Dagens tema

- Innføring i ML - del III
  (Kapittel 7.4.3 & ML-kompendiet.)
  — Unntak
  — Abstrakte datatyper i ML
  — Høyere-ordens funksjoner

Unntak

exception – deklarerer et unntak
```
exception feil;
```

raise — for å gi unntak
```
fun f(..)= ..... raise feil ...;
```

handle — for å fange opp unntak
```
(...f(..) ....) handle feil => "Noe gikk galt!"
```



```
1  - exception finnesikke;
2  exception finnesikke
3
4  - fun forste(l) =
5      case l of nil => raise finnesikke
6    | x::r => x;
7  val forste = fn : 'a list -> 'a
8
9  - forste([1,2,3]);
10 val it = 1 : int
11 - forste([]);
12 [...]
13 uncaught exception finnesikke
14 [...]
15
16 - forste([]) handle finnesikke => 0;
17 val it = 0 : int
18 - forste([1,2,3]) handle finnesikke => 0;
19 val it = 1 : int
```


Eksempel: Stakk

datatype Stakk = empty | push of int * Stakk;

Vi lar top og pop være udefinert for tom stakk (empty).

```
1  - datatype Stakk = empty | push of int * Stakk;
2
3  - fun pop (s:Stakk) =
4    case s of push(_,x) => x;
5  [...] Warning: match nonexhaustive
6  - fun top (s:Stakk) =
7    case s of push(x,_) => x;
8  [...] Warning: match nonexhaustive
9
10 - pop(push(123,empty));
11 val it = empty : Stakk
12 - pop(empty);
13 [...] uncaught exception nonexhaustive match failure [...]
```

Det er bedre å bruke unntak!

Eksempel: Stakk med unntak

datatype Stakk = empty | push of int * Stakk;

Vi kan nå lage en stakk slik:
$push(x1, push(x2, ...push(xn, empty)...))$

```
1  - datatype Stakk = empty | push of int * Stakk;
2  - exception TomStakk; (* deklarerer exception TomStakk *)
3
4  - fun pop (s:Stakk) =
5    case s of empty => raise TomStakk (* feil *)
6    | push(_,x) => x;
7  val pop = fn : Stakk -> Stakk
8  - fun top (s:Stakk) =
9    case s of empty => raise TomStakk (* feil *)
10   | push(x,_) => x;
11 val top = fn : Stakk -> int
```


Abstrakte datatyper i ML

I ML kan man lage abstrakte datatyper ved først å definere grensesnitt og så implementasjon:

Grensesnitt angis ved en **signatur**, som inneholder deklarasjon av det som skal være synlig utad. Funksjoner og typer implementeres ikke, men typeinformasjon angis.

Implementasjon angis ved en **struktur**, som angir hvordan deklarasjonene i det tilsvarende grensesnittet skal implementeres. (Her må man angi datastrukturer for typer og implementasjon av alle funksjoner.)

Både signaturer og strukturer navngis. Man kan ha flere strukturer som implementerer samme signatur.

Grensesnittet — en abstrakt datatype

```
1  signature Stakk_def =
2  sig
3    type 'elem Stakk
4    exception TomStakk
5
6    val empty: 'elem Stakk
7    val nonempty: 'elem Stakk -> bool
8
9    val push: 'elem * 'elem Stakk -> 'elem Stakk
10   val pop : 'elem Stakk -> 'elem Stakk
11   val top : 'elem Stakk -> 'elem
12 end;
```


Implementasjon — ved en egen datatype

```
1  structure Stakk_impl: Stakk_def =
2  struct
3    datatype 'elem Stakk = empty | push of 'elem * ('elem Stakk);
4    exception TomStakk;
5    fun nonempty (s) = case s of
6      empty => false
7      | _  => true;
8
9    fun pop (s) = case s of
10     empty => raise TomStakk
11     | push(_,x) => x;
12
13   fun top (s) = case s of
14     empty => raise TomStakk
15     | push(x,_) => x;
16 end;
```

Stakk_impl kan også implementere andre funksjoner, men bare de vi har i Stakk_def vil være synlige utenfra.

Eksempel på bruk

```
- val s = Stakk_impl.empty;
val s = empty : 'a Stakk_impl.Stakk

- val s = Stakk_impl.push(1, Stakk_impl.push(2, s));
val s = push (1,push (2,empty)) : int Stakk_impl.Stakk

- Stakk_impl.top(s);
val it = 1 : int

- Stakk_impl.pop(s);
val it = push (2,empty) : int Stakk_impl.Stakk
```

For å få direkte tilgang til innholdet (slippe å skrive "Stakk_impl." hele tiden), kan strukturen åpnes med:
open Stakk_impl;

Implementasjon — ved en liste

```
structure Stakk_impl: Stakk_def =
struct
 type 'elem Stakk = 'elem list;

 exception TomStakk;
 val empty = [];

 fun push(x,s) = x::s;

 fun top(s) = case s of [] => raise TomStakk
   | x::xs => x;

 fun pop(s) = case s of [] => raise TomStakk
   | x::xs => xs;

 fun nonempty(s) = case s of [] => false | _ => true;
end;
```


Høyere-ordens funksjoner

Høyere-ordens funksjoner er basert på at funksjoner er data på samme måte som tall og tekst er det. Det betyr at de kan sendes som parametre, mottas som returverdier og bindes til variable. Dette kaller vi "funksjoner som fullverdige borgere".

En høyere-ordens funksjon er altså en funksjon som opererer på andre funksjoner.

Funksjoner som parametre

I det følgende skal vi se på tre vanlige høyere-ordens funksjoner som alle tar en funksjon som parameter: oppdater, gjenta og plukk. (Andre vanlige navn er map, reduce (eller fold) og filter.)

Funksjonen *oppdater*

Verdien til $oppdater(f, l)$ skal være listen vi får ved å anvende f på hvert element i listen l.

```
- fun oppdater(f, ls) =
    case ls of [] => []
             | x :: resten => f(x) :: oppdater(f, resten);
val oppdater = fn : ('a -> 'b) * 'a list -> 'b list
```

```
- fun dobbel(x) = x + x;
val dobbel = fn : int -> int
- oppdater(dobbel, [1, 2, 3]);
val it = [2,4,6] : int list
```

Funksjonen *plukk*

Verdien til $plukk(f, l)$ skal være listen av elementer x fra l der $f(x)$ er $true$.

```
1  - fun plukk(f, ls) =
2      case ls of [] => []
3               | x :: resten => if f(x) then x :: plukk(f, resten)
4                                else plukk(f, resten);
5
6  val plukk = fn : ('a -> bool) * 'a list -> 'a list
```

```
1  - fun negativ(x) = x < 0;
2  val negativ = fn : int -> bool
3  - plukk(negativ, [1, ~1, 2, ~4, ~5]);
4  val it = [~1,~4,~5] : int list
```


Funksjonen *gjenta*

$gjenta(f, d, l)$ gjentar 2-arguments funksjonen f over alle elementene i listen l (fra høyre mot venstre). d angir defaultverdi for tom liste.

```
1  - fun gjenta(f, d, ls) =
2      case ls of [] => d
3               | x :: resten => f(x, gjenta(f, d, resten));
4
5  val gjenta = fn : ('a * 'b -> 'b) * 'b * 'a list -> 'b
```

```
1  - gjenta(op+, 0, [1, 3, 5]);
2  val it = 9 : int
```