



Programkonstruktion

Moment 6 Om listor och polymorfa typer

Program- och datastrukturer

	<u>Programstruktur</u>	<u>Datastruktur</u>
Sammansättning:	uttryck som argument	tupler
Alternativ:	if/case/matching	?? (inte ännu)
Upprepning:	rekursion	listor (strängar)

↑
dagens ämne

PK 2008/2009 moment 6

Sida 1

Uppdaterad 2007-10-21

PK 2008/2009 moment 6

Sida 2

Uppdaterad 2007-10-21

Listor

En *lista* är en *ordnad* uppsättning (*sekvens*, *följd*) av *element*, som kan ha godtycklig typ.

- [18, 12, ~5, 12, 10] : int list
- [2.0, 5.3, 3.7, ~1E5] : real list
- ["Lars", "Henrik", "PK"] : string list
- [(1,"A"), (2,"B")] : (int*string) list
- [Math.sin, fn x=>x+1.0] : (real->real) list
- [[1], [2, 3]] : int list list

Karakteristiskt för listor:

- Alla element i en viss lista måste ha *samma typ*.
- En lista kan innehålla *godtyckligt många* element (eller *inga alls*).
- *Samma element* kan finnas *flera gånger* i samma lista.

PK 2008/2009 moment 6

Sida 3

Uppdaterad 2007-10-21

Praktisk användning av listor

En förteckning (böcker i ett bibliotek):

```
["Introduction to Programming Using Standard ML",
 "Mathematical Logic", "Software Engineering",
 "Applications of Formal Methods"] : string list
```

En tabell (telefonanknytningar på inst. för IT):

```
[("Lars-Henrik Eriksson", 1057), ("Jesper Bengtson", 3156),
 ("Mattias Wiggberg", 3176), ("Aletta Nylén", 7595)]
 : (string*int) list
```

En kalender:

```
[((11,5),[(10,"Föreläsning PK"),(13,"Föreläsning PK"),
 (15,"Möte med Arne")]),
 ((11,6),[(10,"Ledningsgruppsmöte"),(13,"Föreläsning PK")])]
 : ((int*int)*(int*string) list) list
```

PK 2008/2009 moment 6

Sida 4

Uppdaterad 2007-10-21

Listuttryck

Precis som tupler kan listor konstrueras av uttryck som beräknar ett visst värde.

- [18, 3+9, 5-7, size("abc"), 10] → [18, 12, ~2, 3, 10]
- ["A" ^ "B", Int.toString(5)] → ["AB", "5"]
- [(1+1,2), qplus((1,2),(1,3))] → [(2,2), (5,6)]

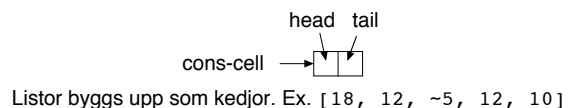
En tom lista: []

PK 2008/2009 moment 6

Sida 5

Uppdaterad 2007-10-21

Struktur hos listor



Man kommer bara åt element genom att gå igenom länkarna. Tidsåtgången (antal steg) är proportionell mot elementets plats i listan.

Pilarna (*pekarna*) kan inte ändras. Listor måste byggas *bakifrån*!

I ML (och t.ex. Java) är pekarna *implicita* – de är ett sätt att *beskriva* datastrukturer men är *inte värden* i språket. I en del andra språk (t.ex. C) är de *explicita*, dvs. separata värden med egen datatyp.

PK 2008/2009 moment 6

Sida 6

Uppdaterad 2007-10-21

Grundläggande operationer på listor.

`hd` (head) ger *första elementet* i en lista.
`hd [1,2,3] → 1` (Resultatet blir *inte* [1])
`hd []` ger exekveringsfel (Empty).
`tl` (tail) ger en lista *utom* första elementet.
`tl [1,2,3] → [2,3]`
`tl ["Hej"] → []`
`tl []` ger exekveringsfel (Empty).
`null` talar om ifall en lista är tom.
`null [1,2,3] → false`
`null (tl ["Hej"]) → true`
`::` (*cons*) skapar en ny listcell av ett *element* och en lista.
`1::[2,3] → [1,2,3]` (`::` är en *infix operator*)
`[1,2,3]` är egentligen `1::(2::(3::[]))`, vilket även kan skrivas `1::2::3::[]` (`::` är *högerassociativ*).

PK 2008/2009 moment 6

Sida 7

Uppdaterad 2007-10-21

Polymorfa typer

`hd [1,2,3] → 1`, `hd ["Kalle","Anka"] → "Kalle"`
Vilken typ har `hd`? `int list → int`? `string list → string`?
`hd` har den *polymorfa* (mångformiga) typen
`'a list → 'a`
Där `'a` är en *typvariabel*.
Alla typvariabler i ML har namn som börjar med apostrof.
`hd` kan användas i alla sammanhang där det behövs en typ som man kan få genom att ersätta `'a` med en annan typ, t.ex. `int` eller `string`. (Detta kallas en *instans* av `'a list → 'a`).
`= : 'a*'a -> bool` (Dubbla apostrofer anger en *likhetstyp*).
Listtyper är likhetstyper omm elementtypen är det!
Polymorfi är inte samma sak som överlagring (t.ex. `hos +, <`).

PK 2008/2009 moment 6

Sida 8

Uppdaterad 2007-10-21

Specifikation av listoperationerna

`hd l`
Type: `'a list → 'a`
Pre: `l <> []`
Post: första elementet i `l`
`tl l`
Type: `'a list → 'a list`
Pre: `l <> []`
Post: `l` utan sitt första element.
`null l`
Type: `'a list → bool`
Pre: (ingen)
Post: `l = []`
Av tekniska skäl bör man använda `null` (eller matchning) och inte direkt jämförelse med tom lista.
`x :: l`
Type: `'a*'a list → 'a list`
Pre: (ingen)
Post: En lista som har `x` som första element och fortsätter som listan `l`.

PK 2008/2009 moment 6

Sida 9

Uppdaterad 2007-10-21

Typen av tomma listan

Vilken typ har den tomma listan, `[]`?
`[]` kan vara en tom lista av heltal, en tom lista av strängar etc....
`tl [1] → [], tl ["Kalle"] → []`
`[]` måste alltså höra till både typen `int list` och `string list` (och alla andra sorters listor).
`[]` är ett *värde* som har den polymorfa typen `'a list`.

PK 2008/2009 moment 6

Sida 10

Uppdaterad 2007-10-21

Definierade polymorfa funktioner

Polymorfi fungerar därför att den polymorfa funktionen (t.ex. `hd`) inte utför någon operation på *värden* av den obestämda typen.

Definierade funktioner blir också polymorfa om de har argument som de inte gör någon operation med:

```
(* swap(x,y)
   Type: 'a*'b -> 'b*'a *)
fun swap(x,y) = (y,x)

(* second l
   Type: 'a list -> 'a *)
fun second l = hd(tl l)

(* second0 l
   Type: int list -> int *) second0 blir inte polymorf.
fun second0 l = hd(tl l)+0
```

PK 2008/2009 moment 6

Sida 11

Uppdaterad 2007-10-21

Strängar och teckenlistor

Strängar är en följd av tecken: `"Hej, hopp"`
Teckenlistor (`char list`):
`["#H", "#e", "#j", "# ", "#h", "#o", "#p", "#p"]`
är också en följd av tecken – vad är skillnaden?

- Strängar och teckenlistor representeras (troligen) helt olika.
- Strängar kräver mindre minnesutrymme.
- Man har direkt åtkomst till delar av strängar utan att behöva "läsa förbi" den del av strängen som ligger före.
- Man kan matcha listor men inte strängar.
- Rekursion över listor är mer naturlig än över strängar.

Konverteringsfunktioner:

```
explode: string -> char list
implode: char list -> string
```

PK 2008/2009 moment 6

Sida 12

Uppdaterad 2007-10-21

Räkna antalet element i listan

```
(* length l
  Type: 'a list->int
  Pre: (ingen)
  Post: Antal element i listan
  Ex: length [17,42] = 2
      length [] = 0 *)
(* Variant: längden av l *)
fun length l = if null l then
  0
  else
    1+length(tl l)

length [17,42]
-> if null [17,42] then 0 else 1+length(tl [17,42])
-> 1+length(tl [17,42])
-> 1+length [42]
-> 1+(if null [42] then 0 else 1+length(tl [42]))
-> 1+(1+length(tl [42]))
-> 1+(1+length [])
-> 1+(1+(if null [] then 0 else 1+length(tl [])))
-> 1+(1+0)
-> 2
```

PK 2008/2009 moment 6

Sida 13

Uppdaterad 2007-10-21

Sätt samman två listor

```
(* append(x,y)
  Type: 'a list*'a list -> 'a list
  Pre: (ingen)
  Post: En lista med elementen i x följda av elementen i y.
  Ex: append([1,2],[3,4]) = [1,2,3,4]
      append([1,2],[]) = [1,2] *)
(* Variant: längden av x *)
fun append(x,y) = if null x then
  y
  else
    hd x :: append(tl x,y)

append([1,2],[3,4])
-> if null [1,2] then [3,4] else hd [1,2] ::append(tl [1,2],[3,4])
-> hd [1,2]::append(tl [1,2],[3,4])
-> 1::append([2],[3,4])
-> 1::(if null [2] then [3,4] else hd [2]::append(tl [2],[3,4]))
-> 1::(hd [2]::append(tl [2],[3,4]))
-> 1::(2::append([], [3,4]))
-> 1::(2::(if null [] then [3,4] else hd [] ::append(tl [], [3,4]))
-> 1::(2::[3,4])
-> 1::[2,3,4]
-> [1,2,3,4]
```

PK 2008/2009 moment 6

Sida 14

Uppdaterad 2007-10-21

Mönstermatchning med listor

Listnotationen kan användas i matchning:

[1,2,3] matchar mönstret [x,y,z].

x binds till 1, y binds till 2 och z binds till 3.

:: är egentligen ingen funktion utan en (*datastruktur*)konstruktor.

Symbolen :: representerar en listcell – och kan också användas i matchning.

[1,2,3] matchar mönstret x::y. x binds till 1, y binds till [2,3].

[1,2,3] matchar mönstret x::y::z. x binds till 1, y binds till 2

och z binds till [3].

PK 2008/2009 moment 6

Sida 15

Uppdaterad 2007-10-21

Rekursion med mönstermatchning

```
(* append(x,y)
  Type: 'a list*'a list -> 'a list
  Pre: (ingen)
  Post: Elementen i x följda av de i y.
  Ex: append([1,2],[3,4]) = [1,2,3,4]
      append([1,2],[]) = [1,2] *)
(* Variant: längden av x *)
fun append([],y) = y
  | append(first::rest, y) = first::append(rest,y)
```

Obs. betydelsen av att namnge första argumentet i specifikationen – annars kan man inte skriva eftervillkoret på vettigt sätt!

```
append([1,2],[3,4])
-> 1::append([2],[3,4])
-> 1::(2::append([], [3,4]))
-> 1::(2::[3,4])
-> 1::[2,3,4]
-> [1,2,3,4]
```

PK 2008/2009 moment 6

Sida 16

Uppdaterad 2007-10-21

Vänd på en lista

append använde vanlig (icke svans-) rekursion, vilket bevarade ordningen hos listelementen när listan byggs med ::.

Svansrekursion ger omvänd ordning hos beräkningen, vilket är precis vad vi vill om vi skall vända ordningen på en lista.

```
(* rev l
  Type: 'a list -> 'a list
  Pre: (ingen)
  Post: En lista med elementen i l i omvänd ordning.
  Ex: rev [1,2,3,4] = [4,3,2,1]
      rev [] = [] *)
fun rev l = revAux(l,[])
(* revAux(l,ack)
  Type: 'a list*'a list -> 'a list
  Pre: (ingen)
  Post: En lista med elementen i l i omvänd ordning följda av elementen i ack i rättvänd ordning.
  Ex: revAux([3,4],[2,1]) = [4,3,2,1] *)
(* Variant: längden av l *)
fun revAux([],ack) = ack
  | revAux(first::rest,ack) = revAux(rest,first::ack)
```

PK 2008/2009 moment 6

Sida 17

Uppdaterad 2007-10-21

Vänd på en lista (forts.)

```
(* revAux l ack *)
fun revAux([],ack) = ack
  | revAux(first::rest,ack) =
    revAux(rest,first::ack)
```

```
(* rev l *)
fun rev l = revAux(l,[])

rev [1,2,3,4]
-> revAux([1,2,3,4],[])
-> revAux([2,3,4],[1])
-> revAux([3,4],[2,1])
-> revAux([4],[3,2,1])
-> revAux([], [4,3,2,1])
-> [4,3,2,1]
```

PK 2008/2009 moment 6

Sida 18

Uppdaterad 2007-10-21

Fler inbyggda listfunktioner

length, rev och append är så vanliga att de är fördefinierade.
append heter som inbyggd funktion @ och är en infix operator:

`[1,2] @ [3,4] = [1,2,3,4]`

OBS! Skilj på `::` (cons) och `@` (append)!

`::` skapar en ny listcell, medan `@` sätter ihop hela listor:

```
"A" :: ["B"] = ["A","B"]
      |> nil   |> nil
      "B"     "A"  "B"

["A"] @ ["B"] = ["A","B"]
  |> nil   |> nil
  "A"     "B"  "A"  "B"

"A" @ ["B"] eller ["A"] :: ["B"] ger typfel!
```

PK 2008/2009 moment 6

Sida 19

Uppdaterad 2007-10-21

Komplexiteten hos append (@)

För att koppla ihop två listor måste den första listan *kopieras*.

Tidsåtgången för detta är $O(n)$, där n är längden på första listan.

Antag att man försöker bygga en lista "framifrån" med hjälp av @:

```
[] @ [1] -> [1]           0 rekursiva anrop av @
[1] @ [2] -> [1,2]       1 rekursiva anrop av @
[1,2] @ [3] -> [1,2,3]  2 rekursiva anrop av @
[1,2,3] @ [4] -> [1,2,3,4] 3 rekursiva anrop av @
```

För att på detta sätt bygga en lista med n element krävs alltså

$0+1+\dots+n = n(n+1)/2 = (n^2+n)/2$ rekursiva anrop av @.

Tidsåtgången är *kvadratisk* – $O(n^2)$!

Bygg listan bakifrån med `::` istället! `1 :: (2 :: (3 :: (4 :: [])))`.

Tidsåtgången blir linjär – $O(n)$.

PK 2008/2009 moment 6

Sida 20

Uppdaterad 2007-10-21

Sökning i listor

Är ett bestämt värde ett element i en given lista?

```
(* member(x,l)
   Type: 'a*'a list -> bool
   Pre: (ingen)
   Post: true om x är ett element i l, false annars
   Ex: member(42,[17,42]) = true
       member(25,[17,42]) = false *)
(* Variant: längden av l *)
fun member(_,[]) = false
  | member(x,first::rest) = x = first orelse
                           member(x,rest)

member(42,[17,42])
-> 42 = 17 orelse member(42,[42])
-> 42 = 42 orelse member(42,[])
-> true
```

Notera att member är polymorf över *likhetstyper* (typvariabeln har dubbla apostrofer) – detta för att man gör en likhetsjämförelse.

PK 2008/2009 moment 6

Sida 21

Uppdaterad 2007-10-21

Sökning i listor (2)

Hämta ett bestämt värde ur en tabell.

```
(* lookup(key,table)
   Type: 'a*('a*'b) list -> 'b
   Pre: Exakt en post i tabellen table har nyckeln key
   Post: table är en lista av par (k,v) där k är en nyckel och v
        ett tabellvärde. Värdet av lookup är det tabellvärde som
        motsvarar nyckeln key,
   Ex: lookup("Mattias Wiggberg",
              [("Lars-Henrik Eriksson",1057),
               ("Mattias Wiggberg",3176)]) = 3176 *)
(* Variant: längden av table *)
fun lookup(key,(k,v)::rest) = if key = k then
                               v
                             else
                               lookup(key,rest)
```

Om key inte finns så avbryts exekveringen med ett fel. Vilket?

ML ger en varning när detta program läses in. Vilken?

Notera att man *inte* kan skriva: Varför?

```
fun lookup(key,(key,v)::rest) = v
  | lookup(key,::_:rest) = lookup(key,rest)
```

PK 2008/2009 moment 6

Sida 22

Uppdaterad 2007-10-21

Struktur hos program som arbetar med listor

Eftersom listor har godtycklig längd blir program som arbetar med listor i allmänhet rekursiva.

Oftast har rekursiva program över listor följande form:

```
(* f(...,l,...) *)
(* Variant: längden av l *)
fun f(...,[],...) = ...
  | f(...,first::rest,...) = ... f(...,rest,...) ...
```

Denna typ av rekursion kallas *strukturell rekursion* – rekursion över strukturen hos listan.

PK 2008/2009 moment 6

Sida 23

Uppdaterad 2007-10-21

Hanoi-programmet med listrepresentation

```
(* hanoi(n,from,via,to)
   Type: int*string*string*string->
        (string*string) list
   Pre: n>=0. from, via och to är olika strängar.
   Post: En lista av par som beskriver hur man
        skall flytta n brickor i Hanoi-spelet
        från pinne from till pinne to med hjälp
        av pinne via. Varje par anger att en
        bricka skall flyttas från den pinne som
        anges av första komponenten till den
        pinne som anges av andra komponenten.
   Ex.: hanoi(0,"A","B","C") = []
        hanoi(1,"A","B","C") = [("A","C")]
        hanoi(2,"A","B","C") = [("A","B"),("A","C"),("B","C")] *)
(* Variant: n *)
fun hanoi(0,_,_,_) = []
  | hanoi(n,from,via,to) = hanoi(n-1,from,to,via) @
                          (from,to) :: hanoi(n-1,via,from,to)
```

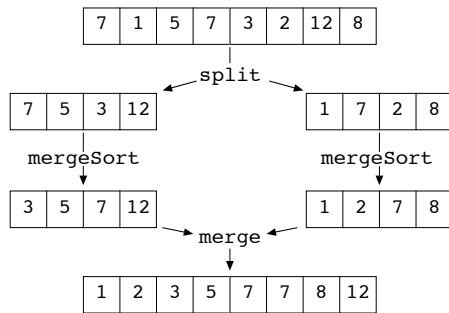
PK 2008/2009 moment 6

Sida 24

Uppdaterad 2007-10-21

Sortering

Princip för funktionen mergeSort.

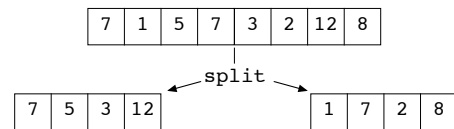


PK 2008/2009 moment 6

Sida 25

Uppdaterad 2007-10-21

Uppdelning (split) av två listor



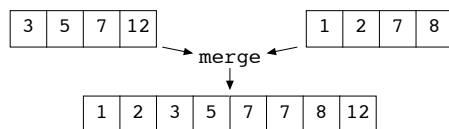
```
(* split l
  Type: 'a list -> 'a list*'a list
  Pre: (ingen)
  Post: (l1,l2) sådan att l1 och l2 innehåller elementen ifrån
        l i godtycklig ordning och längden av l1 och l2 som mest
        skiljer ett element.
  Ex.: split [7,1,5,7,3,2,12,8] = ([7,5,3,12],[1,7,2,8] *)
  (* Variant: length l *)
  fun split [] = ([],[])
    | split (first::rest) = let
        val (l2,l1) = split rest
        in (first::l1, l2)
        end
```

PK 2008/2009 moment 6

Sida 26

Uppdaterad 2007-10-21

Sammanvävning (merge) av två listor



```
(* 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)
```

PK 2008/2009 moment 6

Sida 27

Uppdaterad 2007-10-21

mergeSort

```
(* mergeSort l
  Type: int list -> int list
  Pre: (ingen)
  Post: En lista med alla element i l i stigande ordning.
  Ex.: mergeSort [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 l = let
        val (l1,l2) = split l
        in merge(mergeSort l1, mergeSort l2)
        end
```

Varför krävs två basfall – tom lista och en elementslista?

Varför är inte mergeSort polymorf?

Komplexiteten hos mergeSort är $O(n \log n)$, vilket är typiskt för sorteringsalgoritmer.

(n är alltså storleken på problemet – i detta fall listans längd.)

PK 2008/2009 moment 6

Sida 28

Uppdaterad 2007-10-21

Vektorer

- För att komma åt ett element i en lista krävs en tidsåtgång som ökar när listan blir större.
- Man har ibland behov av att komma åt data i *konstant tid*.
- Nästan alla datorer har arbetsminne med sekventiellt ordnade celler där olika delar går att komma åt i konstant tid.
- De flesta programspråk har en datastruktur som direkt motsvarar strukturen hos minnet, kallade vektorer eller arrayer.

0	1	2	3	4
18	12	~5	12	10

En heltalsvektor med 5 element och deras *index*.

Till skillnad från en lista som består av ett antal celler i kedja så är en vektor *ett* objekt – att komma åt varje element går alltså lika fort.

Jämför med listan på sidan 6!

PK 2008/2009 moment 6

Sida 29

Uppdaterad 2007-10-21

Vektorer i ML

I ML skrivs vektorer på samma sätt som listor, men med tecknet # framför. Vektorn i exemplet skrivs #[18, 12, ~5, 12, 10] och har typen int vector.

Funktioner för vektorer i ML finns i biblioteket vector. Exempel:

- vector.fromList skapar en vektor från en lista
- vector.length ger längden av en vektor
- vector.sub tar fram ett element ur en vektor
- vector.extract tar ut en följd av element ur en vektor
- vector.concat slår ihop vektorer

Eftersom en vektor är *ett* objekt så finns ingen motsvarighet till hd, tl, :: o.dyl. för vektorer. Man får alltså betala för åtkomst i konstant tid med att det är besvärligare att skapa/ändra vektorer.

PK 2008/2009 moment 6

Sida 30

Uppdaterad 2007-10-21

Vektorfunktioner (1)

```
Vector.fromList x
TYPE: 'a list -> 'a vector
POST: En vektor av samma längd som listan
      där cellerna i tur och ordning
      innehåller elementen i listan
EXAMPLE: Vector.fromList [18, 12, ~5, 12, 10] =
         #[18, 12, ~5, 12, 10]

Vector.length a
TYPE: 'a vector -> int
POST: Antal element i vektorn a
EXAMPLE: Vector.length #[18,12,~5,12,10] = 5

Vector.sub(a,i)
TYPE: 'a vector*int -> 'a
PRE: 0 ≤ i < Vector.length a
POST: Elementet med index i inom vektorn a.
EXAMPLE: Vector.sub(#[18,12,~5,12,10],2) = ~5
```

PK 2008/2009 moment 6

Sida 31

Uppdaterad 2007-10-21

Vektorfunktioner (2)

```
Vector.concat l
TYPE: 'a vector list -> 'a vector
POST: En vektor som innehåller elementen, i tur
      och ordning, från vektorerna i listan l.
EXAMPLE: Vector.concat(#[18],#[12,~5],#[],#[12,10])
         = #[18,12,~5,12,10]

Vector.extract(a,i,x)
TYPE: 'a vector*int*int option -> 'a
PRE: 0 ≤ i ≤ Vector.length a, x=NONE eller
      x=SOME j, där 0 ≤ j ≤ Vector.length a - i
POST: En vektor med element från a med början
      vid index i. Om x=SOME j tas j element ut
      om x=NONE tas alla element till slutet.
EXAMPLE: Vector.extract(#[18,12,~5,12,10],1,NONE)
         = #[12,~5,12,10]
         Vector.extract(#[18,12,~5,12,10],1,SOME 2)
         = #[12,~5]
```

PK 2008/2009 moment 6

Sida 32

Uppdaterad 2007-10-21

Summera elementen i en heltalsvektor

```
(* sumVector a
TYPE: int vector -> int
POST: Summan av elementen i a *)
fun sumVector a =
  let
    (* sumVectorAux(a,i)
    TYPE: int vector*int -> int
    PRE: 0 ≤ i ≤ Vector.length a
    POST: Summan av elem. i a fr.o.m index i *)
    (* VARIANT: Vector.length a - i *)
    fun sumVectorAux(a,i) =
      if i >= Vector.length a then
        0
      else
        Vector.sub(a,i) + sumVectorAux(a,i+1)
    in
      sumVectorAux(a,0)
    end;
  - sumVector #[18,12,~5,12,10];
  > val it = 47 : int
```

PK 2008/2009 moment 6

Sida 33

Uppdaterad 2007-10-21