested, the inner expression `abs (x - y)` has access to both `x`, which was bound by the outer function application, and `y`, which was bound by the inner one.0 comments

This style of function is called a *curried* function. (Currying is named after Haskell Curry, a logician who had a significant impact on the design and theory of programming languages.) The key to interpreting the type signature of a curried function is the observation that `->` is right-associative. The type signature of `abs_diff` can therefore be parenthesized as follows:0 comments

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

OCaml ∗ variables-and-functions/abs_diff.mli ∗ all code

The parentheses don't change the meaning of the signature, but they make it easier to see the currying.0 comments

Currying is more than just a theoretical curiosity. You can make use of currying to specialize a function by feeding in some of the arguments. Here's an example where we create a specialized version of `abs_diff` that measures the distance of a given number from `3`:0 comments

```
# let dist_from_3 = abs_diff 3;;
 val dist_from_3 : int -> int = <fun>
# dist_from_3 8;;
 - : int = 5
# dist_from_3 (-1);;
 - : int = 4
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 19) ∗ all code

The practice of applying some of the arguments of a curried function to get a new function is called *partial application*.0 comments

Note that the `fun` keyword supports its own syntax for currying, so the following definition of `abs_diff` is equivalent to the previous one.0 comments

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

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 20) ∗ all code

You might worry that curried functions are terribly expensive, but this is not the case. In OCaml, there is no penalty for calling a curried function with all of its arguments. (Partial application, unsurprisingly, does have a small extra cost.)0 comments

Currying is not the only way of writing a multiargument function in OCaml. It's also possible to use the different parts of a tuple as different arguments. So, we could write:0 comments

```
# let abs_diff (x,y) = abs (x - y);;
 val abs_diff : int * int -> int = <fun>
# abs_diff (3,4);;
 - : int = 1
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 21) ∗ all code

OCaml handles this calling convention efficiently as well. In particular it does not generally have to allocate a tuple just for the purpose of sending arguments to a tuple-style function. You can't, however, use partial application for this style of function.0 comments

There are small trade-offs between these two approaches, but most of the time, one should stick to currying, since it's the default style in the OCaml world.0 comments

Recursive Functions

A function is *recursive* if it refers to itself in its definition. Recursion is important in any programming language, but is particularly important in functional languages, because it is the way that you build looping constructs. (As will be discussed in more detail in Chapter 8, *Imperative Programming*, OCaml also supports imperative looping constructs like `for` and `while`, but these are only useful when using OCaml's imperative features.)0 comments

In order to define a recursive function, you need to mark the `let` binding as recursive with the `rec` keyword, as shown in this function for finding the first sequentially repeated element in a list:0 comments

```
# let rec find_first_stutter list =
    match list with
    | [] | [_] ->
      (* only zero or one elements, so no repeats *)
      None
    | x :: y :: tl ->
      if x = y then Some x else find_first_stutter (y::tl)
   ;;
 val find_first_stutter : 'a list -> 'a option = <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 22) ∗ all code

Note that in the code, the pattern `| [] | [_]` is what's called an *or-pattern*, which is a disjunction of two patterns, meaning that it will be considered a match if either pattern matches. In this case, `[]` matches the empty list, and `[_]` matches any single element list. The _ is there so we don't have to put an explicit name on that single element.0 comments

We can also define multiple mutually recursive values by using `let rec` combined with the `and` keyword. Here's a (gratuitously inefficient) example:0 comments

```
# let rec is_even x =
    if x = 0 then true else is_odd (x - 1)
  and is_odd x =
    if x = 0 then false else is_even (x - 1)
 ;;
 val is_even : int -> bool = <fun>
 val is_odd : int -> bool = <fun>
# List.map ~f:is_even [0;1;2;3;4;5];;
 - : bool list = [true; false; true; false; true; false]
# List.map ~f:is_odd [0;1;2;3;4;5];;
 - : bool list = [false; true; false; true; false; true]
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 23) ∗ all code

OCaml distinguishes between nonrecursive definitions (using `let`) and recursive definitions (using `let rec`) largely for technical reasons: the type-inference algorithm needs to know when a set of function definitions are mutually recursive, and for reasons that don't apply to a pure language like Haskell, these have to be marked explicitly by the programmer.0 comments

But this decision has some good effects. For one thing, recursive (and especially mutually recursive) definitions are harder to reason about than nonrecursive ones. It's therefore useful that, in the absence of an explicit `rec`, you can assume that a `let` binding is nonrecursive, and so can only build upon previous bindings.0 comments

In addition, having a nonrecursive form makes it easier to create a new definition that extends and supersedes an existing one by shadowing it.0 comments

Prefix and Infix Operators

So far, we've seen examples of functions used in both prefix and infix style:0 comments

```
# Int.max 3 4  (* prefix *);;
 - : int = 4
# 3 + 4        (* infix  *);;
 - : int = 7
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 24) ∗ all code

You might not have thought of the second example as an ordinary function, but it very much is. Infix operators like + really only differ syntactically from other functions. In fact, if we put parentheses around an infix operator, you can use it as an ordinary prefix function:0 comments

```
# (+) 3 4;;
 - : int = 7
# List.map ~f:((+) 3) [4;5;6];;
 - : int list = [7; 8; 9]
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 25) ∗ all code

In the second expression, we've partially applied `(+)` to create a function that increments its single argument by `3`.0 comments

A function is treated syntactically as an operator if the name of that function is chosen from one of a specialized set of identifiers. This set includes identifiers that are sequences of characters from the following set:0 comments

```
! $ % & * + - . / : < = > ? @ ^ | ~
```

Syntax ∗ variables-and-functions/operators.syntax ∗ all code

`or` is one of a handful of predetermined strings, including `mod`, the modulus operator, and `lsl`, for "logical shift left," a bit-shifting operation.0 comments

We can define (or redefine) the meaning of an operator. Here's an example of a simple vector-addition operator on `int` pairs:0 comments

```
# let (+!) (x1,y1) (x2,y2) = (x1 + x2, y1 + y2);;
 val ( +! ) : int * int -> int * int -> int * int = <fun>
# (3,2) +! (-2,4);;
 - : int * int = (1, 6)
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 26) ∗ all code

Note that you have to be careful when dealing with operators containing `*`. Consider the following example:0 comments

```
# let (***) x y = (x ** y) ** y;;
 Characters 17-18:
 Error: This expression has type int but an expression was expected of type
          float
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 27) ∗ all code

What's going on is that `(***)` isn't interpreted as an operator at all; it's read as a comment! To get this to work properly, we need to put spaces around any operator that begins or ends with `*`:0 comments

```
# let ( *** ) x y = (x ** y) ** y;;
 val ( *** ) : float -> float -> float = <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 28) ∗ all code

The syntactic role of an operator is typically determined by its first character or two, though there are a few exceptions. Table 2.1, "Precedence and associativity" breaks the different operators and other syntactic forms into groups from highest to lowest precedence, explaining how each behaves syntactically. We write `!`... to indicate the class of operators beginning with `!`.0 comments

Table 2.1. Precedence and associativity

Operator prefix	Associativity
!..., ?..., ~...	Prefix
., .(, .[	-
function application, constructor, assert, lazy	Left associative
-, -.	Prefix
**..., lsl, lsr, asr	Right associative
*..., /..., %..., mod, land, lor, lxor	Left associative
+..., -...	Left associative
::	Right associative
@..., ^...	Right associative
=..., <..., >..., |..., &..., $...	Left associative
&, &&	Right associative
or, ||	Right associative
,	-
<-, :=	Right associative
if	-
;	Right associative

There's one important special case: - and -., which are the integer and floating-point subtraction operators, and can act as both prefix operators (for negation) and infix operators (for subtraction). So, both -x and x - y are meaningful expressions. Another thing to remember about negation is that it has lower precedence than function application, which means that if you want to pass a negative value, you need to wrap it in parentheses, as you can see in this code:0 comments

```
# Int.max 3 (-4);;
 - : int = 3
# Int.max 3 -4;;
 Characters -1-9:
 Error: This expression has type int -> int
        but an expression was expected of type int
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 29) ∗ all code

Here, OCaml is interpreting the second expression as equivalent to:0 comments

```
# (Int.max 3) - 4;;
 Characters 1-10:
 Error: This expression has type int -> int
        but an expression was expected of type int
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 30) ∗ all code

which obviously doesn't make sense.0 comments

Here's an example of a very useful operator from the standard library whose behavior depends critically on the precedence rules described previously:0 comments

```
# let (|>) x f = f x ;;
 val ( |> ) : 'a -> ('a -> 'b) -> 'b = <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 31) ∗ all code

It's not quite obvious at first what the purpose of this operator is: it just takes a value and a function and applies the function to the value. Despite that bland-sounding description, it has the useful role of a sequencing operator, similar in spirit to using the pipe character in the UNIX shell. Consider, for example, the following code for printing out the unique elements of your PATH. Note that List.dedup that follows removes duplicates from a list by sorting the list using the provided comparison function:0 comments

```
# let path = "/usr/bin:/usr/local/bin:/bin:/sbin";;
 val path : string = "/usr/bin:/usr/local/bin:/bin:/sbin"
#   String.split ~on:':' path
  |> List.dedup ~compare:String.compare
  |> List.iter ~f:print_endline
  ;;


 /bin
 /sbin
 /usr/bin
 /usr/local/bin
 - : unit = ()
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 32) ∗ all code

Note that we can do this without |>, but the result is a bit more verbose:0 comments

```
#   let split_path = String.split ~on:':' path in
  let deduped_path = List.dedup ~compare:String.compare split_path in
  List.iter ~f:print_endline deduped_path
  ;;


 /bin
 /sbin
 /usr/bin
 /usr/local/bin
 - : unit = ()
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 33) ∗ all code

An important part of what's happening here is partial application. For example, `List.iter` normally takes two arguments: a function to be called on each element of the list, and the list to iterate over. We can call `List.iter` with all its arguments:0 comments

```
# List.iter ~f:print_endline ["Two"; "lines"];;


 Two
 lines
 - : unit = ()
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 34) ∗ all code

Or, we can pass it just the function argument, leaving us with a function for printing out a list of strings:0 comments

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

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 35) ∗ all code

It is this later form that we're using in the preceding |> pipeline.0 comments

But |> only works in the intended way because it is left-associative. Let's see what happens if we try using a right-associative operator, like (^>):0 comments

```
# let (^>) x f = f x;;
 val ( ^> ) : 'a -> ('a -> 'b) -> 'b = <fun>
# Sys.getenv_exn "PATH"
  ^> String.split ~on:':' path
  ^> List.dedup ~compare:String.compare
  ^> List.iter ~f:print_endline
  ;;
 Characters 98-124:
 Error: This expression has type string list -> unit
        but an expression was expected of type
          (string list -> string list) -> 'a
        Type string list is not compatible with type
          string list -> string list
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 36) ∗ all code

The type error is a little bewildering at first glance. What's going on is that, because `^>` is right associative, the operator is trying to feed the value `List.dedup ~compare:String.compare` to the function `List.iter ~f:print_endline`. But `List.iter ~f:print_endline` expects a list of strings as its input, not a function.0 comments

The type error aside, this example highlights the importance of choosing the operator you use with care, particularly with respect to associativity.0 comments

Declaring Functions with Function

Another way to define a function is using the `function` keyword. Instead of having syntactic support for declaring multiargument (curried) functions, `function` has built-in pattern matching. Here's an example:0 comments

```
# let some_or_zero = function
```

```
      | Some x -> x
      | None -> 0
  ;;
 val some_or_zero : int option -> int = <fun>
# List.map ~f:some_or_zero [Some 3; None; Some 4];;
 - : int list = [3; 0; 4]
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 37) ∗ all code

This is equivalent to combining an ordinary function definition with a `match`:0 comments

```
# let some_or_zero num_opt =
    match num_opt with
    | Some x -> x
    | None -> 0
  ;;
 val some_or_zero : int option -> int = <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 38) ∗ all code

We can also combine the different styles of function declaration together, as in the following example, where we declare a two-argument (curried) function with a pattern match on the second argument:0 comments

```
# let some_or_default default = function
      | Some x -> x
      | None -> default
  ;;
 val some_or_default : 'a -> 'a option -> 'a = <fun>
# some_or_default 3 (Some 5);;
 - : int = 5
# List.map ~f:(some_or_default 100) [Some 3; None; Some 4];;
 - : int list = [3; 100; 4]
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 39) ∗ all code

Also, note the use of partial application to generate the function passed to `List.map`. In other words, `some_or_default 100` is a function that was created by feeding just the first argument to `some_or_default`.0 comments

Labeled Arguments

Up until now, the functions we've defined have specified their arguments positionally, *i.e.*, by the order in which the arguments are passed to the function. OCaml also supports labeled arguments, which let you identify a function argument by name. Indeed, we've already encountered functions from Core like `List.map` that use labeled arguments. Labeled arguments are marked by a leading tilde, and a label (followed by a colon) is put in front of the variable to be labeled. Here's an example:0 comments

```
# let ratio ~num ~denom = float num /. float denom;;
 val ratio : num:int -> denom:int -> float = <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 40) ∗ all code

We can then provide a labeled argument using a similar convention. As you can see, the arguments can be provided in any order:0 comments

```
# ratio ~num:3 ~denom:10;;
 - : float = 0.3
# ratio ~denom:10 ~num:3;;
 - : float = 0.3
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 41) ∗ all code

OCaml also supports *label punning*, meaning that you get to drop the text after the `:` if the name of the label and the name of the variable being used are the same. We were actually already using label punning when defining `ratio`. The following shows how punning can be used when invoking a function:0 comments

```
# let num = 3 in
let denom = 4 in
ratio ~num ~denom;;
 - : float = 0.75
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 42) ∗ all code

Labeled arguments are useful in a few different cases:0 comments

- When defining a function with lots of arguments. Beyond a certain number, arguments are easier to remember by name than by position.0 comments

- When the meaning of a particular argument is unclear from the type alone. Consider a function for creating a hash table whose first argument is the initial size of the array backing the hash table, and the second is a Boolean flag, which indicates whether that array will ever shrink when elements are removed:0 comments

  ```
  val create_hashtable : int -> bool -> ('a,'b) Hashtable.t
  ```
  OCaml * variables-and-functions/htable_sig1.ml * all code

  The signature makes it hard to divine the meaning of those two arguments. but with labeled arguments, we can make the intent immediately clear:0 comments

  ```
  val create_hashtable :
    init_size:int -> allow_shrinking:bool -> ('a,'b) Hashtable.t
  ```
  OCaml * variables-and-functions/htable_sig2.ml * all code

  Choosing label names well is especially important for Boolean values, since it's often easy to get confused about whether a value being true is meant to enable or disable a given feature.0 comments

- When defining functions that have multiple arguments that might get confused with each other. This is most at issue when the arguments are of the same type. For example, consider this signature for a function that extracts a substring:0 comments

  ```
  val substring: string -> int -> int -> string
  ```
  OCaml * variables-and-functions/substring_sig1.ml * all code

  Here, the two `ints` are the starting position and length of the substring to extract, respectively. We can make this fact more obvious from the signature by adding labeled:0 comments

  ```
  val substring: string -> pos:int -> len:int -> string
  ```
  OCaml * variables-and-functions/substring_sig2.ml * all code

  This improves the readability of both the signature and of client code that makes use of `substring` and makes it harder to accidentally swap the position and the length.0 comments

- When you want flexibility on the order in which arguments are passed. Consider a function like `List.iter`, which takes two arguments: a function and a list of elements to call that function on. A common pattern is to partially apply `List.iter` by giving it just the function, as in the following example from earlier in the chapter:0 comments

  ```
  #   String.split ~on:':' path
    |> List.dedup ~compare:String.compare
    |> List.iter ~f:print_endline
    ;;


   /bin
   /sbin
   /usr/bin
   /usr/local/bin
   - : unit = ()
  ```
  OCaml Utop * variables-and-functions/main.topscript , continued (part 43) * all code

  This requires that we put the function argument first. In other cases, you want to put the function argument second. One common reason is readability. In particular, a multiline function passed as an argument to another function is easiest to read when it is the final argument to that function.0 comments

Higher-order functions and labels

One surprising gotcha with labeled arguments is that while order doesn't matter when calling a function with labeled arguments, it does matter in a higher-order context, *e.g.*, when passing a function with labeled arguments to another function. Here's an example:0 comments

```
# let apply_to_tuple f (first,second) = f ~first ~second;;
```

```
val apply_to_tuple : (first:'a -> second:'b -> 'c) -> 'a * 'b -> 'c = <fun>
```
OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 44) ∗ all code

Here, the definition of `apply_to_tuple` sets up the expectation that its first argument is a function with two labeled arguments, `first` and `second`, listed in that order. We could have defined `apply_to_tuple` differently to change the order in which the labeled arguments were listed:0 comments

```
# let apply_to_tuple_2 f (first,second) = f ~second ~first;;
val apply_to_tuple_2 : (second:'a -> first:'b -> 'c) -> 'b * 'a -> 'c = <fun>
```
OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 45) ∗ all code

It turns out this order matters. In particular, if we define a function that has a different order0 comments

```
# let divide ~first ~second = first / second;;
val divide : first:int -> second:int -> int = <fun>
```
OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 46) ∗ all code

we'll find that it can't be passed in to `apply_to_tuple_2`.0 comments

```
# apply_to_tuple_2 divide (3,4);;
Characters 17-23:
Error: This expression has type first:int -> second:int -> int
       but an expression was expected of type second:'a -> first:'b -> 'c
```
OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 47) ∗ all code

But, it works smoothly with the original `apply_to_tuple`:0 comments

```
# let apply_to_tuple f (first,second) = f ~first ~second;;
val apply_to_tuple : (first:'a -> second:'b -> 'c) -> 'a * 'b -> 'c = <fun>
# apply_to_tuple divide (3,4);;
- : int = 0
```
OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 48) ∗ all code

As a result, when passing labeled functions as arguments, you need to take care to be consistent in your ordering of labeled arguments.0 comments

Optional Arguments

An optional argument is like a labeled argument that the caller can choose whether or not to provide. Optional arguments are passed in using the same syntax as labeled arguments, and, like labeled arguments, can be provided in any order.0 comments

Here's an example of a string concatenation function with an optional separator. This function uses the `^` operator for pairwise string concatenation:0 comments

```
# let concat ?sep x y =
    let sep = match sep with None -> "" | Some x -> x in
    x ^ sep ^ y
 ;;
val concat : ?sep:string -> string -> string -> string = <fun>
# concat "foo" "bar"             (* without the optional argument *);;
- : string = "foobar"
# concat ~sep:":" "foo" "bar"    (* with the optional argument    *);;
- : string = "foo:bar"
```
OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 49) ∗ all code

Here, `?` is used in the definition of the function to mark `sep` as optional. And while the caller can pass a value of type `string` for `sep`, internally to the function, `sep` is seen as a `string option`, with `None` appearing when `sep` is not provided by the caller.0 comments

The preceding example needed a bit of boilerplate to choose a default separator when none was provided. This is a common enough pattern that there's an explicit syntax for providing a default value, which allows us to write `concat` more concisely:0 comments

```
# let concat ?(sep="") x y = x ^ sep ^ y ;;
val concat : ?sep:string -> string -> string -> string = <fun>
```
OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 50) ∗ all code

Optional arguments are very useful, but they're also easy to abuse. The key advantage of optional arguments is that they let you write functions with multiple arguments that users can ignore most of the time, only worrying about them when they specifically want to invoke those options. They also allow you to extend an API with new functionality without changing existing code.0 comments

The downside is that the caller may be unaware that there is a choice to be made, and so may unknowingly (and wrongly) pick the default behavior. Optional arguments really only make sense when the extra concision of omitting the argument outweighs the corresponding loss of explicitness.0 comments

This means that rarely used functions should not have optional arguments. A good rule of thumb is to avoid optional arguments for functions internal to a module, *i.e.*, functions that are not included in the module's interface, or `mli` file. We'll learn more about `mli`s in Chapter 4, *Files, Modules, and Programs*.0 comments

Explicit passing of an optional argument

Under the covers, a function with an optional argument receives `None` when the caller doesn't provide the argument, and `Some` when it does. But the `Some` and `None` are normally not explicitly passed in by the caller.0 comments

But sometimes, passing in `Some` or `None` explicitly is exactly what you want. OCaml lets you do this by using `?` instead of `~` to mark the argument. Thus, the following two lines are equivalent ways of specifying the `sep` argument to `concat`:0 comments

```
# concat ~sep:":" "foo" "bar" (* provide the optional argument *);;
 - : string = "foo:bar"
# concat ?sep:(Some ":") "foo" "bar" (* pass an explicit [Some] *);;
 - : string = "foo:bar"
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 51) ∗ all code

And the following two lines are equivalent ways of calling `concat` without specifying `sep`:0 comments

```
# concat "foo" "bar" (* don't provide the optional argument *);;
 - : string = "foobar"
# concat ?sep:None "foo" "bar" (* explicitly pass `None` *);;
 - : string = "foobar"
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 52) ∗ all code

One use case for this is when you want to define a wrapper function that mimics the optional arguments of the function it's wrapping. For example, imagine we wanted to create a function called `uppercase_concat`, which is the same as `concat` except that it converts the first string that it's passed to uppercase. We could write the function as follows:0 comments

```
# let uppercase_concat ?(sep="") a b = concat ~sep (String.uppercase a) b ;;
 val uppercase_concat : ?sep:string -> string -> string -> string = <fun>
# uppercase_concat "foo" "bar";;
 - : string = "FOObar"
# uppercase_concat "foo" "bar" ~sep:":";;
 - : string = "FOO:bar"
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 53) ∗ all code

In the way we've written it, we've been forced to separately make the decision as to what the default separator is. Thus, if we later change `concat`'s default behavior, we'll need to remember to change `uppercase_concat` to match it.0 comments

Instead, we can have `uppercase_concat` simply pass through the optional argument to `concat` using the `?` syntax:0 comments

```
# let uppercase_concat ?sep a b = concat ?sep (String.uppercase a) b ;;
 val uppercase_concat : ?sep:string -> string -> string -> string = <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 54) ∗ all code

Now, if someone calls `uppercase_concat` without an argument, an explicit `None` will be passed to `concat`, leaving `concat` to decide what the default behavior should be.0 comments

Inference of labeled and optional arguments

One subtle aspect of labeled and optional arguments is how they are inferred by the type system. Consider the following example for computing numerical derivatives of a function of two real variables. The function takes an argument `delta`, which determines the scale at which to compute the derivative; values `x` and `y`, which determine at which point to compute the derivative; and the function `f`, whose derivative is being computed. The function `f` itself takes two labeled arguments, `x` and `y`. Note that you can use an apostrophe as part of a variable name, so `x'` and `y'` are just ordinary variables:0 comments

```
# let numeric_deriv ~delta ~x ~y ~f =
    let x' = x +. delta in
    let y' = y +. delta in
    let base = f ~x ~y in
    let dx = (f ~x:x' ~y -. base) /. delta in
    let dy = (f ~x ~y:y' -. base) /. delta in
    (dx,dy)
  ;;
 val numeric_deriv :
   delta:float ->
   x:float -> y:float -> f:(x:float -> y:float -> float) -> float * float =
   <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 55) ∗ all code

In principle, it's not obvious how the order of the arguments to `f` should be chosen. Since labeled arguments can be passed in arbitrary order, it seems like it could as well be `y:float -> x:float -> float` as it is `x:float -> y:float -> float`.0 comments

Even worse, it would be perfectly consistent for `f` to take an optional argument instead of a labeled one, which could lead to this type signature for `numeric_deriv`:0 comments

```
val numeric_deriv :
  delta:float ->
  x:float -> y:float -> f:(?x:float -> y:float -> float) -> float * float
```

OCaml ∗ variables-and-functions/numerical_deriv_alt_sig.mli ∗ all code

Since there are multiple plausible types to choose from, OCaml needs some heuristic for choosing between them. The heuristic the compiler uses is to prefer labels to options and to choose the order of arguments that shows up in the source code.0 comments

Note that these heuristics might at different points in the source suggest different types. Here's a version of `numeric_deriv` where different invocations of `f` list the arguments in different orders:0 comments

```
# let numeric_deriv ~delta ~x ~y ~f =
    let x' = x +. delta in
    let y' = y +. delta in
    let base = f ~x ~y in
    let dx = (f ~y ~x:x' -. base) /. delta in
    let dy = (f ~x ~y:y' -. base) /. delta in
    (dx,dy)
  ;;
 Characters 130-131:
 Error: This function is applied to arguments
 in an order different from other calls.
 This is only allowed when the real type is known.
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 56) ∗ all code

As suggested by the error message, we can get OCaml to accept the fact that `f` is used with different argument orders if we provide explicit type information. Thus, the following code compiles without error, due to the type annotation on `f`:0 comments

```
# let numeric_deriv ~delta ~x ~y ~(f: x:float -> y:float -> float) =
    let x' = x +. delta in
    let y' = y +. delta in
    let base = f ~x ~y in
    let dx = (f ~y ~x:x' -. base) /. delta in
    let dy = (f ~x ~y:y' -. base) /. delta in
    (dx,dy)
  ;;
 val numeric_deriv :
   delta:float ->
   x:float -> y:float -> f:(x:float -> y:float -> float) -> float * float =
   <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 57) ∗ all code

Optional arguments and partial application

Optional arguments can be tricky to think about in the presence of partial application. We can of course partially apply the optional argument itself:0 comments

```
# let colon_concat = concat ~sep:":";;
 val colon_concat : string -> string -> string = <fun>
# colon_concat "a" "b";;
 - : string = "a:b"
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 58) ∗ all code

But what happens if we partially apply just the first argument?0 comments

```
# let prepend_pound = concat "# ";;
 val prepend_pound : string -> string = <fun>
# prepend_pound "a BASH comment";;
 - : string = "# a BASH comment"
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 59) ∗ all code

The optional argument `?sep` has now disappeared, or been *erased*. Indeed, if we try to pass in that optional argument now, it will be rejected:0 comments

```
# prepend_pound "a BASH comment" ~sep:":";;
 Characters -1-13:
 Error: This function has type string -> string
        It is applied to too many arguments; maybe you forgot a `;'.
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 60) ∗ all code

So when does OCaml decide to erase an optional argument?0 comments

The rule is: an optional argument is erased as soon as the first positional (i.e., neither labeled nor optional) argument defined *after* the optional argument is passed in. That explains the behavior of `prepend_pound`. But if we had instead defined `concat` with the optional argument in the second position:0 comments

```
# let concat x ?(sep="") y = x ^ sep ^ y ;;
 val concat : string -> ?sep:string -> string -> string = <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 61) ∗ all code

then application of the first argument would not cause the optional argument to be erased.0 comments

```
# let prepend_pound = concat "# ";;
 val prepend_pound : ?sep:string -> string -> string = <fun>
# prepend_pound "a BASH comment";;
 - : string = "# a BASH comment"
# prepend_pound "a BASH comment" ~sep:"--- ";;
 - : string = "# --- a BASH comment"
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 62) ∗ all code

However, if all arguments to a function are presented at once, then erasure of optional arguments isn't applied until all of the arguments are passed in. This preserves our ability to pass in optional arguments anywhere on the argument list. Thus, we can write:0 comments

```
# concat "a" "b" ~sep:"=";;
 - : string = "a=b"
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 63) ∗ all code

An optional argument that doesn't have any following positional arguments can't be erased at all, which leads to a compiler warning:0 comments

```
# let concat x y ?(sep="") = x ^ sep ^ y ;;


 Characters 15-38:
 Warning 16: this optional argument cannot be erased.val concat : string -> string -
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 64) ∗ all code

And indeed, when we provide the two positional arguments, the `sep` argument is not erased, instead returning a function that expects the `sep` argument to be provided:0 comments

```
# concat "a" "b";;
 - : ?sep:string -> string = <fun>
```

OCaml Utop ∗ variables-and-functions/main.topscript , continued (part 65) ∗ all code

As you can see, OCaml's support for labeled and optional arguments is not without its complexities. But don't let these complexities obscure the usefulness of these features. Labels and optional arguments are very effective tools for making your APIs both more convenient and safer, and it's worth the effort of learning how to use them effectively.0 comments

< Previous **Next >**