1 Higher-Order Functions

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. 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 1.1.

· higher-order function

· closure

Listing 1.1 higherOrderMap.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
```

```
$ fsharpc --nologo higherOrderMap.fsx && mono
  higherOrderMap.exe
[3; 4; 6]
```

Here `List.map` applies the function `inc` to every element of the list. higher-order functions are often used together with *anonymous functions*, where the anonymous functions is given as argument. For example, Listing 1.1 may be rewritten using an anonymous function as shown in Listing 1.2.

· anonymous functions

Listing 1.2 higherOrderAnonymous.fsx:
An anonymous function is a higher-order function used here as an unnamed argument. Compare with Listing 1.1.

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

```
$ fsharpc --nologo higherOrderAnonymous.fsx
$ mono higherOrderAnonymous.exe
[3; 4; 6]
```

The code may be compacted even further, as shown in Listing 1.3.

Listing 1.3 higherOrderAnonymousBrief.fsx: A compact version of Listing 1.1.

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

```
$ fsharpc --nologo higherOrderAnonymousBrief.fsx
$ mono higherOrderAnonymousBrief.exe
[3; 4; 6]
```

What was originally three lines in Listing 1.1 including bindings to the names `inc` and `newList` has in Listing 1.3 been reduced to a single line with no bindings. 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 allows 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. Further, for compact programs like Listing 1.3, it is not possible to perform a unit test of the function arguments. Finally, bindings emphasize semantic aspects of the evaluation 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.

Anonymous functions are also useful as return values of functions, as shown in Listing 1.4

Listing 1.4 higherOrderReturn.fsx: The procedure `inc` returns an increment function. Compare with Listing 1.1.

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

```
$ fsharpc --nologo higherOrderReturn.fsx && mono
  higherOrderReturn.exe
[3; 4; 6]
```

Here the `inc` function produces a customized incrementation function as argument to `List.map`: It adds a prespecified number to an integer argument. Note that the closure of this customized function is only produced once, when the arguments for `List.map` is prepared, and not every time `List.map` maps the function to the elements of the list. Compare with Listing 1.1.

Piping is another example of a set of higher-order function: `(<|)`, `(|>)`, `(<||)`, `(||>)`, `(<|||)`, `(|||>)`. E.g., the functional equivalent of the right-to-left piping operator takes a value and a function and applies the function to the value, as demonstrated in Listing 1.5. · piping

Listing 1.5 higherOrderPiping.fsx:
The functional equivalent of the right-to-left piping operator is a higher-order function.

```
let inc x = x + 1
let aValue = 2
let anotherValue = (|>) aValue inc
printfn "%d -> %d" aValue anotherValue
```

```
$ fsharpc --nologo higherOrderPiping.fsx && mono
  higherOrderPiping.exe
2 -> 3
```

Here the piping operator is used to apply the `inc` function to `aValue`. A more elegant way to write this would be `aValue |> inc`, or even just `inc aValue`.

1.1 Function Composition

Piping is a useful shorthand for composing functions, where the focus is on the transformation of arguments and results. Using higher-order functions, we can forgo the arguments and compose functions as functions directly. This is done with the "`>>`" and "`<<`" operators. An example is given in Listing 1.6.

· function composition
· »@>>
· «@<<

Listing 1.6 higherOrderComposition.fsx:
Functions defined as compositions of other functions.

```
let f x = x + 1
let g x = x * x
let h = f >> g
let k = f << g
printfn "%d" (g (f 2))
printfn "%d" (h 2)
printfn "%d" (f (g 2))
printfn "%d" (k 2)
```

```
$ fsharpc --nologo higherOrderComposition.fsx
$ mono higherOrderComposition.exe
9
9
5
5
```

In the example we see that `(f >> g) x` gives the same result as `g (f x)`, while `(f << g) x` gives the same result as `f (g x)`. A memo technique for remembering the order of the application, when using the function composition operators, is that `(f >> g) x` is the same as `x |> f |> g`, i.e., the result of applying `f` to `x` is the argument to `g`. However, there is a clear distinction between the piping and composition operators. The type of the piping operator is

```
(|>) : ('a, 'a -> 'b) -> 'b
```

i.e., the piping operator takes a value of type `'a` and a function of type `'a -> 'b`, applies the function to the value, and produces the value `'b`. In contrast, the composition operator has type

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

1.2 Currying

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 called *partial specification* or *currying* in tribute of Haskell Curry[1]. An example is given in Listing 1.7.

· partial specification
· currying

Listing 1.7 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
```

Here, `mul 2.0` is a partial application of the function `mul x y`, where the first argument is fixed, and hence `timesTwo` is a function of 1 argument being the second argument of `mul`. The same can be achieved using tuple arguments, as shown in Listing 1.8.

Listing 1.8 higherOrderTuples.fsx:
Partial specification of functions using tuples is less elegant. Compare with Listing 1.7.

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

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

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

Conversion between multiple and tuple arguments is easily done with higher-order functions, as demonstrated in Listing 1.9.

Listing 1.9: Two functions to convert between two and 2-tuple arguments.

```
> let curry f x y = f (x,y)
- let uncurry f (x,y) = f x y;;
val curry : f:('a * 'b -> 'c) -> x:'a -> y:'b -> 'c
val uncurry : f:('a -> 'b -> 'c) -> x:'a * y:'b -> 'c
```

Conversion between multiple and tuple arguments are useful when working with higher-order functions such as `Lst.map`. E.g., if `let mul (x, y) = x * y` as in Listing 1.8, then `curry mul` has the type `x:'a -> y:'b -> 'c` as can be seen in Listing 1.9, and thus is equal to the anonymous function `fun x y -> x * y`. Hence, `curry mul 2.0` is equal to `fun y -> 2.0 * y`, since the precedence of function calls is `(curry mul) 2.0`.

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.** Advice