6
Functions

Recall that abstraction is the process of viewing a set of apparently dissimilar things as instantiating an underlying identity. Plato in his *Phaedrus* has Socrates adduce two rhetorical principles. The first Socrates describes as

> That of perceiving and bringing together in one idea the scattered particulars, that one may make clear by definition the particular thing which he wishes to explain. (Plato, 1927)

that is, a principle of abstraction. (Socrates's second principle shows up in Chapter 8.)

Abstraction in programming is this process applied to code, and can be enabled by appropriate language constructs. Programming abstraction is important because it enables programmers to satisfy perhaps the most important edict of programming:

Edict of irredundancy:
Never write the same code twice.

A standard technique that beginning programmers use is "cut and paste" programming – you find some code that does more or less what you need, perhaps code you've written before, and you cut and paste it into your program, adjusting as necessary for the context the code now appears in. There is a high but mostly hidden cost to the cut and paste approach. If you find a bug in one of the copies, it needs to be fixed in all of the copies. If some functionality changes in one of the copies, the other copies don't benefit unless they are modified too. As documentation is added to clarify one of the copies, it must be maintained for all of them. When one of the copies is tested, no assurance is thereby gained for the other copies. There's a theme here. Having written the same code twice, all of the problems of debugging, maintaining, documenting, and testing code have been similarly multiplied.

The edict of irredundancy is the principle of avoiding the problems introduced by duplicative code. Rather than write the same code twice, the edict calls for viewing the apparently dissimilar pieces of code as instantiating an underlying identity, and factoring out the common parts using an appropriate abstraction mechanism.

Given the emphasis in the previous chapters, it will be unsurprising to see that the abstraction mechanism we turn to for satisfying the edict of irredundancy is the function itself. but before getting there, there is much to be introduced about how functions are defined and used in OCaml.

We will thus introduce how OCaml supports functions, their application and their definition, including some notational issues that simplify writing functions and connections to the typing constraints that make sure that code works properly. Then, we'll have the tools to provide an example of how functions can factor out redundancies from code in keeping with the edict of irredundancy. Finally, we'll extend the expressivity of functions even further with recursive functions, and introduce the idea of unit testing of functions to help verify their correctness.

6.1 Function application

We introduced functions in Section 4.4 as mappings from an argument to the function's value at that argument. We can make use of a function by APPLYING it to its argument. You'll be most familiar with the traditional and ubiquitous mathematical notation for function application, in which a symbol naming the function precedes a parenthesized, comma-separated list of the arguments, as, for instance, $f(1,2,3)$.[1] It is thus perhaps surprising that OCaml doesn't use this notation for function application. Instead, it follows the notational convention proposed by Church in his lambda calculus. (See Section 1.2.) In the lambda calculus, functions and their application are so central (indeed, there's basically nothing else in the logic) that the addition of the parentheses in the function application notation becomes onerous. Instead, Church proposed merely prefixing the function to its argument. Instead of $f(1)$, Church's notation would have $f\,1$. Instead of $f(g(1))$, he would have $f\,(g\,1)$, using the parentheses for grouping, but not for demarcating the arguments.

[1] Some historical background on this notation is provided in Section B.1.2.

Similarly, in OCaml, the function merely precedes its argument. The successor of `41` is simply `succ 41`. The square root of two is `sqrt 2.0`.

```
# succ 41 ;;
- : int = 42
# sqrt 2.0 ;;
- : float = 1.41421356237309515
```

Recall from Section 4.4 that functions (as all values) have types, which can be expressed as type expressions using the `->` operator. For instance, the successor function `succ` has the type given by the type expression `int -> int` and the `string_of_int` function the type `int -> string`.

Figure 6.1: Moses Schönfinkel (1889–1942), Russian logician and mathematician, first specified the use of higher-order functions to mimic the effect of multiple-argument functions.

6.2 *Multiple arguments and currying*

The simple prefix notation for function application is only appropriate when functions take exactly one argument. But it turns out that this is not a substantial limitation in a system (like the lambda calculus and like OCaml) in which functions are themselves values. Suppose we have a function that we think of as taking multiple arguments simultaneously (like $f(1,2,3)$). We can reconceptualize f as taking only one argument (in this case, the argument 1), returning a function that takes the second argument 2, again returning a function that takes the third and final argument 3, returning the final value. The type of such a function, which takes three integers returning an integer result, say, is thus

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

In essence, the function takes its three arguments *one at a time*, returning a function after each argument before the last. Although this trick was first discussed by Schönfinkel (1924), it is referred to as CURRYING a function, the resulting function being *curried*, so named after Haskell Curry who popularized the approach.

Figure 6.2: Haskell Curry (1900–1982), American logician, promulgator of the use of higher-order functions to simulate functions of multiple arguments, which is referred to as *currying* in his honor.

Because in OCaml functions take one argument, the language makes extensive use of currying, and language constructs facilitate its use. For instance, the `->` type expression operator is right associative (see Section 3.2) in OCaml, so that the type of the curried three-argument function above can be expressed as

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

Application, conversely, is left associative, so that applying a curried function `f` to its arguments can be notated `f 1 2 3` instead of `((f 1) 2) 3`.

We've already used some curried functions without noticing. The two-argument arithmetic and boolean operators, like `+`, `/.`, and `&&`, are curried. As usual, the REPL reveals their type:

```
# (+) ;;
- : int -> int -> int = <fun>
# (/.) ;;
- : float -> float -> float = <fun>
# (&&) ;;
- : bool -> bool -> bool = <fun>
```

Normally, we write these operators INFIX, placing the operator *between* its two arguments, but by placing the operator in parentheses[2] as we've done, the OCaml REPL interprets them as regular PREFIX functions, in which the function appears *before* its argument. Making use of this ability, they can even be applied in the one-by-one manner, as we've done here both parenthesized and unparenthesized:

```
# ((+) 3) 4 ;;
- : int = 7
# (+) 3 4 ;;
- : int = 7
```

[2] Care must be taken when parenthesizing the multiplication operators `*` and `*.` to convert them to prefix functions. Since OCaml comments are provided as `(* ⟨⟩ *)`, parenthesizing as `(*)` will be misinterpreted as the beginning of a comment. To avoid this problem, place spaces between the parentheses and the operator: `( * )`.

Exercise 21

What (if anything) are the types and values of the following expressions? Try to figure them out yourself before typing them into the REPL to verify your answer.

1. `(-) 5 3`
2. `5 - 3`
3. `- 5 3`
4. `"O" ^ "Caml"`
5. `(^) "O" "Caml"`
6. `(^) "O"`
7. `( ** )` – See footnote 2.

6.3 *Defining anonymous functions*

Now we get to the whole point of functional programming: *defining your own functions*. Suppose we want to specify a function that maps a certain input, call it `x`, to an output, say the doubling of `x`. The following expression does the trick: `fun x : int -> 2 * x`.

```
# fun x : int -> 2 * x ;;
- : int -> int = <fun>
```

The keyword `fun` introduces the function definition. The arrow `->` separates the typing of a variable that represents the input, the integer `x`, from an expression that represents the output value, `2 * x`. The output expression can, of course, make free use of the input variable as part of the computation.

We can apply this function to an argument (`21`, say). We use the usual OCaml prefix function application syntax, placing the function before its argument:

```
# (fun x : int -> 2 * x) 21 ;;
- : int = 42
```

In general, we construct such a "function without a name", an ANONYMOUS FUNCTION, with the OCaml `fun` construct:[3]

```
fun ⟨var⟩ : ⟨type⟩ -> ⟨expr⟩
```

[3] Warning: The same arrow symbol `->` is used in defining both function *values* and function *types*. This sometimes leads to confusion. Be aware that though the same symbol is used for both, the two are quite distinct.

Here, ⟨*var*⟩ is a variable, the name of the argument of the function, and ⟨*expr*⟩ is an expression defining the output of the function, which will be of the given ⟨*type*⟩.

The `fun` construct, like the `let` construct, is a binding construct. The `fun` construct introduces a variable and binds occurrences of that variable in its scope. The scope of the variable is the body of the `fun`, the expression ⟨*expr*⟩ after the arrow.

As was the case for `let` expressions, when the type of the variable can be inferred from how it is used in the definition part, as is typically the case, the typing part can be left off. So, for instance, the doubling function could be written

```
# fun x -> 2 * x ;;
- : int -> int = <fun>
```

and the same type `int -> int` still inferred.

Exercise 22

Try defining your own functions, perhaps one that squares a floating point number, or one that repeats a string.

6.4 *Named functions*

Now that we have the ability to define functions (with `fun`) and the ability to name values (with `let`), we can put them together to name newly-defined functions. Here, we give a global naming of the doubling function and use it:

```
# let double = fun x -> 2 * x ;;
val double : int -> int = <fun>
# double 21 ;;
- : int = 42
```

Here are functions for the circumference and area of circles of given radius:

```
# let pi = 3.1416 ;;
val pi : float = 3.1416

# let area =
#   fun radius ->
#     pi *. radius ** 2. ;;
val area : float -> float = <fun>

# let circumference =
#   fun radius ->
#     2. *. pi *. radius ;;
val circumference : float -> float = <fun>

# area 4. ;;
- : float = 50.2656
```

```
# circumference 4. ;;
- : float = 25.1328
```

6.4.1 Compact function definitions

This method for defining named functions, though effective, is a bit cumbersome. For that reason, OCaml provides a simpler syntax for defining functions, in which a definition for the calling pattern itself is provided. Instead of the phrasing

$$\texttt{let}\ \langle var_{func} \rangle\ \texttt{=}\ \texttt{fun}\ \langle var_{arg} \rangle\ \texttt{->}\ \langle expr \rangle$$

OCaml allows the following equivalent phrasing

$$\texttt{let}\ \langle var_{func} \rangle\ \langle var_{arg} \rangle\ \texttt{=}\ \langle expr \rangle$$

This syntax for defining functions may be more familiar from other languages. It is also consistent with a more general pattern-matching syntax that we will come to in Section 7.2.

This compact syntax for function definition is an example of SYNTACTIC SUGAR,[4] a bit of additional syntax that serves to abbreviate a more complex construction. By adding some syntactic sugar, the language can provide simpler expressions without adding underlying constructs to the language; a language with a small core set of constructs can still have a sufficiently expressive concrete syntax that it is pleasant to program in. As we introduce additional syntactic sugar constructs, notice how they allow for idiomatic programming without increasing the core language.

[4] The term “syntactic sugar” was first used by Landin (1964) (Figure 17.7) to describe just such abbreviatory constructs.

We can use this more compact function definition notation to provide a more elegant definition of the doubling function:

```
# let double x = 2 * x ;;
val double : int -> int = <fun>
# double (double 3) ;;
- : int = 12
```

This compact notation applies to local definitions as well.

```
# let double x = 2 * x in
# double (double 3) ;;
- : int = 12
```

It even extends to multiple-argument curried functions. The definition

```
# let hypotenuse x y =
#   sqrt (x ** 2. +. y ** 2.) ;;
val hypotenuse : float -> float -> float = <fun>
```

is syntactic sugar for (and hence completely equivalent to) the definition

```
# let hypotenuse =
#   fun x ->
#     fun y ->
#       sqrt (x ** 2. +. y ** 2.) ;;
val hypotenuse : float -> float -> float = <fun>
```

6.4.2 Providing typings for function arguments and outputs

As in all definitions, you can provide a typing for the variable being defined, as in

```
# let hypotenuse : float -> float -> float =
#   fun x ->
#     fun y ->
#       sqrt (x ** 2. +. y ** 2.) ;;
val hypotenuse : float -> float -> float = <fun>
```

and it is good practice to do so for top-level definitions. That way, you are registering your intentions as to the types – remember the edict of intention? – and the language interpreter can verify that those intentions are satisfied. (See Section C.3.4.)

In the compact notation, typings can and should be provided for the application of the function to its arguments, as well as for the arguments itself. In the `hypotenuse` function definition, the application `hypotenuse x y` is of type `float`, which can be recorded as

```
# let hypotenuse x y : float =
#   sqrt (x ** 2. +. y ** 2.) ;;
val hypotenuse : float -> float -> float = <fun>
```

Each of the arguments can be explicitly typed as well.

```
# let hypotenuse (x : float) (y : float) : float =
#   sqrt (x ** 2. +. y ** 2.) ;;
val hypotenuse : float -> float -> float = <fun>
```

Here, we have recorded that `x` and `y` are each of `float` type, and the result of an application `hypotenuse x y` is also a `float`, which together capture the full information about the type of `hypotenuse` itself. Consequently, the type inferred for the `hypotenuse` function itself is, as before, `float -> float -> float`, that is, a curried binary function from `float`s to `float`s.

Exercise 23

Consider the following beginnings of function declarations. How would these appear using the compact notation?

1. `let foo : bool -> int -> bool = ...`
2. `let foo : bool -> (int -> bool) = ...`
3. `let foo : (float -> int) -> float -> bool = ...`

Exercise 24

What are the types for the following expressions?

1. `let greet y = "Hello" ^ y in greet "World!" ;;`

2. `fun x -> let exp = 3. in x ** exp ;;`

Exercise 25

Define a function `square` that squares a floating point number. For instance,

```
# square 3.14 ;;
- : float = 9.8596
# square 1234567. ;;
- : float = 1524155677489.
```

Exercise 26

Define a function `abs : int -> int` that computes the absolute value of an integer.

```
# abs (-5) ;;
- : int = 5
# abs 0 ;;
- : int = 0
# abs (3 + 4) ;;
- : int = 7
```

Exercise 27

The `Stdlib.string_of_bool` function returns a string representation of a boolean. Here it is in operation:

```
# string_of_bool (3 = 3) ;;
- : string = "true"
# string_of_bool (0 = 3) ;;
- : string = "false"
```

What is the type of `string_of_bool`? Provide your own function definition for it.

Exercise 28

Define a function `even : int -> bool` that determines whether its integer argument is an even number. It should return `true` if so, and `false` otherwise. Try using both the compact notation for the definition and the full desugared notation. Try versions with and without typing information for the function name.

Exercise 29

Define a function `circle_area : float -> float` that returns the area of a circle of a given radius specified by its argument. Try all of the variants described in Exercise 28.

Exercise 30

A frustrum (Figure 6.3) is a three-dimensional solid formed by slicing off the top of a cone parallel to its base. The volume V of a frustrum with radii r_1 and r_2 and height h is given by the formula

$$V = \frac{\pi h}{3}(r_1^2 + r_1 r_2 + r_2^2) \quad .$$

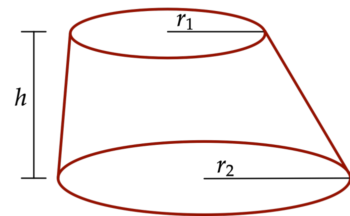

Figure 6.3: The frustrum of a cone, with top and bottom radii r_1 and r_2 respectively, and height h.

Implement a function to calculate the volume of a frustrum given the radii and height.

Problem 31

The calculation of the date of Easter, a calculation so important to early Christianity that it was referred to simply as COMPUTUS (“the computation”), has been the subject of innumerable algorithms since the early history of the Christian church. An especially simple method, published in *Nature* in 1876 and attributed to “A New York correspondent” (1876), proceeds by sequentially calculating the following values on the basis of the

year Y:

$$a = Y \bmod 19 \qquad h = (19a + b - d - g + 15) \bmod 30$$
$$b = \frac{Y}{100} \qquad i = \frac{c}{4}$$
$$c = Y \bmod 100 \qquad k = c \bmod 4$$
$$d = \frac{b}{4} \qquad l = (32 + 2e + 2i - h - k) \bmod 7$$
$$e = b \bmod 4 \qquad m = \frac{a + 11h + 22l}{451}$$
$$f = \frac{b+8}{25} \qquad month = \frac{h + l - 7m + 114}{31}$$
$$g = \frac{b - f + 1}{3} \qquad day = ((h + l - 7m + 114) \bmod 31) + 1$$

Write two functions, `computus_month` and `computus_day`, which take an integer year argument and return, respectively, the month and day of Easter as calculated by the method above. Use them to verify that the date of Easter in 2018 was April 1.

6.5 *Function abstraction and irredundancy*

We have enough background in place to see directly how functions are key to obeying the edict of irredundancy. Recall the comparison of the areas of two triangles from Section 5.4. By appropriate use of naming subcalculations, the computation was defined as

```
# 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
```

But some obvious redundancies remain. The calculation of `left_area` and `right_area` are structured identically, composed of first a calculation of the semiperimeter for the three sides and then the area calculation itself, again using the three side lengths in corresponding places.

Of course, they are not strictly identical; if they were, we could just use the naming trick (Section 5.4) to remove the redundancy. However, except for the three side lengths, the two calculations are the same. The two area values involve the same computation over the side lengths, the same mapping from side lengths to area, the same *function* of the side lengths so to speak. We can view these two dissimilar

expressions as manifesting an underlying identity by thinking of them as applications of one and the same function (call it `area`) to the three side lengths.

We start with a definition of this `area` function.

```
# let area x y z =
#   let sp = (x +. y +. z) /. 2. in
#   sqrt (sp *. (sp -. x) *. (sp -. y) *. (sp -. z)) ;;
val area : float -> float -> float -> float = <fun>
```

The two original computations of `left_area` and `right_area` match this pattern exactly, just with different values substituted for the three side lengths `x`, `y`, and `z`.

To generate these two instances, we apply the `area` function to the two sets of side lengths and compare the results as before.

```
# let left_area = area 1. 1. 1.41 in
# let right_area = area 1.5 0.75 2. in
# if left_area > right_area then left_area else right_area ;;
- : float = 0.499991149296665216
```

It is worth noting that this solution to the triangle area comparison problem specifies each of the six side lengths exactly once. Compare that with the initial version, in which each of the six side lengths appears ten times in the calculation, providing the risk of accidentally modifying some of the occurrences but not others and introducing bugs that way. Similarly, the definition of semiperimeter occurs once in this version, but 16 times in the original version. The definition of area by Heron's method appears only once here but four times in the original. This is the essence of abstraction, capturing the underlying idea once that unifies many instances.

We've now seen two abstraction techniques for eliminating redundancies. For trivial redundant expressions, exact duplications, it suffices to name the expression once and refer to it by its name multiple times. When the redundancy is a bit more subtle, involving systematic differences as to particular values in particular places, we can introduce a function that abstracts over those places, applying it to the particular values. But there are cases where mere substitution of simple values (like in the `area` example) is not sufficient. The true power of functions comes in with these even more sophisticated cases, which we explore in detail in Chapter 8.

To prepare for those abstraction techniques, we extend the expressivity of functions even further by allowing functions to be defined in terms of themselves, recursive functions.

"abstraction" — 2024/2/14 — 22:06 — page 69 — #69

6.6 *Defining recursive functions*

Consider the FACTORIAL function, which maps its integer input onto the product of all the positive integers that are no larger. Thus, the factorial of 3, traditionally notated with a suffixed exclamation mark as 3!, is the product of 1, 2, and 3, that is, 6; and 4! is 24. Notice that 4! is $4 \cdot 3!$, which makes sense because 3! has already incorporated all the integers up to 3, so the only remaining integer to multiply in is 4 itself. Indeed, in general,

$$n! = n \cdot (n-1)!$$

for all integers n greater than 1, and if we take the value of 0! to be 1, the equation even holds for $n = 1$. This serves to completely define the factorial function. We can take its definition to be given by the two equations[5]

$$0! = 1$$
$$n! = n \cdot (n-1)! \quad \text{for } n > 0$$

[5] See Section B.1.1 for more background on defining mathematical functions by equations.

We can implement the factorial function directly from this definition. The first line of the definition, setting up the name of the function (`fact`), its single integer argument (`n`), and its output type (`int`) is straightforward.

```
let fact (n : int) : int =
  ...
```

The body of the function starts by distinguishing the two cases, when `n` is zero and when `n` is positive.

```
let fact (n : int) : int =
  if n = 0 then ...
  else ...
```

The zero case is simple; the output value is 1.

```
let fact (n : int) : int =
  if n = 0 then 1
  else ...
```

The non-zero case involves multiplying `n` by the factorial of `n - 1`.

```
let fact (n : int) : int =
  if n = 0 then 1
  else n * fact (n - 1) ;;
```

Let's try it.

```
# let fact (n : int) : int =
#   if n = 0 then 1
#   else n * fact (n - 1) ;;
```

```
Line 3, characters 9-13:
3 | else n * fact (n - 1) ;;
            ^^^^
Error: Unbound value fact
Hint: If this is a recursive definition,
you should add the 'rec' keyword on line 1
```

There seems to be a problem. Recall from Section 5.5 that the scope of a `let` is the body of the `let` (or the code following a global `let`), but not the definition part of the `let`. Yet we've referred to the name `fact` in the definition of the `fact` function. The scope rules for the `let` constructs (both local and global) disallow this.

In order to extend the scope of the naming to the definition itself, to allow a recursive definition, we add the `rec` keyword after the `let`.

```
# let rec fact (n : int) : int =
#   if n = 0 then 1
#   else n * fact (n - 1) ;;
val fact : int -> int = <fun>
```

The `rec` keyword means that the scope of the `let` includes not only its body but also its definition part. With this change, the definition goes through, and in fact, the function works well:

```
# fact 0 ;;
- : int = 1
# fact 1 ;;
- : int = 1
# fact 4 ;;
- : int = 24
# fact 20 ;;
- : int = 2432902008176640000
```

You may in the past have been admonished against defining something in terms of itself, such as "comb: an object used to comb one's hair; to comb: to run a comb through." You may therefore find something mysterious about recursive definitions. How can we make use of a function in its own definition? We seem to be using it before it's even fully defined. Isn't that problematic?

Of course, recursive definition *can* be problematic. For instance, consider this recursive definition of a function to add "just one more" to a recursive invocation of itself:

```
# let rec just_one_more (x : int) : int =
#   1 + just_one_more x ;;
val just_one_more : int -> int = <fun>
```

The *definition* works just fine, but any attempt to *use* it fails impressively:

```
# just_one_more 42 ;;
Stack overflow during evaluation (looping recursion?).
```

The error message "Stack overflow during evaluation (looping recursion?)" gives a hint as to what's gone wrong; there is indeed a looping recursion that would go on forever if the computer didn't run out of memory ("stack overflow") first.

But a recursion that is well founded can be quite useful.[6] In the case of factorial, each recursive invocation of `fact` is given an argument that is one smaller than the previous invocation, so that eventually an invocation on argument 0 will occur and the recursion will end. Because there are branches of computation (namely, the first arm of the conditional) without recursive invocations of `fact`, and those branches will eventually be taken, all is well.

[6] In fact, the computer scientist C. A. R. Hoare in his 1981 Turing Award lecture described his own introduction to recursion this way:

> Around Easter 1961, a course on ALGOL 60 was offered in Brighton, England, with Peter Naur, Edsger W. Dijkstra, and Peter Landin as tutors. . . . It was there that I first learned about recursive procedures and saw how to program the sorting method which I had earlier found such difficulty in explaining. It was there that I wrote the procedure, immodestly named QUICKSORT, on which my career as a computer scientist is founded. Due credit must be paid to the genius of the designers of ALGOL 60 who included recursion in their language and enabled me to describe my invention so elegantly to the world. I have regarded it as the highest goal of programming language design to enable good ideas to be elegantly expressed. (Hoare, 1981)

But will those branches always be eventually taken? Unfortunately not.

```
# fact (~-5) ;;
Stack overflow during evaluation (looping recursion?).
```

This looks familiar. Counting down from any non-negative integer will eventually get us to zero. But counting down from a negative integer won't. We intended the factorial function to apply only to non-negative integers, the values for which it's defined, but we didn't express that intention – the edict of intention again – with this unfortunate result.

You might think that we could solve this problem with types. Instead of specifying the argument as having integer type, perhaps we could specify it as of non-negative integer type. Unfortunately, OCaml does not provide for this more fine-grained type, and in any case, other examples might require different constraints on the type, perhaps odd integers only, or integers larger than 7, or integers within a certain range.

Exercise 32

For each of the following cases, define a recursive function of a single argument for which the definition is well founded (and the computation terminates) only when the argument is

1. an odd integer;
2. an integer less than or equal to 5;
3. the integer 0;
4. the truth value `true`.

OCaml's type system isn't expressive enough to capture these fine-grained distinctions.[7] Instead, we'll have to deal with such anomalous conditions using different techniques, which will be the subject of Chapter 10.

[7] If you are interested in the issue, you might explore the literature on DEPENDENT TYPE SYSTEMS, which provide this expanded expressivity at the cost of much more complex type inference computations.

Exercise 33

Imagine tiling a floor with square tiles of ever-increasing size, each one abutting the previous two, as in Figure 6.4. The sides of the tiles grow according to the FIBONACCI SEQUENCE, in which each number is the sum of the previous two. By convention, the

first two numbers in the sequence are 0 and 1. Thus, the third number in the sequence is $0+1=1$, the fourth is $1+1=2$, and so forth.

The first 10 numbers in the Fibonacci sequence are

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

The Fibonacci sequence has connections to many natural phenomena, from the spiral structure of seashells (as alluded to in the figure) to the arrangement of seeds in a sunflower to the growth rate of rabbits. It even relates to the golden ratio: the tiled area depicted in the figure tends toward a golden rectangle (see Exercise 8) as more tiles are added. (Exercise 174 explores this fact.)

Define a recursive function `fib : integer -> integer` that given an index into the Fibonacci sequence returns the integer at that index. For instance,

```
# fib 1 ;;
- : int = 0
# fib 2 ;;
- : int = 1
# fib 8 ;;
- : int = 13
```

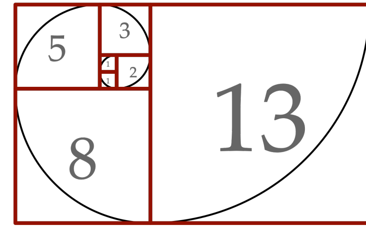

Figure 6.4: A Fibonacci tiling.

Exercise 34

Define a function `fewer_divisors : int -> int -> bool`, which takes two integers, `n` and `bound`, and returns `true` if `n` has fewer than `bound` divisors (including 1 and `n`). For example:

```
# fewer_divisors 17 3 ;;
- : bool = true
# fewer_divisors 4 3 ;;
- : bool = false
# fewer_divisors 4 4 ;;
- : bool = true
```

Do not worry about zero or negative arguments or divisors. Hint: You may find it useful to define an auxiliary function to simplify the definition of `fewer_divisors`.

6.7 *Unit testing*

Having written some functions, how can we have some assurance that our code is correct? Best might be a mathematical proof that the code does what it's supposed to do. Such a proof would guarantee that the code generates the appropriate values regardless of what inputs it is given. This is the domain of FORMAL VERIFICATION of software. Unfortunately, the difficulty of providing formal specifications that can be verified, along with the arduousness of carrying out the necessary proofs, means that this approach to program correctness is used only in rare circumstances. It is, in any case, beyond the scope of this book.

But if we can't have a proof that a program generates the appropriate values on *all* input values, perhaps we can at least verify that it generates the appropriate values on *some* of them – even better if the values we verify are representative of a full range of cases. This leads us to the approach of UNIT TESTING, the systematic evaluation of code on known inputs, comparing the behavior to the expected.

In this section, we begin the development of a simple unit testing framework for OCaml, continuing the development in Sections 10.5 and 17.6. We do so not because OCaml lacks a good unit testing tool of its own; in fact, there are several such full-featured packages, such

as ounit, alcotest, qcheck, ppl_inline_tests, crowbar, bun, and broken, providing functionality far beyond what we develop in this running example. Rather, seeing the construction should make clearer what is going on in such unit testing tools, making their utility clearer. In addition, the subtle issues that arise provide a nice opportunity to demonstrate the use of abstractions (exceptions and laziness) that we introduce later. But we start here using only functions.

Consider the fact function defined above. It exhibits the following (correct) behavior:

```
# fact 1 ;;
- : int = 1
# fact 2 ;;
- : int = 2
# fact 5 ;;
- : int = 120
# fact 10 ;;
- : int = 3628800
```

We can describe the correctness conditions for these inputs as a series of boolean expressions.

```
# fact 1 = 1 ;;
- : bool = true
# fact 2 = 2 ;;
- : bool = true
# fact 5 = 120 ;;
- : bool = true
# fact 10 = 3628800 ;;
- : bool = true
```

A unit testing function for fact, call it fact_test, verifies that fact calculates the correct values for representative examples. (Let’s start with these.) One approach is to simply evaluate each of the conditions and make sure that they are all true.

```
# let fact_test () =
#   fact 1 = 1
#   && fact 2 = 2
#   && fact 5 = 120
#   && fact 10 = 3628800 ;;
val fact_test : unit -> bool = <fun>
```

We run the tests by calling the function:

```
# fact_test () ;;
- : bool = true
```

If all of the tests pass (as they do in this case), the testing function returns true. If any test fails, it returns false. Unfortunately, in the latter case it provides no help in tracking down the tests that fail.

In order to provide information about which tests have failed, we'll print an indicative message associated with the test. An auxiliary function to handle the printing will be helpful:[8]

```
# let unit_test (test : bool) (msg : string) : unit =
#   if test then
#     Printf.printf "%s passed\n" msg
#   else
#     Printf.printf "%s FAILED\n" msg ;;
val unit_test : bool -> string -> unit = <fun>
```

[8] We're making use here of two language constructs that, strictly speaking, belong in later chapters, as they involve side effects, computational artifacts that don't affect the value expressed: the sequencing operator (`;`) discussed in Section 15.3, and the `printf` function in the `Printf` library module. Side effects in general are introduced in Chapter 15.

Now the `fact_test` function can call `unit_test` to verify each of the conditions.

```
# let fact_test () =
#   unit_test (fact 1 = 1) "fact 1";
#   unit_test (fact 2 = 2) "fact 2";
#   unit_test (fact 5 = 120) "fact 5";
#   unit_test (fact 10 = 3628800) "fact 10" ;;
val fact_test : unit -> unit = <fun>
```

Running `fact_test` provides a report on the performance of `fact` on each of the unit tests.

```
# fact_test () ;;
fact 1 passed
fact 2 passed
fact 5 passed
fact 10 passed
- : unit = ()
```

We'll want to unit test `fact` as completely as is practicable. We can't test *every* possible input, but we can at least try examples representing as wide a range of cases as possible. We're missing an especially important case, the base case for the recursion, `fact 0`. We'll add a unit test for that case:

```
# let fact_test () =
#   unit_test (fact 0 = 1) "fact 0 (base case)";
#   unit_test (fact 1 = 1) "fact 1";
#   unit_test (fact 2 = 2) "fact 2";
#   unit_test (fact 5 = 120) "fact 5";
#   unit_test (fact 10 = 3628800) "fact 10" ;;
val fact_test : unit -> unit = <fun>
```

We haven't tested the function on negative numbers, and probably should. But `fact` as currently written wasn't intended to handle those cases. We postpone discussion about unit testing in such cases to Section 10.5, when we'll have further tools at hand. (See Exercise 83.)

Testing the `hypotenuse` function presents further issues. We might want to check the simple case of the hypotenuse of a unit triangle, whose hypotenuse ought to be about 1.41421356, as well as the limiting case of a “triangle” with zero-length sides.

```
# let hypotenuse_test () =
#   unit_test (hypotenuse 0. 0. = 0.) "hyp 0 0";
#   unit_test (hypotenuse 1. 1. = 1.41421356) "hyp 1 1" ;;
val hypotenuse_test : unit -> unit = <fun>

# hypotenuse_test () ;;
hyp 0 0 passed
hyp 1 1 FAILED
- : unit = ()
```

The test reveals a problem. The unit triangle test has failed, not because the `hypotenuse` function is wrong but because the value we've proposed isn't exactly the floating point number calculated. The `float` type has a fixed capacity for representing numbers, and can't therefore represent all numbers exactly. The best we can do is check that floating point calculations are approximately correct, within some tolerance. Rather than checking the condition as above, instead we can check that the value is within, say, 0.0001 of the value in the test, a condition like this:

```
# hypotenuse 1. 1. -. 1.41421356 < 0.0001 ;;
- : bool = true
```

Instead of writing out these more complex conditions each time they're needed, we'll devise another unit testing function for approximate floating point calculations:

```
# let unit_test_within (tolerance : float)
#                      (test_value : float)
#                      (expected : float)
#                      (msg : string)
#                    : unit =
#   unit_test (abs_float (test_value -. expected) < tolerance) msg ;;
val unit_test_within : float -> float -> float -> string -> unit =
  <fun>
```

We can restate the `hypotenuse_test` function to make use of these approximate tests. (We've added a few more for other conditions.)

```
# let hypotenuse_test () =
#   unit_test_within 0.0001 (hypotenuse 0. 0.) 0.       "hyp 0 0";
#   unit_test_within 0.0001 (hypotenuse 1. 1.) 1.4142    "hyp 1 1";
#   unit_test_within 0.0001 (hypotenuse ~-.1. 1.) 1.4142 "hyp -1 1";
#   unit_test_within 0.0001 (hypotenuse 2. 2.) 2.8284    "hyp 2 2" ;;
val hypotenuse_test : unit -> unit = <fun>
```

Calling the function demonstrates that all of the calculations hold within the required tolerance.

```
# hypotenuse_test () ;;
hyp 0 0 passed
hyp 1 1 passed
hyp -1 1 passed
```

```
hyp 2 2 passed
- : unit = ()
```

We'll return to the question of unit testing in Sections 10.5 and 17.6, when we have more advanced tools to use.

6.8 Supplementary material

- Lab 1: Basic functional programming
- Problem set A.1: The prisoners' dilemma