ME 2017-08-22

15 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 functions~~, functions evaluate to ~~and through~~ closures, Which ~~functions~~ 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, as demonstrated ~~e.g., shown~~ in Listing 15.1.

· higher order function

not really maybe "lambdas"

maybe delete...

Listing 15.1 higherOrderMap.fsx:
List.map is a higher order function, since it takes a function as argument.

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

```
$ fsharpc --nologo higherOrderMap.fsx && mono higherOrderMap.exe
[2; 3; 5] -> [3; 4; 6]
```

Here `List.map` applies the function `inc` to every element of the list. * ~~Another is anonymous functions as shown in Listing 15.2.~~

Listing 15.2 higherOrderAnonymous.fsx:
An anonymous function is a higher order function, since the expression returns a function.

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

(fun x -> x+1)

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

The *anonymous function* is here bound to the `inc`-identifier and passed to `List.map` to produce the same result as in Listing 15.1. Similarly, we can make a function that returns a function as shown in Listing 15.3.

· anonymous function

* Until now, we have seen that functions can be introduced using the let-syntax. Another possibility for introducing a function is to use the expression syntax "fun x → e" where x is a variable and e is an expression. Such functions are called "anonymous" as the function is not directly given a name. For instance, the program in Listing 15.1 can instead be written as shown in Listing 15.2.

Function Types ← perhaps introduce a section about function types.

Listing 15.3 higherOrderReturn.fsx:
The procedure inc returns an increment function. Compare with Listing 15.2.

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

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

Here the inc function takes ~~no~~ an integer argument and returns a function. Thus, in order to supply a function to `List.map` we first need to call the inc function, and therefore the syntax is slightly different.

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 15.4. · piping

Listing 15.4 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`.

15.1 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 15.5.

space

· function composition
· “>>”
· “<<”

Listing 15.5 higherOrderComposition.fsx:
Functions defined as composition 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
```

flows the result of applying f to x into g.

In the example we see that `(f >> g) x` gives the same result as `g (f x)`, while `(f << g)` space `x` gives the same result as `f (g x)`. A memotechnique for remembering the order of the application, when using the function composition operators, is that `(f >> g) x` ~~is similar to `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~~ `(|>)` is `('a, 'a -> 'b) -> 'b`, i.e., the piping operator takes a value of type `'a` and a function of type `'a -> 'b` and applies the function to the value and produces the value `'b`. In contrast, the composition operator ~~is~~ `(>>)` 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`.

15.2 Currying

rephrase! Function can be defined as partial specification of another. This is called *currying* in tribute of Haskell Curry~~, and~~. An example is given in Listing 15.6. · currying

Listing 15.6 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
```

Here, `mul 2.0` is a partial ~~specification~~ 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 15.7.

Listing 15.7 higherOrderTuples.fsx:
Partial specification of functions using tuples is less elegant. Compare with Listing 15.6.

```
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
```

However, currying is more elegant, and is thus often used in functional programming. Nevertheless, ~~currying tends to lower readability of code, and in generally~~ **~~currying~~ should be used with care and be well documented for proper readability of code.** Advice

Introduce

```
let curry f x y = f (x, y)
let uncurry f (x, y) = f x y
```

Discuss their types and demonstrate their usefulness, for instance, in ~~relation to~~ cooperation with List.map.