



Dagens tema

• Innføring i ML - del IV

- Mønstergjenkjenning
- Litt om funksjoner & likhetstyper
- Anonyme funksjoner
- Begrenset skop - let
- Funksjonsrom og currierte funksjoner
- Litt av hvert - greit å ta med seg
- Avsluttende kommentarer - spørsmål?

1/28

Forelesning 5 – 24.9.2003

Mønstergjenkjenning

Et *mønster* kan ha følgende form:

1. Et variabelmønster: *variabelnavn* (: *type*)
2. Wildcard: `_` matcher alt
3. Et sammensatt mønster, f.eks.: (*mønster-1*, ..., *mønster-n*)
(Mønstre kan også settes sammen ved hjelp av *konstanter* for tall, strenger, tegn, etc., *konstruktører* for datatyper og lister, *records*, mm.)

Deklarasjonen

val *mønster* = *uttrykk*;

er velformet sålenge *mønster* og *uttrykk* har samme type.

Forelesning 5 – 24.9.2003

2/28

Mønstergjenkjenning

Eksempel på mønster i variabeldeklarasjon:

```
1 - val ((x,_),(_y)) = ((100,200),("abcde","fghij"));
2 val x = 100 : int
3 val y = "fghij" : string
```

Mønstergjenkjenning i case-uttrykk:

<pre>1 - fun f x = 2 case x of (_0) => true 3 (0,_) => true 4 _ => false; 5 val f = fn : int * int -> bool 6 - f (1,2); 7 val it = false : bool</pre>	<pre>- fun entotre ls = case ls of [] => false 1 :: 2 :: 3 :: resten => true x :: resten => entotre resten; val entotre = fn : int list -> bool - entotre [9,9,9,9,9,1,2,3,4,5,6]; val it = true : bool</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Forelesning 5 – 24.9.2003

3/28

Mønstergjenkjenning

Mønstergjenkjenning i funksjonsdefinisjoner:

I stedet for å bruke case kan vi gjøre slik:

<pre>1 - fun f (_0) = true 2 f (0,_) = true 3 f _ = false; 4 val f = fn : int * int -> bool</pre>	<pre>- fun entotre [] = false entotre (1 :: 2 :: 3 :: resten) = true entotre (x :: resten) = entotre resten; val entotre = fn : int list -> bool</pre>
--------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------

(Dette er *nøyaktig* de samme funksjonene som på forrige side!)

Rekkefølgen spiller en rolle:

```
1 - fun entotre [] = false
2   | entotre (x :: resten) = entotre resten
3   | entotre (1 :: 2 :: 3 :: resten) = true;
4 Error: match redundant
```

Den samme feilen får vi ved tilsvarende case-uttrykk!

Forelesning 5 – 24.9.2003

4/28

Litt om funksjoner

Hva er en funksjon?

- Utenfraperspektiv: en mengde av par
- Innenfraperspektiv: instruksjoner / en algoritme

Er $f(x) = 2 * x$ og $g(x) = x + x$ samme funksjon?

Utenfra: **JA!** Innenfra: **NEI!**

Dette er skillet mellom et **ekstensjonalt** og **intensjonalt** syn på funksjoner.

Likhetstyper

- En **likhetstype** er type hvor verdiene tillater likhetstester.
- Likhetstester kan f.eks. ikke utføres på funksjoner!
Hva betyr at det at funksjonene f og g er like?
Svar: at $f(x) = g(x)$ for alle mulige argumenter x !
Hvordan kan vi sjekke det? Umulig.

```
1 - fun f x = 2*x;
2 - fun g x = x + x;
3 - f = g;
4 stdIn:152.1-152.6 Error: operator and operand don't agree [equality type required]
5   operator domain: "Z * "Z
6   operand:      (int -> int) * (int -> int)
```

I ML brukes "a for å angi at a er en likhetstype.

`int * (int * bool)` er en likhetstype.

`int -> (int * bool)` er *ikke* en likhetstype!

Anonyme funksjoner

To måter å skrive det samme på:

- `fun f x = x + x;`
- `val f = fn x => x + x;`

Syntaksen for `fn` er slik: `fn var : type => uttrykk`
(i stedet for `var : type` kan man også ha et *mønster*)

Denne evaluerer til en funksjon som tar et argument av typen *type* og som gir en verdi av typen til *uttrykk*.

var kalles en parameter og *uttrykk* en kropp.

Anonyme funksjoner

Eksempler

```
1 - map (fn x => 1000 + x) [1,2,3,4];
2 val it = [1001,1002,1003,1004] : int list
```

```
1 - fun toganger f = fn x => f (f x);
2 val toganger = fn : ('a -> 'a) -> 'a -> 'a
3 - toganger (fn str => str ^ " hopp") "hei";
4 val it = "hei hopp hopp" : string
5 - toganger (toganger (fn str => str ^ " hopp")) "hei";
6 val it = "hei hopp hopp hopp hopp" : string
```

Funksjonen `toganger` tar en funksjon f og returnerer en funksjon som anvender f to ganger.

Husk at funksjoner er verdier!

(som kan brukes som alle andre verdier ...)

```

1 - fn arg => [arg, arg];
2 val it = fn : 'a -> 'a list
3
4 - val lagto = (fn arg => [arg, arg]);
5 val lagto = fn : 'a -> 'a list
6
7 - lagto 1234;
8 val it = [1234,1234] : int list;
9
10 - val tofunksjoner = lagto (fn x => 5 * x);
11 val tofunksjoner = [fn,fn] : (int -> int) list
12
13 - (hd tofunksjoner) 100;
14 val it = 500 : int
  
```

Begrenset skop - let

let gjør en deklarasjon tilgjengelig i en begrenset del av koden:

let D in E end;

hvor D er en liste av deklarasjoner og E et uttrykk (som bruker disse deklarasjonene).

Eksempel:

```

1 - let val x = "hurra " in x^x^x end;
2 val it = "hurra hurra hurra " : string
  
```

Dette er som lokale variable i en blokk i imperative språk.

Begrenset skop - let

Let-uttrykk kan spare skrivearbeid, ved at et deluttrykk skrives bare én gang, og kan gi bedre effektivitet, ved at det regnes ut bare én gang.

```

1 - fun dobbel(x) = x + x;
2 val dobbel = fn : int -> int
3 - fun beregn (x) =
4     let val s = dobbel(x)
5     in
6     (s,~s)
7     end;
8 val beregn = fn : int -> int * int
9 - beregn(3);
10 val it = (6,~6) : int * int
  
```

Begrenset skop - let

Eksempel: let og en sammensatt funksjonsverdi

Oppgave:

Gitt en liste ønsker vi å beregne summen av alle elementene i listen, samt antall elementer i listen.

```

sumlengde([1,2,3,4])  ~>  (10,4)
sumlengde([1,2,3,4,5]) ~>  (15,5)
  
```

Vi tar dette eksemplet mot slutten hvis vi får tid.

Eksempel: let og en sammensatt funksjonsverdi

Første forsøk:

```

1 - fun sum(ls) =
2   case ls of [] => 0
3     | x :: r => x + sum(r);
4 val sum = fn : int list -> int
5
6 - fun lengde(ls) =
7   case ls of [] => 0
8     | x :: r => 1 + lengde(r);
9 val lengde = fn : 'a list -> int
10
11 - fun sumlengde(ls) = (sum(ls), lengde(ls));
12 val sumlengde = fn : int list -> int * int
13
14 - sumlengde([2,3,4,5,6,7,8,9,22,33,44,55,66,77,88,99]);
15 val it = (528,16) : int * int

```

Vi går her gjennom den samme listen to ganger.

Eksempel: let og en sammensatt funksjonsverdi

Andre forsøk: slå sammen sum og lengde til én funksjon:

```

1 - fun sumlengde2(ls) =
2   case ls of [] => (0,0)
3     | x :: r => (x + #1 (sumlengde2(r)),
4                 1 + #2 (sumlengde2(r)));
5 val sumlengde2 = fn : int list -> int * int
6
7 - sumlengde2([2,3,4,5,6,7,8,9,22,33,44,55,66,77,88,99]);
8 val it = (528,16) : int * int

```

Alt er i nå en funksjon, men vi gjør det samme rekursive kallet to ganger!

Eksempel: let og en sammensatt funksjonsverdi

Tredje forsøk: slå sammen sum og lengde til én funksjon:

```

1 - fun sumlengde3(ls) =
2   case ls of [] => (0,0)
3     | x :: r => let val (sum_r, lengde_r) = sumlengde3(r)
4                 in (x + sum_r, 1 + lengde_r) end;
5 val sumlengde3 = fn : int list -> int * int
6
7 - sumlengde3([2,3,4,5,6,7,8,9,22,33,44,55,66,77,88,99]);
8 val it = (528,16) : int * int

```

Dette gir bedre effektivitet!

Legg merke til bruken av mønster i let val (sum_r, lengde_r)

Funksjonsrom og currierte funksjoner

```

1 - fun pluss(a,b) = a + b;
2 val pluss = fn : int * int -> int
3 - pluss(3,4);
4 val it = 7 : int

```

```

- fun pluss a b = a + b;
val pluss = fn : int -> int -> int
- pluss 3 4;
val it = 7 : int

```

pluss er her en funksjon som tar **et par av heltall** (a,b) som argument og som gir et heltall som verdi. (Det er mer presist enn å si at den simpelthen tar to argumenter, a og b.)

pluss er her en funksjon som tar **et heltall a** som argument og som gir en **funksjon fra heltall til heltall** som verdi. Funksjoner på denne formen kalles **currierte**.

Med et **funksjonsrom** mener vi en mengde funksjoner fra en type til en annen. `int -> int` kan ses på som et enkelt funksjonsrom. En funksjon av type `a -> b -> c` tar argumenter av type `a` og gir funksjoner fra **funksjonsrommet** over funksjoner fra `b` til `c`.

Funksjonsrom og currierte funksjoner

```

1 - fun pluss a b = a + b;
2 val pluss = fn : int -> int -> int
3 - val f = pluss 3;
4 val f = fn : int -> int
5 - f 4;
6 val it = 7 : int
7 - f 8;
8 val it = 11 : int

```

Kallet **pluss 3** kalles **ufullstendig/partielt**, fordi det har en "ufullstendig liste av aktuelle parametre". Dette kallet gir altså en funksjon som funksjonsverdi.

Funksjonsrom og currierte funksjoner

En curriert versjon av *gjenta* eller *foldr*:

```

1 - fun foldr f d l =
2   case l of [] => d
3           | x :: r => f(x, foldr f d r);
4 val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

```

Merk at **f** her er antatt å være en funksjon som tar et par som argument; den er altså *ikke* en curriert funksjon!

foldr, foldl og map (*gjenta*) er predefinert i ML, filter (*plukk*) er det ikke.

Funksjonsrom og currierte funksjoner

Eksempel på bruk av *foldr*:

<pre> 1 - fun pluss(a,b) = a + b; 2 val pluss = fn : int * int -> int 3 4 - val sum = foldr pluss 0; 5 val sum = fn : int list -> int 6 7 - sum [1,2,3,4,5]; 8 val it = 15 : int </pre>	<pre> 1 - fun gange(x,y) = x*y; 2 val gange = fn : int * int -> int 3 4 - val prod = foldr gange 1; 5 val prod = fn : int list -> int 6 7 - prod [1,2,3,4,5]; 8 val it = 120 : int </pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Funksjonsrom og currierte funksjoner

Curry og uncurry

```

1 - fun curry f x y = f(x,y);
2 val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
3
4 - fun uncurry f(x,y) = f x y;
5 val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c

```

Funksjonen **curry** lager curriert versjon av en ikke-curriert funksjon, og **uncurry** det motsatte.

Funksjonsrom og currierte funksjoner

Curry og uncurry – eksempler:

1 - fun curry f x y = f(x,y); 2 val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c 3 4 - fun pluss(a,b) = a + b; 5 val pluss = fn : int * int -> int 6 7 - curry pluss 1 2; 8 val it = 3 : int	- fun uncurry f(x,y) = f x y; val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c - fun gange a b = a * b; val gange = fn : int -> int -> int - uncurry gange (3,4); val it = 12 : int
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Eksempel: Bag som abstrakt datatype

```

1 signature Bag_def =
2 sig
3   type 'elem Bag
4   val empty: 'elem Bag
5   val add : 'elem Bag * 'elem -> 'elem Bag
6   val del : 'elem Bag * 'elem -> 'elem Bag
7   val has : 'elem Bag * 'elem -> int
8   val union: 'elem Bag * 'elem Bag -> 'elem Bag
9 end;

```

Husk: to fnutt'er på "elem gir oss tilgang til likhet over denne typen.

Eksempel: Bag som abstrakt datatype

Implementasjon I: ved liste

```

1 structure Bag_impl: Bag_def =
2 struct
3   type "elem Bag = ("elem * int) list;
4   val empty = [];
5   fun add(b,x) = case b of [] => (x,1)::[]
6     | (y,i)::b' => if x=y then (y,i+1)::b' else (y,i)::add(b',x);
7   fun del(b,x) = case b of [] => []
8     | (y,i)::b' => if x=y then if i=1 then b' else (y,i-1)::b'
9     | _ => (y,i)::del(b',x);
10
11   fun has(b,x) = case b of [] => 0
12     | (y,i)::b' => if x=y then i else has(b',x);
13   fun union(a,b)= case b of [] => a
14     | (y,i)::b' => if i=1 then union(add(a,y), b')
15     | _ => union(add(a,y),(y,i-1)::b');
16 end;

```

Eksempel: Bag som abstrakt datatype

Implementasjon II: ved funksjonsrom

```

1 structure Bag_impl: Bag_def =
2 struct
3   type "elem Bag = "elem -> int;
4   fun empty y = 0;
5   fun add(b,x) y = if x=y then (b x)+1 else (b y);
6   fun del(b,x) y = if x=y then (b x)-1 else (b y);
7   fun has(b,x) = b x;
8   fun union(a,b) (x) = (a x) + (b x);
9 end;

```

Merk at funksjonsrom ikke har noen likhetsfunksjon i ML; dermed kan vi ikke teste om to "bagger" er like med denne implementasjonen.

En fordel: vi kan definere og regne på uendelige bager. F.eks.: en bag som består av en forekomst av hvert tall.

Eksempel: Bag som abstrakt datatype

Implementasjon IIb: ved funksjonsrom

```

1 structure Bag_impl: Bag_def =
2 struct
3   type "elem Bag = "elem -> int;
4   val empty      = fn(y) => 0;
5   fun add(b,x)   = fn(y) => if x=y then (b x)+1 else (b y);
6   fun del(b,x)   = fn(y) => if x=y then (b x)-1 else (b y);
7   fun has(b,x)   = b x;
8   fun union(a,b) = fn(y) => (a y) + (b y);
9 end;
```

Her har vi gjort det samme som på forrige foil, men med en litt annen syntaks.

Litt av hvert - greit å ta med seg

explode : string -> char list:

```

1 - explode "test";
2 val it = ["t","e","s","t"] : char list
```

implode : char list -> string:

```

1 - implode ["t","e","s","t"];
2 val it = "test" : string
```

Litt av hvert - greit å ta med seg

infix *navn*:

```

1 - infix cons;
2 infix cons
3 - fun (a cons b) = a :: b;
4 val cons = fn : 'a * 'a list -> 'a list
5 - 123 cons [4,5,6];
6 val it = [123,4,5,6] : int list
```

infix gjør at funksjonen som heter *navn* bare kan anvendes infix.

op⊕:

```

1 - op cons
2 val it = fn : 'a * 'a list -> 'a list
3 - (op cons)(123, [4,5,6]);
4 val it = [123,4,5,6] : int list
5 - foldr op+ 0 [1,2,3,4,5];
6 val it = 15 : int
```

Hvis ⊕ er infix, så gjør **op**⊕ at man kan benytte funksjonen ⊕ på par med vanlig prefiks notasjon.

Noen fordeler med ML

- Gjør at vi kan programmere og abstrahere **slik vi tenker**.
- **Enkel, elegant kode**, selv for svært komplekse problemer. (Jfr. dronningproblemet.) **Rekursjon** blir svært enkelt.
- **Verifikasjon** blir enklere. Få side-effekter gjør matematisk analyse greiere.
- ... (statisk typet, polymorfi, typesikkerhet, moduler, høyere-ordens funksjoner, mønstergjenkjenning, støtte for imperativ programmering, klar semantikk)

Noen ulemper: ikke objekt-orientert, subtyper, etc.