Functional Programming

Crash course, part 2

Jon Sporring

2019/07/08

UNIVERSITY OF COPENHAGEN

Updated Schedule

Time	Monday	Tuesday	Wednesday	Thursday	Friday
09:00-9:45	Introduction to F#	Solutions to assignments 1-3	Solutions to assignment 4-5 Report writing	Solutions to assignment 6 Open question session	Assignment 7
10:00-10:45	Assignments 1-3	Pattern matching	Assignment 6	Assignment 7	Assignment 7
10:00-11:55	Imperative programming in F#	Assignment 4	Assignment 6	Assignment 7	Assignment 7
14:00-14:45	Programming with recursion	Working with lists			
15:00-17:00	Assignments 1-3	Assignment 5			
20:00			Assignment 6 deadline		Assignment 7 deadline

Patterns, match-with

Patterns are rules to transform data

Patterns in let-bindings:

```
let v = (3.5, 4.1)
let (x, y) = v
printfn "%A consists of %f and %f" v x y
```

Patterns in argument-bindings:

```
let f (x, y) = sqrt (x*x+y*y)
printfn "%A has length %f" v (f v)
```

Patterns have many forms:

- Constants, e.g., 1
- Variable, e.g., m
- Tuple, e.g., (x,y)
- Wildcard, _

match-with:

- Evaluated from top down until a match
- Patterns must cover all cases
- All results must have the same type

```
/// faculty function using recursion
let rec facRec n =
  if n > 1 then
    n * facRec (n-1)
  else
    1

let n = 5
printfn "fac %d = %d" n (facRec n)
```

Patterns in match-with:

```
/// faculty function using patterns
let rec facPat n =
  match n with
    1 -> 1
    | m -> m * facPat (m - 1)

let n = 5
printfn "fac %d = %d" n (facPat n)
```

All cases covered but, what about facPat -1?

Patterns, match-with

All cases covered but,

```
/// faculty function using patterns
let rec facPat n =
  match n with
    1 -> 1
    | m -> m * facPat (m - 1)

let n = 5
printfn "fac %d = %d" n (facPat n)
```

Guards

```
/// faculty function using patterns
let rec facGua n =
  match n with
    m when m > 1 -> m * facGua (m - 1)
    | _ -> 1

let n = 5
printfn "fac %d = %d" n (facGua n)
```

Things on lists

Tuples:

```
> let a = (3, "three");;
val a : int * string = (3, "three")
```

Different types, size determined at the time of definition

Strenge:

```
> "Hello world".[6..];;
val it : string = "world"
```

Samme type (char), og operatorer til indicering og sammensætning

Lists:

Like strings, elements have to have identical type and are immutable

- Example:

```
> let a = ['h'; 'e'; 'l'; 'l'; 'o'];;
val a : char list = ['h'; 'e'; 'l'; 'l'; 'o']
```

Generic type

- The empty list:

```
> let a = [];;
val a : 'a list
```

Slicing (as strings)

- Indexing

```
> ['h'; 'e'; 'l'; 'l'; 'o'].[1..];;
val it : char list = ['e'; 'l'; 'l'; 'o']
```

'Alphabetic' comparison

- Comparison

```
> [2; 3; 5] > [2; 2; 6];;
val it : bool = true
```

Concatenation of lists

- Append (@)

```
> ['h'; 'e'; 'l'; 'l'; 'o'] @ [' '; 'w'; 'o'; 'r'; 'l'; 'd'];;
val it : char list = ['h'; 'e'; 'l'; 'l'; 'o'; ' '; 'w'; 'o'; 'r'; 'l'; 'd']
```

Prepend an element

- Cons (::)

```
> 'h' :: ['e'; 'l'; 'l'; 'o'];;
val it : char list = ['h'; 'e'; 'l'; 'l'; 'o']
```

Concatenation is slow, Cons is fast

listAppendLarge.fsx

```
let mutable lst = []
for i = 1 to 40000 do
  lst <- lst @ [i]
```

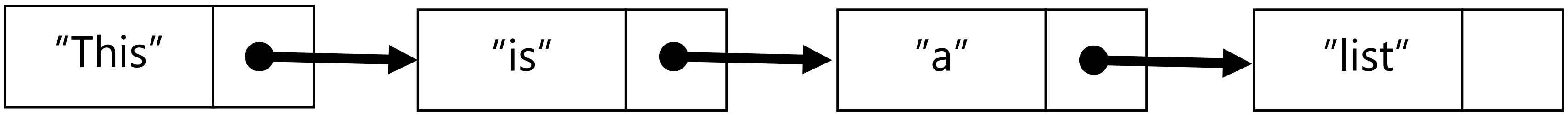
Demo: listAppendLarge.fsx

listConsLarge.fsx

```
let mutable lst = []
for i = 1 to 40000 do
  lst <- i :: lst
```

Demo listConsLarge.fsx

Lists are linked lists:



List library: init and iter

Creation of lists

```
> let a = List.init 5 (fun i -> i)
- let b = [0..4];;
```

Iterating over lists

```
> let a = List.init 5 (fun i -> pown 2 i)
- for i = 0 to a.Length - 1 do
-   printf "%d " a.[i]
- printfn "";;
```

Better iteration over lists

```
> let a = List.init 5 (fun i -> pown 2 i)
- List.iter (fun e -> printf "%d " e) a
- printfn "";;
```

List patterns

```
/// faculty function using patterns
let rec prodLst lst =
  match lst with
    elm :: rest -> elm * prodLst rest
    | [] -> 1

let l = List.init 5 (fun i -> i+1)
printfn "prod %A = %d" l (prodLst l)
```

Another example

```
/// reverse a list using patterns
let rec rev lst =
  match lst with
    elm :: rest -> (rev rest) @ [elm]
    | [] -> []

let l = List.init 5 (fun i -> i+1)
printfn "rev %A = %A" l (rev l)
```

Demo

List library: map and fold

List.map: f:('T -> 'U) -> lst:'T list -> 'U list

Apply a function to every element of a lists as

lst = [1;2;3] => [f 1; f 2; f 3]

```
> let lst = [1..5]
- List.map (fun e -> e * e) lst;;
```

List.fold: f:('State -> 'T -> 'State) -> elm:'State -> lst:'T list -> 'State

Fold a list into a single entity as

lst = [1;2;3] og elm = 0 => f (f (f 0 1) 2) 3

```
> let lst = [1..5]
- let prod acc elm = acc * elm
- List.fold prod 1 lst;;
```

prodFold.fsx

```
let lst = [1..5]
let prod acc elm =
  let res = acc * elm
  printfn "acc = %A, elm = %A, res = %A" acc elm res
  res
printfn "prod %A = %A" lst (List.fold prod 1 lst)
```

List library: more fold

List.fold: f:('State -> 'T -> 'State) -> elm:'State -> lst:'T list -> 'State

Fold works well with types, e.g., concatenate list of int to a string.:

appFold.fsx

```
let lst = [1..5]
let app acc elm =
  let res = acc + (string elm)
  printfn "acc = %A, elm = %A, res = %A" acc elm res
  res
printfn "app %A = %A" lst (List.fold app "" lst)
```

Ellegant reversal of a list:

revFold.fsx

```
let lst = [1..5]
let rev acc elm =
  let res = elm :: acc
  printfn "acc = %A, elm = %A, res = %A" acc elm res
  res
printfn "rev %A = %A" lst (List.fold rev [] lst)
```