

# Programmering og Problemløsning, 2017

## Typer og Mønstergenkendelse

Martin Elsman

Datalogisk Institut  
Københavns Universitet  
DIKU

23. Oktober, 2017

# 1 Typer og Mønstergenkendelse

- Typer
- Typeforkortelser
- Records
- Mønstergenkendelse
- Selection Sort–genbesøgt

## Typer

*Det er ofte hensigtsmæssigt at forstå typer som mængder af værdier.*

I dag vil vi se på type-begrebet, på hvordan vi kan udvide begrebet til at klassificere flere slags værdier og på hvordan vi kan skrive genbrugelig type-invariant kode (generiske funktioner).

Vi vil også se på forholdet mellem konstruktion og dekonstruktion af værdier. Specifikt vil vi se på begrebet mønstergenkendelse (pattern matching).

- 1 **Typer kan forstås som mængder.**
- 2 **Type-forkortelser (type-abbreviations).**
- 3 **Typer for generisk kode.**
- 4 **Generiske type-forkortelser.**
- 5 **Mønstergenkendelse (pattern matching).**
- 6 **Selection-sort – revisited.**

## Typer kan forstås som mængder af værdier

### Eksempler:

<code>int</code>	$\approx$	$\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$
<code>float</code>	$\approx$	$\mathbb{R}$
<code>int * float</code>	$\approx$	$\mathbb{Z} \times \mathbb{R} = \{\dots, (3, 1.2), \dots\}$ (sæt af alle par med elementer fra $\mathbb{Z}$ og $\mathbb{R}$ )
<code>int list</code>	$\approx$	$\{\ [], [1], [2], \dots, [1; -2], \dots\}$
<code>int -&gt; float</code>	$\approx$	$\mathbb{Z} \rightarrow \mathbb{R}$ (sæt af alle afbildninger fra $\mathbb{Z}$ til $\mathbb{R}$ )
<code>bool</code>	$=$	$\{\text{true}, \text{false}\}$
<code>unit</code>	$=$	$\{0\}$

### Spørgsmål:

- 1 Hvorfor  $\approx$  for de første fire tilfælde?
- 2 Hvorfor  $\approx$  for funktioner?

## Typeforkortelser

Det er nemt i F# at give et navn til en type for derved at gøre kode lettere læselig.

```
type department = string
type costs = (department * float) list
let total (costs:costs) : float =
    List.fold (fun acc (_,f) -> acc+f) 0.0 costs
```

### Bemærk:

- Typen department er blot et synonym for typen string.
- Funktionen total kan derfor benyttes på alle værdier af typen (string\*float) list.

## Type-generiske type-forkortelser

Type-forkortelser kan være generiske således at det er muligt at skrive generisk kode der henviser til en type-forkortelse:

```
// association lists mapping strings to values of type 'a  
type 'a alist = (string * 'a) list  
let add (m:'a alist) (s:string) (v:'a) : 'a alist =  
  (s,v)::m  
let rec look (m:'a alist) (s:string) : 'a option =  
  if List.isEmpty m then None  
  else if fst(List.head m) = s then Some(snd(List.head m))  
  else look (List.tail m) s  
let empty () : 'a alist = []
```

### Bemærk:

- Vi kan benytte den tomme liste [] til at repræsentere den tomme associationsliste.
- Vi skal senere se hvorledes vi med moduler kan sikre at typen 'a alist bliver "fuldt abstrakt" således at kun de nævnte funktioner kan benyttes til at operere på de konstruerede associationslister.

## Record-typer

Records i F# giver mulighed for at navngive elementer i et tuple.

Syntaksen for at definere en record-type er ganske enkelt:

```
type person = {first:string; last:string; age:int}

let xs = [{first="Lene"; last="Andersen"; age=56};
          {last="Hansen"; first="Jens"; age=39}]
let name (p:person) : string = p.first + " " + p.last
let incr_age (p:person) : person = {p with age=p.age+1}
let ys = List.map incr_age xs
```

### Bemærk:

- Ved konstruktion af en record er felt-rækkefølgen ubetydelig.
- Elementer i en record kan udtrækkes ved brug af **dot-notationen** (`p.first`).
- En **ny** record kan konstrueres (med et opdateret element) ved brug af **with**-konstruktionen.

## Mønstergenkendelse (Pattern matching)

Generelt set giver mønstergenkendelse mulighed for at **undersøge** og **nedbryde** en værdi i dens bestanddele.

Vi vil se på mønstergenkendelse ud fra typen på de værdier vi undersøger.

I F# kan mønstergenkendelse optræde i flere forskellige programkonstruktioner:

- I simple **let**-bindinger.
- I **match-with**-konstruktioner.
- I funktionsparametre.



## Mønstergenkendelse på Tupler

```
let x = (34, "hej", 2.3) // construct triple
let (_, b, f) = x       // use of wildcard (_)
do printfn "%s:%f" b f  // b and f are available here
```

## Mønstergenkendelse på Records

```
type person = {first:string; last:string; age:int}
let name ({first=f;last=l}:person) = f + " " + l
```

### Bemærk:

- 1 Matching på records kræver blot at et udvalg af felt-navne er nævnt.
- 2 Hvis flere record-typer benytter samme felt-navne kan det være nødvendigt med type-annoteringer.
- 3 Mønstergenkendelse for tupler og records er også meget anvendte i funktionsparametre:

```
let swap (x:'a,y:'b) : 'b * 'a = (y,x)
```

## Mønstergenkendelse på heltal og andre grundtyper

Implementation af Fibonacci med mønstergenkendelse:

```
let rec fib n =  
  match n with  
  | 1 -> 1  
  | 2 -> 1  
  | _ -> fib(n-1) + fib(n-2)  
let v = fib 10
```

### Bemærk:

- 1 Den første bar (|) i en match-case er optional.
- 2 Den første match-case der “matcher” vinder.
- 3 Wildcards (\_) kan benyttes i en match-case.
- 4 Tilsvarende kan der matches på andre grundtyper, såsom karakterer, booleans og strenge.

## Mønstergenkendelse på option-værdier

Typen `int option` er et eksempel på en simpel såkaldt “sum-type”, også kaldt “discriminated union”, som repræsenterer værdier der enten er værdien `None` eller er en værdi `Some(v)`, hvor `v` er en værdi af typen `int`.

Her er en funktion der “løfter” addition til værdier af typen `int option`:

```
let add_opt (a:int option) (b:int option) : int option =  
  match a, b with  
  | Some a, Some b -> Some(a+b)  
  | _ -> None
```

### Bemærk:

- 1 Der benyttes her en form for “nested pattern matching” på par af værdier af typen `int option`.
- 2 Ved konstruktion og matching af tupler kan man ofte undvære brugen af parenteser.
- 3 Variabler kan **bindes** i en match-case og henvises til i højre-siden af en match, hvor de vil varetage de matchede værdier.

## Mønstergenkendelse på lister

Liste-værdier **konstrueres** grundlæggende set ved brug af to forskellige konstruktører:

- `[]` (Nil) Konstruktion af den tomme liste.
- `x::xs` (Cons) Konstruktion af et listeelement med hovedet `x` og halen `xs` (en anden liste).

Samme to konstruktører benyttes ved mønstergenkendelse på en liste:

```
let rec length (l: 'a list) : int =  
  match l with  
  | [] -> 0  
  | x::xs -> 1 + length xs
```

## Mønstergenkendelse med **function**-konstruktionen

```
let rec length : 'a list -> int =  
  function [] -> 0  
           | x::xs -> 1 + length xs
```

**Bemærk:** Funktionsparameter og **match-with**-konstruktionen sammentrækkes.

## Nestede mønstergenkendelser på lister

Liste-værdier kan matches til dybere niveauer end første cons-celle:

```
let lengthy (l: 'a list) : bool =  
  match l with  
  | _::_::_ -> true  // at least two cells  
  | _ -> false
```

Mønstre kan være mere komplekse:

```
let rec ones (l: int list) : int =  
  match l with  
  | [] -> 0  
  | 1::xs -> 1 + ones xs  // match-cases are tested in order  
  | _::xs -> ones xs
```

## Selection Sort–uden mønstergenkendelse

- Udtræk det mindste element i listen.
- Gentag processen rekursivt indtil der ikke længere er elementer i listen.

### En implementation i F#:

```
let rec select (xs:int list) (m,ys) =
    if List.isEmpty xs then (m,ys)
    else let x = List.head xs
         let xs = List.tail xs
         in if x < m then
             if m <> System.Int32.MaxValue then
                 select xs (x,m::ys)
             else select xs (x,ys)
             else select xs (m,x::ys)
let rec ssort xs =
    if List.isEmpty xs then xs
    else let (m,xs) = select xs (System.Int32.MaxValue,[])
         in m :: ssort xs
```

## Selection Sort–med mønstergenkendelse

Ny implementation i F#:

```
let maxInt = System.Int32.MaxValue
let rec select (xs:int list) (m,ys) =
    match xs with
    | [] -> (m,ys)
    | x::xs -> if x < m then
                if m <> maxInt then
                    select xs (x,m::ys)
                else select xs (x,ys)
            else select xs (m,x::ys)

let rec ssort =
    function [] -> []
           | xs -> let (m,xs) = select xs (maxInt,[])
                  in m :: ssort xs
```

## Analyse af Selection Sort – Set tidligere

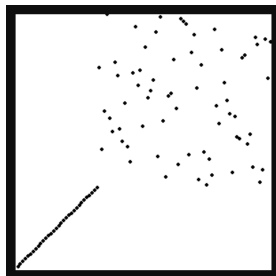
Funktionen `select` køres  $N$  gange og for hver kørsel gennemløbes listen (i gennemsnit  $N/2$  elementer).

### Summary:

Best time:  $O(N^2)$

Worst time:  $O(N^2)$

Average time:  $O(N^2)$



(animation)