6
Funktionsorienteret programmering i ML.

Typeforhold i ML.
Basale indbyggede typer.
Erklæringer og bindinger af navne.
Patternmatching.
Funktionsbegrebet.
Brugerdefinerede datatyper.
Modulbegrebet.

PS4 - ML

Noter

Vi har hidtil benyttet Lisp dialekten Scheme som programmeringssprog til illustration af funktionsorienteret programmering. Vores primære interesse i Scheme har været som et konkret og enkelt sprog til udtrykkelse af funktionsorienterede programmer. Som en sidegevinst har vi stiftet bekendtskab med listehåndtering i et sprog i Lisp familien.

Vi vil i denne forelæsning studere et alternativt funktionsorienteret programmeringssprog: ML. ML indeholder ligesom Scheme imperative primitiver. ML er altså ikke et rent funktionsorienteret programmeringssprog. Vi vil dog kun beskæftige os med den funktionsorienterede del af ML i denne forelæsning.

Det er ikke vores formål at gøre os bekendt med detaljer i ML. Vi vil derimod sætte fokus på nogle udvalgte aspekter i ML, som udgør kontraster eller "nyheder" i forhold til Scheme.

Litteratur:

"Introduction to Standard ML" af Robert Harper, ESC-LFCS-86-14, fra Edingburgh, er en fin tutorial til ML (ca 65 sider). Mange af eksemplerne i dette kapitel er lånt fra dette skrift.

De fleste ML **eksempler** fra dette kapitel findes på filen

/user/normark/public/ps4/ml

Man **starter ML systemet** med

/pack/sml/bin/sml

(hvilket i skrivende stund i det mindste virker på Solaris 1 maskiner).

Typeperspektiv i ML.

- ML understøtter *statisk typning*..
- Typen af et ML udtryk erkendes så vidt muligt ved *typeinferens*.
  - Typen af et udtryk findes ud fra anvendelsessammenhængen .
  - Det er sjældent nødvendigt at erklære typen af navne og parametre statisk i programmet.
- Som resultat af evalueringen af et udtryk giver ML:
  - Udtrykkets værdi.
  - Udtrykkets type.

Noter

Statisk typning implicerer at alle typeforhold kan checkes ud fra en analyse af programmet inden udførelsestidspunktet. I ML opnåes statisk typning ved typeinferens, hvilket er grundlæggende forskellig fra den "traditionelle måde" at opnår statisk typning (nemlig eksplicit dekoration af programmet med typeinformation).

Pointerne på denne slide bliver i rigt mål illustreret i resten af forelæsningen. Vi vil gentagne gange se eksempler på effekten af typeinferens, og vi vil se, hvordan ML konsekvent melder ud om både værdien og typen af udtryk, som beregnes i "read eval print" loop'en.

Oversigt over basale, sprogdefinerede typer.

Unit ()
Booleans bool
Integers int
Reals real

atomare typer

Tupler

- En tupel type: $t_1 \times t_2$, givet to andre typer t_1 og t_2.
- En tupel værdi: (e_1, e_2), hvor e_i er af typen t_i.
- Generaliserer til *n*-tupler.

Lister

- Modelleret efter Lisp lister:
  - e :: lst ~ (cons el lst).
  - nil ~ '()
  - *patternmatching* bruges til destruktuerering (~car, cdr).

Records

- {$label_1$ = $værdi_1$, $label_2$ = $værdi_2$, ...}

sammensatte typer

PS4 - ML

Noter

Typen unit består af en enkelt værdi, som skrives (). Denne værdi bruges i de tilfælde, hvor et udtryk ikke har nogen interessant værdi, eller når funktioner ikke har noget argument. I denne situation er notationen () elegant afstemt.

Der er ikke meget interessant at sige om booleans, integers og reals.

Vi har tidligere været motiveret for at indføre tupler, nemlig da vi definerede højereordensfunktion construct (se slide "funktionssammensætning og krydsprodukt" i kapitlet om højereordensfunktioner). ML har netop den specielle type, som vi savnede på dette tidspunkt.

Ligesom i Lisp har vi også i ML mulighed for at notere en liste mere direkte end gennem sammensatte cons-udtryk. I ML skriver vi [3, 4, 5] i stedet for som i Lisp (3 4 5). Bemærk at en liste i ML skal have elementer af samme type. Med andre ord, listen skal være *homogen*. Vi kan altså ikke som i Lisp og Scheme lave en liste, hvis første element er et tal, og hvis andet element er en anden liste. Hvis dette var tilladt, ville ML komme i problemer med en fornuftig typeudmelding om en sådan liste. Hvis man i ML programmeringssammenhæng skulle savne en in-homogen liste, skal man nok overveje at bruge en tupel, som jo pr. definition kan være in-homogen (men til gengæld er af fast længde). Man kan også i nogle sammenhænge erstatte inhomogene lister med rekursive datatyper, som man er i stand til at definere i ML. Dette vender vi tilbage til lidt senere i dette kapitel.

Erklæringer.

- I en erklæring bindes et navn til en værdi.
- ML understøtter flere forskellige erklæringsformer, afhængig af hvilken slags værdi der skal bindes til et navn.
- Værdibindinger:
  - val *navn* = *værdi*;

```
- val x = 9 * 5;
> val x = 45  :  int

- val y = "xyz";
>val y = "xyz": string

- val pair = (x, y);
> val pair = (45, "xyz"):   int * string
```

- Simultan navnebinding:
  - val $navn_1$ = $værdi_1$ and $navn_2$ = $værdi_2$

```
-  val x = 9 * 5;
> val x = 45  :  int

- val x = true and y = x;
> val x = true: bool
   val y = 45 : int
```

PS4 - ML

Noter

I det ML system, vi forestiller os at anvende i disse forelæsningsnoter, er prompten blot '-'. Resultatet af en beregning, eller mere præcist værdien af det udtryk vi beregner, er prefixet med tegnet '>'. (Dette er ikke helt det samme i det ML system vi har kørende, men det er selvfølgelig en ret ligegyldig detalje).

I forhold til Scheme svarer ovenstående værdibinding med val til en define form: (define navn værdi). Begge påvirker den navndebindings omgivelse (engelsk: environment), hvori erklæringen forekommer.

Den simultane navnebindingsform svarer til Scheme's let, blot binder let altid relativt til et udtryk (som beregnes med hensyn til de navnebindinger, vi er ved at etablere).

Vi vil senere se andre erklæringsformer, bl.a. funktionsbindinger, som starter med fun.

Sammensatte navnebindingsformer.

- Sekventielle navnebindinger.

```
- val x = 17; val x = true; val y = x;
> val x = 17: int
> val x = true: bool
> val y = true: bool
```

- Lokale navnebindinger

```
- local
    val x = 10
  in
     val u = x * x ;
     val v = 2 * x + 1
  end;
> val u = 100: int
> val v = 21: int
```

- Lokale navnebindinger i udtryk.

```
- let
     val x = 10
   in
      x * x + 2 * x + 1
   end;
> 121 : int
```

PS4 - ML

Noter

Patternmatching: destrukturering af sammensatte typer.

- Patternmatching og en generaliseret værdibinding gør det muligt at destrukturere sammensatte datastrukturer.
- val *pattern = sammensat-værdi.*

Eksempler:

```
- val  x = ( 17, true);
> val x = (17, true): int * bool
-  val (left, right) = x;
> val left = 17: int
   val right = true: bool
-  val lst = [“a”,“b” ,“c”];
> val lst = [“a”, “b”, “c”] : string list
-  val [first, middle, last] = lst;
> val first = “a” : string
   val middle = “b”: string
   val last = “c” : string
- val hd::tl = lst;
 >val hd = “a”: string
   val tl = [“b”, “c”]: string list
-  val hd ::_ = lst;
> val hd = “a”: string
```

_ kaldes et “wildcard”

PS4 - ML

Noter

Patternmaching er et elegant alternativ til en hel række funktioner, hvis formål det er at selektere bestandele af sammensatte datastrukturer (af sammensatte typer).

Patternmatching er endnu rigere end eksemplificeret på denne slide. Der henvises til litteraturen (se første side i dette kapitel for referencer).

Funktionsbegrebet i ML.

- Enhver funktion tager *netop én parameter*.
  - I tilfælde af ingen parametre: unit typen ()
  - I tilfælde af to eller flere parametre: tupler ala (x, y).
- Hvis en funktionsdefinition har to eller flere parametre, opfattes funktionen som en *curried funktion*.
- ML anvender *strict funktionssemantik*, svarende til *call-by-value parameteroverførsel*.
- ML anvender *clausal funktionsdefintion*:
  - En funktionsdefinition indeholder én 'clause' for hver form af værdier af argumentet.
- ML understøtter *polymorfe funktioner*:
  - En funktion kan anvendes på en parameter af flere forkellige typer, altså
  - En funktions parameter kan antage mange former.
- Det er muligt at definere *højereordensfunktioner*.
- ML anvender *statisk navnebinding*.

Noter

Eksempler på funktionsdefinitioner og kald (1).

Simpel funktionsdefinition:

```
- fun twice x = 2 * x;                    - twice 4;
> val twice = fn:  int -> int             > 8 : int
```

Eksplicit, statisk typedefinition:

```
- fun plus(x, y): int = x + y;            - plus (4, 5);
> val plus = fn:  int * int -> int        > 9 : int
```

Clausal definition, pattern i parameter liste, polymorf funktion.

```
- fun append(nil, l) = l                  - append([1, 2], [3, 4, 5]);
    | append(hd::tl, l) = hd :: append(tl, l);   > [1, 2, 3, 4, 5]: int list
> val append = fn:  ( 'a list * 'a list ) -> 'a list
```

PS4 - ML

Noter

En funktionstype noteres som t -> u, hvor t og u er typer. Den pågældende funktion afbilder altså værdier af typen t over i typen u. Vi vil kalde t for definitionstypen, og u for værditypen af den pågældende funktion.

Funktionskald er syntaktisk set blot en *sidestilling* e f (engelsk: *juxaposition*) af to udtryk, hvor det første e skal evaluere til et funktionsobjekt, og det andet skal evaluere til en værdi i funktionens definitionstype.

For at kaldet e f skal være lovlig skal e være af typen t -> u og f skal være af typen t. Resultatet af udtrykket e f er af typen u.

Kommentarer til plus funktionen: plus udfører blot et kald til den sprogindbyggede aritmetiske funktion +, som i ML anvendes på infix form (i kontrast bl.a. til Scheme, som konsekvent insisterer på prefix form). I ML er + *overloadet*, således at a + b alt afhængig af typen af a og b betegner heltals addition, reel tals addition etc. Typeinference apparatet kan derfor ikke finde typerne af parametrene til plus, og ej heller typen af resultatet, medmindre programmøreren eksplicit "giver en hånd". Dette er gjort ved at sige, at resultatet af plus skal være af typen int. Dermed kan det sluttes, at der er tale om heltalsaddition, og dermed at begge parametre bliver nødt til at være heltal.

Kommentarer til append: Her illustreres hvad der kaldes "clausal function definition": altså definition af en funktion ved et antal tilfælde, som hver modsvarer en mulig form af parametrene (i tuplen). Det er værd at bemærke at den *formelle parameter tupel er et pattern*, som matches op imod de aktuelle parametre, hvorved destruktureringen finder sted. Sammenlign dette med Scheme (og andre tilsvarende sprog), hvor vi adskiller de to tilfælde ved et selektivt udtryk (if, cond). Endvidere skal vi i dette udtryk påkalde os diverse selektorer, som vi slipper for i ML, på grund af bidraget fra patternmatchingen.

Append er en *polymorf funktion*. Der er intet i append der gør det nødvendigt at forpligte os til bestemte parametertyper eller resultattype. Derfor indfører ML systemet en typevariabel, som skrives 'a. Værdien af 'a kan være hvad som helst. I anvendelsen af append ovenfor bliver 'a typen int.

Eksempler på funktionsdefinitioner og kald (2).

Lokal funktion, polymorfe funktioner:

```
-  fun reverse l =                                   - reverse [1, 2, 3];
    let fun  rev(nil, y) = y                         > [3, 2, 1]: int list
           | rev (hd::tl, y) = rev(tl, hd::y)
    in
        rev(l, nil)
    end;
> val reverse = fn : 'a list -> 'a list
```

Curried funktion:

```
-  fun times x y: int  =  x * y                      - times 2 4;
> val times = fn  : int -> int -> int               > 8 : int

-  val twice = times 2;                              - twice 4;
> val twice = fn:  int -> int                       > 8: int
```

PS4 - ML

Noter

Bemærk at twice er venstresektioneringen af multiplikations funktionen, hvor det første argument fastfryses til 2. Bemærk også, at vi i funktionen times har behov for at supplere med eksplicit typeinformation, fordi den artimetiske funktion * er overloadet, og man derfor ikke kan bestemme om der er tale om heltals multiplikation, reel talts multiplikations, eller ...

Eksempler på funktionsdefinitioner og kald (3).

Curried polymorf funktion:

```
- fun map f nil = nil                          -  map twice [1, 2, 3];
    | map f (hd::tl) = f(hd)::map f tl ;        > [2, 4, 6] : int list
> val map = fn :
       ('a -> 'b) -> ('a list) -> ('b list)     -  map (fn x => [x])  [1, 2, 3];
                                                > [[1],[2],[3]] : int list list
```

Curried polymorf funktion:

```
- fun compose(f, g)(x) = f (g(x));              - val fourtimes = compose(twice, twice);
> val compose = fn:                            > val fourtimes = fn: int -> int
    ('a -> 'b *  'c -> 'a) -> 'c -> 'b
                                                - fourtimes 5;
                                                > 20 : int
```

PS4 - ML

Noter

Øverst viser vi mapping funktionalet, som vi tidligere i kapitlet om højereordensfunktioner har studeret i detaljer.

I forbindelse med kaldene af map ser vi, hvordan vi kan overføre en anonym funktion, som i ML noteres ved

```
fn x => [x]
```

Denne funktion svarer til Scheme funktionen

```
(lambda (x) (list x))
```

Nederst vises funktionssammensætningsfunktionalet, som vi også har set på tidligere i kapitlet om højereordensfunktioner.

Hvordan finder ML ud af signaturen af compose? Det fremgår af kroppen af compose, at definitions typen af f skal være den samme som værditypen af g. Endvidere er det klart, at definitionstypen af den resulterende funktion skal være den samme som definitionstypen af g; og værdi typen af den resulterende funktion skal være lig med værditypen af f. Disse observationer giver netop signaturen

```
('a -> 'b *  'c -> 'a) -> 'c -> 'b
```

af compose funktionalet.

Eksempler på funktioner: Quicksort.

```
- fun quick(le:'a*'a->bool) =
   let fun sort [] = []
         | sort [x] = [x]
         | sort (a::rest) = (* the head "a" is the pivot *)
            let fun split(left,right,[]) = sort left @ (a::sort right)
                  | split(left,right,x::l) =
                   if le(x,a) then split(x::left,right,l)
                              else split(left,x::right,l)
            in split([],[],rest)
           end
    in sort
   end;
> val quick = fn : ('a * 'a -> bool) -> 'a list -> 'a list

-  quick (fn (x:int,y) => (x <= y)) [3,2,1,5,7,3,2,1];
> val it = [1,1,2,2,3,3,5,7] : int list
```

PS4 - ML

Noter

Det ses, at quicksort funktionen quick tager en funktion som parameter, der sammenligner to værdier af type ‘a, som jo er en typevariabel, der betegner en vilkårlig type. Quick returnerer den lokale funktion sort, som vi per typeinferens let erkender, tager en liste som input og giver en liste som output. Det er endvidere klart, at denne liste har elementtype ‘a, idet vi ellers ikke meningsfyldt ville kunne sammenligne et element fra listen med et andet element via funktionen le. Alt i alt ser vi, at signaturen af quick er, som den er udmeldt efter > ...

I sort håndteres to specialtilfælde først: den tomme liste og lister af længde 1. I det mere generelle tilfælde sorteres en liste af formen a::rest.

Funktionen split, som er lokal til sort, deler sin tredie parameter op i to lister, med a som pivot element. Hvis det første element E i den 3. parametere er mindre end eller lig med pivot elementet, putter vi E i den venstre delliste, ellers i den højre. Når den 3. parameter er tom, er opdelningen ført tilende, og de to rekursive kald af sort kan aktiveres.

Bemærk, at @ er en infix append funktion.

Brugerdefinerede typer.

- I ML er det muligt for programmøren af definere sine egne typer.
- Tre slags typedefinitioner:
  - Typeforkortelser: **type**.
  - Nye sammensatte data typer: **datatype**.
  - Abstrakte datatyper: **abstype**.

- Ad **datatype**
  - Kan anvendes som 'enumeration types'.
  - Kan anvendes til typekonstruktion ud fra andre (mere primitive) typer.
    - Med eller uden alterntering
  - Kan anvendes til definition af rekursive datatyper.

Noter

Type forkortelser er udtryk for et ganske overfladisk fænomen: giv en simpel eller sammensat typeudtryk et navn.

Vi koncentrerer os om datatype definitioner, som er de mest interessante for os. Til venstre viser vi en pendent til en enumeration type ala Pascal. De tre farver er konstanter, eller rettere i ML sammenhæng parameterløse konstruktorer. Til højre viser vi, hvordan man kunne have defineret ML typen bool (hvis den ikke havde være indbygget); vi viser også den boolske funktion not.

Der er store ligheder mellem ML datatyper og algebraiske specifikationer af datatyper. I begge formalismer er der konstruktorer, som tillader os at generere værdier i typen. Vi kan definere funktioner, som kan observere på typen, og vi kan definere transformatorer.

Eksempler på datatyper (1).

Enumeration type

```
-  datatype color = red | blue  | yellow ;
> type color
   con red: color
   con blue: color
   con yellow: color

-  red;
> red : color
```

Boolean

```
-  datatype bool = true | false ;
> type bool
   con true : bool
   con false : bool

- fun not(true) = false
      | not(false) = true;
> val not = fn : bool -> bool
```

Enumeration typer og typekonstruktion ud fra andre typer:

```
datatype suit = Clubs | Diamonds | Hearts | Spades;
datatype cardvalue = Two | Three | Four | Five | Six | Seven | Eight |
                     Nine | Ten | Jack | Queen | King | Ace;
datatype cardtype = Card of (cardvalue * suit);

map Card [(Six, Hearts), (Ten, Hearts)];
```

PS4 - ML

Noter

Vi ser øverst til venstre hvordan man kan lave en simple opremsning af en række konstanter. Dette svarer til en enumeration type ala Pascal.

Øverst til højre ses, hvordan man i princippet kan definere type Boolean. Der vises også en boolsk funktion, defineret ved patternmatching i to "clauses".

Nederst vises igen to enumeration typer suit og cardvalue, samt en type cardtype, der er konstrueret ud fra suit og cardvalue. Bemærk at Card er en funktion, som kaldes en *konstruktor*. Denne slags funktioner er analoge til konstruktorerne i algebraiske funktioner. Der er i det hele taget store ligheder mellem algebraiske funktioner og ML datatypes.

Eksempler på datatyper (2).

Rekursiv datatype:

```
-  datatype tree = empty |  leaf |  node of tree * tree ;
> type tree
   con empty : tree
   con leaf : tree
   con node:  tree * tree -> tree

-  fun countleaves (empty) = 0
     | countleaves(leaf) = 1
     | countleaves(node (tree1, tree2)) =
           countleaves(tree1) + countleaves(tree2);
> val countleaves = fn:  tree -> int
```

Rekursiv parametrisk datatype:

```
-  datatype 'a list =  nil |  :: of 'a * 'a list ;
> type 'a list
   con nil : 'a list
   con ::  :  'a   *   'a list -> ' a list
```

PS4 - ML

Noter

Det øverste eksempel viser hvordan vi kan definere et binært træ (uden knudeindhold). node er en konstruktor funktion, der tager to binære træer og returnerer et nyt binært træ. Countleaves er en simpel funktion, der tæller bladene i træet.

Observer igen de store ligheder ift. en algebraisk specifikation af et tilsvarende træbegrebt, der ville se ud som følger:

Type tree
Functions
  empty: -> tree
  leaf: -> tree
  node: tree x tree -> tree
  countleaves: tree -> int
Axioms
  for all t, s **in** tree
  countleaves(empty) = 0
  countleaves(leaf) = 1
  countleaves(node(t, s))= countleaves(t) + countleaves(s)
 End

Den nederste datatype på sliden er en mulig definition af datatypen List, som dog er predefineret i ML. Denne type er parametriseret med type variablen 'a. Når man ser bort fra typeparametriseringen af datatypen, og fra konstruktoren med det lidt sære navn ::, er der igen principielle forskelle mellem tree og list.

Eksempler på datatyper (3).

```
datatype forfatter = Forfatter of string | UdenForfatter
datatype bog = Paperback of (string * forfatter list) |
               Hardback of (string * forfatter list)
```

Konstruktorerne:

Forfatter: String -> forfatter
UdenForfatter: -> forfatter
Paperback: string x forfatter list -> bog
Hardback: string x forfatter list -> bog

En selektor funktion

```
fun title (Paperback(s, fl)) = s
|   title (Hardback(s, fl)) = s
```

```
datatype mix = Int of int
             | Str og string
```

En mix liste:
[Int 5, Str "pip", Int 6, Str "slut"]

Noter

Abstrakte datatyper.

- ML stiller en variation af *data type* tilrådighed, som tillader definition af abstrakte datatyper. Denne form for typedefinition kaldes en *abstype.*
- I en abstrakt datatype (abstype) skjules type konstruktorerne (implementationstypen) for klienterne.

Eksempel:

```
-  abstype color = blend of  int * int * int
   with val white = blend(0, 0, 0)
        and red = blend(15, 0, 0)
        and blue = blend(0, 15, 0)
        and yellow = blend(0, 0, 15)
        fun mix(parts1: int, blend(r,b,y),
                parts2: int, blend(rr,bb,yy) ) =
          ...
    end;
```

← Scopet af blend er begrænset til definitionerne mellem with og end.

PS4 - ML

Noter

Implementationstypen i den abstrakte datatype color er konstruktoren blend, som tager tre heltalsparametre. Disse er tænkt som intensiteten af tre grundfarver. Der er ingen principiel forskel på denne konstruktor og konstruktorerne i tree og list, på forrige slide.

Det nye i forhold til datatype er at værdier og funktioner i typen er erklæret lokalt til typen, inde i typen mellem with og end. Men det er jo den sædvanlige måde at håndtere abstrakte datatyper på, som vi kender fra mange andre sammenhænge.

Bemærk at klienter aldrig konfronteres med konstruktoren blend. Klienten arbejder med navngivne værdier, der refererer til bestemte parametriseringer af konstruktorerne, og med funktionen mix.

Vi har ikke her vist kroppen af funktionen mix. På de tre prikkers plads skal vi naturligvis give et udtryk, som giver en ny farveblanding. Bemærk den formelle parameterliste af mix, som indeholder en kraftig destrukturering (med deraf følgende behov for patternmatching) i forhold til kommende aktuelle parametre.

På næste side viser vi et måske endnu mere bekendt eksempel på en abstrakt datatype.

Abstrakt datatype: stack (1).

```
- abstype 'a stack = empty | ins of 'a * 'a stack
  with val emptystack = empty
       exception stackIsEmpty
       fun push(x,s) = ins(x,s)
       fun top(ins(x,s)) = x
         | top(empty) = raise stackIsEmpty
       fun pop(ins(x,s)) = s
         | pop(empty) = empty
  end;

> type 'a  stack
val emptystack = - : 'a stack
exception stackIsEmpty
val push = fn : 'a * 'a stack -> 'a stack
val top = fn : 'a stack -> 'a
val pop = fn : 'a stack -> 'a stack
```

```
- val mystack = push(5, push(6, (push(8,
                          emptystack))));
> val mystack = - : int stack

- top(emptystack);
uncaught exception stackIsEmpty

- top(pop(mystack));
> val it = 6 : int

- val myOtherStack = push(true,push(false,
                                    emptystack));
> val myOtherStack = - : bool stack

- top(pop(myOtherStack));
> val it = false : bool

- val myErrorStack = push(true,
                          push(5, emptystack));
std_in:5.1-5.49 Error:
  operator and operand don't agree (tycon mismatch)
  operator domain: bool * bool stack
  operand:         bool * int stack
  in expression:
    push (true,push (5,emptystack))
```

PS4 - ML

Noter

Dette er en ML elaborering af en det klassiske eksempel: en stak. Bemærk at repræsentationen (i termer af konstruktorerne) ikke kan tilgåes fra klienter; dette ud fra en filosofi om information hiding. Derfor bliver vi nødt til at have en eksplicit defineret funktion push, som blot afspejler funktionaliten af konstruktoren, som vi har kaldt ins.

Bemærk hvorledes resultatet af push udtrykket meldes tilbage:

val mystack = - : int stack

Altså en eller anden værdi, her skrevet som "-" af typen int stack. Der røbes altså intet ud af til om repræsentationen af en stak. Stakken kan blot observeres og påvirkes.

Nederst viser vi, hvordan der opstår en fejl, hvis vi først pusher et heltal, og dernæst en boolsk værdi på en stak.

På næste side viser vi en variation af ovenstående, hvor repræsentationstypen er mere ligefrem, nemlig en liste, som jo er en eksisterende, indbygget type i ML.

Abstrakt datatype: stack (2).

```
- abstype 'a stack1 = stack1 of 'a list
  with val emptystack = stack1([])
       exception stackIsEmpty
       fun push(x, stack1(y ::r)) = stack1(x :: y :: r)
         | push(x, stack1([])) = stack1([x])
       fun top(stack1(x :: r)) = x
         | top(stack1([])) = raise stackIsEmpty
       fun pop(stack1(x :: r)) = stack1(r)
         | pop(stack1([])) = stack1([])
  end;

> type 'a  stack1
val emptystack = - : 'a stack1
exception stackIsEmpty
val push = fn : 'a * 'a stack1 -> 'a stack1
val top = fn : 'a stack1 -> 'a
val pop = fn : 'a stack1 -> 'a stack1
```

```
- val myStack = push(5,
                     push(8, emptystack));
> val myStack = - : int stack1

- top(pop(myStack));
>val it = 8 : int
```


Noter

Bemærk hvor relativ omstændelig push operationen er blevet, i definitionen ovenfor.

Bemærk igen i applikationen til højre, at repræsentationen af stakken ikke røbes i forhold til klienten.

Modulbegrebet i ML.

- Et '*environment*' er en samling af navnebindinger.
- En *ML struktur* er et environment.
  - Der er en syntaktisk form, kaldet et *strukturudtryk*, for ML strukturer.
  - Værdien af et strukturudtryk kan bindes til et navn via en *strukturbinding*.
  - Værdien af et strukturudtryk (en ML struktur) er ikke af 1. klasse.
- Typen af en ML struktur kaldes en *signatur*.
  - Analogt til strukturer understøtter ML *signaturudtryk* og *signaturbindinger*.
  - Man taler om signaturmatching mellem en struktur T og en signatur S, i betydningen at T er af typen S.

```
structure S =                                    signature G =
struct                                            sig
  type t = int;                                     type t
  val x = 3 ;                                       val x: int
  fun fak(x)  =  if x=0 then 1 else x*fak(x-1)      val fak: int -> int
end;                                              end;
```

PS4 - ML

Noter

I ovenstående kan man endvidere udtrykke, at strukturen S skal matche signaturen G. Dette gøres ved:

```
structure S: G =

struct

  type t = int;

  val x = 3 ;

  fun fak(x)  =  if x=0 then 1 else x*fak(x-1)

end;
```

Det er muligt at forstå en signatur som et *interface* af et modul (struktur). En struktur skal mindst have de elementer, som er beskrevet i signaturen; der må godt være flere.

Man kan tilgå de enkelte erklæringer i en struktur ved dot-notation, som vi kender det fra mange andre programmeringssprog. Eksempelvis kan man tilgå heltallet x i S ved at skrive

```
S.x
```

Scheme understøtter ikke håndtering af environments direkte. Men indirekte kan man godt sige, at environments spiller en væsentlig rolle i Scheme (og andre funktionsorienterede sprog), idet et environment er den ene væsentlige del af *closure* begrebet. (Den anden del er selve den syntaktiske form af et lambda-udtryk). Så environments kan i Scheme håndteres indirekte (som 1. klasses objekter til og med) via et nært beslægtet 1. klasses objekt: funktionen.

Når vi i Scheme aggregerede en række definitioner gjorde vi det via lokale definitioner i en funktion. I ML kan vi direkte udtrykke en sådan aggregering i en *structure*. I Scheme var vi nødt til at kalde funktionen (som har de lokale definitioner), og returnere en lokal funktion (dispatch, også kaldet self) for at få hånd på det environment, hvori de lokale, aggregerede definitioner befinder sig. Der henvises til kapitlet om "Funktionsorienteret programmering i Scheme" for yderligere detaljer om dette.

Dataabstraktion via ML modulbegrebet.

- *Structures* og *signaturer* kan benyttes som "byggeklodser" til abstrakte datatyper med information hiding.
  - Lav en structure med alle nødvendige navnebindinger.
  - Lav en signatur, som afspejler interfacet (eksport listen), og pres denne ned over strukturen.
- Dette er et alternativ til brug af *abstypes*

Eksempel:

```
- signature SET =
   sig
     type 'a set
     val empty_set:  'a set
     val union: 'a set * 'a set -> 'a set
   end;
```

```
- abstraction Set:  SET =
   struct
      datatype 'a set = set of 'a list
      val empty_set = set([])
      fun union(set(l1), set(l2)) = ...
   end;
```

PS4 - ML

Noter

Ved at presse signaturen SET ned over struct-en Set opnår vi, at repræsentationen i termer af lister ikke kan ses udaftil.

Bemærk at vi har introduceret en ny type binding, abstraction.

Funktorer.

- En funktor (eng.: functor) er en funktion fra strukturer til strukturer.
- En funktor er begrænset til at være af 1. orden.
- Der er store ligheder mellem *kald af en funktor* og *instantiering af en klasse*.
  - Parametre til funktorer svarer til typeparametre af klasser.

```
signature EQ =
  sig
    type t
    val eq: t * t -> bool
end;

functor F(P: EQ): EQ =
  struct
    type t = P.t * P.t
    fun eq((x,y),(u,v)) = P.eq(x,u) andalso P.eq(y,v)
  end;
```

```
structure X: EQ =
  struct
    type t = int;
    fun eq(x,y) = (x = y)
  end;

structure Y = F(X);
```

PS4 - ML

Noter

Konsekvensen af begrænsningen til 1. ordens funktorer er, at funktorer ikke kan overføres som parametre til funktorer, og at funktorer ikke kan returneres som resultatet af funktorer.

Eksempel på parametriseret abstrakt datatype.

Skitse

```
functor BinarySearchTree(x: Comparable): BinarySearchTreeSig =
struct
 datatype tree = ...   (* heri refereres x.element *)
 local
    val atree: tree = ...
 in
  fun find(y) = ...
  fun insert(y) = ...
  fun delete(y) = ...
 end
end
```

```
structure tree1 = BinarySearchTree(int);
structure tree2 = BinarySearchTree(anothertype)
```

```
signature Comparable
 sig
    type element
    val leq: element * element -> bool
end
```

PS4 - ML

Noter

Bemærk, at indholdet af denne slide kun er en grov skitse, som ikke nødvendigvis overholder ML spillereglerne. Det vigtigste budskab er analogien til den objektorienterede programmeringsstil: klasser og instantieringer.