Programmering og Problemløsning, 2019

Programmering med Lister (Del 2)

Martin Elsman

Department of Computer Science
University of Copenhagen
DIKU

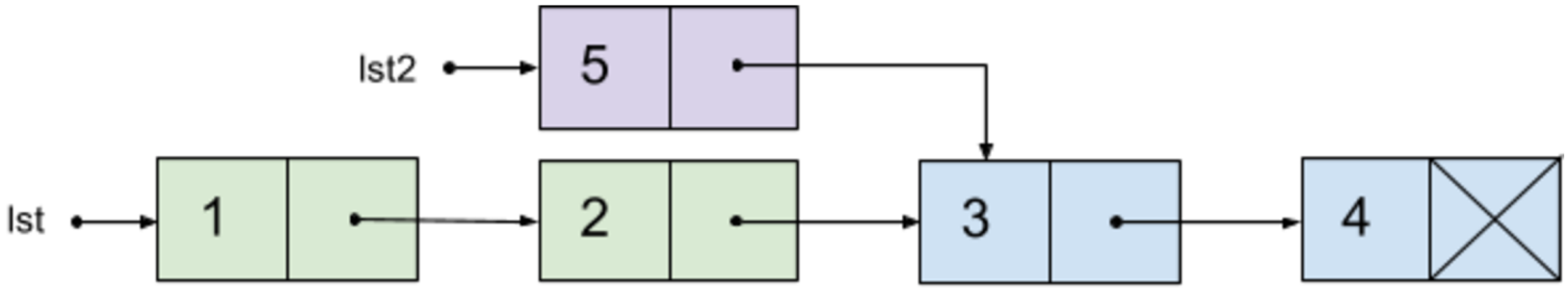
3. oktober, 2019

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?

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
```

```
map f [v0; v1; v2; ...; vn]
  = [f v0; f v1; f v2; ...; f vn]
```

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

```
map2 f [a0; a1; a2; ...; an] [b0; b1; b2; ...; bm]
  = [f a0 b0; f a1 b1; f a2 b2; ...; f an bn]   // if n=m
  = error                                       // 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:

```
// bad_summation.fs
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. _____

Oversættelse og kørsel af `bad_summation.fs`

```
// bad_summation.fs
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
```

Oversættelse og kørsel

```
bash-3.2$ fsharpc --nologo bad_summation.fs
bash-3.2$ time mono bad_summation.exe
1249975000

real 0m8.112s
```

En grim løsning på listesummation

```
// ugly_summation.fs
let lst = List.init 50000 (fun x -> x)      // UGLY
let mutable i = 0                           // UGLY
let mutable sum = 0                         // UGLY
let len = lst.Length                        // UGLY
let mutable lst2 = lst                      // UGLY
while (i < len) do                          // UGLY
  sum <- sum + List.head lst2               // UGLY
  lst2 <- List.tail lst2                    // UGLY
  i <- i + 1                                // UGLY
printf "%d\n" sum                           // UGLY
```

Oversættelse og kørsel

```
bash-3.2$ fsharpc --nologo ugly_summation.fs
bash-3.2$ time mono ugly_summation.exe
1249975000

real 0m0.078s
```

Gennemløb af lister med `for-in-do` konstruktionen

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

```
// better_summation.fs
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...

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
```

$$\texttt{fold f s } [x_0;x_1;x_2;\ldots;x_n] = \texttt{f } \ldots \texttt{ (f (f (f s } x_0\texttt{) } x_1\texttt{) } x_2\texttt{) } \ldots \texttt{ } x_n$$

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

$$\texttt{foldBack f } [x_0;x_1;x_2;\ldots;x_n] \texttt{ s} = \texttt{f } x_0 \texttt{ (f } x_1 \texttt{ (f } x_2 \texttt{ ...(f } x_n \texttt{ s)...))}$$

Husk:

- En funktion kaldes først når argumenterne er evalueret til værdier!
- Dette princip kaldes “Call-by-value”.

Spørgsmål:

1. Hvorfor er typerne for `List.fold` og `List.foldBack` forskellige?

Eksempel: summation af elementerne i en liste

```
  let sum = List.fold (+) 0 [3;6;2;5]  // See also
                                       // good_summation.fs
//   = (((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`