



# Programkonstruktion

## Moment 10 Om sidoeffekter In/utmatning och imperativ programmering

## 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.

## Inmatning

Hittills 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
```

## 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.

## 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 ; ... ; e_{n-1} ; e_n
```

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.)

## Utmatning

```
print x
TYPE: 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.

## 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 ingen ersättning för `load`.)

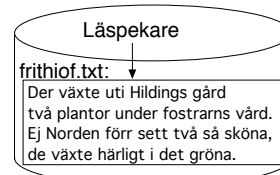
PK 2008/09 moment 10

Sida 7

Uppdaterad 2006-09-06

## 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 `outstream` 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 `outstream`.

PK 2008/09 moment 10

Sida 8

Uppdaterad 2006-09-06

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

PK 2008/09 moment 10

Sida 9

Uppdaterad 2006-09-06

## 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. filslut)
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. filslut).
SIDE-EFFECTS: Läspekaren för is flyttas fram.

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

PK 2008/09 moment 10

Sida 10

Uppdaterad 2006-09-06

## Ett urval funktioner för inmatning (2)

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

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

PK 2008/09 moment 10

Sida 11

Uppdaterad 2006-09-06

## 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 endOfStream is then
      (closeIn is; [])
    else
      inputLine is :: readToListAux is
in
  readToListAux (openIn f)
end;
```

PK 2008/09 moment 10

Sida 12

Uppdaterad 2006-09-06

## Terminering av rekursion vid inläsning

När man läser in data från en ström i ett rekursivt program så bryts i allmänhet rekursionen av filslut. 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 filslut.

## 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 endOfStream 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.

## 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);
```

## Ett urval funktioner för utmatning (2)

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

## 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.
```

## 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 Frithiof\n");
> val it = () : unit
- closeOut x;
> val it = () : unit
```

så kommer `testut.txt` att innehålla:

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

## 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 besvärligt. 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.

PK 2008/09 moment 10

Sida 19

Uppdaterad 2006-09-06

## 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.

PK 2008/09 moment 10

Sida 20

Uppdaterad 2006-09-06

## 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 "kedjeräkning" 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!	

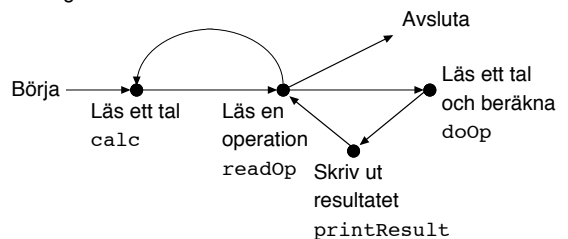
PK 2008/09 moment 10

Sida 21

Uppdaterad 2006-09-06

## Tillståndsdigram

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



Programmet skrivs som en *tillståndsmaskin* som motsvarar

PK 2008/09 moment 10

Sida 22

Uppdaterad 2006-09-06

diagrammet. Varje tillstånd motsvaras av en funktion i programmet.

## 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;
```

PK 2008/09 moment 10

Sida 23

Uppdaterad 2006-09-06

PK 2008/09 moment 10

Sida 24

Uppdaterad 2006-09-06

## 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))
```

PK 2008/09 moment 10

Sida 25

Uppdaterad 2006-09-06

## 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);
```

PK 2008/09 moment 10

Sida 26

Uppdaterad 2006-09-06

## 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;
```

PK 2008/09 moment 10

Sida 27

Uppdaterad 2006-09-06

## 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 ()));
```

PK 2008/09 moment 10

Sida 28

Uppdaterad 2006-09-06

## 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!
```

PK 2008/09 moment 10

Sida 29

Uppdaterad 2006-09-06

## Modifierbara celler

ML ger möjlighet att skapa datastrukturer som verkligen kan ändras.

ref är en datatypskonstruktor som nästan 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
```

PK 2008/09 moment 10

Sida 30

Uppdaterad 2006-09-06

## 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     Obs. blank mellan + !
- y := 2;
> val it = () : unit
- z = !y;
> val it = false : bool      y är nu ändrad
- f 2;
> val it = 4 : int           f använder det nya
                               värdet!
```

PK 2008/09 moment 10

Sida 31

Uppdaterad 2006-09-06

## Ändringar kan ske "bakom ryggen"

Anrop av funktioner med sidoeffekter 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!

PK 2008/09 moment 10

Sida 32

Uppdaterad 2006-09-06

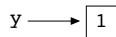
## Bindningsomgivningar och lagret

- Definitioner (val, fun) binder 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
```



PK 2008/09 moment 10

Sida 33

Uppdaterad 2006-09-06

## Likhet och olikhet hos celler

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

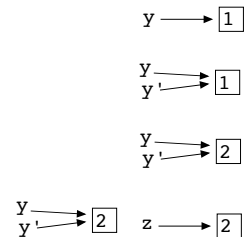
- val y' = y;
> val y' = ref 1 : int ref

- 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
```



PK 2008/09 moment 10

Sida 34

Uppdaterad 2006-09-06

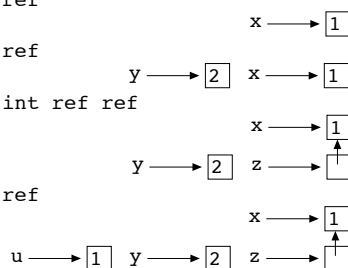
## Referenser från celler till celler

```
- 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!

PK 2008/09 moment 10

Sida 35

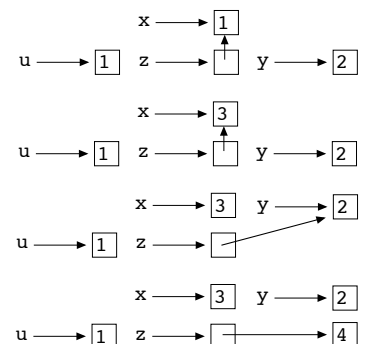
Uppdaterad 2006-09-06

## Referenser från celler till celler (forts.)

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

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

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



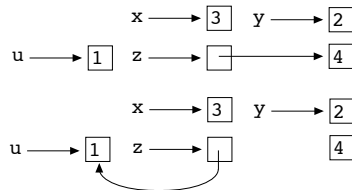
PK 2008/09 moment 10

Sida 36

Uppdaterad 2006-09-06

## 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.

PK 2008/09 moment 10

Sida 37

Uppdaterad 2006-09-06

## Imperativ programmeringsstil

I imperativ programmeringsstil arbetar man med att

- *ändra* innehållet i celler snarare än binda identifierare.
- utföra beräkningar i steg (*satser*) som har sideeffekter, 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 derefererar (utför !-operationen på) variabler automatiskt när det behövs.

PK 2008/09 moment 10

Sida 38

Uppdaterad 2006-09-06

## 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
      (); <----- ; skiljer uttryck som fungerar som satser.
      Math.sqrt(!x')
  end;
```

PK 2008/09 moment 10

Sida 39

Uppdaterad 2006-09-06

## Loopar

En loop *upprepar* ett uttryck (sats).

```
while e1 do e2
```

Uttrycket  $e_2$  beräknas upprepade gånger så länge uttrycket  $e_1$  beräknas till *true*. Värdet av *while* är alltid *()*.

Utän sideeffekter är detta meningslöst eftersom  $e_1$  och  $e_2$  alltid skulle ge samma värden!

Loopar är det normala sättet att utföra iteration i imperativa språk som Java och C (i stället för svansrekursion som i ML).

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

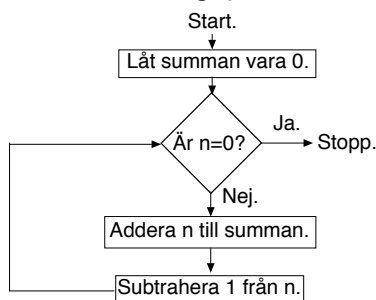
```
let val i = ref 1
in while !i <= 10 do
  (print "Hej igen!\n";
   i := !i + 1)
end;
```

PK 2008/09 moment 10

Sida 40

Uppdaterad 2006-09-06

## Analys av summeringsproblemet (repris)



Konstruktionen kallas för en *loop*.

(Det spelar ingen roll om vi adderar  $n+\dots+0$  eller  $0+\dots+n$ )

PK 2008/09 moment 10

Sida 41

Uppdaterad 2006-09-06

## 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);
```

PK 2008/09 moment 10

Sida 42

Uppdaterad 2006-09-06

## Imperativ genomgång av listor

```
(* countGreater(x,g)
   TYPE: int list*int -> 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;
```

PK 2008/09 moment 10

Sida 43

Uppdaterad 2006-09-06

## 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)
```

PK 2008/09 moment 10

Sida 44

Uppdaterad 2006-09-06

## 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 sumUpTo x = sumUpToAux(x,0);
fun sumUpToAux(n,sum) = if n > 0 then
  sumUpToAux(n-1,sum+n)
  else
  sum
```

PK 2008/09 moment 10

Sida 45

Uppdaterad 2006-09-06

## Imperativ summering av flyttalsvektor

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

PK 2008/09 moment 10

Sida 46

Uppdaterad 2006-09-06