



Programmeringsmetodik DV1 Programkonstruktion 1

Moment 10 Om sidoeffekter In/utmatning och imperativ programmering

PK16PM1 HT-05 moment 10 Sida 1 Uppdaterad 2005-09-12

Sekvensering

När sidoeffekter finns blir beräkningsordningen mycket viktig. Tänk på att ML beräknar uttryck från vänster till höger (och inifrån och ut). Ibland kan man vilja beräkna flera uttryck efter varandra endast för deras sidoeffekter utan att bry sig om värdet. För detta ändamål finns ML-konstruktionen

```
e1 ; e2 ; ... ; en-1 ; en
```

vars värde är värdet av e_n , men som dessutom först beräknar e_1 t.o.m. e_{n-1} .

$e_1; e_2; \dots; e_{n-1}; e_n$ är meningsfull bara om beräkningen av e_1 t.o.m. e_{n-1} har sidoeffekter.

(Obs att ; även avslutar uttryck som man matar in till ML ; som sekvensoperator måste stå inom parenteser, let...end eller dylikt.)

PK16PM1 HT-05 moment 10 Sida 5 Uppdaterad 2005-09-12

Skapa strömmar

`openIn s`
TYPE: string -> instream
PRE: s skall vara namnet på en fil som finns.
POST: En instream kopplad till filen s.
SIDE-EFFECTS: Olika anrop till openIn ger olika strömmar.
EXAMPLE : openIn "frithiof.txt"

`openOut s : string -> ostream`
PRE: s skall vara namnet på en fil som går att skriva/skapa.
POST: En ostream kopplad till filen s.
SIDE-EFFECTS: Filen s skapas om den inte finns.
Olika anrop till openOut ger olika strömmar.
EXAMPLE : openOut "nyfil"

PK16PM1 HT-05 moment 10 Sida 9 Uppdaterad 2005-09-12

Terminering av rekursion vid inläsning

När man läser in data från en ström i ett rekursivt program så bryts i allmänhet rekursionen av filsut. Man kan inte veta i förväg hur många rekursiva anrop som skall ske utan att veta hur strömmen ser ut. Det vet man normalt inte innan den är läst! Desutom är det principiellt omöjligt om man läser från en ström som skapas successivt (t.ex. en ström som läser från ett tangentbord).

Som rekursionsvariant kan man ange t.ex.

```
(* VARIANT: oläst längd på strömmen is *)
```

Om funktionen läser från is och rekursionen avslutas vid filsut.

PK16PM1 HT-05 moment 10 Sida 13 Uppdaterad 2005-09-12

Referentiell transparens

I ren funktionell programmering kan man byta "lika mot lika" utan att programmets funktion ändras. Detta kallas referentiell transparens. Orsaken är att en beräkning inte har någon effekt annat än att beräkna ett värde (om man bortser från tids/minnesåtgång).

```
f x*g y kan bytas mot g y*f x.
f x ^ f x kan bytas mot let
    val fx = f x
    in
    fx^fx
end
```

Det sista bytet lönar sig resursmässigt om beräkningen av f x tar lång tid eller har ett stort objekt som värde.

Ett sådant byte kan t.o.m. göras automatiskt av ML-systemet.

PK16PM1 HT-05 moment 10 Sida 2 Uppdaterad 2005-09-12

Utmatning

```
print x : string -> unit
SIDE-EFFECTS: x skrivs ut på terminalfönstret.
POST: ()
EXAMPLE: print "Hej\n" = () och skriver samtidigt raden "Hej\n" på terminalfönstret.
```

```
print "Hej\n"; print "Hej\n"
-> (); print "Hej\n" (och "Hej\n" skrivs ut.)
-> (); (); (och "Hej\n" skrivs ut.)
-> ();
```

Alltså print "Hej\n" = print "Hej\n" – trots det kan man inte byta print "Hej\n"; print "Hej\n" mot let val ph = print "Hej\n" in ph; ph end print beräknar inte bara ett värde utan skriver också på terminalen. En ny rad skrivs vid varje anrop – print har en sidoeffekt. Antalet anrop och deras inbördes ordning blir viktig.

PK16PM1 HT-05 moment 10 Sida 6 Uppdaterad 2005-09-12

Ett urval funktioner för inmatning (1)

`inputLine is`
TYPE: instream -> string
POST: Nästa rad från strömmen is (inkl. \n), eller tom sträng om inget kunde läsas (t.ex. filsut)
SIDE-EFFECTS: Läspekaren för is flyttas fram.

Exempelfunktionen readline kan definieras så här:
fun readline() = inputLine stdIn;

`input1 is`
TYPE: instream -> char option
POST: SOME c om nästa tecken i strömmen är c, NONE om inget kunde läsas (t.ex. filsut).
SIDE-EFFECTS: Läspekaren för is flyttas fram.

`closeIn is`
TYPE: instream -> unit
POST: ().
SIDE-EFFECTS: Avslutar användningen av strömmen is

PK16PM1 HT-05 moment 10 Sida 10 Uppdaterad 2005-09-12

Två vanliga fel...

```
(* readToList f
TYPE: string -> string list
PRE: f är namnet på en fil som kan läsas
POST: En lista av raderna i filen f *)
(* VARIANT: återstående rader i f *)
fun readToList f =
let
val is = openIn f
in
if eofStream is then
[]
else
inputLine is :: readToList f
end;
```

- readToList öppnar en ny ström vid varje anrop och läser därför första raden i filen om och om igen.
- Inga strömmar stängs någonsin.

PK16PM1 HT-05 moment 10 Sida 14 Uppdaterad 2005-09-12

Inmatning

Hitills har all in/utmatning av data gått via ML-systemets användargränssnitt. I praktiken räcker inte det – programmen måste själva kunna mata in/ut data – t.ex. till/från filer.

Låt oss anta att det finns en funktion readLine med spec.:

```
readLine x
TYPE : unit -> string
SIDE-EFFECTS: En rad läses från terminalfönstret.
POST: en sträng med den rad som lästs in.
EXAMPLE: readLine() = "Hej\n" om man skriver in raden "Hej\n" på tangentbordet.
```

Körexempel:

```
- readLine();
Hej, hopp.
> val it = "Hej, hopp.\n" : string
```

PK16PM1 HT-05 moment 10 Sida 3 Uppdaterad 2005-09-12

In/utmatning i ML – TextIO

För in/utmatning (I/O) finns olika bibliotek i ML, bl.a. biblioteket TextIO för in- och utmatning av text. För att slippa skriva TextIO, före alla namn på funktioner och värden i detta bibliotek kan man använda deklARATIONEN

```
open TextIO;
```

i sitt program. Den gör att ML automatiskt söker i biblioteket TextIO efter namn den inte känner igen annars. (Detta fungerar med alla bibliotek, även t.ex. String och List.) Man kan använda flera open-deklARATIONER. Jag förutsätter i fortsättningen att man använt deklARATIONEN open TextIO.

(Obs! open är ingens ersättning för load.)

PK16PM1 HT-05 moment 10 Sida 7 Uppdaterad 2005-09-12

Ett urval funktioner för inmatning (2)

`inputAll is`
TYPE: instream -> string
POST: Resterande text i strömmen is till filsut.
SIDE-EFFECTS: Läspekaren för is flyttas fram.

`eofStream is`
TYPE: instream -> bool
POST: true om läsningen av strömmen is har nått till slutet av filen, false annars.
SIDE-EFFECTS: Inga.

PK16PM1 HT-05 moment 10 Sida 11 Uppdaterad 2005-09-12

Ett urval funktioner för utmatning (1)

`output(os,s)`
TYPE: ostream*string -> unit
POST: ()
SIDE-EFFECTS: Strängen s skrivs ut på strömmen os.

`flushOut os`
TYPE: ostream -> ()
POST: ().
SIDE-EFFECTS: Skriver ut bufferten för strömmen os

`closeOut os`
TYPE: ostream -> unit
POST: ().
SIDE-EFFECTS: Avslutar användningen av strömmen os

Den inbyggda funktionen print skulle kunnat definieras så här:

```
fun print s = (output(stdOut,s);flushOut stdOut);
```

PK16PM1 HT-05 moment 10 Sida 15 Uppdaterad 2005-09-12

Sidoeffekter

Vad blir readLine() ^ readLine()? Låt oss anta att användaren först skriver raden "Hej" sedan "hopp". (Varför slutar strängarna med \n?)

```
readLine() ^ readLine()
-> "Hej\n" ^ "readLine()"
-> "Hej\n" ^ "hopp\n"
-> "Hej\nhopp\n"
```

Observera att readLine() # readLine()

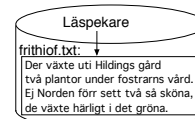
Man kan alltså inte byta readLine() ^ readLine() mot let val r1 = readLine() in r1 ^ r1 end

Det beror på att readLine inte bara beräknar ett värde utan också läser en ny rad från terminalen vid varje anrop – readLine har en sidoeffekt. Antalet anrop och deras inbördes ordning blir viktig.

PK16PM1 HT-05 moment 10 Sida 4 Uppdaterad 2005-09-12

Strömmar

Vid I/O krävs omfattande administration, bl.a. skall filer lokaliseras och man behöver hålla reda på hur långt i en fil man har läst/skrivit.



TextIO finns abstrakta datatyper instream och ostream kallade strömmar. All information rörande I/O finns i strömmarna.

För I/O till terminalfönster finns i TextIO två fördefinierade strömmar stdIn av typ instream och stdOut av typ ostream.

PK16PM1 HT-05 moment 10 Sida 8 Uppdaterad 2005-09-12

Exempel på inläsning från ström

```
(* readToList f
TYPE: string -> string list
PRE: f är namnet på en fil som kan läsas
POST: En lista av raderna i filen f *)
fun readToList f =
let
(* readToListAux is
TYPE: instream -> string list
POST: En lista av återstående rader i is *)
(* VARIANT: återstående rader i is *)
fun readToListAux is =
if eofStream is then
(closeIn is; [])
else
inputLine is :: readToListAux is
in
readToListAux (openIn f)
end;
```

PK16PM1 HT-05 moment 10 Sida 12 Uppdaterad 2005-09-12

Ett urval funktioner för utmatning (2)

`output1(os,c)`
TYPE: ostream*char -> unit
POST: ()
SIDE-EFFECTS: Tecknet c skrivs ut på strömmen os.

PK16PM1 HT-05 moment 10 Sida 16 Uppdaterad 2005-09-12

Tillägg till befintliga filer

openOut skapar en ny fil eller skriver över en befintlig fil. Man kan också vilja *lägga till* ny information i slutet av en befintlig fil.

```
openAppend s
TYPE: string -> ostream
PRE: s skall vara namnet på en fil som går att skriva eller skapa.
POST: En ostream kopplad till filen s.
SIDE-EFFECTS: Filen s skapas om den inte finns.
Skrivningen börjar i slutet av filen.
Olika anrop till openAppend ger olika strömmar.
```

PK1&PM1 HT-05 moment 10

Sida 17

Uppdaterad 2005-09-12

Tillägg till filer – exempel

Om vi har en fil testut.txt, med innehåll:

```
Hej, Hopp
I det gröna
```

och gör:

```
- val x = openAppend "testut.txt";
> val x = <ostream> : ostream
- output(x, "sade Prithiof\n");
> val it = () : unit
- closeOut x;
> val it = () : unit
så kommer testut.txt att innehålla:
```

```
Hej, Hopp
I det gröna
sade Prithiof
```

PK1&PM1 HT-05 moment 10

Sida 18

Uppdaterad 2005-09-12

I/O av annat än text

Vill man mata in/ut data av annan typ än string måste man omvandla den till till/från strängar först.

Omvandlingen måste man programmera själv om inte inbyggd funktion finns (t.ex. Int.toString)

Konvertering från strängar är besvärlig. I allmänhet har strängen olika typer av information blandade efter varandra, t.ex:

"beräkna 1+23*4 slut". Ur denna sträng skulle man vilja få: strängen "beräkna", talet 1, tecknet +, talet 23, tecknet *, talet 4, och strängen "slut".

Dessutom vill man kanske ha 1+23*4 representerat som ett träd! Detta problem kallas *syntaxanalys* eller *parsning*. Parsning behandlas i senare kurser.

PK1&PM1 HT-05 moment 10

Sida 19

Uppdaterad 2005-09-12

Metodik för utveckling av program med i/o

- Tänk *noga* på i vilken ordning saker och ting utförs
- Försök dela upp programmet i en del som utför all in/utmatning och en annan del som inte alls utför någon i/o.
- Testa alla data som läses in – lita inte på att de är riktiga!

För *interaktiva* program (program som arbetar i dialog med en användare) bör man dessutom tänka på:

- Programmet får inte kunna avbrytas pga fel så att användaren förlorar data.
- Tänk igenom dialogen med användaren – strukturera programmet efter den.

PK1&PM1 HT-05 moment 10

Sida 20

Uppdaterad 2005-09-12

En kalkylator i ML

Skriv en enkel fyrfunktionskalkylator i ML! Den skall läsa in ett tal, en operation och ett tal till och visa resultatet. Därefter skall man kunna utföra "kedjering" med nya operationer och tal.

```
Skriv ett tal: 3           Skriv ett tal: 3.5
Skriv en operation: +     Skriv en operation: d
Skriv ett tal: 4           Felaktig operation!
Resultatet är: 7.0        Skriv en operation: /
Skriv en operation: +     Skriv ett tal: 0
Skriv ett tal: 5.2        Fel!
Resultatet är: 12.2       Skriv ett tal: 0
Skriv en operation: c     Skriv en operation: q
Skriv ett tal: abc
Felaktigt tal!
```

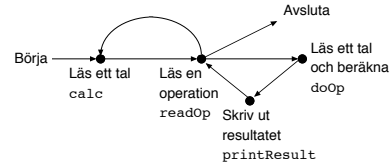
PK1&PM1 HT-05 moment 10

Sida 21

Uppdaterad 2005-09-12

Tillståndsdigram

När man planerar programmet kan man göra ett *tillståndsdigram* över dialogen.



Programmet skrivs som en *tillståndsmaskin* som motsvarar diagrammet. Varje tillstånd motsvaras av en funktion i programmet.

PK1&PM1 HT-05 moment 10

Sida 22

Uppdaterad 2005-09-12

Programstrukturen

Programkoden visas i *top-down* ordning med huvudfunktionerna först och hjälpfunktionerna sist. I programfilen måste funktionerna ligga i motsatt ordning eftersom en funktion måste definieras innan de används.

Funktionerna i tillståndsmaskinen är ömsesidigt rekursiva.

Programmet inleds med deklARATIONERNA:

```
load "Real";
open TextIO;
```

PK1&PM1 HT-05 moment 10

Sida 23

Uppdaterad 2005-09-12

Tillståndsmaskinen (1)

```
(* calc x
  TYPE: unit -> unit
  SIDE-EFFECTS: Huvudfunktion i kalkylatorn *)
fun calc() = readOp(readNum())

(* readOp x
  TYPE: real -> unit
  SIDE-EFFECTS: Läser en operation i kalkylatorn *)
and readOp x =
  (print "Skriv en operation: ";
   case inputLine stdin of
     "q\n" => ()
   | "c\n" => calc()
   | opr => case selectOp opr of
             SOME f => doOp(x,f)
             | NONE => (print "Felaktig op.\n";
                       readOp x))
```

PK1&PM1 HT-05 moment 10

Sida 24

Uppdaterad 2005-09-12

Tillståndsmaskinen (2)

```
(* doOp(x,f)
  TYPE: real*(real*real -> real) -> unit
  SIDE-EFFECTS: Läser en tal och utför en operation i kalkylatorn *)
and doOp(x,f) =
  (printResult(f(x,readNum()))
   handle _ => (print "Fel!\n"; calc()))

(* printResult x
  TYPE: real -> unit
  SIDE-EFFECTS: Skriver resultat i kalkylatorn *)
and printResult x =
  (print ("Resultatet är: " ^
         Real.toString x ^ "\n");
   readOp x);
```

PK1&PM1 HT-05 moment 10

Sida 25

Uppdaterad 2005-09-12

Välj operation

```
(* selectOp x
  TYPE: string -> (real*real -> real) option
  POST: Om x är namnet på en känd operation
  returneras SOME f, där f är en funktion som utför operationen. NONE annars.
  EXAMPLE: selectOp "+\n" = SOME (op +) *)
fun selectOp "+\n" = SOME (op +)
  selectOp "-\n" = SOME (op -)
  selectOp "*\n" = SOME (op *)
  selectOp "/\n" = SOME (op /)
  selectOp _ = NONE;
```

PK1&PM1 HT-05 moment 10

Sida 26

Uppdaterad 2005-09-12

Läs in tal

```
(* readNum x
  TYPE: unit -> real
  SIDE-EFFECTS: Läser från terminalen
  POST: Ett inläst flyttal *)
fun readNum () =
  (print("Skriv ett tal: ");
   case Real.fromString (inputLine stdin) of
     SOME x => x
   | NONE => (print "Felaktigt tal!\n";
             readNum ());)
```

PK1&PM1 HT-05 moment 10

Sida 27

Uppdaterad 2005-09-12

Återblick: ingenting ändras...

I ren funktionell programmering kan man aldrig ändra någonting – bara skapa nytt. Detta är en förutsättning för att byta "lika mot lika".

```
- val x = [1,2,3,4];
> val x = [1, 2, 3, 4] : int list
- rev x;
> val it = [4, 3, 2, 1] : int list
- x;
> val it = [1, 2, 3, 4] : int list
x är oförändrad efter anropet av rev x!

- val y = 1;
> val y = 1 : int
- fun f x = x+y;
> val f = fn : int -> int
- val y = 2;
> val y = 2 : int
- f 2;
> val it = 3 : int
f använder den gamla bindningen av y!
```

PK1&PM1 HT-05 moment 10

Sida 28

Uppdaterad 2005-09-12

Modifierbara celler

ML ger möjlighet att skapa datastrukturer som verkligen kan ändras. ref är en datatypskonstruktor som *nåstan* beter sig som

```
datatype 'a ref = ref of 'a;
```

ref 3 kallas en *referens* till värdet 3. (Kallas också cell, box, låda).

För att komma åt refererade värden använder man funktionen !

```
fun ! (ref x) = x;
```

!x kallas *dereferering* av x.

Referenser kan *ändras* med den infix funktionen :=:

```
r := v
TYPE: 'a ref*'a -> unit
SIDE-EFFECTS: Värdet i r ändras till v.
POST: ()
EXAMPLE: val x = ref 3; !x = 3
         x := 2;       Nu är !x = 2
```

PK1&PM1 HT-05 moment 10

Sida 29

Uppdaterad 2005-09-12

Ingenting är säkert längre...

Använder man referenser så kan man i allmänhet inte längre byta "lika mot lika".

```
- val y = ref 1;
> val y = ref 1 : int ref
```

y är nu bunden till en referens till ett heltal.

```
- val z = !y;
> val z = 1 : int
- fun f x = x+ !y;
> val f = fn : int -> int
- y := 2;
> val it = () : unit
- z = !y;
> val it = false : bool
- f 2;
> val it = 4 : int
```

Obs, blank mellan + !

y är nu *ändrad*

f använder det nya värdet!

PK1&PM1 HT-05 moment 10

Sida 30

Uppdaterad 2005-09-12

Ändringar kan ske "bakom ryggen"

Anrop av funktioner med sideeffter kan förändra data – beräkningsordningen blir kritisk.

```
(* bump x
  TYPE: int ref -> int
  SIDE-EFFECTS: Ökar innehåller i cellen x med 1.
  POST: Det nya värdet av x. *)
fun bump x = (x := !x + 1; !x);
```

```
- val x = ref 1;
> val x = ref 1 : int ref
- bump x;
> val it = 2 : int
- !x;
> val it = 2 : int
```

Obs att (bump x)+ !x och !x+(bump x) beräknar olika värden!

PK1&PM1 HT-05 moment 10

Sida 31

Uppdaterad 2005-09-12

Bindningsomgivningar och lagret

- Definitioner (val, fun) *bindar* identifierare till värden.
- Bindningar kan aldrig ändras.
- Bindningsomgivningar är en samling bindningar.
- Bindningsomgivningar kan skapas och försvinna.

Dessutom har ML ett *lager* (eng. *store*) som är en samling celler.

- ref skapar celler som *innehåller* ett värde
- Innehållet i celler kan ändras
- Lagret är en samling celler
- Det finns bara ett lager och det försvinner aldrig.

```
- val y = ref 1;
> val y = ref 1 : int ref
```

PK1&PM1 HT-05 moment 10

Sida 32

Uppdaterad 2005-09-12

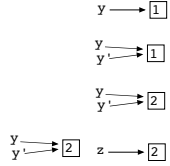
Likhet och olikhet hos celler

```
- val y = ref 1;
> val y = ref 1 : int ref
- val y' = y;
> val y' = ref 1 : int ref

- y := 2;
> val it = () : unit

- val z = ref 2;
> val z = ref 2 : int ref

- y = y';
> val it = true : bool
- y = z;
> val it = false : bool
```

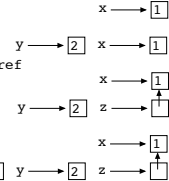


Referenser från celler till celler

```
- val x = ref 1;
> val x = ref 1 : int ref
- val y = ref 2;
> val y = ref 2 : int ref
- val z = ref x;
> val z = ref ref 1 : int ref ref

- val u = ref (!x);
> val u = ref 1 : int ref

z får typ int ref ref - en pekare till en annan cell!
```

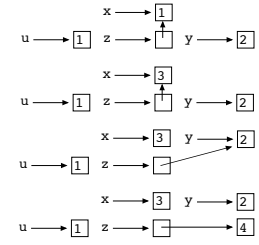


Referenser från celler till celler (forts.)

```
- !z := 3;
> val it = () : unit

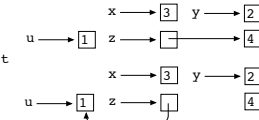
- z := y;
> val it = () : unit

- z := ref 4;
> val it = () : unit
```



Skräpsamling

```
- z := u;
> val it = () : unit
```



Cellen med värdet 4 är inte åtkomlig längre – den har blivit *skräp*. Skräp tas bort automatiskt med *skräpsamling* (garbage collection). Alla sorters data i ML kan bli skräp – inte bara celler utan t.ex. listor – men fenomenet märks kanske tydligast med celler. Tack vare skräpsamlingen fungerar det att ML-program arbetar med att ständigt konstruera nya objekt utan att någonsin ta bort gamla.

Imperativ programmeringsstil

- I imperativ programmeringsstil arbetar man med att
 - *ändra* innehållet i celler snarare än binda identifierare.
 - utföra beräkningar i steg (*satser*) som har sidoeffekter, snarare än som uttryck. (Enkla uttryck är ofta *delar* av satser.)
 - utföra upprepningar med *loopar* snarare än rekursion.

Imperativ programmeringsstil är det normala i programspråk som Pascal, C, Java...

Det som kallas "variabler" i dessa språk är normalt namngivna *ref*-celler.

Uttryck av *ref*-typ i ML motsvarar "lvalues" i C – alltså något som kan *tilldelas* ett värde. C och andra imperativa språk *dererfererar* (utför !-operationen på) variabler automatiskt när det behövs.

Ett imperativt program

```
(* absSqrt x
   TYPE: real->real
   POST: Kvadratroten av absolutbeloppet av x. *)
fun absSqrt(x) =
  let
    val x' = ref x
  in
    if !x' < 0.0 then
      x' := -(!x')
    else
      ();
    Math.sqrt(!x')
  end;
```

Loopar

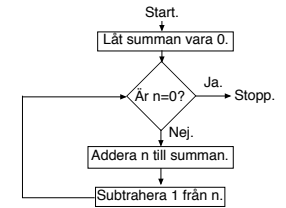
En loop *upprepar* ett uttryck (sats).
while e1 do e2

Uttrycket e2 beräknas upprepade gånger så länge uttrycket e1 beräknas till true. Värdet av while är alltid (). Utan sidoeffekter är detta meningslöst eftersom e1 och e2 alltid skulle ge samma värden!

Detta uttryck innehåller en loop som skriver ut "Hej igen!" 10 gånger:

```
let
  val i = ref 1
in
  while !i <= 10 do
    (print "Hej igen!\n";
     i := !i + 1)
end;
```

Analys av summeringsproblemet (repris)



Konstruktionen kallas för en *loop*. (Det spelar ingen roll om vi adderar n+...+0 eller 0+...+n)

Imperativ summeringsfunktion

```
(* sumUpTo
   TYPE: int -> int
   PRE: x >= 0
   POST: 1+2+...+x *)
fun sumUpTo x =
  let
    val n = ref x
    val sum = ref 0
  in
    while !n > 0 do
      (sum := !sum + !n;
       n := !n - 1);
    !sum
  end;
  Samma funktion i funktionell variant:
  fun sumUpTo n = if n = 0 then 0
                 else n + sumUpTo(n-1);
```

Imperativ genomgång av listor

```
(* countGreater(x,g)
   TYPE: int list->int
   POST: Antal element i x som är större än g *)
fun countGreater(x,g) =
  let
    val l = ref x
    val n = ref 0
  in
    while not (null(!l)) do
      (if hd(!l) > g then
         n := !n + 1
       else
         ());
      l := tl(!l);
    !n
  end;
```

countGreater imperativ och funktionell

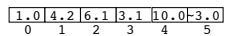
```
fun countGreater(x,g) = (* imperativ *)
  let
    val l = ref x
    val n = ref 0
  in
    while not (null(!l)) do
      (if hd(!l) > g then
         n := !n + 1
       else
         ());
      l := tl(!l);
    !n
  end;
  fun countGreater(l,g) = (* funktionell *)
    if null l then 0
    else (if hd l > g then 1 else 0)
        + countGreater(tl l,g)
```

Loopar och svansrekursion

```
(* Med loop *)
fun sumUpTo x =
  let
    val n = ref x
    val sum = ref 0
  in
    while !n > 0 do
      (sum := !sum + !n;
       n := !n - 1);
    !sum
  end;
  (* Med svansrekursion *)
  fun sumUpToAux(x,0) = sumUpToAux(x,0);
  fun sumUpToAux(n,sum) = if n > 0 then
    sumUpToAux(n-1,sum+n)
  else
    sum
```

Arrayer

- För att komma åt ett element i en lista eller ett träd krävs en tidsåtgång som ökar när datastrukturen 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 – *arrayen* (vektor, fält).

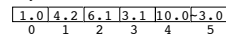


En flyttalsarray med 6 element och deras *index*.

Till skillnad från en lista som består av ett antal ::-konstruktörer är en array *ett* objekt – att komma åt alla element går alltså lika fort.

Arrayer i ML

Arrayer i ML definieras i biblioteket `Array` och liknar en abstrakt `datatype`. Det är dock en likhetstyp. (Men observera att liksom hos `ref` så kan två arrayer vara olika även om innehållet är lika!)



Arrayen ovan har i ML typen `real array`.

Liksom `ref`-celler kan innehållet i arrayer ändras.

Arrayer skiljer sig från `ref` i princip genom att `ref` är *en* cell, medan arrayer normalt har *flera* celler.

Skapa arrayer

```
Array.fromList x
  TYPE: 'a list -> 'a array
  POST: En array av samma längd som listan
        där cellerna i tur och ordning
        innehåller elementen i listan
  SIDE-EFFECTS: En ny array skapas.

Array.array(n,x)
  TYPE: int*'a -> 'a array
  PRE: n >= 0
  POST: En array av längd n
        där cellerna innehåller x
  SIDE-EFFECTS: En ny array skapas.
```

Använda arrayer

```
Array.length a
  TYPE: 'a array -> int
  POST: Antal celler i arrayen a

Array.sub(a,i)
  TYPE: 'a array*int -> 'a
  PRE: 0 <= i < length a
  POST: Innehållet hos cellen med index i
        inom arrayen a.

Array.update(a,i,x)
  TYPE: 'a array*int*'a -> unit
  PRE: 0 <= i < Array.length a
  POST: ()
  SIDE-EFFECTS: I arrayen a sätts innehållet
               i cellen med index i till x.
```

Högre ordningens funktioner för arrayer

```
Array.foldr f e a
TYPE: ('a*'b->'b)->'b->'a array->'b
POST: f(x1,f(x2,...f(xn,e)...)) om arrayen a
       innehåller elementen x1,x2,...,xn
```

Jämför med specifikationen av foldr för listor:

```
foldr f e l
TYPE: ('a*'b->'b)->'b->'a list->'b
POST: f(x1,f(x2,...f(xn,e)...)) om l=[x1,x2,...,xn]
```

Array.foldr kan användas för att definiera en funktion toList som är bra att ha för att se hur arrayer ser ut.

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

PK16PM1 HT-05 moment 10

Slid 49

Uppdaterad 2005-09-12

Exempel på arrayfunktioner

```
- val a = Array.fromList [10,20,30,40];
> val a = <array> : int array
- toList a;
> val it = [10, 20, 30, 40] : int list
- Array.length a;
> val it = 4 : int
- Array.sub(a,3);
> val it = 40 : int
- Array.update(a,2,42);
> val it = () : unit
- toList a;
> val it = [10, 20, 42, 40] : int list

- val b = Array.array(3,0);
> val b = <array> : int array
- toList b;
> val it = [0, 0, 0] : int list
```

PK16PM1 HT-05 moment 10

Slid 50

Uppdaterad 2005-09-12

Summera elementen i en flyttalsarray

Funktionellt:

```
(* sumArray a
   TYPE: real array -> real
   POST: Summan av elementen i a *)
fun sumArray a =
  let
    (* sumArrayAux(a,i)
       TYPE: real array*int -> real
       POST: Summan av elementen i a fr.o.m.
             elementet med index i *)
    fun sumArrayAux(a,i) =
      if i >= Array.length a then
        0.0
      else
        Array.sub(a,i) + sumArrayAux(a,i+1)
  in
    sumArrayAux(a,0)
  end;
```

PK16PM1 HT-05 moment 10

Slid 51

Uppdaterad 2005-09-12

Summera elementen i en flyttalsarray

Imperativt:

```
(* sumArray a
   TYPE: real array -> real
   POST: Summan av elementen i a *)
fun sumArray a =
  let
    val sum = ref 0.0
    val i = ref 0
  in
    while !i < Array.length a do
      (sum := !sum + Array.sub(a,!i);
       i := !i + 1);
    !sum
  end;
```

Bekvämt:

```
fun sumArray a = Array.foldr (op +) 0.0 a
```

PK16PM1 HT-05 moment 10

Slid 52

Uppdaterad 2005-09-12