Dagens tema

- Typer
  (Kapittel 3 frem til 3.3.1.)
- Innføring i ML
  (Kapittel 7.4.3 & ML-kompendiet.)

Typer

En (data-)type består av:

- en mengde verdier
- en mengde operasjoner man kan anvende på disse verdiene

Eksempel:

```
1  Type: boolean
2  Verdimengde: {true, false}
3  Operatorer: NOT, AND, OR ...
```

Programmeringsspråk har vanligvis en del predefinerte typer, samt mekanismer for å lage mer komplekse datatyper fra disse.

Hva er fint med typer?

Det gir oss muligheten til å

- klassifisere våre data
- beskytte data fra feil eller meningsløs bruk
- abstrahere og gjemme den underliggende representasjonen

Predefinerte typemekanismer

elementære typer , ofte integer, real, character, string,. . .

oppramstyper , eks. {mandag, tirsdag, onsdag, ...}

intervall-typer , f.eks. tallene fra 1 til 100

lister, sekvenser , predefinert i LISP, ML

tabeller, arrayer , gjerne predefinert i imperative språk

pekere . . .

Generelle typemekanismer I

produkt-typer med flere komponenter

- Pascal/Ada: record
- C/C++: strukturer
- Java/Simula: klasser

union-typer gir flere alternativer (gjensidig ekskluderende)

- Eksempel (i ML-syntaks):

  ```
  datatype boolean = true | false;
  ```

- Pascal/Ada: variant record
- C/C++: union

Generelle typemekanismer II

rekursive typer inneholder noe av sin egen type
Eksempel: liste av heltall

```
1  class Node { int verdi; Node neste; }
2  Node liste;
```

Med rekursjon kan man lage strukturer som er vilkårlig store og som kan ha vilkårlig stor kompleksitet.

parametriserte typer muliggjør datastrukturer over vilkårlige "element"-typer. Eksempel: en generell liste

Ved de generelle mekanismene kan man uttrykke de fleste andre type-konstruksjoner implisitt (med vekslende effektivitet).

Abstrakte datatyper

Innkapsling: Kan definere verdimengden og de tilhørende funksjonene i én modul.

Beskyttelse/skjuling: Skjule visse funksjoner og variable i en modul slik at de ikke kan brukes utenfor modulen.

Abstraksjon - vi slipper å tenke på den underliggende representasjonen.

Eksempel: Rasjonale tall som data-type:

```
1  Type RAT =
2    modulebegin
3    t, n : Int
4    FUNC mult(a,b:RAT):RAT ==
5      RETURN((a.t*b.t),(a.n*b.n))
6      ...
7    endmodule RAT
```

Verdimengde / datastruktur
Tilhørende funksjoner

Eventuelt andre funksjoner

Funksjonelle (applikative) språk

Motivasjon: Matematisk tilnærming til problemene uten tanke på maskinens oppbygning. Abstraksjonsnivå tilpasset programmereren, ikke hardware.

Eksempel: Euclids algoritme for å finne største felles divisor

Algoritmen: `gcd(0, n) = n`
`gcd(m, n) = gcd(n mod m, m) for m > 0`

```
1  // Java
2  int gcd(int m, int n){
3    int prev;
4    while (m != 0) { prev = m; m = n % m; n = prev;}
5    return n;}
```

```
(* ML *)
fun gcd(m, n) =
  if m = 0 then n
  else gcd(n mod m, m);
```

Rent funksjonelle språk

Rent funksjonelle språk har ikke tilordning! Variable kan ikke oppdateres.

Kan definere:

- konstanter,
- funksjoner,
- typer (hvis typet språk)

og kan skrive uttrykk!

Vi har ikke tilstander som i imperative språk, men verdier og applikasjon av funksjoner.

Rent funksjonelle språk

Typiske fordeler

- enkel, elegant kode
- ingen side-effekter
- aliasing ikke problem
- kan lett interpreteres

Typiske ulemper

- effektivitet
- mister kontroll med tid og rom
- input/output (ML gjør dette imperativt)

Rent funksjonelle språk

	imperativt	applikativt
oppdatering	tilordning	parameter-mekanismen
sekv. sammensetning	;	funksjons-sammensetning
valg	if-setning	if-uttrykk
gjentakelse	løkke	rekursjon

Eksempel: den imperative koden

$x := f(x, y);$
$x := g(x, y);$

blir funksjonelt seende ut som:

$$g(f(x, y), y)$$

Introduksjon til standard ML

Pensum:

- Bjørn Kristoffersen: *Funksjonell Programmering i Standard ML*, kompendium 61.
- Læreboka, kapittel 7 (untatt 7.4.1, 7.4.2 og 7.5).

Standard ML er et *interpreterende* språk.

Ml-interpreteren startes fra Unix ved kommandoen sml:

```
1  > sml
2  Standard ML of New Jersey, Version 110.0.3,
3  January 30, 1998 [CM; autoload enabled]
4  -
```

\- angir at systemet er klart til å ta imot kommandoer.

= angir at systemet venter på flere instruksjoner.

Husk: Alle uttrykk og deklarasjoner skal avsluttes med ;.

Kommentarer skrives ved (* . . . *).

CTRL-D avslutter systemet.

Man kan også starte ML fra emacs (sml-mode).

Eksempler på enkle beregninger

```
1 - 2 + 2;
2 val it = 4 : int
```

```
- 5 div 2;
val it = 2 : int
```

```
- 5.0 / 2.0;
val it = 2.5 : real
```

```
- 8 mod 3;
val it = 2 : int
```

```
- it + 5;
val it = 7 : int
```

Deklarasjon av funksjoner

fun f(<parameterliste>) = <uttrykk>;

```
- fun plussto(x : int) = x + 2;
val plussto = fn : int -> int
- plussto(5);
val it = 7 : int
- plussto 1;
val it = 3 : int
- plussto(plussto it);
val it = 7 : int
```

Deklarasjon av konstanter

```
- val konstanten = 5;
val konstanten = 5 : int
- konstanten + 3;
val it = 8 : int
```

Predefinerte typer:

```
int     med = < * + -  div mod  0, 1, 2, ... ( ~ unær minus)
real    med = < * + -  /        2.56
bool    med = true  false  andalso  orelse
string  med = <  "Olav Olavsen"   ^ (konkatenering)
list    med = ::   @ (konkatenering)  []  [1,2,3]  list
```

Merk: list er ikke en type, men en type*konstruktør*.

Ingen overlasting

Ny deklarasjon med samme navn medfører at siste deklarasjon gjelder!

Eksempel:

```
fun f(...) = ...;
.
.
.
fun f(...) = ...;
```

nå vil f binde til siste deklarasjon!

Pass på skopet!

```
1  - val a = 5;
2  val a = 5 : int
3  - fun plussa x = x + a;
4  val plussa = fn : int -> int
5  - plussa 2;
6  val it = 7 : int
7  - plussa 3;
8  val it = 8 : int
9  - val a = 100;
10 val it = false : bool
11 - plussa 2;
12 val it = 7 : int
```

Funksjonen plussa bruker en **omgivelse** der a har verdien 5. Vi har statisk binding.

Lister

Eksempler med lister:

- [1,2,3] – listen av 1, 2 og 3
- [] – den tomme listen (skrives også nil)
- [1,2]@[3,4]=[1,2,3,4] – konkatenering
- 0::[1,2] = [0,1,2] – innsetting først
- ["Ole", "Petter"] – liste av strenger
- [[1,2],[2,3,4]] – liste av lister

Typisk mønster for rekursive funksjoner over lister:

```
1 - fun f (lst: <type> list) =
2     case lst of [] => ...
3               | x::xs => ... f(xs) ...;
```

To eksempler på rekursive funksjoner over lister:

```
1  - fun finn(x:int ,ls:int list) =
2      case ls of [] => false
3              | y :: resten => x = y
4                               orelse finn(x,resten);
5  val finn = fn : int * int list -> bool
6  - finn(4, [2,4,6]);
7  val it = true : bool
8  - fun fjern(x:int ,ls:int list) =
9      case ls of [] => []
10                | y :: resten => if x = y
11                                 then fjern(x,resten)
12                                 else y :: fjern(x,resten);
13 val fjern = fn : int * int list -> int list
14 - fjern(1,[1,3,4,1,3]);
15 val it = [3,4,3] : int list
```