



UPPSALA
UNIVERSITET

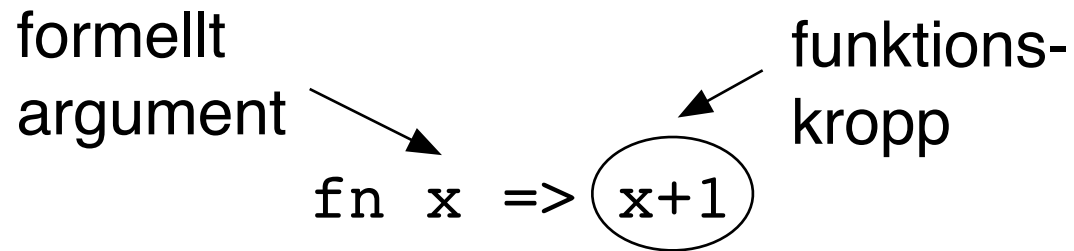
Programkonstruktion och datastrukturer

Moment 8

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
```

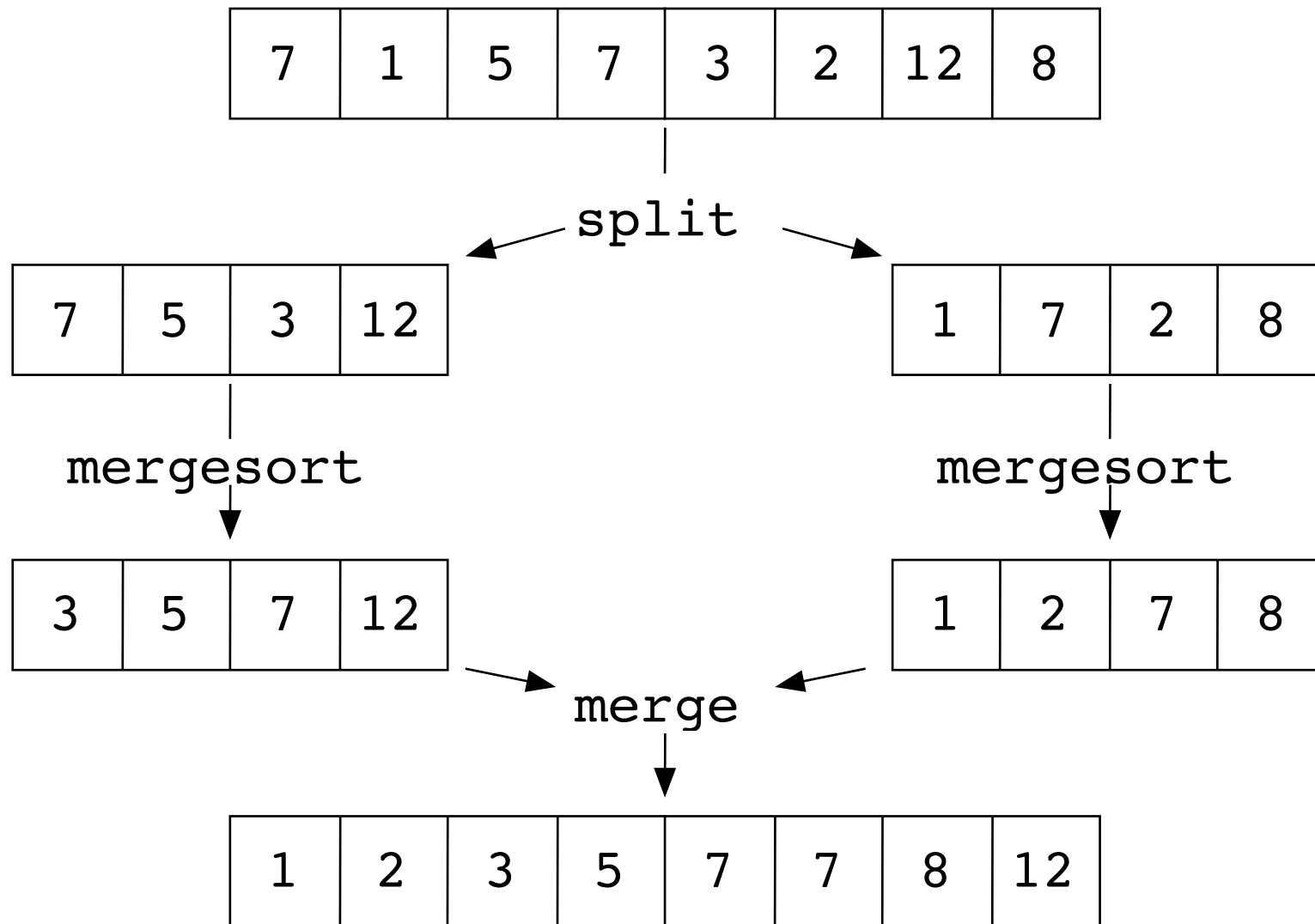
Vilken är typen av dotwice?

(Prova gärna genom att skriva in definitionen i ML.)

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

En *högre ordningens funktion* har *något* funktionsargument.

merge sort igen

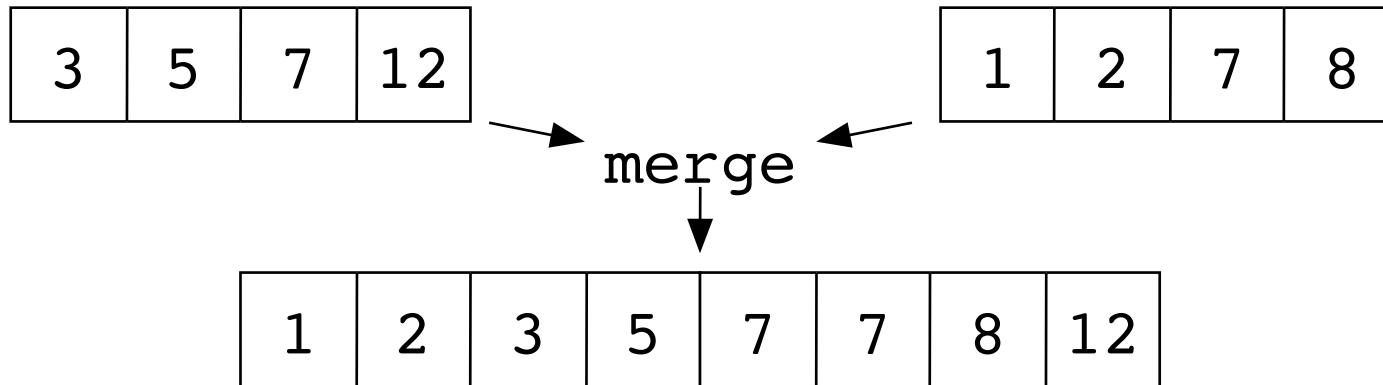


Varför är inte merge sort polymorf?

```
> fun sort [] = []  
#   | sort [x] = [x]  
#   | sort xs =  
#       let  
#           val (ys, zs) = split xs  
#       in  
#           merge(sort ys, sort zs)  
#       end;  
val sort = fn : int list -> int list
```

Algoritmen för merge sort bryr sig inte om vilken typ det är på de data man sorterar. Så varför är inte funktionen `sort` polymorf?

Original-merge



```
(* merge(L,M)
  TYPE: int list*int list -> int list
  PRE: L och M är sorterade i stigande ordning.
  POST: Listan L@M med elementen 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 ([], M) = M
| merge (L, []) = L
| merge (L as x::xs, M as y::ys) =
    if x > y then
      y :: merge (L, ys)
    else
      x :: merge (xs, M);
```

Polymorf merge

```
(* merge(less,L,M)
   TYPE: ('a*'a->bool)*'a list*'a list -> 'a list
   PRE:  less är en ordningsrelation. L och M är
          sorterade i stigande ordning enligt less.
   POST: Listan L@M med elementen 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 L)*(length M) *)
fun merge(_, [], M) = M
  | merge(_, L, []) = L
  | merge(less, L as x::xs, M as y::ys) =
    if less(y,x) then
      y :: merge(less,L,ys)
    else
      x :: merge(less,xs,M);
```


Notationen op


Funktioner som är infix operatorer kan inte direkt vara argument:

```
> merge(<,[3,5,7,12],[1,2,7,8]);  
Warning-(<) has infix status but was not  
preceded by op.
```

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 *)`
Varför? blank 

(Fast just Poly/ML är snäll när det gäller dessa saker.)

Polymorf merge sort

```
(* polysort(less,L)
  TYPE: ('a*'a->bool)*'a list -> 'a list
  PRE: less är en ordningsrelation
  POST: En lista med alla element i L i stigande
        ordning enligt less.
  EXAMPLE: polysort(op <,[5,7,3,12,1,7,2,8,13])
            = [1,2,3,5,7,7,8,12,13] *)

(* VARIANT: length L *)
fun polysort(_,[]) = []
  | polysort(_,[x]) = [x]
  | polysort(less,xs) =
    let
      val (ys, zs) = split xs
    in
      merge(less,polysort(less,ys),
            polysort(less, zs))
    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)?

```
polysort(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 -  
  | "times" => op *  
  | "div" => op div  
  | "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 hos selectOpr?

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`.

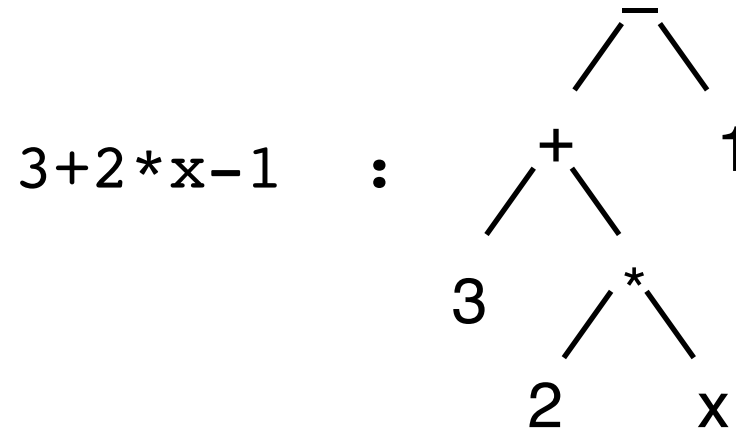
```
> selectOpr "plus";  
val it = fn : int * int -> int
```

Typen för `selectOpr` är `string->(int*int->int)`.

Typkonstruktorn `->` är *högerassociativ* så vi kan i detta fall ta bort parenteserna och skriva: `string->int*int->int`.

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

`selectOpr` används för att välja funktion givet informationen om operator i trädet.

```
(* eval(expr,table)
  TYPE: expr*(string*int) list->int
  PRE: Alla variabler i expr finns i table.
       Ingen division med 0 förekommer i expr.
  POST: Värdet av expr uträknat.
  EX: eval(Op("minus",
              Op("plus",Const 3,
                  Op("times",Const 2,Var "x")),
              Const 1),
          insert([], "x", 4)) = 10 *)
(* VARIANT: Storleken hos expr *)
fun eval(Const c, _) = c
  | eval(Var v, values) = search(values, v)
  | eval(Op(opr, e1, e2), values) =
    selectOpr opr (eval(e1, values), eval(e2, values));
```

Tabell med funktionsvärden

I stället för att använda `selectOpr` för att välja funktion kan vi skapa en tabell som kopplar operatorerna i trädet till ML-funktioner.

```
val optable =  
  [ ("plus", op +),  
    ("minus", op -),  
    ("times", op * ),  
    ("div", op div),  
    ("sumto", sumRange),  
    ("mean", fn (x,y) => (x+y) div 2) ];
```

Vilken typ har `optable`?

Beräkning av uttryck med funktionstabell

search används för att hitta avsedd funktion givet informationen om operator i trädet.

```
(* eval(expr,table)
```

```
  TYPE: expr*(string*int) list->int
```

```
  PRE: Alla variabler i expr finns i table.
```

```
        Ingen division med 0 förekommer i expr.
```

```
  POST: Värdet av expr uträknat.
```

```
  EX: eval(Op("minus",  
             Op("plus",Const 3,  
                Op("times",Const 2,Var "x")),  
             Const 1),  
        insert([], "x",4)) = 10 *)
```

```
(* VARIANT: Storleken hos expr *)
```

```
fun eval(Const c, _) = c
```

```
  | eval(Var v, values) = search(values,v)
```

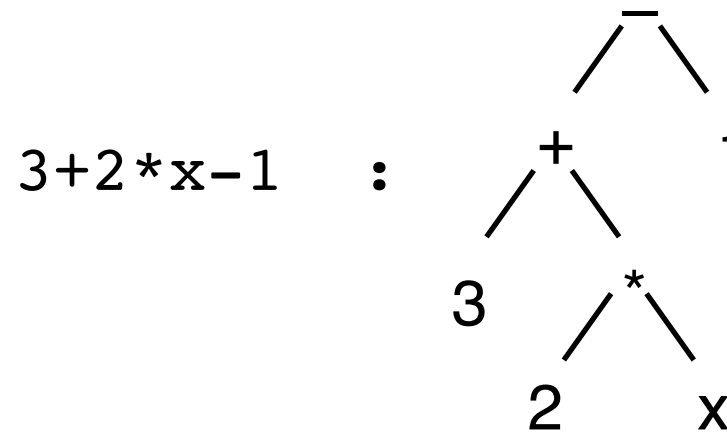
```
  | eval(Op(opr, e1, e2),values) =
```

```
    search(optable,opr)
```

```
      (eval(e1,values),eval(e2,values));
```

Syntaxträd med funktioner i noderna

Själva ML-funktionen som skall utföra beräkningen lagras i ett fält i varje nod.



```
datatype expr = Const of int  
              | Var of string  
              | Op of (int*int->int)*expr*expr;
```

```
Op(op -, Op(op +, Const 3,  
              Op(op *, Const 2, Var "x" ) ),  
  Const 1)
```

Beräkning med funktioner i noderna

Nu behövs ingen översättning utan ML-funktionen kan anropas direkt!

```
(* eval(expr, table)
  TYPE: expr*(string*int) list->int
  PRE: Alla variabler i expr finns i table.
       Ingen division med 0 förekommer i expr.
  POST: Värdet av expr uträknat.
  EX: eval(Op(op -,
              Op(op +, Const 3,
                  Op(op *, Const 2, Var "x")),
                  Const 1),
          insert([], "x", 4)) = 10 *)

(* VARIANT: Storleken hos expr *)
fun eval(Const c, _) = c
  | eval(Var v, values) = search(values, v)
  | eval(Op(opr, e1, e2), values) =
    opr(eval(e1, values), eval(e2, values));
```

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 — *abstraherar bort* skillnaderna — och som har ett funktionsargument för skillnaden:

```
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 `map1` finns inbyggd i ML under namnet `map`.

```
map1(f,l) = map f l
```

```
fun squareList l = map square l;
```

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

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

Beräkning av specialiserade sorteringsfunktioner

Antag att vi skrev om den polymorfa `polysort` som en stuvad funktion med typ `('a * 'a -> bool) -> 'a list -> 'a list`

Då går det lätt att göra specialversioner för olika typer:

```
> val sortInt = polysort op <;
```

```
val sortInt = fn : int list -> int list
```

```
> sortInt [5,7,3,12,1,7,2,8,13];
```

```
val it = [1,2,3,5,7,7,8,12,13] : int list
```

```
> val sortName = polysort lessName;
```

```
val sortName = fn : (string * string) list -> (string * string) list
```

```
> sortName namnlista;
```

```
val it = [("Andersson", "Arne"), ("Andersson", "Petra"),  
("Berg", "Stina"), .....] : (string * string) list
```


Mer komplicerade programmonster

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

fun addList [] = 0
  | addList (first::rest) = first+addList rest;

fun append([],y) = y
  | append(first::rest,y) = first::append(rest,y);
```

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).

Att `length` passar in i mönstret syns bättre om man skriver så här:

```
| length (first::rest) =
    (fn (x,y) => 1+y)(first,length rest);
```

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
```

Högre ordningens funktioner för vektorer

I biblioteket `Vector` finns varianter av funktionerna `map`, `foldr` och `foldl` som arbetar på vektorer i stället för listor.

Exempel: Summera heltal i en vektor (svansrekursivt):

```
(* sumVector a
   TYPE: int vector -> int
   POST: Summan av elementen i a *)
fun sumVector a = Vector.foldl (op +) 0 a;
```

Exempel: Bilda en lista av elementen i en vektor:

```
(* toList a
   TYPE: 'a vector -> 'a list
   POST: En lista av elementen i vektorn a. *)
fun toList a = Vector.foldr (op ::) [] a;

> sumVector (Vector.fromList [18,12,-5,12,10]);
val it = 47 : int
```

Funktionell programmering och multiprocessorer

Hastighetsökningen vid utveckling av datorer bygger i dag huvudsakligen på att datorerna kan göra flera beräkningar samtidigt (t.ex. flerkärniga processorer) än att varje enskild beräkning går snabbare.

För att utnyttja detta måste program skrivas så att de ingående beräkningarna kan fördelas på flera processorer. Detta är *svårt*.

Eftersom beräkningsordningen inte spelar roll i funktionell programmering kan olika delberäkningar i ett funktionellt program delas upp på olika processorer och utföras samtidigt.

I ML görs beräkningarna från vänster till höger, men det är inget principiellt krav. I det följande exemplet avviker vi när det behövs.

Exempel: parallelisering

Antag att vi har ett problem där en och samma operation skall utföras på en mängd olika data och man sedan skall kombinera resultatet på något sätt.

Sådana problem lämpar sig väl för uppdelning på olika processorer, men hur skall man skriva programmen så att det blir möjligt?

Ett exempel kan vara att ifrån en samling med textdokument (lagrade som strängar) söka ut precis de dokument som innehåller en given sträng.

Sökningarna kan göras i flera dokument samtidigt på olika processorer

Exempel: mapReduce

För att uttrycka detta definierar vi funktionen mapReduce som är en kombination av map och foldr.

```
(* mapReduce r m e l
  TYPE: ('a->'b)->('b*'c->'c)->'c->'a list->'c
  PRE: (ingen)
  POST: r(m x1,r(m x2,...r(m xn,e)...)) om
        l=[x1,x2,...,xn]
*)
(* VARIANT: längden av l *)
fun mapReduce r m e [] = []
  | mapReduce r m e (x::l) =
    r(m x,mapReduce r m e l);
```

Exempel: findString

Vi behöver en funktion för att söka i ett dokument...

```
(* findString'(s,x,pos)
  TYPE: string*string*pos->bool
  PRE: 0≤pos≤size x-size s
  POST: true om s är en delsträng av x vid eller
        eller efter position pos, false annars
  EXAMPLE: findString("bc","abcd",1) = true
            findString("bc","abcd",2) = false    *)

(* VARIANT: size x - pos *)
fun findString'(s,x,pos) =
  pos + size s <= size x andalso
  (String.substring(x,pos,size s)=s orelse findString'(s,x,pos+1));

(* findString(s,x)
  TYPE: string*string->bool
  PRE: (ingen)
  POST: true om s är en delsträng av x,
        false annars
  EXAMPLE: findString("bc","abcd") = true
            findString("bb","abcd") = false    *)

fun findString(s,x) = findString'(s,x,0);
```

Exempel: checkDoc

En hjälpfunktion...

```
(* checkDoc(s,doc)
   TYPE: string*string->string list
   PRE: (ingen)
   POST: [x] om s är en delsträng av x, annars []
   EXAMPLE: checkDoc("bc","abcd") = ["abcd"]
             checkDoc("bc","abbd") = []          *)

fun checkDoc(s,doc) =
    if findString(s,doc) then
        [doc]
    else
        [];
```

Exempel: searchDocuments

Så till sist huvudfunktionen...

```
(* searchDocuments(s,l)
  TYPE: string*string list->string list
  PRE: (ingen)
  POST: En lista av de element i l som innehåller
        s som delsträng.
  EXAMPLE: searchDocuments("bc",["abcd","abbd"])=
            ["abcd"]      *)
fun searchDocuments(s,docs) =
  mapReduce (op @)
            (fn doc => checkDoc(s,doc))
            [] docs;
```

searchDocuments: hur funkar det?

```
searchDocuments("bc", ["abcd", "abbd"])  
—> mapReduce op @ (fn x => ...) [] ["abcd", "abbd"]  
—> (fn x => ...) "abcd" @  
    mapReduce op @ (fn x => ...) [] ["abbd"]  
—> checkDoc("bc", "abcd") @  
    mapReduce op @ (fn x => ...) [] ["abbd"]
```

Här väljer vi att *inte* ta vänstraste uttrycket först!

```
—> checkDoc("bc", "abcd") @ (fn x => ...) "abbd" @  
    mapReduce op @ (fn x => ...) [] []  
—> checkDoc("bc", "abcd") @ checkDoc("bc", "abbd") @  
    mapReduce op @ (fn x => ...) [] [] inte här heller!  
—> checkDoc("bc", "abcd") @ checkDoc("bc", "abbd") @ []
```

Anropen av checkDoc kan i princip *utföras samtidigt!*

```
—> ["abcd"] @ [] @ [] —> ["abcd"]
```

Googles MapReduce

Den här idén ligger bakom Googles programmeringsmetod "MapReduce" som de använder för att köra program med massiva parallellberäkningar på hundratusentals processorer.

Se t.ex. Wikipediaartikeln om MapReduce!