Functional Programming in F#

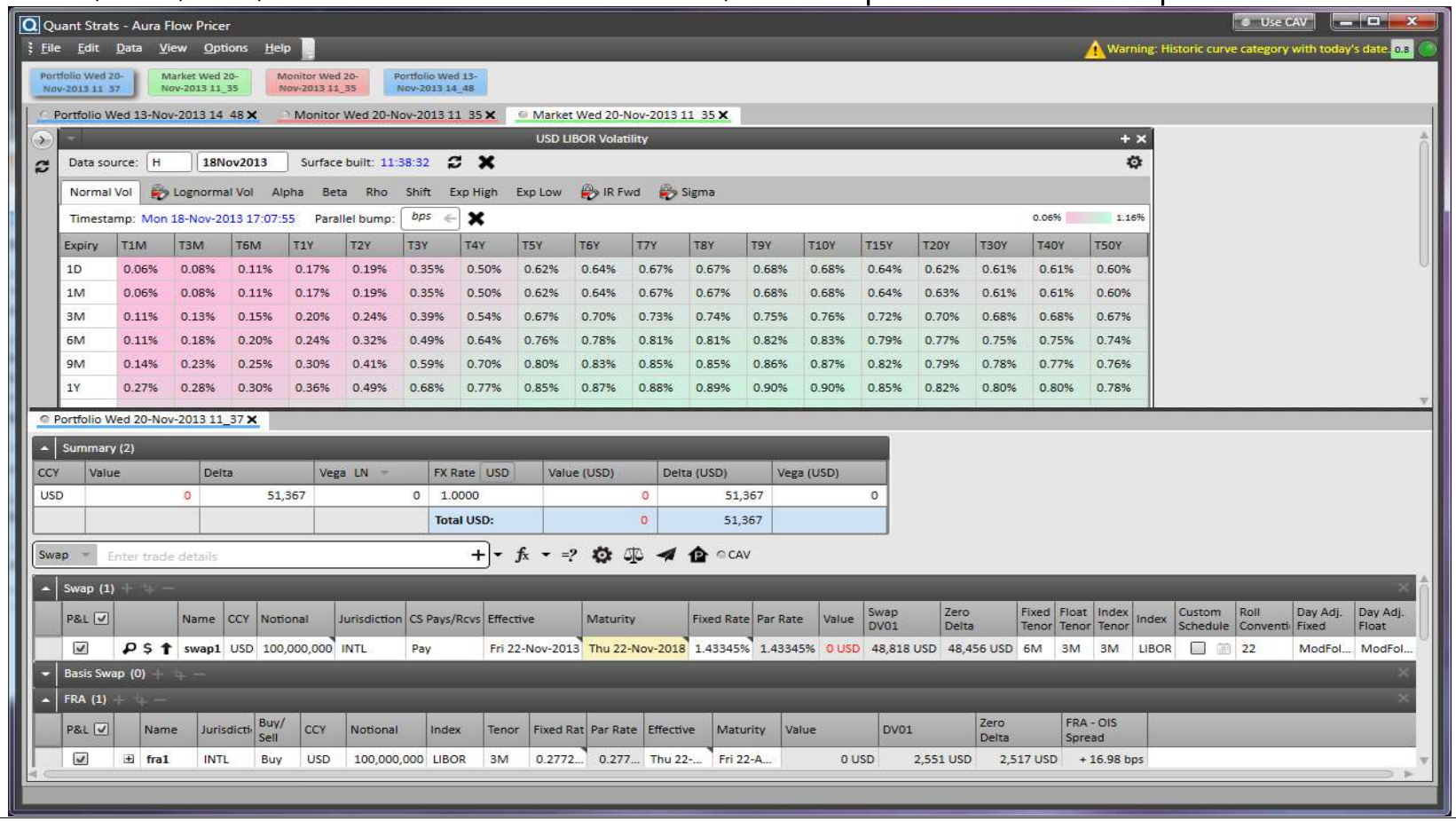
Quantitative Analysis and Technology

Zbigniew Fiałkiewicz
Artur Tadrała

CREDIT SUISSE

Quantitative Analysis and Technology

- Modeling, trading analytics, and risk management across the Investment Bank
- C++, C#, F#, 10 million lines of code, 1000 production components



Functional programming

- Your program is composition of mathematical functions

- Functions have arguments and a single result
- Functions do not have side effects
- Functions do not have state
- Functions are first-class citizens in F#

- Values cannot be mutated

- Aim for most of your program to be written in a functional manner

- Avoid mutating of objects\variables

Functional programming

It is a tool suitable for:

- Modelling mathematical equations
- Data-rich applications
- Concurrency

Why F#

Functional programming in a production environment

- (IDE, Intellisense, Debugger, Libraries)

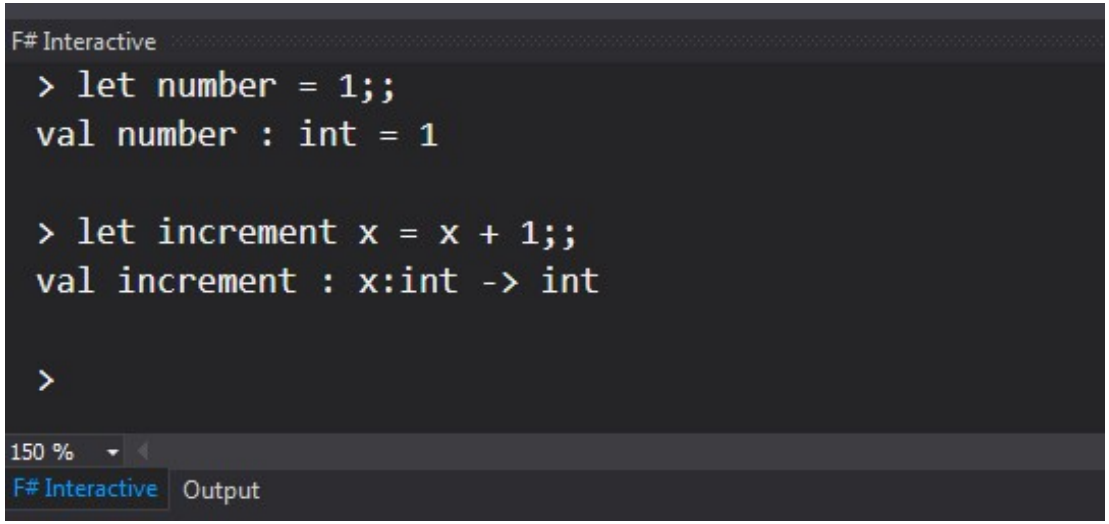
- Conciseness – no noise in the code
- Correctness – types prevent many common errors (no null ref ex)
- Convenience – optimizing for common tasks, scripts, REPL (**R**ead, **E**valuate, **P**rint **L**oop)
- Concurrency – asynchronous, thread safe by design
- Completeness – part of .net

[http://fsharpforfunandprofit.com/]

- Active community
- Credit Suisse Quant Strats is a big user of F#

F# Environment

- Online compilers (e.g. repl.it)
- Jetbrains Rider
- F# interactive(fsi) - REPL – Visual Studio Build Tools
- Visual Studio Code with Ionide plugin - free
- **Visual Studio Community** – free for students (https://visualstudio.microsoft.com/downloads/)
- Visual Studio Professional



Let binding – immutable value

```
let a = 1
```
val a: int = 1

```
let a = 1
a <- 2    // immutable, #ERROR
```

Let binding – one result
– indentation matters

```
let c =
      let a = 1
      let b = 2
      let d =
            let k = 5
            k
      a + b + d
```

val c: int = 8

Let binding - shadowing

```
let b =
      let a = 1
      let a = 2
      a
```

val b: int = 2

Basic types

```
let i = 1          // int
let l = 1L         // long
let f = 1.         // float
let d = 2m         // decimal
let b = true       // bool
let s = "text"     // string
let c = 'c'        // char
let u = ()         // unit, i.e. 'nothing'
```

Type inference

```
let a = 1
```
val a: int = 1

```
let a : int = 1 //annotation not needed
```
val a: int = 1

Type inference

```
let result =
      let a = 1
      let b = 2
      let result = a + b
      let str = string(result)
      "1 and 2 gives" + str
```

val result : string = "1 and 2 gives: 3"

```
let r= "1 and 2 gives" + string(1 + 2)
```

val r : string = "1 and 2 gives: 3"

Functions

```
let myfunc a = a + 1

let myfunc (a) = a + 1

let myfunc = fun a -> a + 1
```

val myfunc : a:int -> int

Functions – unit argument

```
let func () = printf "hello"
```

val func : unit -> unit

Functions – first class citizens

```
let func a = a + 1
```
val func : a:int -> int
```
let myfunc = func
```
val myfunc : a:int -> int

```
let outer =
    let inner a = a + 1
    inner
```
val outer: int -> int

```
let callFunc myFunc = myFunc(1)
```
val callFunc : myFunc:(int -> 'a) -> 'a

Functions – automatically generic

```
let identity b = b
```
val identity: 'a -> 'a

```
identity 1
```
val it : int = 1

```
identity "hello"
```
val it : string = "hello"

Lists

```
let l = [1;2;3]
```
val l : int list = [1; 2; 3]

```
let w = 4::[1;2;3]
```
val w : int list = [4; 1; 2; 3]

```
let x = [5..10]
```
val w : int list = [5; 6; 7; 8; 9; 10]

Example #1

Tuples

```
let t1 = (1,2)
```
*val t1 : int * int = (1, 2)*

```
let t3 = (1, t1, "T")
```
*val t3 : int * (int * int) * string = (1, (1, 2), "T")*

```
let (one,two) = t1
```
val two : int = 2
val one : int = 1

```
let (one,two,three) = t1
```
-> błąd!

Tuples

```
let long = (1,2,2.3)
```
*val longTuple : int * int * float = (1, 2, 2.3)*

```
let (x,y,_) = long
```
val y : int = 2
val x : int = 1

```
let add (a,b) = a + b
```
*val add : int * int -> int*

Tuple – usage, advantages, disadvantages

+ Return more then one value from a function
+ temporary, local variable
+ Piping (pipe operator |>)
+ Pattern matching of more then one value

- Hard to infere meaning of a tuple
- For longer tuples it’s possible to confuse elements

Records

```
type ZNumber = { real: float; img: float }
type GeoCoord= { lat: float; long: float }

let one  = { real=1.0; img=0.0 }
```
val one : ZNumber = {real = 1.0; img = 0.0;}

let r = one.real

```
let {real=r; img=i} = one
```
val r : float = 1.0
val i : float = 0.0

Records

```
let {real=rl; img=_} = one
```
val rl : float = 1.0

```
let {real=rl} = one
```
val rl : float = 1.0

```
let add {real=r1; img=i1} {real=r2; img=i2} =
  {
    real=r1 + r2;
    img =i1 + i2
  }
```
val add : ZNumber -> ZNumber -> ZNumber

Discriminated unions

```
type ValueOrError =
      | Value of int
      | Error of string

let v = Value 2
```

val v: ValueOrError = Value 2

```
let e = Error "failed"
```

val e: ValueOrError = Error “failed”

Discriminated unions

```
type Example =
        | NoParam
        | Tup of int*string*float
        | R of ZNumber
        | N of int
        | V of ValueOrError
        | D of System.DateTime
```

```
let v = V (Value 42)
```
val v : Example = V (Value 42)

Patter matching

```
let s n = match n with
          | 0 -> "zero!"
          | 1 -> "one!"
          | n when n < 100 -> string n
          | _ -> "BIG"
```

```
let rec sum l =
    match l with
    | [] -> 0
    | x::xs -> x + sum xs
```

val sum : int list -> int

```
sum [1;2;3]
```

val it : int = 6

Pattern match

```
match znumber with
| {real=r; img=0.0}-> string r
| {real=0.0; img=i}-> string i + "i"
| {real=r; img=i}-> string r + "+" + string i+ "i"

match (x, y) with
| 0,_ -> true
| _,0 -> true
| _   -> false

match n with
| 1 | 3 | 5 | 7 | 9 -> "odd"
| 0 | 2 | 4 | 6 | 8 -> "even"
| _ -> "dont know"
```

Example #2

Further reading

https://fsharpforfunandprofit.com/series/thinking-functionally.html: 1-4, 8-9

https://fsharpforfunandprofit.com/series/understanding-fsharp-types.html 1-2, 4-7

https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/lists

Exercises

(* You should aim to answer these exercises without using mutation *)

1. Using visual studio create a F# Library and inside a module 'Lecture1,
2. Define functions fst, mid, lst for 3-tuple that return first, middle and last element
3. Let tuple (x,y) : float*float be a 2D coordinate. Define functions:
    a) flipX v
    b) flipY v
    c) rotate v angle
    d) transpose v v
    e) isOrtogonal v v
4. Define a record containing: Name, Salary, Department.
    a. Department should be a Discriminated Union of IT, SALES, PR, HR
    b. Define a list containing 10 examples
    c. Define a function that prints only records for IT (HINT: use pattern matching)
    d. Define a function that sums salaries only for a specified department (argument)
5. Using recursion, define a function 'removeDups : list<int> -> list<int>' that removes duplicates from the provided list. Your solution should maintain the remaining lements in the same order as they were previously. It's fine for it to be slow (quadratic time). (HINT: use pattern matching to deconstruct list into head and tail)
6. Write removeDups2(as above) using List.foldBack