

# Programmering og Problemløsning, 2017

## Programmering med Lister og Arrays

Martin Elsman

Department of Computer Science  
University of Copenhagen  
DIKU

October 3, 2017

## 1 Programmering med Lister (continued)

- Definition af Lister
- List modulet
- Beregninger på lister

## 2 Programmering med Arrays

- 1D Arrays
- 2D Arrays

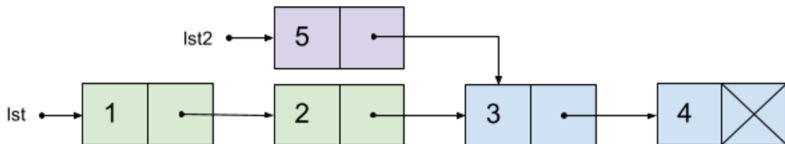
## Repræsentationen af lister

### ■ Syntax:

```
let lst = [1;2;3;4]
```

```
let lst2 = 5 :: List.tail (List.tail lst)
```

### ■ Lagerrepræsentation:



- Det er nemt at hægte et ekstra element på starten af en liste (`::`).
- Det er **IKKE** nemt (læs: hurtigt) at tilgå det sidste element i en liste.
- Lister er *immutable*, dvs elementer kan ikke opdateres.
- Hvorfor kan immutabilitet være godt?

## Modulet `List`

Modulet `List` indeholder en lang række operationer på lister.

### *// list creation*

```
val init      : int -> (int -> 'a) -> 'a list
val length   : 'a list -> int    // length l = l.Length
```

### *// list transformers*

```
val map       : ('a -> 'b) -> 'a list -> 'b list
val map2     : ('a->'b->'c) -> 'a list -> 'b list -> 'c list
val filter   : ('a -> bool) -> 'a list -> 'a list
```

### *// list traversing*

```
val fold     : ('s -> 'a -> 's) -> 's -> 'a list -> 's
val foldBack : ('a -> 's -> 's) -> 'a list -> 's -> 's
val find     : ('a -> bool) -> 'a list -> 'a option
...

```

## Listefoldninger – fold og foldBack

Listefoldninger er generiske funktioner der gør det muligt at gennemløbe en liste for samtidig at foretage beregninger på elementerne, f.eks. for at opbygge en ny datastruktur.

**val** fold : ('s -> 'a -> 's) -> 's -> 'a list -> 's

$$\begin{aligned} \text{fold } f \ s \ [x_0; x_1; x_2; \dots; x_n] \\ = f \ \dots \ (f \ (f \ (f \ s \ x_0) \ x_1) \ x_2) \ \dots \ x_n \end{aligned}$$

**val** foldBack : ('a -> 's -> 's) -> 'a list -> 's -> 's

$$\begin{aligned} \text{foldBack } f \ [x_0; x_1; x_2; \dots; x_n] \ s \\ = f \ x_0 \ (f \ x_1 \ (f \ x_2 \ \dots \ (f \ x_n \ s) \ \dots)) \end{aligned}$$

## Eksempel: summation af elementerne i en liste

```
let sum = List.fold (+) 0 [3;6;2;5]

// = (((0 + 3) + 6) + 2) + 5
// → ((3 + 6) + 2) + 5
// → (9 + 2) + 5 → 11 + 5 → 16
```

## Eksempel: Find det mindste element i en liste

```
let min x y = if x < y then x else y
let maxInt = System.Int32.MaxValue // = 2147483647
let min_elem = List.fold min maxInt [3;6;2;5]

// = min (min (min (min 2147483647 3) 6) 2) 5
// → min (min (min 3 6) 2) 5
// → min (min 3 2) 5 → min 2 5 → 2
```

## Spørgsmål:

- 1 Kunne man også have benyttet `List.foldBack`?
- 2 Hvorfor tager `List.fold` og `List.foldBack` et initielt element?

## Eksempel: Det mindste element i en liste med `List.foldBack`

```

let min x y = if x < y then x else y
let maxInt = System.Int32.MaxValue // = 2147483647
let min_elem = List.foldBack min maxInt [3;6;2;5]

// = min 3 (min 6 (min 2 (min 5 2147483647)))
// → min 3 (min 6 (min 2 5))
// → min 3 (min 6 2)
// → min 3 2
// → 2
    
```

## Funktionen `min` er associativ og `maxInt` er det neutrale element:

- 1 For alle  $x, y, z$ :  

$$\min x (\min y z) = \min (\min x y) z$$
- 2 For alle  $x$ :  

$$\min 2147483647 x = \min x 2147483647 = x$$

## Eksempel: Reverser en liste

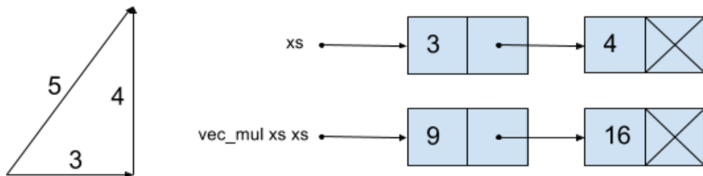
```
let f s x = x :: s
let rev xs = List.fold f [] xs
```

```
let ex = rev [1;2;3]
```

```
// = f (f (f [] 1) 2) 3
// → f (f (1 :: []) 2) 3 → f (2 :: 1 :: []) 3
// → 3 :: 2 :: 1 :: []
```



## Eksempel: dot-produktet og vectorlængde



```
let vec_mul (xs:float list) ys = List.map2 (*) xs ys
let dot xs ys = List.fold (+) 0.0 (vec_mul xs ys)
let vec_len xs = sqrt (dot xs xs)
let ex = vec_len [3.0; 4.0]
```

```
// = sqrt (List.fold (+) 0.0 (vec_mul [3.0; 4.0] [3.0; 4.0]))
// ~> sqrt (List.fold (+) 0.0 [9.0; 16.0])
// ~> sqrt 25.0
// ~> 5.0
```

## Funktionen `List.find`

```
val find : ('a -> bool) -> 'a list -> 'a option
```

Udtrykket `(find p xs)` returnerer `(Some x)` hvis `x` er det første element i `xs` for hvilket `(p x)` evaluerer til `true`. Udtrykket returnerer `None` hvis der ikke findes et sådan element.

## Implementation af `List.find` ved brug af `List.fold`

```
let find p xs =
  List.fold (fun s x -> if s = None && p x then Some x else s)
           None xs
```

```
// find (fun x -> x > 4) [3;2;5;6;45]
// ~> 5
```

## For-in løkker—et alternativ til `List.fold`

Et alternativ til `List.fold` er at benytte den specielle **for-in-do** syntax til liste-gennemløb:

```
let lst = List.init 50000 (fun x -> x)
let mutable sum = 0
for x in lst do sum <- sum + x
do printf "%d\n" sum
```

### Fordele:

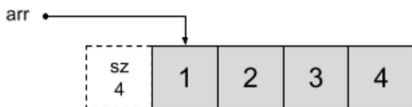
- 1 Ingen overhead ved liste-indicering eller kald til `lst.Length`.
- 2 Simplere form for imperativ programmering; dog er `sum` stadig mutable...

## Repræsentationen af arrays

- Syntax:

```
let arr = [|1;2;3;4|]
```

- Lagerrepræsentation:



- Det er **IKKE** nemt at tilføje ekstra elementer.
- Det er nemt (hurtigt) at læse ethvert element i et array.
- Arrays er *mutable*, dvs det er muligt (hurtigt) at opdatere ethvert element.
- Typen for integer arrays: `int[]`
- Typen for arrays indeholdende int-float par: `(int*float)[]`

## Gennemløb af arrays

Arrays kan gennemløbes tilsvarende som lister:

```
let arr = Array.init 50000 (fun x -> x)
let mutable sum = 0
for x in arr do sum <- sum + x
do printf "%d\n" sum
```

## Arrays kan muteres:

```
let arr = [|1;2;3;4|]
for i in [0..arr.Length-1] do arr.[i] <- arr.[i]*arr.[i]
do printf "%A\n" arr
```

## Kørsel:

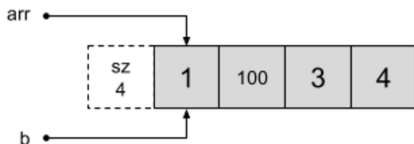
```
bash-3.2$ fsharpc --nologo arr_square.fs && mono arr_square.exe
[|1; 4; 9; 16|]
```

## Array aliasing

To forskellige variabler kan referere til det samme array:

```
let arr = [|1;2;3;4|]
let b = arr
do b[1] <- 100
do printf "%A\n" arr
```

## Lagerrepræsentation:



## Bemærk:

- 1 Aliasing kan observeres fordi arrays er mutable!
- 2 Aliasing (og deling) kan ikke observeres (på samme måde) med lister, da disse er immutable.

## Modulet `Array`

### *// array creation*

```

val init      : int -> (int -> 'a) -> 'a []
val length   : 'a [] -> int    // length a = a.Length
val toList   : 'a [] -> 'a list
val ofList   : 'a list -> 'a []
    
```

### *// array transformers*

```

val map      : ('a -> 'b) -> 'a [] -> 'b []
val map2    : ('a->'b->'c) -> 'a [] -> 'b [] -> 'c []
val filter  : ('a -> bool) -> 'a [] -> 'a []
    
```

### *// array traversing*

```

val fold     : ('s -> 'a -> 's) -> 's -> 'a [] -> 's
val foldBack : ('a -> 's -> 's) -> 'a [] -> 's -> 's
val find     : ('a -> bool) -> 'a [] -> 'a option
...
    
```

## To-dimensionelle arrays

To-dimensionelle *regulære arrays*, dvs. 2d-arrays hvor alle rækker indeholder det samme antal elementer, kan udtrykkes ved brug af modulet `Array2D`.

Modulet `Array2D` kan benyttes til f.eks. at konstruere en multiplikationstabel:

```
let a = Array2D.init 5 5 (fun r c -> (r+1) * (c+1))
let prA (a : int[,]) =
    for r in [0..Array2D.length1 a - 1] do // 1 2 3 4 5
        for c in [0..Array2D.length2 a - 1] do // 2 4 6 8 10
            printf "%2d " (a.[r,c]) // 3 6 9 12 15
        printf "\n" // 4 8 12 16 20
    do prA a // 5 10 15 20 25
```

### Bemærk:

- 1 Typen på et to-dimensionelt int-array skrives: `int[,]`.
- 2 Vidden på de udskrevne heltal kontrolleres med format-specifieren `"%2d "`.