



ML (kap 5)

■ Bakgrunn

- Litt historikk
- Funksjonelle språk (kap 4.4)
- Bakgrunnen for ML

■ Programmeringsspråket ML

- Hvordan kjøre ML-programmer
- Enkle uttrykk
- Deklarasjoner
- Funksjoner
- Datatyper
- Mønstre

1. generasjons språk

Data var statisk: alle variable eksisterte under hele kjøringen.

Eksempler: Fortran, Cobol, Basic.

Funksjoner: Kun faste funksjoner. Ikke rekursive kall.

2. generasjons språk

Blokker ga lokale variable og rekursive kall.

Eksempler: Algol-60 (med BNF).

Funksjoner: Funksjoner som parametre.

3. generasjons språk

Egendefinerte datatyper.

Eksempler: Simula (objektorientert programmering), C, Java, Pascal.

Funksjoner: C med pekere til funksjoner, for eksempel

```
void (*signal (int sig, void (*disp)(int)))(int)
```

4. generasjons språk

Fleksible datastrukturer (lister, tabeller);
avanserte innebygde operasjoner (søking
med regulære uttrykk).

Eksempler: Perl.

Funksjoner: Som for 3. gen.

5. generasjons språk

Basert på ideer fra AI.

Eksempler: ??

Andre språk

Funksjonelle språk: Lisp, ...

Grafiske språk, logiske språk, ...

Hvorfor funksjonelle språk?

Det er flere grunner:

- ❶ Ønsket om større fleksibilitet når det gjelder funksjoner: anonyme funksjoner, kunne lage nye funksjoner utifra andre.
- ❷ Mer generelle funksjoner (kunne returnere alle typer).
- ❸ Matematikerne likte dårlig tilordning.

Få språk er *rent funksjonelle*. Lisp og ML er blandingspråk men kalles *funksjonelle*.

Men: Flere moderne algoritmiske språk (som Java og C#) har nå fått funksjonelle elementer.

Hvordan ble ML til?

I Edinburgh på slutten av 1970-tallet jobbet man med automatisk bevis av teoremer og laget bevissystemet LCF («Logic for Computable Functions»). Dette ble programmert i ML («Meta Language»). Hovedmannen var Robin Milner.

Vi kjører *Standard ML of New Jersey* utviklet av flere universiteter og forskningsinstitusjoner der. Informasjon finnes på [//www.smlnj.org/](http://www.smlnj.org/); der finnes også nedlastbar kode.

Hvordan kjøre ML

ML er i utgangspunktet et interaktivt språk:

```
> sml
Standard ML of New Jersey v1 10.49 [FLINT v1.5], September 13, 2004
- "Hallo, alle sammen.";
val it = "Hallo, alle sammen." : string
-  + 
```

Man kan også oppgi en fil med ML-kode:

```
> more tall.sml
(5+3)-2;
1+2+3+4+5;
> sml tall.sml
Standard ML of New Jersey v1 10.49 [FLINT v1.5], September 13, 2004
[opening tall.sml]
val it = 6 : int
val it = 15 : int
-  + 
```

Følgende gjelder:

- ML leser uttrykk og utfører dem direkte. Så skrives svaret ut (med typeangivelse).
- Alle uttrykk avsluttes med semikolon.

```
- (~5 + 7) - 2 ;  
val it = 0 : int  
- (1+ ~2)*(4+5);  
val it = ~9 : int  
- 100 div 3;  
val it = 33 : int  
- 100 mod 3;  
val it = 1 : int  
- 24.0 / 2.5; (* Use / for reals *)  
val it = 9.6 : real
```

- Tegnet ~ er en unær minus mens - er binær (dvs for subtraksjon).
- div og mod brukes på heltall mens / er for flyt-tall.
- Man kan ikke blande hel- og flyt-tall; de må konverteres med real eller floor.
- Kommentarer skrives i (* ... *).

Deklarasjoner

Nye konstanter defineres med val:

```
- val x = 7+2;  
val x = 9 : int  
- val y = x+3;  
val y = 12 : int  
- val z = x*y - (x div y);  
val z = 108 : int  
- x + y + z;  
val it = 129 : int
```

Løse uttrykk definerer konstanten it.

NB!

Konstanter kan aldri endre verdien.

```
- val pi = 3.14;  
val pi = 3.14 : real  
- fun omkrets (r) = 2.0 * pi * r;  
val omkrets = fn : real -> real  
- omkrets(33.0);  
val it = 207.24 : real  
- val pi = 3.14159265;  
val pi = 3.14159265 : real  
- omkrets(33.0);  
val it = 207.24 : real  
-
```


Lokale definisjoner

Man kan definere lokale konstanter med let:

```
fun findroot (a, x, acc) =
  let val next_x = (a/x+x)/2.0 in
    if abs(x-next_x) < acc*x then next_x
    else findroot(a, next_x, acc)
  end;

fun sqrt (x) = findroot(x, 1.0, 1.0E~10);

(* Heron's formula for area of triangle: *)
fun areal (a, b, c) =
  let val s = (a+b+c)/2.0 in
    sqrt(s*(s-a)*(s-b)*(s-c))
  end;

areal(1.0, 1.0, 1.0);
areal(3.0, 4.0, 5.0);
```

Kjøring gir:

```
Standard ML of New Jersey v110.49 ...
[opening let1.sml]
val findroot = fn : real * real * real -> real
val sqrt = fn : real -> real
val areal = fn : real * real * real -> real
val it = 0.433012701892 : real
val it = 6.0 : real
-
```

Funksjoner

Funksjoner defineres med fun:

```
- fun f(x) = x + 5;  
val f = fn : int -> int  
- f(2);  
val it = 7 : int  
- f(f(0));  
val it = 10 : int  
- val g = fn x => x-2;  
val g = fn : int -> int  
- g(2);  
val it = 0 : int  
- f(g(0)) = g(f(0));  
val it = true : bool
```

fn er en funksjonstype som kan brukes til å angi en anonym funksjon.

Datatyper

Heltall

De velkjente tallene med de vanlige operatorene: +, -, *, div og mod.

Flyt-tallene

De like velkjente tallene med sine operatører: +, -, * og /.

Tekster

Typen string er for tekster; tekstkonstanter skrives med doble anførselstegn.

Operatører er

^ skjøter tekster.

size gir antall tegn i teksten.

```
- val navn = "langmyhr";  
val navn = "langmyhr" : string  
- size navn;  
val it = 8 : int  
- val fullt_navn = "dag" ^ " " ^ navn;  
val fullt_navn = "dag langmyhr" : string  
- size fullt_navn;  
val it = 12 : int
```

Logiske verdier

Typen `bool` har to verdier: `false` og `true`.

Man kan teste logiske verdier med

```
if e1 then e2 else e3
```

Testene \neg , \wedge og \vee skrives `not`, `andalso` og `orelse`. De beregner ikke sine parametre unødvendig.

```
- fun equiv(x,y) = (x andalso y) orelse ((not x)andalso(not y));  
val equiv = fn : bool * bool -> bool  
- equiv(true,false);  
val it = false : bool  
- fun e2(x,y) = if x then y else not y;  
val e2 = fn : bool * bool -> bool  
- equiv(false,true);  
val it = false : bool
```

Par, tripler, n-tupler

Tupler er vilkårlige samlinger av verdier som kan være av ulike typer.

```
- (3,4);  
val it = (3,4) : int * int  
- (4, 5, true);  
val it = (4,5,true) : int * int * bool  
- val beatles = ("John", "Paul", "George", "Ringo");  
val beatles = ("John","Paul","George","Ringo")  
  : string * string * string * string  
- #3 beatles;  
val it = "George" : string  
- #2(3,4);  
val it = 4 : int  
- ();  
val it = () : unit
```

#n brukes til å plukke frem et gitt element i tuplet.

() har typen `unit` og brukes som «ingen type» (tilsvarende `void` i C).

Poster

Poster («records») ligner på tupler, men elementene er navngitt.

```
- val guru = {fornavn="Donald", initial="E", etternavn="Knuth"};
val guru = {etternavn="Knuth", fornavn="Donald", initial="E"}
: {etternavn:string, fornavn:string, initial:string}
- #initial guru;
val it = "E" : string
- #fornavn guru = #etternavn guru;
val it = false : bool
- {};
val it = () : unit
```

Lister

Lister er den viktigste datastrukturen i ML.

Alle elementene i en liste har samme type.
nil er den tomme listen.

```
- [1, 2, 3, 4];  
val it = [1,2,3,4] : int list  
- [true, false];  
val it = [true,false] : bool list  
- ["red", "yellow", "blue"];  
val it = ["red","yellow","blue"] : string list  
- [fn x => x+1, fn x => x+2];  
val it = [fn,fn] : (int -> int) list  
- nil;  
val it = [] : 'a list
```

(Notasjonen 'a betegner en generell type.)

Listeoperasjoner

Nye elementer kan legges inn først i listen med ::. To lister kan skjøtes med @.

```
- val a = [1, 2, 3];  
val a = [1,2,3] : int list  
- val b = 0 :: a;  
val b = [0,1,2,3] : int list  
- a @ b;  
val it = [1,2,3,0,1,2,3] : int list  
- 21 :: 22 :: 23 :: nil;  
val it = [21,22,23] : int list
```

Mønstre

Hvordan plukker man fra hverandre en liste eller en post?

Man kan bruke *mønstre* som er uttrykk med variable i stedet for verdier.

```
- val t = (1, 2, 3);  
val t = (1,2,3) : int * int * int  
- val (x, y, z) = t;  
val x = 1 : int  
val y = 2 : int  
val z = 3 : int  
- val liste = [1, 2, 3, 4];  
val liste = [1,2,3,4] : int list  
- val a :: b = liste;  
stdIn:7.1-7.19 Warning: binding not exhaustive  
  a :: b = ...  
val a = 1 : int  
val b = [2,3,4] : int list  
- val halv = {teller=1, nevner=2};  
val halv = {nevner=2,teller=1} : {nevner:int, teller:int}  
- val {teller=x, nevner=y} = halv;  
val y = 2 : int  
val x = 1 : int
```

(Advarselen skyldes at a::b ikke virker for tomme lister.)

Funksjonsparametre

Egentlig har funksjoner i ML bare én parameter, men den er et generelt mønster!

```
- fun sum (x, y) = x+y;
val sum = fn : int * int -> int
- sum(2, ~4);
val it = ~2 : int
- fun f (x, (y,z)) = y;
val f = fn : 'a * ('b * 'c) -> 'b
- f("a", ("b","c"));
val it = "b" : string
- val par = (1, 2);
val par = (1,2) : int * int
- f(0, par);
val it = 1 : int
- fun g (x::y::z) = x::z;
stdIn:13.1-13.23 Warning: match nonexhaustive
      x :: y :: z => ...

val g = fn : 'a list -> 'a list
- g([1,2,3,4,5]);
val it = [1,3,4,5] : int list
- fun h {a=x, b=y, c=z} = {d=y, e=z};
val h = fn : {a:'a, b:'b, c:'c} -> {d:'b, e:'c}
- h{a="A", b="B", c="C"};
val it = {d="B",e="C"} : {d:string, e:string}
```

Dette kan vi bruke ved å lage flere utgaver av funksjonen med ulike variasjoner over mønsteret:

```
- fun f (x, 0) = x | f (0, y) = y | f (x, y) = x+y;  
val f = fn : int * int -> int  
- f(0, 0);  
val it = 0 : int  
- f(0, 1);  
val it = 1 : int  
- f(1, 0);  
val it = 1 : int  
- f(1, 1);  
val it = 2 : int
```

Men alle må passe med samme mønster:

```
> sm1  
Standard ML of New Jersey v1 10.49 [FLINT v1.5], September 13, 2004  
- fun f (x) = 10*x  
  = | f (a,b) = a*b;  
stdln:1.13-1.17 Error: operator and operand don't agree [literal]  
operator domain: int * int  
operand:         int * ('Z * 'Y)  
in expression:  
10 * x
```

Et annet eksempel på hvor nyttige slike mønstre kan være er

```
- fun length (nil) = 0 | length (x::s) = 1+length(s);  
val length = fn : 'a list -> int  
- length([1,2,3]);  
val it = 3 : int  
- length [1,2];  
val it = 2 : int
```

Programmering i ML

Når man programmerer i ML, bør man huske følgende:

- Vi har ikke løkker[†], så bruk rekursjon i stedet.
- Vi har ikke variable[‡] så glem alt om variable som endres.
- Sett inn ekstra blanke rundt operatorene:

```
> sml
Standard ML of New Jersey v110.49 [FLINT v1.5], September 13, 2004
- val a = 3;
val a = 3 : int
- a=~3;
stdIn:2.2-2.4 Error: unbound variable or constructor: =~
stdIn:2.1-2.5 Error: operator is not a function [tycon mismatch]
operator: int
in expression:
a <errorvar>
```

[†] Dette er ikke helt sant – se neste uke.

[‡] Dette er heller ikke helt sant.