5
Naming and scope

In this chapter, we introduce the ability to give names to values, an ability with multiple benefits.

5.1 Variables are names for values

We introduced the concept of a variable in Chapter 1 as seen in the imperative programming paradigm – the variable as a locus of mutable state, which takes on different values over time. But in the functional paradigm, variables are better thought of simply as *names* for values. To introduce a name for a value for use in some other expression, OCaml provides the local naming expression, introduced by the keyword `let`:

⟨*expr*⟩ ::= `let` ⟨*var*⟩ `:` ⟨*type*⟩ `=` ⟨$expr_{def}$⟩ `in` ⟨$expr_{body}$⟩

In this construct, ⟨*var*⟩ is a variable,[1] which will be the name of a value of the given ⟨*type*⟩; ⟨$expr_{def}$⟩ is an expression defining a value of the given ⟨*type*⟩; and ⟨$expr_{body}$⟩ is an expression within which the variable can be used as the name for the defined value. The expression as a whole specifies whatever the ⟨$expr_{body}$⟩ evaluates to. We say that the construction BINDS the name ⟨*var*⟩ to the value ⟨$expr_{def}$⟩ for use in ⟨$expr_{body}$⟩.[2] For this reason, the `let` expression is referred to as a BINDING CONSTRUCT. We'll introduce other binding constructs in Chapters 6 and 7.

As an example,[3] we might provide a name for the important constant π in the context of calculating the area of a circle of radius 2:

```
# let pi : float = 3.1416 in
# pi *. 2. *. 2. ;;
- : float = 12.5664
```

Informally speaking (and we'll provide a more rigorous description in Chapter 13), the construct operates as follows: The ⟨$expr_{def}$⟩ expression

[1] Variables in OCaml are required to be sequences of alphabetic and numeric characters along with the underscore character (_) and the prime character (’). The first character in the variable name must be alphabetic or an underscore. The special role of the latter case is discussed later in Section 7.2.

[2] The name being defined is sometimes referred to as the DEFINIENDUM, the expression it names being the DEFINIENS.

[3] In these examples, we follow the stylistic guidelines described in Section C.1.7 in indenting the body of a `let` to the same level as the `let` keyword itself. The rationale is provided there.

is evaluated to a value, and then the $\langle expr_{body} \rangle$ is evaluated, but as if occurrences of the definiendum $\langle var \rangle$ were first replaced by the value of the definiens $\langle expr_{def} \rangle$.

Notice how by naming the value `pi`, we document our intention that the value serves as the mathematical constant, π, consistent with the edict of intention.

5.2 *The type of a `let`-bound variable can be inferred*

It may seem obvious to you that in an expression like

```
let pi : float = 3.1416 in
pi *. 2. *. 2. ;;
```

the variable `pi` is of type `float`. What else could it be, given that its value is a `float` literal, and it is used as an argument of the `*.` operator, which takes `float` arguments? You would be right, and OCaml itself can make this determination, inferring the type of `pi` without the explicit typing being present. For that reason, the type information in the `let` construct is optional. We can simply write

```
let pi = 3.1416 in
pi *. 2. *. 2. ;;
```

and the calculation proceeds as usual. This ability to infer types is what we mean when we say (as in Section 4.2.1) that OCaml is implicitly typed.

Although these typings when introducing variables are optional, nonetheless, it can still be useful to provide explicit type information when naming a value. First, (and again following the edict of intention), it allows the programmer to make clear the intended types, so that the OCaml interpreter can verify that the programmer's intention was followed and so that readers of the code are aware of that intention. Second, there are certain (relatively rare) cases (Section 9.6) in which OCaml cannot infer a type for an expression in context; in such cases, the explicit typing is necessary.

5.3 *`let` expressions are expressions*

Remember that all expressions in OCaml have values, even `let` expressions. Thus we can use them as subexpressions of larger expressions.

```
# 3.1416 *. (let radius = 2.
#            in radius *. radius) ;;
- : float = 12.5664
```

Exercise 14

Are the parentheses necessary in this example? Try out the expression without the parentheses and see what happens.

A particularly useful application of the fact that `let` expressions can be used as first-class values is that they may be embedded in other `let` expressions to get the effect of defining multiple names. Here, we define both the constant π and a radius to calculate the area of a circle of radius 4:

```
# let pi = 3.1416 in
# let radius = 4. in
# pi *. radius ** 2. ;;
- : float = 50.2656
```

Exercise 15

Use the `let` construct to improve the readability of the following code to calculate the length of the hypotenuse of a particular right triangle:

```
# sqrt (1.88496 *. 1.88496 +. 2.51328 *. 2.51328) ;;
- : float = 3.1416
```

5.4 *Naming to avoid duplication*

We introduce an extended example to more crisply demonstrate the advantages of naming. Suppose we wanted to determine the area of the larger of the two triangles in Figure 5.1.

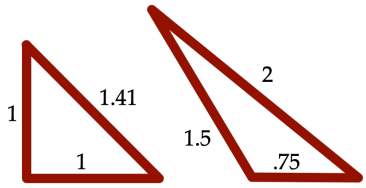

Figure 5.1: Two triangles, the left with sides of length 1, 1, and 1.41, and the right with sides of length 1.75, .75, and 2. Which has the larger area?

To demonstrate some of the advantages of naming, we attempt to calculate the area of the larger without recourse to the `let` construct. To calculate the areas, we'll use a method attributed to Heron of Alexandria around 60 CE.

Calculating the area of the larger triangle without defining local names is possible, but ungainly:

```
# if sqrt ( ((1. +. 1. +. 1.41) /. 2.)
#           *. ((1. +. 1. +. 1.41) /. 2. -. 1.)
#           *. ((1. +. 1. +. 1.41) /. 2. -. 1.)
#           *. ((1. +. 1. +. 1.41) /. 2. -. 1.41) )
#    > sqrt ( ((1.5 +. 0.75 +. 2.) /. 2.)
#             *. ((1.5 +. 0.75 +. 2.) /. 2. -. 1.5)
#             *. ((1.5 +. 0.75 +. 2.) /. 2. -. 0.55)
#             *. ((1.5 +. 0.75 +. 2.) /. 2. -. 2.) )
# then
#   sqrt ( ((1. +. 1. +. 1.41) /. 2.)
#          *. ((1. +. 1. +. 1.41) /. 2. -. 1.)
#          *. ((1. +. 1. +. 1.41) /. 2. -. 1.)
#          *. ((1. +. 1. +. 1.41) /. 2. -. 1.41) )
# else
#   sqrt ( ((1.5 +. 0.75 +. 2.) /. 2.)
#          *. ((1.5 +. 0.75 +. 2.) /. 2. -. 1.5)
#          *. ((1.5 +. 0.75 +. 2.) /. 2. -. 0.75)
#          *. ((1.5 +. 0.75 +. 2.) /. 2. -. 2.) ) ;;
- : float = 0.477777651606895504
```

It's extraordinarily difficult to tell what's going on in this code. Certainly, the various side lengths appear repeatedly, and in fact, calculations making use of them repeat as well. Lines 1–4 and 10–13 both separately calculate the area of the left triangle in the figure, and lines 5–8 and 15–18 calculate the area of the right triangle. The calculations are redundant, and worse, provide the opportunity for bugs to creep in if the copies aren't kept in perfect synchrony.

Appropriate use of naming can partially remedy these problems. (We'll address their solution more systematically in Chapters 6 and 8.) First, by naming the two area calculations, we need calculate each only once.

```
# let left_area = sqrt ( ((1. +. 1. +. 1.41) /. 2.)
#                        *. ((1. +. 1. +. 1.41) /. 2. -. 1.)
#                        *. ((1. +. 1. +. 1.41) /. 2. -. 1.)
#                        *. ((1. +. 1. +. 1.41) /. 2. -. 1.41) ) in
# let right_area = sqrt ( ((1.5 +. 0.75 +. 2.) /. 2.)
#                          *. ((1.5 +. 0.75 +. 2.) /. 2. -. 1.5)
#                          *. ((1.5 +. 0.75 +. 2.) /. 2. -. 0.75)
#                          *. ((1.5 +. 0.75 +. 2.) /. 2. -. 2.) ) in
# if left_area > right_area then left_area else right_area ;;
- : float = 0.499991149296665216
```

We also correct a bug in line 7, which you may not have noticed, that uses inconsistent values for one of the side lengths in the area calculations. By defining the area once and using the value twice, we remove the possibility for such inconsistencies to even arise.

Finally, notice the repeated calculation of, for instance, `(1. +. 1. +. 1.41) /. 2.`, which is calculated some four times, and similarly for `(1.5 +. 0.75 +. 2.) /. 2.`. Each of these is the SEMIPERIMETER of a triangle (that is, half the perimeter). The semiperimeter features heavily in Heron's method of calculating triangle areas. By naming these two subexpressions, we clarify even further what is going on in the example.

```
# let left_area =
#   let left_sp = (1. +. 1. +. 1.41) /. 2. in
#   sqrt ( left_sp
#          *. (left_sp -. 1.)
#          *. (left_sp -. 1.)
#          *. (left_sp -. 1.41) ) in
# let right_area =
#   let right_sp = (1.5 +. 0.75 +. 2.) /. 2. in
#   sqrt ( right_sp
#          *. (right_sp -. 1.5)
#          *. (right_sp -. 0.75)
#          *. (right_sp -. 2.) ) in
# if left_area > right_area then left_area else right_area ;;
- : float = 0.499991149296665216
```

There's still much room for improvement, but to make further

progress on this example awaits additional techniques beyond naming, as described in Section 6.5.

5.5 Scope

The name defined in the `let` expression is available only in the body of the expression. The name is LOCAL to the body, and unavailable outside of the body. We say that the SCOPE of the variable – that is, the code region within which the variable is available as a name of the defined value – is the body of the `let` expression. This explains the following behavior:

```
# (let s = "hi ho " in
#  s ^ s) ^ s ;;
Line 2, characters 9-10:
2 | s ^ s) ^ s ;;
             ^
Error: Unbound value s
```

The body of the `let` expression in this example ends at the closing parenthesis, and thus the variable `s` defined by that construct is unavailable ("unbound") thereafter.

Exercise 16

Correct the example to provide the triple concatenation of the defined string.

Exercise 17

What type do you expect is inferred for `s` in the example?

In particular, the scope of a local `let` naming does not include the definition itself (the $\langle expr_{def} \rangle$ part between the `=` and the `in`). Thus the following expression is ill-formed:

```
# let x = x + 1 in
# x * 2 ;;
Line 1, characters 8-9:
1 | let x = x + 1 in
            ^
Error: Unbound value x
```

And a good thing too, for what would such an expression mean? This kind of recursive definition isn't well founded. Nonetheless, there *are* useful recursive definitions, as we will see in Section 6.6.

What if we define the same name twice? There are several cases to consider. Perhaps the two uses are disjoint, as in this example:

```
# sqrt ((let x = 3. in x *. x)
#       +. (let x = 4. in x *. x)) ;;
- : float = 5.
```

Since each `x` is introduced with its own `let` and has its own body, the scopes are disjoint. The occurrences of `x` in the first expression name

the number `3.` and in the second name the number `4.`. But in the following case, the scopes are not disjoint:

```
# sqrt (let x = 3. in
#       x *. x +. (let x = 4. in x *. x)) ;;
- : float = 5.
```

The scope of the first `let` encompasses the entire second `let`. Do the highlighted occurrences of x in the body of the second `let` name `3.` or `4.`? The rule used in OCaml (and most modern languages) is that the occurrences are bound by the *nearest enclosing binding construct for the variable*. The same binding relations hold as if the inner `let`-bound variable x and the occurrences of x in its body were uniformly renamed, for instance, as y:

```
# sqrt (let x = 3. in
#       x *. x +. (let y = 4. in y *. y)) ;;
- : float = 5.
```

By virtue of this convention that variables are bound by the closest binder, when an inner binder for a variable falls within the scope of an outer binder for the same variable, the outer variable is inaccessible in the inner scope. We say that the outer variable is SHADOWED by the inner variable. For instance, in

```
# let x = 1 in
# x + let x = 2 in
#       x + let x = 4 in
#             x ;;
- : int = 7
```

the innermost x (naming 4) shadows the outer two, and the middle x (naming 2) shadows the outer x (naming 1). Thus the three highlighted occurrences of x name 1, 2, and 4, respectively, which the expression as a whole sums to 7.

Since the scope of a `let`-bound variable is the body of the construct, but not the definition, occurrences of the same variable in the definition must be bound outside of the `let`. Consider the highlighted occurrence of x on the second line:

```
let x = 3 in
let x = x * 2 in
x + 1 ;;
```

This occurrence is bound by the `let` in line 1, not the one in line 2. That is, it is equivalent to the renaming

```
let x = 3 in
let y = x * 2 in
y + 1 ;;
```

Exercise 18

For each occurrence of the variable x in the following examples, which `let` construct binds it? Rewrite the expressions by renaming the variables to make them distinct while preserving the bindings.

1.
```
let x = 3 in
let x = 4 in
x * x ;;
```

2.
```
let x = 3 in
let x = x + 2 in
x * x ;;
```

3.
```
let x = 3 in
let x = 4 + (let x = 5 in x) + x in
x * x ;;
```

5.6 *Global naming and top-level* `let`

The `let` construct introduced above introduces a local name, local in the sense that its scope is just the body of the `let`. OCaml provides a global naming construct as well, defined by this BNF rule:[4]

⟨*definition*⟩ ::= `let` ⟨*var*⟩ `:` ⟨*type*⟩ `=` ⟨$expr_{def}$⟩

By simply leaving off the '`in` ⟨$expr_{body}$⟩' part of the `let` construct, the name can continue to be used thereafter; the scope of the naming extends all the way through the remainder of the REPL session or to the end of the program file.

```
# let pi = 3.1416 ;;
val pi : float = 3.1416
# let radius = 4.0 ;;
val radius : float = 4.
# pi *. radius *. radius ;;
- : float = 50.2656
# 2. *. pi *. radius ;;
- : float = 25.1328
```

The REPL indicates that new names have been introduced by presenting typings for the names (`pi : float` or `radius : float`) as well as displaying their values.

This global naming may look a bit like assignment in imperative languages. We can have, for instance,

```
# let x = 3 ;;
val x : int = 3
# let x = x + 1 ;;
val x : int = 4
# x + x ;;
- : int = 8
```

The second line may look like it is assigning a new value to x. But no, all that is happening is that there is a new name (coincidentally the same as a previous name) for a new value. The old name x for the value

[4] Unlike the local naming construct, the global naming construct expressed in this BNF rule is not an expression (that is, of syntactic class ⟨*expr*⟩). Rather, we categorize it as a DEFINITION (of syntactic class ⟨*definition*⟩). Such definitions are allowed only at the top level of program files or the REPL.

3 is still around; it's just inaccessible, shadowed by the new name x. (In Chapter 15, we provide a demonstration that this is so.)

Exercise 19

In the sequence of expressions

```
let tax_rate = 0.05 ;;
let price = 5. ;;
let price = price * (1. +. tax_rate) ;;
price ;;
```

what is the value of the final expression? (You can use the REPL to verify your answer.)

Global naming is available only at the top level. A global name cannot be defined from within another expression, for instance, the body of a local `let`. The following is thus not well-formed:

```
# let radius = 4. in
# let pi = 3.1416 in
# let area = pi *. radius ** 2. ;;
Line 3, characters 30-32:
3 | let area = pi *. radius ** 2. ;;
                                 ^^

Error: Syntax error
```

Exercise 20

How might you get the effect of this definition of a global variable `area` by making use of local variables for `pi` and `radius`?

❧

We alluded to the fact that in OCaml, functions are first-class values, and as such they can be named as well. In fact, the ability to name values becomes most powerful when the named values are functions. In the next chapter, we introduce functions and function application in OCaml, and start to demonstrate the power of functions as an abstraction mechanism.