Unit 4 – Advanced Functions

In this unit, we study the various function operators available in F#. We will start with basic function manipulation operators, and then move on to important design patterns that make use of these operators.

Function composition, pipe operator

Functions in F# are not more special than, say, integers or strings. Just like these simple data types can be defined, manipulated in expressions, transformed, taken as parameters from functions, and be returned as parameters, so can functions. This means that *functions may accept as parameters, and return as result, other functions*. A function that accepts another function as parameter is called a *higher order function* (HOF), and a function that returns a function is known as *curried*.

> This characteristic of supporting higher order and curried functions, which was originally typical of cutting-edge functional languages, has been widely recognized to be a fundamental aspect of high quality software engineering, and is now found in most modern programming languages. In this, F# has played an important historical role: the adoption of functional constructs in the mainstream began many years ago in C#, directly integrating F# innovations and know-how. From C#, "lambda functions" spread to other languages, from C++ to Java and more, finally becoming an absolute requirement. Of course other languages helped popularize these concepts, among others JavaScript played a very large role next to C#, in shaping software engineering as we understand it nowadays.

A simple higher order curried function is `compose`, the function that takes two functions as parameters and invokes them in sequence, turning them into a pipeline:

```
let compose f g = fun x -> g(f(x))
```

`compose` is a HOF because `f` and `g`, its parameters, are both functions. Given that `compose` returns a function, we also say that it is curried.

The `compose` function is actually so central to functional programming, that it already exists as a stndard operator in F#: the `(>>)` operator.

```
let compose = (>>)
```

> Notice that the `(>>)` operator, which visually resembles an arrow or a sort of pipeline, comes from the mathematical function composition operator $g \circ f$ (or $g \cdot f$), which is meant to represent the analogous to multiplication, but for functions. Notice that the mathematical notation is not very handy for programmers, as the order of execution will go from right to left, and for this reason

> (>>) flips the order. To respect this analogy with mathematics, languages like Haskell use the `.` operator as function composition, which F# has kept for object and field lookup in order to maintain the similarity to C#, C, C++, and Java. The identity of function composition is `id`, both in F# and in mathematics. `f >> id == id >> f == f`.

We could play around with function composition as follows:

```
let incr x = x + 1
let double x = x * 2
let halve x = x / 2

let f = incr >> double
```

Or we could go for an even longer pipeline:

```
let f = double >> incr >> incr >> halve
```

Notice that `f` does not invoke `double` or `incr`. Rather, `f` is simply a newly created, dynamic function, which will invoke `double`, `incr`, etc. whenever it itself is invoked.

F# knows another operator, `(|>)`, which we can use to pass an argument to a function by swapping the order. This makes it possible to write code in the order `subject |> verb |> verb |> ... |> verb`:

```
let f x = x |> incr
            |> double
```

> When using `(>>)` instead of `(|>)`, we can omit the declaration of parameters, but use of `(|>)` alone can greatly improve code readability.

An interesting pattern that we can build with function composition is function repetition, which basically implements the logic of a `for` loop by composing the body of the loop (a function `f`) `n` times with itself:

```
let rec repeat n f = if n <= 0 then id else f >> repeat (n-1) f
```

Thanks to `repeat`, we can perform an operation many times, easily building regular patterns:

```
let star s = s + "*"
let space s = s + " "
let newline s = s + "\n"

let row n = repeat n star
let square n = repeat n (row n >> newline)
```

Notice that neither `row 3`, nor `square 5` would be pictures. They are both functions, waiting to be either composed with other rendering functions, or invoked with an initial string to render from (`square 5 ""`).

HOF design patterns

Higher order functions can be assembled along standard design patterns which arise very commonly in most data structures.

`map`

A very powerful, and the most common design pattern when it comes to higher order and curried functions is `map` over a data structure. `map` performs a transformation of the *content* of a data structure, without altering the structure itself. We often refer to `map` as a *structure-preserving transformation*.

For example, consider transforming the first element of a pair `(x,y)` according to a function `f`:

```
let mapTupleLeft f (x,y) = (f x,y)
```

Notice that the resulting pair will still be a pair (moreover: the second element is precisely the same!). This is what we mean by *preserving structure*. Had the pair become a triple, then the structure would now be different. The first element is changed, by feeding the original value to function `f`. This leads to a new first element, which not only could be of a different value, but could actually even have a fully different type. We do not consider this change of *internal type* to be a change in structure, because the only structure we wish to preserve is the outer structure of the pair, and not the inner structure of the content to be transformed.

We could call the function as follows:

```
mapTupleLeft incr (1,"a") -> (2,"a")
mapTupleLeft double (2,"a") -> (4,"a")
mapTupleLeft (incr >> double) (2,["a";"b"]) -> (6,["a";"b"])
```

> There is a reason why the `map` design pattern is so common: `map` is the homomorphism associated with a functor. Functors are an incredibly common structure both in mathematics and in programming, and as such they are found everywhere. Many developers, unaware of the existance of functors, rediscover the concept accidentally by noticing that their data structures could indeed implement `map`.

A tuple supports two different `map` functions: one for the left element, the other for the right element. The preserved structure is clearly the same, but the preserved element is not:

```
let mapTupleRight f (x,y) = (x,f y)
```

We can define a structure-preserving transformation for basically anything, but when it comes to containers, the transformation becomes even more interesting, as it allows us to *perform operations on all the content* of the container at once.

For example, consider the `Option` datatype. `Option` is a container, that may contain at most one element. In this sense, its `map` function performs some operation on the content, thus either once or not at all, depending on the structure (`Some` or `None`) of the original input:

```
let mapOption f o = match o with None -> None | Some x -> Some(f x)
```

Notice that the structure-preservation guarantees that `mapOption` will always return `None` if the input was `None`, and `Some` if the input was `Some` (albeit with different content in the second case).

```
mapOption incr None -> None
mapOption incr (Some 3) -> Some 4
mapOption double (Some 3) -> Some 6
```

Similarly, we can extend the `map` concept to `List`. This means that we will transform all elements of the list, in a recursive fashion, according to the given function:

```
let rec mapList f l = match l with [] -> [] | x::xs -> (f x)::mapList f xs
```

The structure preservation can here be seen in the fact that the result of `mapList` always has the same number of list elements as the input list, meaning that `mapList` will never alter the length of the original list:

```
mapList incr [] -> []
mapList incr [1;3;5] -> [2;4;6]
mapList double [1;3;5] -> [2;6;10]
```

Composing multiple `map`'s

It is possible to combine different `map` functions together. We can combine them by simply invoking them in a nested fashion, by explicitly specifying the content transformation:

```
let mapListOption f = mapList (mapOption f)
```

Of course declaring a parameter `f` just to pass it along is not very pretty, as it clutters code without really adding anything interesting or particularly informative. Fortunately, function composition comes to the rescue, because we can simply compose (look out for the inverted order!) the two mapping functions as follows:

```
let mapListOption = mapOption >> mapList
```

Non-structure-preserving transformations: `filter`

`map` is not the only common design pattern in functional programming. Another transformation that frequently arises, but only when dealing with a dynamic collection, is `filter`, which removes elements from a collection based on some predicate.

`filter` on an `Option` would check whether or not there is an element, and it respects the predicate. In all other cases, the element is discarded from the collection:

```
let rec filterOption p l =
  match l with
  | Some x when p x -> Some x
  | _ -> None
```

Filtering a list requires checking all elements in turn, recursively:

```
let rec filterList p l =
  match l with
  | [] -> []
  | x::xs when p x -> x::filterList p xs
  | x::xs -> filterList p xs
```

Notice that, while `map` preserves structure (that is, the number of elements in a list), `filter` does not. Whereas `map` gives a result that might have a different content type (`List<int> -> List<string>`), `filter` will always preserve the type.

`filter` decides which elements to keep, and which elements to discard from a collection, so for example:

```
filterList (fun x -> x % 2 == 0) [1;2;3;4;5;6] -> [2;4;6]
filterList
  (fun (x:string) -> x.StartsWith "a")
    ["a"; "aa"; "baa"; "bb"] -> ["a"; "aa"]
```

`fold`

The most general design pattern, which only applies to data structures with variable length (just like `filter`), is `fold`. `fold` will apply a function to all elements in a sort of cascade, in order to aggregate them all together into a single result. A typical example of folding a sequence is adding all elements together, computing the minimum, computing the maximum, finding a specific element, etc.

`fold` is implemented on lists in two different ways, depending on the order in which we want to visit the elements (left-to-right or right-to-left):

```
let rec foldList z f l =
  match l with
  | [] -> z
  | x::xs -> f x (foldList z f xs)

let rec foldList2 z f l =
  match l with
  | [] -> z
  | x::xs -> foldList2 (f z x) f xs
```

We could, for example, use `fold` to add all elements as follows:

```
foldList 0 (+) [1;2;3] -> (1 + (2 + (3 + 0))) -> 6
```

or

```
foldList2 0 (+) [1;2;3] -> (((0 + 1) + 2) + 3) -> 6
```

> While for many operations it does not matter which version of `fold` we use (and thus, in those cases we should use `foldList2`, as it does not use the stack as much as `foldList` and can thus be compiled into a loop via *tail recursion optimisation*), sometimes the order in which the operations are executed really matters. Be on the lookout for these cases!

We could also compute the minimum of a list, but we must be careful: an empty list has no minimum, thus the result must be `Option`:

```
let minOption x y =
  match y with
  | Some a when a < y -> y
  | _ -> Some x

foldList None minOption [1;2;3] -> Some 1
```

`fold` is very powerful. When a datastructure supports the `fold` pattern, then we can build most (if not all!) of the functions we need on it via `fold` itself. For example, given that `List` supports `fold`, then we can define both `map` and `filter` on `List` via `fold`, instead of building both functions manually:

```
let mapList f = foldList [] (fun x xs -> f x::xs)
let filterList p l =
  foldList [] (fun x xs -> if p x then x::xs else xs)
```

Partial application, curry, and uncurry

Currying is the technique which lets functions return other functions. It is very commonly encountered in functional programming languages such as F#, whenever a function takes its parameters one at a time. For example, consider a function that adds two numbers together:

```
let add x y = x + y
```

The `add` function does not really take as input two numbers and returns their sum. Rather, it takes as input a single number, and returns the function that takes as input the second number and adds it to the first. This means that we could call `add` with only one input:

```
let incr = add 1
```

thereby obtaining the function that adds for example `1` to its input. Add is a *curried* function, and passing some of its parameters like we have just done to produce `incr` is called *partial application* ("partial" because we have provided *only a part* of its input, and not all of it).

We can also perform computations between the arguments. For example, we could define a function that determines what to do based on the value of one of the first parameters:

```
let conditionalPipeline p f g x = if p x then f else g
```

We would call this function as follows:

```
let f = conditionalPipeline (fun x -> x > 0) incr double
```

and then, depending on what we would give to `f`, see either `incr` or `double` happen:

```
f 3 -> 4
f -3 -> -6
```

A simple practical example could be drawn from the `draw` functions we built when presenting `repeat`. Instead of defining `star`, `space`, etc. manually, we could simply define them as the partial application of a curried `draw` function in which the symbol to draw is the first parameter:

```
let draw sym s = s + sym
let star = draw "*"
let space = draw " "
let newline = draw "\n"
```

Interestingly, we can automatically move between curried and uncurried functions, by defining the `curry` and `uncurry` higher order functions that transform an uncurried function that takes a tuple as input into a curried function that takes the inputs one at a time, and vice-versa:

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

Suppose now that we were provided with a small-minded draw function which, for reasons unknown to us (incompetence? malice? who knows!) is not built with currying and thus cannot be partially applied:

```
let drawWrong(sym,s) = s + sym
```

Fortunately, we can use `curry` to turn this function into its proper curried equivalent, and use it to define our utilities without ugly code repetition or the introduction of useless arguments that we only pass along or use just once in uninteresting fashions:

```
let draw = curry drawWrong
let star = draw "*"
let space = draw " "
let newline = draw "\n"
```

As a final example, consider the case of a `Person` record, plus a curried `rename` function:

```
type Person = { name:string; surname:string }
let rename newName p = { p with name=newName }
```

We can now define some (admittedly not very useful) utility functions that change the name of a `Person` to the desired name. These functions all expect a `Person`, thanks to the partial application of `rename` to its first parameter only:

```
let georgeify = rename "George"
let janeify = rename "Jane"
let jackify = rename "Jack"
```

We can then use these functions to rename some instances of `Person` as follows:

```
let p1 = { name="Giorgio"; surname="Ruffa" } |> johnify
let p2 = { name="Alex"; surname="Pomaré" } |> janeify
let p3 = { name="Clint"; surname="Eastwood" }
```

Function records

We can combine all we have seen so far in a powerful design pattern, known as type-classes, where we define a record of functions over some generic datatype in order to implement something that somewhat resembles an interface in object-oriented languages.

> Be careful! While it is true that type-classes can be used in order to simulate interfaces, they also support quite a lot more than object-oriented languages interfaces. For example, type-classes allow us to define static methods, constructors, and operators, which interfaces do not support because of their requirement of at least one instance of the implementing class (the instance that will become `this` in the concrete implementation).

Consider two conceptually similar datatypes, built separately and not necessarily overlapping in attribute names (but indeed overlapping in attribute *meaning*):

```
type Manager = {
  name:string
  surname:string
  birthday:DateTime
  company:string
  salary:int
}

type Student = {
  Name:string
  Surname:string
  Birthday:DateTime
  StudyPoints:int
}
```

We would now like to encode the fact that both of these data structures really represent some interface `Person`, which allows getting and setting `Name`, `Surname`, and `Birthday`. Other attributes cannot really be defined for `Person`, as they are not present in both `Manager` and `Student`. We encode this by defining a record of functions over a generic datatype `'p`, which is the actual (and at this point unknown) type of a `Person`:

```
type IPerson<'p> = {
  getName:'p -> string
  getSurname:'p -> string
  getBirthday:'p -> DateTime
```

```
    setName:string -> 'p -> 'p
    setSurname:string -> 'p -> 'p
    setBirthday:DateTime -> 'p -> 'p
  }
```

We then define the implementation of `IPerson` for a `Manager`, and for a `Student`:

```
  let managerPerson:IPerson<Manager> =
    {
      getName=fun (x:Manager) -> x.name;
      getSurname=fun (x:Manager) -> x.surname;
      getBirthday=fun (x:Manager) -> x.birthday;
      setName=fun v (x:Manager) -> { x with name=v };
      setSurname=fun v (x:Manager) -> { x with surname=v };
      setBirthday=fun v (x:Manager) -> { x with birthday=v }
    }

  let studentPerson:IPerson<Student> =
    {
      getName=fun (x:Student) -> x.Name;
      getSurname=fun (x:Student) -> x.Surname;
      getBirthday=fun (x:Student) -> x.Birthday;
      setName=fun v (x:Student) -> { x with Name=v };
      setSurname=fun v (x:Student) -> { x with Surname=v };
      setBirthday=fun v (x:Student) -> { x with Birthday=v }
    }
```

At this point, we could define a simple, but highly generic program that takes as input both a person and its interface implementation (we call `i` the *witness* of `'p` being a `Person`, which sounds kind of awesome) and does stuff with the input `Person`:

```
  let genericProgram (i:IPerson<'p>) (p:'p) =
    let p1 = i.setSurname ("'o" + i.getSurname p) p
    i.setBirthday (i.getBirthday p1 + System.TimeSpan.FromDays 365.0) p1
```

We can then call this generic program on both a `Student` and a `Manager`, by providing the actual witnesses `studentPerson` and `managerPerson` respectively:

```
  printfn "%A"
    (genericProgram studentPerson
      {
        Name="Jack"
        Surname="Lantern"
        Birthday=System.DateTime(2, 3, 1985)
        StudyPoints=120
      }
    )
  printfn "%A" (
    genericProgram managerPerson
    {
      name="John"
      surname="Connor"
      birthday=System.DateTime(2, 3, 1965)
      company="Microsoft"
      salary=150000
    }
  )
```

Composition of functions and types

Functions have a type. The function from `'a` to `'b` is called `'a -> 'b`. Types built with the function `->` operator can be mixed with all other type constructions.

For example, we could define a function that takes as input a list of curried functions, and invokes them all with the same arguments (thus multiple times). A first implementation to do this could be:

```
  let applyMany : (List<'a->'b->'c> -> 'a -> 'b -> List<'c>) = fun l a b ->
    l |> List.map (fun f -> f a)
      |> List.map (fun f -> f b)
```

Of course, there is no need to go through the list twice, so we can use `List.map` only once, to invoke the function with both parameters:

```
  let applyMany : (List<'a->'b->'c> -> 'a -> 'b -> List<'c>) = fun l a b ->
    l |> List.map (fun f -> f a b)
```

We can make the situation even more complex. We could assume that we have a list of function `Option`'s, that is some functions we have, and some we do not. In this case, we first collect all the functions we have (at most one per `Option`), and then we proceed with their actual invocation:

```
  let applyMany : (List<Option<'a->'b->'c>> -> 'a -> 'b -> List<'c>) = fun l a b ->
    l |> List.collect (
      function
        | None -> []
        | Some f -> [f]
      )
      |> List.map (fun f -> f a b)
```

From a `List` of `Option`'s of functions, we could also choose to only apply the functions when we have them, and to return None when no function was available. This means that, instead of a List of all possible results, we return a List with Option's that might contain the results:

```
let applyMany : (List<Option<'a->'b->'c>> -> 'a -> 'b -> List<Option<'c>>) = fun l
  l |> List.map (
    function None ->
      None
      | Some f -> Some(f a b))
```

We could also have the function itself give back a result which might not be there, leading us to two possible implementations, depending on whether or not we wish to filter away the absent functions or results:

```
let applyMany : (List<Option<'a->'b->Option<'c>>> -> 'a -> 'b -> List<'c>) = fun l
  l |> List.collect (
    function
      None -> []
      | Some f ->
          match f a b with
          | None -> []
          | Some x -> [x]
  )
```

or

```
let applyMany : (List<Option<'a->'b->Option<'c>>> -> 'a -> 'b -> List<Option<'c>>)
  l |> List.map (function None -> None | Some f -> f a b)
```

This goes on to show that functions really are just like all other datatypes in F#, and they can be freely mixed with all other types and all other tools we have defined so far.

> The ability to compose constructs according to our will, without artificial limitations imposed by the language, is crucial, and more often than not underestimated in its impact when building high quality software. Composition makes it possible to build complex abstractions and large programs by leveraging the power of all the tried-and-tested functions and other components that we have already built and tested. Few tools offer the ability that (pure, referentially transparent) functional languages have to build components in isolation, test them, compose them, and obtain flawlessly working programs as a direct result, leading beginning practitioners of functional programming to believe that "programming in [insert functional language here] simply *works*, without bugs!".

Exercises

Exercise 1

Implement a function

```
let mapFold (f : 'a -> 'b) (l : List<'a>) : List<'b> = ...
```

implementing `map` for lists using only `fold`.

Exercise 2

Implement a function

```
let filterFold (f : 'a -> bool) (l : List<'a>) : List<'a> = ...
```

implementing `filter` for lists using only `fold`.

Exercise 3

Implement a function

```
let flatten (l : List<List<'a>) : List<'a> = ...
```

that flattens a list of lists into a single list using `fold`.

Exercise 4

Implement a function

```
let map2 (f :  a -> 'b -> 'c) (l1 : List<'a>)
  (l2 : List<'b) : Option<List<'c>> = ...
```

that applies `f` to two lists of equal length `l1` and `l2`. If the two lists have different length it return `None`.

Exercise 5

Implement a function

```
let fold2 (f : 'state -> 'a -> 'b -> 'state) (init : 'state)
  (l1 : List<'a>) (l2 : List<'b>) : Option<'state> = ...
```

that folds two lists of equal length. If the length is different than None is returned.

Exercise 6

Implement a function

```
let zip (l1 : List<'a>) (l2 : List<'b>) : Option<List<'a * 'a>> = ...
```

that take two lists with the same length and creates a list of pairs containing the elements that are in the same position from both lists. Implement this function by using normal recursion and then by using `fold2`.

Exercise 7

Implement a function

```
let map2Safe (f : List<'a> -> List<'b> -> 'c) (l1 : List<'a>)
  (l2 : List<'b) : List<Option<'c>>
```

that applies the function `f` to the elements in the same position of two lists `l1` and `l2`, possibly with different length. If an element of one list does not have a correspondent element in the second list, then the function returns None.

Example: Summing the elements of `[1;2;3;4]` and `[4;5]` with `map2Safe` returns `[Some(5);Some(7);None;None]`