# Chapter 1

# Higher-Order Functions

**Abstract** A *higher-order function* is a function that takes a function as an argument and/or returns a function. higher-order functions are also sometimes called functionals or functors. In this chapter you will learn how to:

- make functions that take and/or return functions as values.
- create new functions with the function composition operator.
- create new functions with a partial specification of function arguments.

## 1.1 Functions as Values

F# is a functions-first programming language with strong support for working with functions as values: Functions evaluate as *closures*, see **??**, which can be passed to and from functions as any other value. An example of a higher-order function is `List.map` which takes a function and a list and produces a list, demonstrated in Listing 1.3. Here `List.map` applies the function `inc` to every element of the

**Listing 1.1 higherOrderListMap.fsx:**
**`List.map` is a higher-order function, since it takes a function as argument.**

```
let inc x = x + 1
let newList = List.map inc [2; 3; 5]
printfn "%A" newList
```

```
$ dotnet fsi higherOrderListMap.fsx
[3; 4; 6]
```

list. higher-order functions are often used together with *anonymous functions*, where the anonymous function is given as an argument. For example, Listing 1.3 may be rewritten using an anonymous function as shown in Listing 1.2.

**Listing 1.2 higherOrderAnonymous.fsx:**
**An anonymous function is a higher-order function used here as an unnamed argument. Compare with Listing 1.3.**

```
let newList = List.map (fun x -> x + 1) [2; 3; 5]
printfn "%A" newList
```

```
$ dotnet fsi higherOrderAnonymous.fsx
[3; 4; 6]
```

Writing a function that takes other functions as arguments is straightforward. If we were to make our own `map` function, it could look like what is shown in Listing 1.3. In this case, `map` has the type

```
map: f: ('a -> 'b) -> lst: 'a list -> 'b list
```

All three programs result in the same output and as such are equal. Likewise, running times will be equal. However, they differ in readability for a programmer and ease of bug hunting and future maintenance: Bindings allow us to reuse the code at a later stage, but if there is no reuse, then the additional bindings may result in a cluttered program. Nevertheless, programs using anonymous functions can be difficult to debug. Finally, bindings emphasize semantic aspects of the evaluation

**Listing 1.3 higherOrderMap.fsx:**
**A homemade version of `List.map`.**

```
let rec map f lst =
  match lst with
    [] -> []
    | e::rst -> (f e)::map f rst

let newList = map (fun x->x+1) [2; 3; 5]
printfn "%A" newList
```

```
$ dotnet fsi higherOrderMap.fsx
[3; 4; 6]
```

being performed merely by the names we select, and typically long, meaningful names are to be preferred, within reasonable limits. For example, instead of `inc` one could have used `increment_by_one` or similar which certainly is semantically meaningful, but many programmers will find that the short is to be preferred in order to reduce the amount of typing to be performed.

Functions can also be return values. For example, in Listing 1.4, the function `incBy` creates functions, which increments by a given argument. Note that the closure of

**Listing 1.4 higherOrderReturn.fsx:**
**The procedure `inc` returns an increment function. Compare with Listing 1.3.**

```
let incBy n =
  fun x -> x + n
printfn "%A" (List.map (incBy 2) [2; 3; 5])
```

```
$ dotnet fsi higherOrderReturn.fsx
[4; 5; 7]
```

this customized function is only produced once when the arguments for `List.map` are prepared, and not every time `List.map` applies the function to the elements of the list. Compare with Listing 1.3.

## 1.2 The Function Composition Operator

F# has strong support for working with functions on a functional level. In **??** on page **??**, we saw how functions can be composed by passing the result of one function to the next, e.g., using piping. Alternatively, we can compose functions before we apply them to values using the *">>"* and *"<<" composition operators*, which is defined as,

```
(>>) : ('a -> 'b, 'b -> 'c) -> ('a -> 'c)
(<<) : ('b -> 'c, 'a -> 'b,) -> ('a -> 'c)
```

i.e., it takes two functions of type `'a -> 'b` and `'b -> 'c` respectively, and produces a new function of type `a' -> 'c`. As an example, consider the composition of the $\log x$ and $\sqrt{x}$ functions to make $f(x) = \log\left(\sqrt{x}\right), x > 0$. Using the piping operator, this can be written as `x |> sqrt |> log`, which is sufficient, if the function is only to be used once or not passed as an argument. However, the ">>" operator allows us to make a new function by `logSqrt = sqrt >> log` or equivalently `logSqrt = log << sqrt`. As with the piping operators, the precedence and association rules imply differences in the number of parentheses needed, but in the end, the choice mostly boils down to personal preference. In Listing 1.5 is a comparison of regular composition and the composition operator is shown.

**Listing 1.5 functionPipingAdv.fsx:**
**A demonstration of differences in function composition.**

```
let lst = [1.0..3.0]
// regular composition
printfn "%A" (List.map (fun x -> log (sqrt x)) lst)
// piping operator
printfn "%A" (List.map (fun x -> x |> sqrt |> log) lst)
printfn "%A" (List.map (fun x -> log <| (sqrt <| x)) lst)
// composition operator
printfn "%A" (List.map (sqrt>>log) lst)
printfn "%A" (List.map (log<<sqrt) lst)
```

```
$ dotnet fsi functionPipingAdv.fsx
[0.0; 0.3465735903; 0.5493061443]
[0.0; 0.3465735903; 0.5493061443]
[0.0; 0.3465735903; 0.5493061443]
[0.0; 0.3465735903; 0.5493061443]
[0.0; 0.3465735903; 0.5493061443]
```

## 1.3 Currying

Functions, with only the initial list of arguments, also return functions. This is called *partial specification* or *currying* in tribute of Haskell Curry[1]. An example is given in Listing 1.6. Here, `mul 2.0` is a partial application of the function `mul x y`, where

[1] Haskell Curry (1900–1982) was an American mathematician and logician who also has a programming language named after him: Haskell.

**Listing 1.6 higherOrderCurrying.fsx:**
**Currying: defining a function as a partial specification of another.**

```
let mul x y = x*y
let timesTwo = mul 2.0
printfn "%g" (mul 5.0 3.0)
printfn "%g" (timesTwo 3.0)
```

```
$ fsharpc --nologo higherOrderCurrying.fsx && mono
   higherOrderCurrying.exe
15
6
```

the first argument is fixed, and hence `timesTwo` is a function of 1 argument being the second argument of `mul`.

Currying is emphasized by how the type of functions of several values is written. Consider a function `f` of two generic arguments. Its type in F# will be `f : 'a -> 'b -> 'c`, meaning that `f` takes an argument of type `'a` and returns a function of type `'b -> 'c`. That is, if just one argument is given, then the result is a function, not a value. This is illustrated in Figure 1.1.

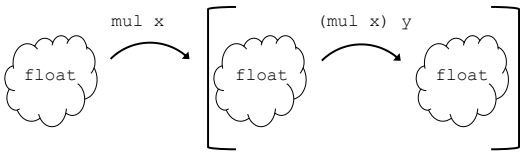


**Fig. 1.1** The type of `mul x y = x*y` is a function of 2 arguments is a function from values to functions to values.

Currying makes elegant programs and is often used in functional programming. Nevertheless, currying may lead to obfuscation, and in general, **currying should be used with care and be well documented for proper readability of code.** ⋆

## 1.4 Key concepts and terms in this chapter

In this chapter, we have looked at higher-order functions. Key concepts have been:

- **Functions are values** and can be used as arguments and return value.
- Functions can be compose using the **composition operator** to produce new functions.

- Function arguments can be **partially specified** to create new functions with less arguments. This is also called **currying**.