



## Dagens tema

- **Innføring i ML - del II**  
(Kapittel 7.4.3 & ML-kompendiet.)
  - Oppvarming og uttøyning
  - Smart typeanalyse
  - Evalueringsmekanismen i ML
  - Typedeklarasjoner (produkt-, union-, rekursive typer ...)

# Oppvarming og repetisjon

```

1 - 0 :: [1,2];
2 - [1,2] @ [3,4];
3 - [1,2] :: [3,4];
4 - 1 :: 2 :: 3 :: nil;
5 - "INF" ^ "3110" ^ " er moro";
6 - rev [1,2,3];
7 - fun plussto(x : int) = x + 2;
8 - plussto(~10);
9 - plussto ~10;
10 - plussto;
11 - it (it 1000);
12 - 2 < 3 andalso 3<2;
13 - 2 < 3 orelse 3<2;
14 - Hurra!;
```

```

val it = [0,1,2] : int list
val it = [1,2,3,4] : int list
Error: operator and operand don't agree
val it = [1,2,3] : int list
val it = "INF3110 er moro" : string
val it = [3,2,1] : int list
val plussto = fn : int -> int
val it = ~8 : int
val it = ~8 : int
val it = fn : int -> int
val it = 1004 : int
val it = false : bool
val it = true : bool
Error: unbound variable or constructor: Hurra
```

# Oppvarming og repetisjon

Rekursjon over lister:

```
1 - fun finn(x:int , ls:int list) =  
2   case ls of [] => false  
3     | y :: resten => x = y orelse finn(x,resten);  
4 val finn = fn : int * int list -> bool
```

Alternativt:

```
1 - fun finn(x:int , []:int list) = false  
2   | finn(x:int , (y :: resten):int list) =  
3     x = y orelse finn(x,resten);  
4 val finn = fn : int * int list -> bool
```

## Smart typeanalyse

Følgende deklarasjoner forstås likt av ML:

```
1 - fun dobbel(x:int)    = x+x; (* vi spesifiserer typen til argumentet *)
2 - fun dobbel(x):int   = x+x; (* vi spesifiserer typen til resultatet *)
3 - fun dobbel(x:int):int = x+x; (* vi spesifiserer begge deler *)
4 - fun dobbel(x)       = x+x; (* vi spesifiserer ikke noe *)
```

I alle tilfellene kvitterer ML med

```
1 val dobbel = fn : int -> int
```

ML kan dedusere typen av funksjonsverdien og/eller (utpede) parametre, så sant en éntydig løsning finnes.

```
1 - fun id(x) = x;
2 val id = fn : 'a -> 'a
```

Her står 'a for en vilkårlig type. Dvs. funksjonen er *polymorf*, den kan brukes for alle typer.

## Evalueringmekanismen i ML

Hvis ML skal regne ut `plussto(<uttrykk>)`, så må først `<uttrykk>` regnes ut!

Eksempel:

```
1 - fun fak(n:int) = if n < 2 then 1 else n * fak(n-1);  
2 val fak = fn : int -> int  
3 - fak(3);  
4 val it = 6 : int  
5 - fak 4;  
6 val it = 24 : int
```

Hvordan regnes dette ut?

```
fun fak(n:int) = if n < 2 then 1 else n * fak(n-1);
```

```
1 fak(4) => 4 * fak(4 -1)
2   => 4 * fak(3)
3   => 4 * (3 * fak(3 - 1))
4   => 4 * (3 * fak(2))
5   => 4 * (3 * (2 * fak(2 - 1)))
6   => 4 * (3 * (2 * fak(1)))
7   => 4 * (3 * (2 * 1))
8   => 4 * (3 * 2)
9   => 4 * 6
10  => 24
```

Denne funksjonen bruker unødvendig mye plass.

Et forslag som bruker mindre plass:

```
1 - fun fakti(n:int, r:int) =  
2   if n < 2 then r else fakti(n-1, r*n);  
3 val fakti = fn : int * int -> int
```

```
1 fakti(4,1) => fakti(4 - 1, 1 * 4)  
2   => fakti(3, 4)  
3   => fakti(3 - 1, 4 * 3)  
4   => fakti(2, 12)  
5   => fakti(2 - 1, 12 * 2)  
6   => fakti(1, 24)  
7   => 24
```

Denne er **halerekursiv** eller **iterativ**.

Gode kompilatorer finner ut av når en funksjon er halerekursiv.

Merk: Ikke programmer slik bortsett fra når plass er essensielt!

# Typedeklarasjoner

Deklarasjon av en type med navn T *uten* parametre:

type/datatype T = <type-uttrykk>;

```
1 type tall = int;  
2 type talliste = tall list;
```

Deklarasjon av en type med navn T og *med* parametre:

type/datatype <typeparametre> T = <type-uttrykk>;

```
1 type 'minType par = 'minType * 'minType; (* mer om produkttyper på side 10 *)  
2 type intpar = int par;  
3 type boolpar = bool par;
```

**Merk:** Navnet på en typeparameter må starte med en apostrof. `par` blir her en *typekonstruktør*.



## Produkttyper med navngitte komponenter - record

```
1 - type Person = {navn:string, alder:int};  
2 type Person = {alder:int, navn:string}  
3  
4 - val Ole = {navn="Ole", alder= 25};  
5 val Ole = {alder=25,navn="Ole"} :  
6     {alder:int, navn:string}  
7  
8 - #navn Ole;  
9 val it = "Ole" : string  
10  
11 - #alder Ole;  
12 val it = 25 : int
```

## Produkttyper med navnløse komponenter - tuple

```
1 - type Person = string * int;  
2 type Person = string * int  
3  
4 - val Ole = ("Ole", 25);  
5 val Ole = ("Ole", 25) : string * int  
6  
7 - #1 Ole;  
8 val it = "Ole" : string  
9  
10 - #2 Ole;  
11 val it = 25 : int;
```

## Produkttyper

### Tomt produkt – unit

Typen unit står for et tomt produkt.

Denne typen er predefinert, og har bare én mulig verdi, nemlig (). Denne verdien kan også skrives som {}.

```
1 - ();  
2 val it = () : unit  
3 - () = {};  
4 val it = true : bool
```

## Uniontyper i ML: “datatyper”

```
1 - datatype PersonData = navn of string | alder of int;  
2 datatype PersonData = alder of int | navn of string  
3 - val n = navn("Ole");  
4 val n = navn "Ole" : PersonData  
5 - val a = alder(25);  
6 val a = alder 25 : PersonData
```

Både `n` og `a` er lovlige, men ulike, verdier av typen `PersonData`.

Ved **case**-uttrykk kan man analysere `PersonData`-verdier:

```
case x of navn(s) => ...s...  
        | alder(i) => ...i...
```

der `x` er et uttrykk av type `PersonData`.

## Eksempel:

```
1 - fun inc_alder(x:PersonData) =  
2   case x of navn(s) => x  
3     | alder(i) => alder(i+1);  
4 val inc_alder = fn : PersonData -> PersonData  
5  
6 - val n = navn("Ole");  
7 val n = navn "Ole" : PersonData  
8  
9 - val a = alder(25);  
10 val a = alder 25 : PersonData  
11  
12 - inc_alder(n);  
13 val it = navn "Ole" : PersonData  
14  
15 - inc_alder(a);  
16 val it = alder 26 : PersonData
```

## Konstanter som alternativer

Et alternativ kan også være en konstant, for eksempel:

```
1 datatype Dag =  
2   mandag | tirsdag | onsdag | torsdag | fredag | lordag | søndag;  
3  
4 fun ukedag(d:Dag):bool=  
5   case d of lordag => false  
6     | søndag => false  
7     | _     => true;  
8  
9 - ukedag(mandag);  
10 val it = true : bool  
11  
12 - ukedag(lordag);  
13 val it = false : bool
```

## Eksempel:

```
1 - datatype Data = tall of int | no;  
2 datatype Data = no | tall of int  
3  
4 - tall(5);  
5 val it = tall 5 : Data  
6 - no;  
7 val it = no : Data  
8  
9 - fun dobbel(x:Data):Data =  
10     case x of tall(i) => tall(i+i)  
11     | no           => no;  
12 val dobbel = fn : Data -> Data  
13  
14 - dobbel(tall(5));  
15 val it = tall 10 : Data  
16 - dobbel(no);  
17 val it = no : Data
```

## Rekursive Typer

datatype T = .... | .... T ....

### Eksempel: liste av tall

```
1 - datatype TallListe = tom | leggtil of int * TallListe;  
2 datatype TallListe = leggtil of int * TallListe | tom  
3  
4 - leggtil(1,leggtil(2,leggtil(3,tom)));  
5 val it = leggtil (1,leggtil (2,leggtil #)) : TallListe
```



## Pass på! — “Uendelige datatyper” i ML

```
1 - datatype DumType = dum of int * DumType;  
2 datatype DumType = dum of int * DumType  
3 - fun makeDum(x:int):DumType = dum(x,makeDum(x));  
4 val makeDum = fn : int -> DumType
```

Hva skjer ved kallet `makeDum(1);?`

```
1 makeDum(1)  
2 => dum(1, makeDum(1))  
3 => dum(1, dum(1, makeDum(1)))  
4 => dum(1, dum(1, dum(1, makeDum(1))))  
5 => dum(1, dum(1, dum(1, dum(1, makeDum(1))))))
```

Terminerer ikke!

En rekursiv datatype må ha minst ett ikke-rekursivt/terminerende alternativ!

## Rekursive datatyper med parametre

Lister av vilkårlig type 'elem:

```
1 - datatype 'elem Liste = tom | leggtil of 'elem * 'elem Liste;
```

MLs egen liste-definisjon:

```
1 - datatype 'elem list = nil | :: of 'elem * 'elem list;
```

Merk at :: er en infiks operator, mens leggtil er prefiks.

leggtil kan gjøres til infiks ved å bruke infix slik:

```
1 - infix leggtil;  
2 infix leggtil  
3 - "test" leggtil tom;  
4 val it = "test" leggtil tom : string Liste
```