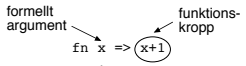
UPPSALA UNIVERSITET

Programmeringsmetodik DV1
Programkonstruktion 1

Moment 9
Om högre ordningens funktioner

Anonyma funktioner igen

En funktion som inte är namngiven kallas för en *anonym* funktion.



När man anropar funktionen så beräknas funktionskroppen *som om* `x` bytts ut mot argumentet i anropet. `x` är *bunden* i funktionen.

`x` är en *godtycklig* identifierare.

Funktionens *typ* bestäms av typen hos det formella argumentet och funktionskroppen. Funktionen ovan har typen `int -> int`.

```
(fn x => x+1)(4*2) ---> (fn x => x+1) 8 ---> 8+1 ---> 9
```


Funktioner är ”första klassens objekt” i ML

Kom ihåg att funktioner kan behandlas som datastrukturer i ML.

- Funktioner kan vara argument till andra funktioner
- Värdet av en funktion kan också vara en funktion
- Funktioner kan förekomma som delar av datastrukturer (t.ex. element i listor).

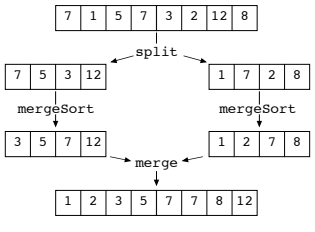
Funktioner kan vara argument

```
fun dotwice(f,x) = f(f x);
fun square x = x*x;

dotwice(square,3) ---> square(square 3)
---> square(3*3) ---> square 9 ---> 9*9 ---> 81

dotwice(fn x=>x+1,3) ---> (fn x=>x+1)((fn x=>x+1) 3)
---> (fn x=>x+1)(3+1) ---> (fn x=>x+1) 4 ---> 4+1 ---> 5
```

Vilken är typen av `dotwice`? `('a->'a)*'a->'a`.

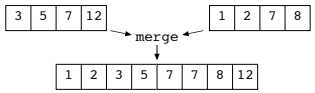
En *första ordningens funktion* har argument som *inte* är funktioner.
En *högre ordningens funktion* har *något* funktionsargument.

`mergeSort` igen



Varför var inte `mergeSort` polymorf?

Original-`merge`




```
(* merge(l1,l2)
   Type: int list*int list -> int list
   Pre: l1 och l2 är sorterade i stigande ordning.
   Post: En lista med alla element i l1 och l2 i stigande ordning.
   Ex.: merge([3,5,7,12],[1,2,7,8]) = [1,2,3,5,7,7,8,12] *)
(* Variant: (length l1)+(length l2) *)
fun merge([],l2) = l2
  | merge(l1,[]) = l1
  | merge(l1,l2) = if hd l1 < hd l2 then
                     hd l1 :: merge(tl l1, l2)
                   else
                     hd l2 :: merge(l1,tl l2)
```


Polymorf `merge`

```
(* merge(less,l1,l2)
   Type: ('a*'a->bool)*'a list*'a list -> 'a list
   Pre: l1 och l2 är sorterade i stigande ordning
        enligt relationen less.
   Post: En lista med alla element i l1 och l2 i
         stigande ordning enligt relationen less.
   Ex.: merge(op <,[3,5,7,12],[1,2,7,8])
                                = [1,2,3,5,7,7,8,12] *)
(* Variant: (length l1)+(length l2) *)
fun merge(_,[],l2) = l2
  | merge(_,l1,[]) = l1
  | merge(less,l1,l2) =
      if less(hd l1,hd l2) then
        hd l1 :: merge(less,tl l1,l2)
      else
        hd l2 :: merge(less,l1,tl l2);
```


Notationen `op`

Funktioner som är infixa operatorer kan inte direkt vara argument:

```
- merge(<,[3,5,7,12],[1,2,7,8]);
! Toplevel input:
! merge(<,[3,5,7,12],[1,2,7,8]);
!
! Ill-formed infix expression
```

ML förväntar sig argument före och efter <. Lösning: skriv `op` före <. `op` ”stänger av” den speciella syntaktiska betydelsen hos <.

```
- merge(op <,[3,5,7,12],[1,2,7,8]);
> val it = [1, 2, 3, 5, 7, 7, 8, 12] : int list
```

Varning: `foo(op *)` gör *inte* vad man tror. Åtgärd: `foo(op * )`
blank

Varför?

Polymorf `mergeSort`

```
(* mergeSort(less,l)
   Type: ('a*'a->bool)*'a list -> 'a list
   Pre: (ingen)
   Post: En lista med alla element i l i stigande
         ordning enligt relationen less.
   Ex.: mergeSort(op <,[7,1,5,7,3,2,12,8])
                              = [1,2,3,5,7,7,8,12] *)
(* Variant: length l *)
fun mergeSort(_,[]) = []
  | mergeSort(_,[x]) = [x]
  | mergeSort(less,l) =
      let
        val (l1,l2) = split l
      in
        merge(less,mergeSort(less,l1),
                   mergeSort(less,l2))
      end;
```


Mer komplicerade relationer

Sortera en lista av namn uttryckta som efternamn-förnamn-par:

```
val namnlista =
[("Bergkvist","Erik"),("Andersson","Arne"),
 ("Björklund","Henrik"),("Eriksson","Lars-Henrik"),
 ("Andersson","Petra"),("Berg","Stina")];
```

När kommer ett namn före ett annat?
Efternamnet skall jämföras först!

```
fun lessName((e1:string,f1:string),(e2,f2)) =
      e1 < e2 orelse e1 = e2 andalso f1 < f2;
```

Varför behövs typdeklarationerna (`:string`)?

```
mergeSort(lessName,namnlista) =
[("Andersson","Arne"),("Andersson","Petra"),
 ("Berg","Stina"),("Bergkvist","Erik"),
 ("Björklund","Henrik"),("Eriksson","Lars-Henrik")]
```


Värdet av en funktion kan vara en funktion

```
fun selectOpr s =
  case s of
      "plus" => op +
    | "minus" => op -
    | "sumto" => sumRange
    | "mean" => fn (x,y) => (x+y) div 2
```

(`sumRange(i,n)` summerar heltalen från `i` till `n`.)

```
selectOpr "plus" ---> op +
selectOpr "mean" ---> fn (x,y) => (x+y) div 2
(selectOpr "plus") (4,6) ---> (op +)(4,6) ---> 10
(selectOpr "sumto") (4,6) ---> sumRange(4,6) ---> 15
```

Vilken är typen av `selectOpr`? `string->(int*int->int)`.
-> är högerassociativ så vi kan utelämna parenteserna och skriva:
`string -> int*int -> int`.

Diverse om användning av funktionsvärden

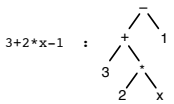
Funktionsapplikation i ML är vänsterassociativ. Ett uttryck som `(f x) y` kan lika gärna skrivas `f x y`.

ML kan inte skriva ut funktionsvärden. Funktionsvärden ersätts med symbolen `fn`.

```
- selectOpr "plus";
> val it = fn : int * int -> int
```


Syntaxträd igen

Strukturen hos en formel kan beskrivas med ett *syntaxträd*.



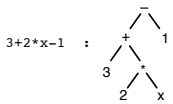

```
datatype expr = Const of int
              | Var of string
              | Plus of expr*expr
              | Minus of expr*expr
              | Times of expr*expr
              | Div of expr*expr;
Minus(Plus(Const 3,Times(Const 2,Var "x")),
      Const 1)
```


Beräkning av uttryck igen

```
exception UnDefVar of string;

(* eval(expr,var,value)
   TYPE: expr*string*int->int
   PRE: var är den enda variabel som förekommer i expr.
        Ingen division med 0 förekommer i expr.
   POST: Värdet av expr uträknat.
   Ex: eval(Minus(Plus(Const 3,Times(Const 2,Var "x")),Const 1),
            "x",4) = 10 *)
(* VARIANT: Storleken hos expr. *)
fun eval(Const c,_,_) = c
  | eval(Var v,var,value) = if v = var then
                              value
                            else
                              raise UnDefVar v
  | eval(Plus(e1,e2),var,value) = eval(e1,var,value) +
                                  eval(e2,var,value)
  | eval(Minus(e1,e2),var,value) = eval(e1,var,value) -
                                   eval(e2,var,value)
  | eval(Times(e1,e2),var,value) = eval(e1,var,value) *
                                   eval(e2,var,value)
  | eval(Div(e1,e2),var,value) = eval(e1,var,value) div
                                 eval(e2,var,value);
```


Lite annorlunda syntaxträd

Operatorn lagras i ett fält i varje nod.




```
datatype expr = Const of int
              | Var of string
              | Op of string*expr*expr;

Op("minus",Op("plus",Const 3,
                     Op("times",Const 2,Var "x")),
           Const 1)
```


Beräkning av uttryck

```
exception UnDefVar of string;

(* eval(expr,var,value)
   TYPE: expr*string*int->int
   PRE: var är den enda variabel som förekommer i expr.
        Ingen division med 0 förekommer i expr.
        Bara operatorer som är kända av selectOpr förekommer i
        expr.
   POST: Värdet av expr uträknat.
   EX: eval(Op("minus",Op("plus",Const 3,
                          Op("times",Const 2,Var "x")),
               Const 1),
            "x",4) = 10 *)
(* VARIANT: Storleken hos expr *)
fun eval(Const c,_,_) = c
  | eval(Var v,var,value) = if v = var then
                              value
                            else
                              raise UnDefVar v
  | eval(Op(opr,e1,e2),var,value) =
      selectOpr opr (eval(e1,var,value),eval(e2,var,value));
```

Datatypen `order`

Ibland behöver man kunna jämföra två objekt både för storlek och likhet. Det finns en fördefinierad datatyp `order` för att beskriva resultatet av jämförelsen:

```
datatype order = LESS | GREATER | EQUAL;
```

Flera inbyggda funktioner har värde av typ `order`:

`Int.compare`, `String.compare`, `Real.compare`, ...

```
Int.compare(3,3) = EQUAL
String.compare("Aha","Ahaaaa!") = LESS
Real.compare(46.4,32.0) = GREATER
```

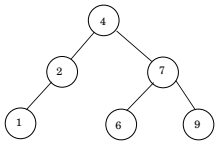
Speciellt för abstrakta datatyper kan det vara praktiskt att ha en `order`-värd jämförelsefunktion.

Exempel på `order`-värd jämförelse

Givet två rationella tal representerade som `(p1,q1)` och `(p2,q2)` så gäller att `(p1,q1)` är mindre än `(p2,q2)` omm `p1*q2<p2*q1`.

```
fun rationalCompare((p1,q1),(p2,q2)) =
  case p1*q2-p2*q1 of
    0 => EQUAL
  | d => if d < 0 then
           LESS
         else
           GREATER;

rationalCompare((7,14),(6,9))
--> case 7*9-14*6 of 0 => EQUAL | d => if ...
--> case ~21 of 0 => EQUAL | d => if ...
--> if ~21 < 0 then LESS else GREATER
--> if true then LESS else GREATER
--> LESS
```


Binära sökträd igen



Kom ihåg att funktionerna för sökning i binära sökträd inte utan vidare kan vara polymorfa eftersom man måste använda storleksjämförelse vid sökning i träden.

Deklaration av polymorfa binära sökträd

```
abstype ('a,'b) table =
    Empty of ('a*'a->order)
  | Bintree of ('a*'a->order)*'a*'b*
               ('a,'b) table*('a,'b) table
  (* …dokumentation… *)
with
  fun empty compare = Empty compare;
  fun update .....;
  fun lookup .....;
  fun remove .....;
  fun toList .....;
end;
```

Den nödvändiga jämförelsefunktionen lagras i trädet när en tom tabell skapas. (OBS att `empty` nu är en funktion.)

Nyckeltypen (`'a`) behöver nu inte vara en likhetstyp. Varför?

Funktionen `lookup`

```
(* lookup(key,table)
   TYPE: 'a*('a*'b) table -> 'b option
   PRE: (ingen)
   POST: SOME v om key finns i table och v är
         motsvarande värde. NONE annars. *)
(* VARIANT: Största djupet hos ett löv i table *)
  fun lookup(key,Empty _) = NONE
    | lookup(key,Bintree(compare,key',value,
                         left,right)) =
        case compare(key,key') of
          EQUAL => SOME value
        | LESS => lookup(key,left)
        | GREATER => lookup(key,right);
```


Funktionen `update`

```
(* update(key,value,table)
   TYPE: 'a*'b*('a*'b) table -> ('a*'b) table
   POST: Tabellen table uppdaterad med värdet
         value för nyckeln key *)
(* VARIANT: största djupet av ett löv i table *)
  fun update(key,value,Empty compare) =
        Bintree(compare,key,value,
                Empty compare,Empty compare)
    | update(key,value,
             Bintree(compare,key',value',
                     left,right)) =
        case compare(key,key') of
          EQUAL =>
            Bintree(compare,key,value,left,right)
        | LESS => Bintree(compare,key',value',
                          update(key,value,left),right)
        | GREATER => Bintree(compare,key',value',
                             left,update(key,value,left));
```


Användning av polymorfa sökträd

```
val tabell =
   update("Lars-Henrik",1057,
    update("Roland", 7606,
     empty String.compare));
```

Tabellen innehåller information om Lars-Henrik och Roland.

```
lookup("Lars-Henrik",tabell) = SOME 1057
lookup("Roland",tabell) = SOME 7606
lookup("Kalle",tabell) = NONE
```


Mönster vid listrekursion

```
fun squareList [] = []
  | squareList (first::rest) =
      square first::squareList rest;
fun negList [] = []
  | negList (first::rest) = ~first::negList rest;
```

Dessa funktioner har samma struktur – de skiljer sig bara i funktionsnamnen och de rödmarkerade delarna.

Man kan skriva en högre ordningens funktion `mapl` som fångar strukturen och som har ett funktionsargument för olikheten:

```
fun mapl(f,[]) = []
  | mapl(f,first::rest) = f first::mapl(f,rest);
fun squareList l = mapl(square,l);
fun negList l = mapl(~,l);
```


Mönster vid listrekursion (forts.)

```
fun mapl(f,[]) = []
  | mapl(f,first::rest) = f first::mapl(f,rest);
fun squareList l = mapl(square,l);

squareList [1,~2,4]
--> mapl(square,[1,~2,4])
--> square 1::mapl(square,[~2,4])
--> 1*1::mapl(square,[~2,4])
--> 1::mapl(square,[~2,4])
--> 1::square ~2::mapl(square,[4])
--> 1:: ~2* ~2::mapl(square,[4])
--> 1::4::mapl(square,[4])
--> 1::4::square 4::mapl(square,[])
--> 1::4::4*4::mapl(square,[])
--> 1::4::16::mapl(square,[])
--> 1::4::16::[]
= [1,4,16]
```


”Stuvade” funktioner

```
fun plus(x,y) = x+y;          (plus: int*int->int)
fun plus' x = fn y => x+y;    (plus': int->int->int)

plus(4,6) --> 4+6 --> 10
plus' 4 6 --> (fn y => 4+y)6 --> 4+6 --> 10
```

- `plus'` är en *stuvad* (*curried* – efter H.B. Curry) form av `plus`.
- `plus'` tar ”ett argument i taget”.
- Funktionen som är värdet av `plus'` ”minns” bindningen av `x`.

```
val add1 = plus' 1;     (add1 nu bunden till fn y => 1+y)
add1 3 --> (fn y => 1+y)3 --> 1+3 --> 4
```

`fun` har en speciell form för att definiera stuvade funktioner:

```
fun plus' x y = x+y;
```

är samma definition av `plus'` som den ovan.

Beräkning av funktioner

En stuvad form av `mapl` finns inbyggd i ML under namnet `map`.

```
mapl(f,l) = map f l

fun squareList l = map square l;
```

`map` kan också användas för att *beräkna funktionen* `squareList`:

```
val squareList = map square;
```


Mer komplicerade programmönster

```
fun length [] = 0
  | length (first::rest) = 1+length rest;
fun addList [] = 0
  | addList (first::rest) = first+addList rest;
fun append([],x) = x
  | append(first::rest,x) = first::append(rest,x);
```

Gemensamt för dessa:

- En operation utförs på första elementet av listan och resultatet av det rekursiva anropet (blått)
- Det finns ett bestämt värde (”startvärde”) vid basfallet (rött).

(Not: även `1+` är principiellt en operation på första elementet i listan och resultatet av det rekursiva anropet även om det är en operation som inte utnyttar värdet av det första listelementet.)

Funktionen `foldr`

Den inbyggda funktionen `foldr` utför mer generell rekursion över listor än `map`.

`foldr` kan utläsas som ”fold from right”.

```
(* foldr f e l
   TYPE: ('a*'b->'b)->'b->'a list->'b
   PRE: (ingen)
   POST: f(x1,f(x2,...f(xn,e)...)) om l=[x1,x2,...,xn]
*)
(* VARIANT: längden av l *)
fun foldr f e [] = e
  | foldr f e (first::rest) =
      f(first,foldr f e rest);
```


Beräkning av funktioner med `foldr`

```
fun foldr f e [] = e
  | foldr f e (first::rest) =
      f(first,foldr f e rest);
```

Vad är värdet av `foldr (op +) 0`? `f` binds till `(op +)` och `e` binds till `0`. Resultatet blir alltså en funktion som är definierad som:

```
fun foo [] = 0
  | foo (first::rest) =
      (op +)(first,foo rest);
```

...vilket (utom syntaxen) är den rekursiva definitionen av `addList`. (Den beräknade funktionen är anonym – jag har kallat den för `foo`.)

`addList` kan alltså definieras som:

```
val addList = foldr (op +) 0;
```

eller – kanske lite tydligare –

```
fun addList l = foldr (op +) 0 l;
```


Användning av `foldr`

Exempel på användning av `foldr`:

```
fun length l = foldr (fn (x,y) => 1+y) 0 l;
fun addList l = foldr (op +) 0 l;
fun append(x,y) = foldr (op ::) y x;
fun smallest (first::rest) =
      foldr Int.min first rest;
fun split l =
      foldr (fn (first,(x,y)) => (first::y,x))
            ([],[]) l;
```

`map` kan definieras med hjälp av `foldr`:

```
fun map f l =
      foldr (fn (x,y) => f x::y) [] l;
```


Funktionen `foldl`

Det finns också en inbyggd funktion för iteration (svansrekursion) över listor:

```
(* foldl f e l
   TYPE: ('a*'b->'b)->'b->'a list->'b
   POST: f(xn,...f(x2,f(x1,e))...) om l=[x1,x2,...,xn]
*)
(* VARIANT: längden av l *)
fun foldl f e l =
  let
    fun foldlAux([],ack) = ack
      | foldlAux(first::rest,ack) =
          foldlAux(rest,f(first,ack))
  in
    foldlAux(l,e)
  end;
```

Exempel: `fun rev l = foldl (op ::) [] l;`

Funktionen `filter`

`List.filter` kan användas för att välja ut vissa delar av en lista.

T.ex. `List.filter (fn x => x>0) [1,~2,4,0] = [1,4]`

```
(* filter p l
   TYPE: ('a -> bool) -> 'a list -> 'a list
   POST: En lista av de element i l som
         gör p sann *)
(* VARIANT: längden av l *)
fun filter p [] = []
  | filter p (first::rest) =
      if p first then
        first::filter p rest
      else
        filter p rest;
```

`filter` kan även definieras med hjälp av `foldr`:

```
fun filter p l =
 foldr (fn (x,y) => if p x then x::y else y) [] l
```