Programmering og Problemløsning, 2017

Programmering med Lister

Martin Elsman

Department of Computer Science
University of Copenhagen
DIKU

September 27, 2017

F# Collections

Vi har ofte behov for at håndtere data vi ikke kender størrelsen af på forhånd.

F# tilbyder en række collection moduler til håndtering af data:

Eksempler:

- Strenge af karakterer (`String` modulet)
- Lister af tal (`List` modulet)
- Mængder af navne (`Set` modulet)
- Afbildninger af navne til telefonnumre (`Map` modulet)
- Muterbare arrays af floats (`Array` modulet)
- ...

Bemærk: Til forskel fra de strukturer vi har set ind til nu (såsom tupler) kan F# collections bestå af et ikke på forhånd defineret antal elementer.

Lister

En liste er en sekvens af elementer af samme type, men hvor antallet af elementer ikke nødvendigvis er kendt på forhånd.

- Tilsvarende som for antallet af tegn i en streng.
- I modsætning til antallet af elementer i et tuple.

Eksempler:

Udtryk	:	Type
`[3; 4; 5]`	:	`int list`
`['h'; 'e'; 'l'; 'l'; 'o']`	:	`char list`
`[true]`	:	`bool list`
`[]`	:	`'a list`

Indicering i og bygning af lister

Vi kan bruge dot-notation til at tilgå dele af lister (ligesom for strenge):

```
['a'; 'e'; 'i'; 'o'; 'u'; 'y'].[2]     = 'i'
['a'; 'e'; 'i'; 'o'; 'u'; 'y'].[2..4]  = ['i'; 'o'; 'u']
['a'; 'e'; 'i'; 'o'; 'u'; 'y'].[2..14] = error message
```

List append (@)

```
['a'; 'e'] @ ['i'; 'o'] = ['a'; 'e'; 'i'; 'o']
              [] @ [] = []
```

List cons (::)

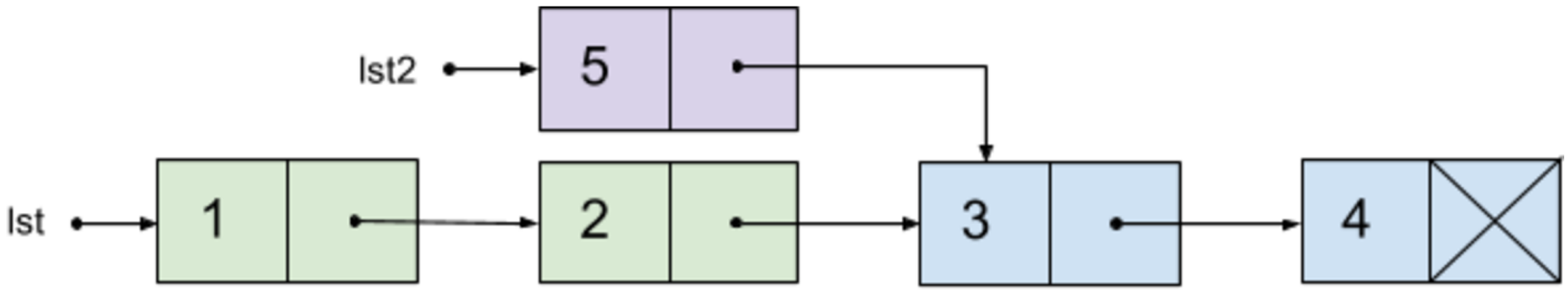
```
      1 ::  [2; 3] = [1; 2; 3]
      false ::  [] = [false]
1.2 ::  2.3 ::  [] = [1.2; 2.3]
        [] ::  [] = [[]]
```

Repræsentationen af lister

- Syntax:

```
let lst = [1;2;3;4]
let lst2 = 5 :: List.tail (List.tail lst)
```

- Lagerrepræsentation:



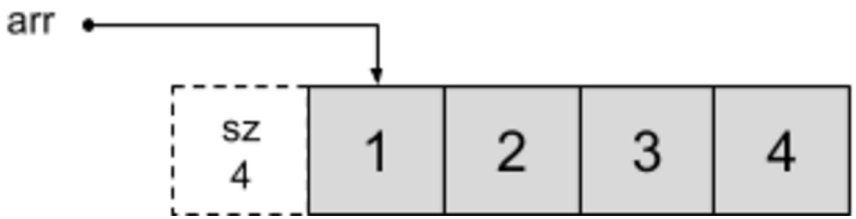
- Det er nemt at hægte et ekstra element på starten af en liste (`::`).
- Det er **IKKE** nemt (læs: hurtigt) at tilgå det sidste element i en liste.
- Lister er *immutable*, dvs elementer kan ikke opdateres.
- Hvorfor kan immutabilitet være godt?

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.

Basale listeoperationer

```
// Immediate lists
let nums = [1;2;3;4]                    // 1 :: 2 :: 3 :: 4 :: []
let caps = [("London",8.8);             // pairs of city name and
            ("Berlin",3.5);             //  population (mill)
            ("Copenhagen",0.7)]
```

Listekonstruktører og append (@)

```
val []     : 'a list                          // empty list
val ::     : 'a -> 'a list -> 'a list         // add element
val @      : 'a list -> 'a list -> 'a list    // append lists
```

Eksempler

```
let allcaps = ("New York",8.5) :: ("Rome",2.9) :: caps
let nums2 = nums @ [100;200]
```

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
...
```

Dynamisk konstruktion af lister

Funktionen `List.init` gør det muligt at opbygge en liste dynamisk fra bunden:

Eksempel

```
let sz = 2 + 3
let lst = List.init sz (fun x -> x * 2 + 1)

//  = [0*2+1; 1*2+1; 2*2+1; 3*2+1; 4*2+1]
// ⇝  [1; 3; 5; 7; 9]
```

Transformation af lister – map og map2

```
val map : ('a -> 'b) -> 'a list -> 'b list
```

$$\texttt{map f } [v_0; v_1; v_2; \ldots; v_n] = [\texttt{f } v_0; \texttt{f } v_1; \texttt{f } v_2; \ldots; \texttt{f } v_n]$$

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

$$\texttt{map2 f } [a_0; a_1; a_2; \ldots; a_n]\ [b_0; b_1; b_2; \ldots; b_m] = [\texttt{f } a_0\ b_0; \texttt{f } a_1\ b_1; \texttt{f } a_2\ b_2; \ldots; \texttt{f } a_n\ b_n] \quad \texttt{// if n=m}$$

$$= \textit{error} \quad \texttt{// if n<>m}$$

Eksempler

```
let vs = List.map (fun x -> x+1) [10; 20; 30]

//  = [10+1; 20+1; 30+1]  ⇝  [11; 21; 31]

let us = List.map2 (+) [10; 20; 30] [1; 2; 3]

//  = [10+1; 20+2; 30+3]  ⇝  [11; 22; 33]
```

Transformation af lister — `List.filter`

```
val filter : ('a -> bool) -> 'a list -> 'a list
```

Udtrykket (`List.filter p xs`) resulterer i en liste indeholdende de elementer i `xs` der opfylder prædikatet `p`.

Eksempel

```
let allcaps = [("London",8.8); ("Berlin",3.5);
               ("Copenhagen",0.7); ("New York",8.5);
               ("Rome",2.9)]
let bigcaps = List.filter (fun (name,sz) -> sz > 5.0) allcaps

// ⇝ [("London",8.8);("New York",8.5)]
```

Beregninger på lister

Her er noget kode for en uheldig implementation af listesummation:

```
let lst = List.init 50000 (fun x -> x)          // BAD
let mutable i = 0                               // BAD
let mutable sum = 0                             // BAD
while (i < lst.Length) do                       // BAD
  sum <- sum + lst.[i]                          // BAD
  i <- i + 1                                    // BAD
printf "%d\n" sum                               // BAD
```

Tre problemer:

1. _____
2. _____
3. _____

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
```

```
fold f s [x0;x1;x2;...;xn]
  = f ... (f (f (f s x0) x1) x2) ... xn
```

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

```
foldBack f [x0;x1;x2;...;xn] s
  = f x0 (f x1 (f x2 ...(f xn s)...))
```

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 min_elem = List.fold min 1000 [3;6;2;5]

//   = min (min (min (min 1000 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?

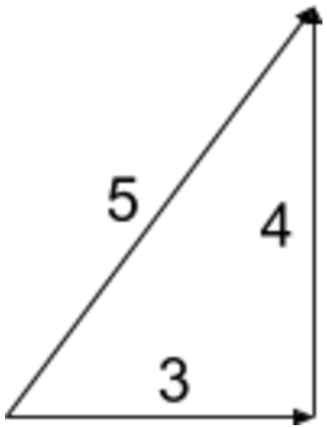
Eksempel: Reverser en liste

```
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
```