Functional Programming in F#

Quantitative Strategies
Paweł Borkowski
Maciej Flis

CREDIT SUISSE

Curring an partial application

```
// Here is a function with single, tuple paramter:
let addTuple (x, y) = x + y
```
*val addTuple : int * int -> int*

```
// Curried version of the function above: Each element of a
tuple is a separate parameter.
let add x y = x + y
```
val add : int -> int -> int

```
let addTwo = add 2
```
val addTwo : (int -> int)

```
let six = addTwo 4
```
val six : int = 6

Partial application

- `//evaluation is proceeded when all parameters are supplied`
- `let problem1 x y z = (y+z) / x`
- *val problem1 : int -> int -> int -> int*

- `let problem2 = problem1 0 // no error`
- *val problem2 : (int -> int -> int)*

- `let problem3 = problem2 1 // no error!`
- *val problem3 : (int -> int)*

- `let problem4 = problem3 2 // error!`
- *System.DivideByZeroException: Attempted to divide by zero.*

Composition

```
let inc x = x + 1 // int -> int
let double x = 2 * x // int -> int

// we can compose functions by applying first function to x
and pass the intermediate value explicitly it to second
function
let doubleInc x =
    let d = double x
    inc d
```

val doubleInc : int -> int

```
let compose f g x = g (f x)
```

val compose : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c

```
let doubleInc2 = compose double inc
```

> val doubleInc2 : (int -> int)

Composition 2

```
let incDoubleToStringAndDublicate = ?

// It may be cumbersome to use compose function so F# offers
bild-in >> operator:
(>>) : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
(<<) : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b

let doubleAndInc = double >> inc
```
val doubleAndInc : (int -> int)

```
let doubleAndInc2 = int << double
```
val doubleAndInc2 : (int -> int)

```
let doStuff = inc >> double >> toStr >> dublicate
```
val doStuff : (int -> string)

Options

```
// How we can express possibly missing value in F# ? - with
Discriminated union?!
// Example for Int
type IntOption =
    |SomeInt of int
    |NoneInt

// This is so common that F# has build in type Option
let someInt = Option.Some 42
val someInt : int option = Some 42

// Example with parsing
let parseDouble str =
    let result = ref 0.
    if System.Double.TryParse(str, result)
        then Some !result
        else None
```
val parseDouble : string -> float option

Options 2

```
let inverse d = // float -> float option
    match d with
    | 0.-> None
    | _ -> Some (1./d)

let doubleDouble x = x *2.0 // : float -> float

let x = Some 2.
```

val x : float option = Some 2.0

```
let doubleX = Option.map doubleDouble x
```

val doubleX : float option = Some 4.0

```
let inverseX = Option.bind inverse x
```

val inverseX : float option = Some 0.5

Options 3

```
let parseDoubleAndIverse str =
    let d = parseDouble str
    let dd = Option.map doubleDouble d
    Option.bind inverse dd
```
val parseAndDouble : string -> float option

```
let parseDoubleAndIverse2 =
      parseDouble >> Option.map doubleDouble >> Option.bind
  inverse
```
val parseDoubleAndIverse2 : (string -> float option)

```
let x = parseDoubleAndIverse2 "0.25"
```
val it : float option = Some 2.0

```
let z = parseDoubleAndIverse2 "0."
```
val z : float option = None

Pipe operator

```
//One of the most used operator in F# is  |>  called "pipe"
//used to chain function calls
let ( |> ) x f = f x
```

val (|>) : 'a -> ('a -> 'b) -> 'b

```
let min x y = if x < y then x else y
let sq x = x * x
let dbl x = x * 2

let calc x = dbl (min 10 (sq x))
let calc2 x = sq x |> min 10 |> dbl

let x = calc2 2
```

val it : int = 8

```
let x = calc2 4
```

val it : int = 20

Lazy evaluation

```
let lazyInt =
        lazy
           printfn "calculating"
           1
```
val lazyInt : Lazy<int> = Value is not created.

```
let lazyInt2 = Lazy.CreateFromValue 1
```
val lazyInt2 : System.Lazy<int> = Value is not created.

```
let lezyFunc x = Lazy.Create(fun () -> x * x)
```
val lezyFunc : int -> System.Lazy<int>

```
//force the computation
let unwrap = lazyInt.Value
```
calculating
val unwrap : int = 1

Arrays

```
//continues block of memory
//mutable
//elements need to be of the same type
//fast for lookup (constant time)
//expensive to extend (need to copy)
let ar = [|1;2;3;4|]
val ar : int [] = [|1; 2; 3; 4|]

let ar2 = [|1..4|]
val ar2 : int [] = [|1; 2; 3; 4|]

ar2.[2] <- 11
val it : unit = () // ar2 is equal to [|1; 2; 11; 4|]

//init elements
let ar3 : int array = Array.zeroCreate 5
val ar3 : int array = [|0; 0; 0; 0; 0|]
let ar4 = Array.create 5 1
val ar4 : int [] = [|1; 1; 1; 1; 1|]
```

Lists

```
//single-linked list in memory
//immutable
//elements need to inherit the same type
//slow to lookup (linear)
//fast to add at the head

//empty list
let e = []
val e : 'a list

//range
let l = [1..10]
val l : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

//increment
let l2 = [1..3..16]
val l2 : int list = [1; 4; 7; 10; 13; 16]

//with sequence expression
let l3 = [ for i in 1..10 -> i*i ]
val l3 : int list = [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```

Lists 2

```
//concat
let j = l @ l2
```

val j : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 1; 4; 7; 10; 13; 16]

```
//deconstruct with pattern matching
let rec print (a: int list) =
    match a with
    | [] -> ()
    | h :: tail -> printfn "%d" h; print tail
```

val print : int list -> unit

```
print [1..4]
```

1
2
3
4
val it : unit = ()

Sequences

```
//lazily evaluated collections, streams of data. Unlike list\arrays no alloc
//seq is an abbreviation for IEnumerable
//return elements with yield keyword (not always required)
let s1 = seq { for i in 1..100 do yield 2*i }
```

val s1 : seq<int>

```
let s2 = s1 |> Seq.iter (fun i -> printfn "%d" i)
```

2
4
…
val s2 : unit = ()

```
//yield! to flatten seq<seq<`a>>
let s3 = seq {yield 555 ; yield! s1 }
```

val s1 : seq<int>

```
//can be constructed from different structures
let fromArray = ar |> Seq.ofArray
//can be translated to a real structure:
let myList = s1 |> List.ofSeq
```

Sequences 2

```
//when seq is reified, only relevant entries are passed through the
computation when Seq calls are chained
//each of the element is looked up only once!
let s3 =
        Seq.initInfinite (fun i -> i)
        |> Seq.filter(fun n -> n%7 = 0)
        |> Seq.map(fun n -> n*n )
        |> Seq.take 20
        |> Seq.rev
        |> Seq.head

//operation above is evaluated in single step and "machinery" of
iterators is created that only access needed elements from an
infinite collection
//DO use Seq operations in the middle of the pipeline, this avoid
unnecessary allocations
//DONT return them into unknown context
```

Map

```
//key -> value mapping
//immutable, logarithmic time lookup
//easiest to create from a collection of tuples
let m1 =
        Map.ofSeq <|
            seq {
                   yield 1, 1
                   let a = 5
                   yield a, 6
                }
```

val m1 : Map<int,int> = map [(1, 1); (5, 6)]

```
//lookup syntax
m1.[5]
```

val it : int = 6

```
Map.tryFind 12 m1
```

val it : int option = None

Set

```
//unordered collection with no duplicates
//immutable
//easiest to create from a different collection

let set1 = Set.ofSeq [1;1;1;1;1]
```
val set1 : Set<int> = set [1]

```
let containsOne = set1.Contains 1
```
val containsOne : bool = true

```
let union = Set.union set1 (Set.ofSeq [2;3;4])
```
val union : Set<int> = set [1; 2; 3; 4]

```
let superset = set1.IsSupersetOf (Set.ofSeq [2;3;4])
```
val superset : bool = false

Exercises

1. Write your own implementation of List.fold and List.foldBack. Recursive if possible. Might use List.rev
2. Write a function SumAhead that sums list of int starting from first i<0 until the end of the list

   E.g. [3;3;-1;5;5;-2;6;4;-3;8;2] -> 24 ; [] -> 0 ; [1,2,3] -> 0

   Use List.fold, then use your own implementation to check if it works the same way.
3. Write a function SumBack that sums back list of int starting from last i<0 until the begining of the list.

   E.g. [3;3;-1;5;5;-2;6;4;-3;8;2] -> 20 ; [] -> 0 ; [1,2,3] -> 0

   Use List.foldBack, then use your own implementation to check if it works the same way.

Ad 2,3: Using Option int as State in fold might help. Return int (rather than Option int) from function SumAhead/SumBack.

4. Write a function to count number of non-white characters plus position in list of strings.

   Remove empty elements/whitespace elements from input list (System.String.IsNullOrWhiteSpace) to create new list.

   For every string count number of non-spaces and add position number (strating from 0) in a new list.

   ["   ala";"";"ma ";" ";"kota";"   "] -> 12 ; [] ->0 ; ["";" "] -> 0
5. Write permutation of words using list sequence expression and functions from Lecture #1

Reading materials

- https://fsharpforfunandprofit.com/series/thinking-functionally.html 5-7
- https://fsharpforfunandprofit.com/posts/list-module-functions/

F# |> I ❤