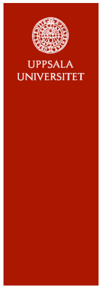


Programmeringsmetodik DV1
Programkonstruktion 1

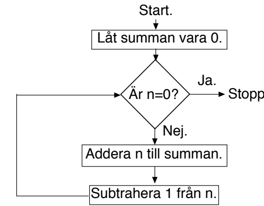
Moment 4
Om rekursion

Summera *godtyckligt* antal tal

```
(* sumUpTo n
   Type: int->int
   Pre: n >= 0, n <= 7
   Post: Summan av talen från 0 till n
   Ex.: sumUpTo 4 = 10
        sumUpTo 0 = 0
*)
fun sumUpTo 0 = 0
  | sumUpTo 1 = 0+1
  | sumUpTo 2 = 0+1+2
  | sumUpTo 3 = 0+1+2+3
  | sumUpTo 4 = 0+1+2+3+4
  | sumUpTo 5 = 0+1+2+3+4+5
  | sumUpTo 6 = 0+1+2+3+4+5+6
  | sumUpTo 7 = 0+1+2+3+4+5+6+7
```

`n>7`, då?? Detta är uppenbarligen ingen framkomlig väg...

Analys av summeringsproblemet



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

Iteration i ML

```
fun sumUpTo n = sumUpToAux(n,0)
fun sumUpToAux(0,sum) = sum                      <- Basfall
  | sumUpToAux(n,sum) = sumUpToAux(n-1,sum+n)    <- Rekursivt anrop
```

En hjälpfunktion `sumUpToAux` implementerar loopen. Tekniken kallas *iteration* (*svansrekursion*) och är ett specialfall av *rekursion*.
Argumentet `sum` kallas för *ackumulator*.
Obs att nya värden för `n` och `sum` beräknas *samtidigt*.

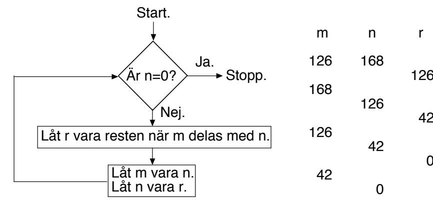
```
sumUpTo 2
—> sumUpToAux(2,0)
—> sumUpToAux(2-1,0+2)
—> sumUpToAux(1,2)
—> sumUpToAux(1-1,2+1)
—> sumUpToAux(0,3)
—> 3
```


Bindningar i iteration

```
fun sumUpToAux(0,sum) = sum
  | sumUpToAux(n,sum) = sumUpToAux(n-1,sum+n)
sumUpToAux(2,0)
—> sumUpToAux(n-1,sum+n)[n—>2,sum—>0]
—> sumUpToAux(2-1,sum+n)[n—>2,sum—>0]
—> sumUpToAux(1,sum+n)  [n—>2,sum—>0]
—> sumUpToAux(1,0+n)    [n—>2,sum—>0]
—> sumUpToAux(1,0+2)    [n—>2,sum—>0]
—> sumUpToAux(1,2)      [n—>2,sum—>0]
—> sumUpToAux(n-1,sum+n)[n—>1,sum—>2]
                        [n—>2,sum—>0] skuggas bort helt…
—> sumUpToAux(1-1,sum+n)[n—>1,sum—>2]
—> sumUpToAux(0,sum+n)  [n—>1,sum—>2]
—> sumUpToAux(0,2+n)    [n—>1,sum—>2]
—> sumUpToAux(0,2+1)    [n—>1,sum—>2]
—> sumUpToAux(0,3)      [n—>1,sum—>2]
—> 3
```


Euklides algoritm: exempel

Största gemensamma delaren (gcd) till 126 och 168



m	n	r
126	168	
		126
168		
	126	
		42
126		
	42	
		0
42		
	0	

gcd i ML

```
(* gcd(m,n)
   Type: int*int->int
   Pre: m,n>=0
   Post: Största gemensamma delaren till m och n
   Ex.: gcd(126,168) = 42 *)
fun gcd(m,0) = m
  | gcd(m,n) = gcd(n, m mod n)
```

(....ingen hjälpfunktion behövs. Varför?)

```
gcd(126,168)
—> gcd(168,126 mod 168)
—> gcd(168,126)
—> gcd(126,168 mod 126)
—> gcd(126,42)
—> gcd(42,126 mod 42)
—> gcd(42,0)
—> 42
```


Terminering

Hur vet man att rekursionen någonsin avslutas (*terminerar*)?
Vad *innebär* det att den inte gör det?

Inte självklart att rekursionen terminerar... felaktiga argument:

```
sumUpTo ~1
—> sumUpToAux(~1,0)
—> sumUpToAux(~1-1,0+ ~1)
—> sumUpToAux(~2,~1)
—> sumUpToAux(~2-1,~1+ ~2)
—> sumUpToAux(~3,~3)
—> .....
```

Förvillkoret måste utesluta sådana argument!

…eller en liten felskrivning:

```
fun sumUpToAux(0,sum) = sum
  | sumUpToAux(n,sum) = sumUpToAux(n+1,sum+n)
```


Rekursionsvarianter

Varje rekursivt anrop måste lösa ett *enklare* fall än det föregående. Ett tillräckligt enkelt fall måste kunna lösas *utan* rekursion.

Varje rekursivt anrop av `sumUpToAux(n,sum)` har ett *mindre* värde av `n` än det föregående. Fallet när `n`=0 (*basfallet*) hanteras utan rekursion. Samtidigt kan inte `n` bli mindre än 0.

En *rekursionsvariant* är ett heltalsuttryck sådant att:
- Det blir *mindre* för varje rekursivt anrop.
- Det *kan inte* bli mindre än något bestämt värde (normalt 0).
- För *detta* bestämda värde (och ev. andra) löses problemet utan rekursion.

Finns en rekursionsvariant är terminering garanterad.
Rekursionsvarianten är en del av programdokumentationen!

Konkret tolkning av varianten

Eftersom varianten minskar med 1 (minst!) för varje rekursivt anrop och inte kan bli mindre än 0 (typiskt) så ger varianten en *gräns* för hur många rekursiva anrop som behövs.

Är varianten 10 krävs *maximalt* 10 rekursiva anrop för att beräkna funktionen. Detta ger även ett tips om hur man kan hitta varianten.

Det finns funktioner där man inte (eller inte med rimlig ansträngning) kan ge en gräns för antalet rekursiva anrop men som ändå säkert terminerar. Då får man använda andra tekniker, t.ex. *välgrundade ordningar*.

Vi kommer endast i undantagsfall att stöta på sådana funktioner i denna kurs.

Rekursion och matematisk induktion

Det finns ett nära förhållande mellan rekursion och induktion.
Exempel: *Bevisa* att en rekursionsvariant garanterar terminering!

Låt T(n) betyda att ett anrop av funktionen med argument som har variantvärdet n terminerar.

Fullständig induktion över n (naturligt tal, ok ty varianten är inte < 0)

- T(0) gäller direkt (*basfallet*).
- Vi vill visa att T(n) gäller.
  Vi vet (*induktionsantagande*) att T(n’) för alla n’<n
  Alla rekursiva anrop som behövs för att beräkna funktionen har en variant n’<n, alltså terminerar de eftersom vi vet att T(n’).
  Alltså kommer det givna anropet att terminera.

gcd terminerar

```
... Pre: m,n>=0 ...
fun gcd(m,0) = m
  | gcd(m,n) = gcd(n, m mod n)
```

`n` är en rekursionsvariant för `gcd`.

- `m mod n` är alltid (definitionsmässigt) mindre än `n`.
- `n` kan inte vara mindre än 0 (eftersom förvillkoret anger att n *från början* inte är mindre än 0 och `m mod n` inte *kan bli* mindre än 0).
- fallet `n`=0 hanteras utan rekursion.

Funktionsspecifikationen för `sumUpToAux`

```
sumUpToAux(n,sum)
Type: int*int->int
```

För att `n` skall vara en rekursionsvariant krävs att `n` inte är mindre än 0 – inte ens från början:

```
Pre: n>=0
```

Eftervillkoret är *inte* att `sumUpToAux` ”summerar av talen 0 till `n`”. Så är den avsedd att *användas* – men det är inte vad den *faktiskt* gör.

```
Post: Summan av sum och talen från 0 till n.
```

Exempel behövs också...

```
Ex.: sumUpToAux(3,0) = 6
     sumUpToAux(4,5) = 15
     sumUpToAux(0,2) = 2
```

Rekursionsvarianten anges som del av programdokumentationen.

```
Variant: n
```

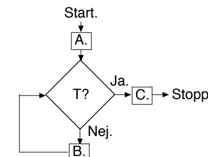

Varianter på defensiv programmering

```
fun sumUpTo n =
     if n < 0 then
       raise Domain
     else
       sumUpToAux(n,0)

fun sumUpToAux(0,sum) = sum
  | sumUpToAux(n,sum) =
      if n < 0 then
        raise Domain
      else
        sumUpToAux(n-1,sum+n)

fun sumUpToAux(n,sum) =
      if n <= 0 then
        sum
      else
        sumUpToAux(n-1,sum+n)
```

Påverkas förvillkoret i detta fall?

Kodningsmönster för iteration

Flödesschema för funktionen `foo`:



Kod för funktionen `foo`:

```
fun foo x = fooAux A

fun fooAux x =
      if T then
        C
      else
        fooAux B
```


Exempel: räkna uppåt

```
(* sumUpToAux(n,i,sum)
   Type: int*int*int->int
   Pre: n>=i-1
   Post: Summan av sum och talen från i till n.
   Ex.: sumUpToAux(4,0,0) = 10
        sumUpToAux(4,2,1) = 10 *)
(* Variant: n-i+1 *)
fun sumUpToAux(n,i,sum)=if i>n then
                          sum
                        else
                          sumUpToAux(n,i+1,sum+i);
(* sumUpTo n
   Type: int->int
   Pre: n >= 0
   Post: Summan av talen från 0 till n
   Ex.: sumUpTo 4 = 10
        sumUpTo 0 = 0 *)
fun sumUpTo n = sumUpToAux(n,0,0);
```

Exempel: räkna tecken 1

```
(* countCharAux(s,c,count)
   Type: string*char*int->int
   Pre: (ingen)
   Post: Summan av count och antalet förekomster av c i s.
   Ex.: countChar("Hej, du glade",#"d",0)=2
        countChar("Hej, du glade",#"q",5)=5 *)
(* Variant: size s  *)
fun countCharAux("",c,count) = count
  | countCharAux(s,c,count) =
      countCharAux(String.substring(s,1,size s -1), c,
                   if String.sub(s,0)=c then
                     count+1
                   else
                     count)
(* countChar(s,c)
   Type: string*char->int
   Pre: (ingen)
   Post: Antalet förekomster av c i s
   Ex.: countChar("Hej, du glade",#"d")=2
        countChar("Hej, du glade",#"q")=0 *)
fun countChar(s,c) = countCharAux(s,c,0);
```


En annan ansats till teckenräkning

Vid rekursion över strängar är rekursionsargumentet en *delsträng*.

Delsträngen behöver inte konstrueras uttryckligen – istället kan den vara underförstådd (*implicit*) genom att använda en *position i strängen* som extra argument och göra rekursion över *positionen*.

Strängargumentet `s` ersätts av *två* argument `s` och `pos`.
`s` är den *ursprungliga* strängen, `pos` är positionen.

När man använder `s` får man skriva
`String.sub(s,pos)` istället för `String.sub(s,0)`
Det rekursiva anropet har
`s` och `pos+1` istället för `String.substring(s,1,size s-1)`

Denna teknik är viktig om det är komplicerat att konstruera uttrycket i det rekursiva anropet.

Exempel: räkna tecken på annat sätt

```
(* countCharAux2(s,c,pos,count)
   Type: string*char*int*int->int
   Pre: 0<=pos<=size s
   Post: Summan av count och antalet förekomster av c i s
         från och med position pos.
   Ex.: countCharAux2("Hej, du glade",#"d",7,0)=1
        countCharAux2("Hej, du glade",#"d",3,5)=7 *)
(* Variant: size s - pos *)
fun countCharAux2(s,c,pos,count) =
      if pos >= size s then
        count
      else
        countCharAux2(s,c,pos+1,
            if String.sub(s,pos)=c then count+1 else count);
(* countChar2(s,c)
   Type: string*char->int
   Pre: (ingen)
   Post: Antalet förekomster av c i s
   Ex.: countChar2("Hej, du glade",#"d")=2
        countChar2("Hej, du glade",#"q")=0 *)
fun countChar2(s,c) = countCharAux2(s,c,0,0)
```


Rekursion

Svansrekursion innebär att värdet av det rekursiva anropet *direkt* blir värdet av funktionen. Genom att ta bort denna begränsning får man oftast enklare program:

```
fun sumUpTo 0 = 0
  | sumUpTo n = sumUpTo(n-1) + n

sumUpTo 2
—> sumUpTo(2-1)+2
—> sumUpTo(1)+2
—> (sumUpTo(1-1)+1)+2
—> (sumUpTo(0)+1)+2
—> (0+1)+2
—> 1+2
—> 3
```

(PS: Kalla det inte ”generell rekursion” – det är något annat...)

Hur funkar det?

```
fun sumUpTo 0 = 0
  | sumUpTo n =
      n + sumUpTo(n-1)
```

Rekursion är en variant av principen att dela upp problemet i enklare delar och sedan sätta ihop resultatet av delarna.
En rekursiv funktion anropar sig själv, men problemet är ändå enklare eftersom argumenten (egentligen rekursionsvarianten) är mindre. När det rekursiva anropet är klart sätter man ihop resultatet (additionen i exemplet).
Varje rekursivt anrop är som en ”rysk docka”.

Bindningar i rekursion

```
fun sumUpTo 0 = 0
  | sumUpTo n = sumUpTo(n-1) + n

sumUpTo 2
—> sumUpTo(n-1)+n                    [n-->2]
—> sumUpTo(2-1)+n                    [n-->2]
—> sumUpTo 1 +n                      [n-->2]
—> (sumUpTo(n-1)+n [n-->1])+n        [n-->2]
—> (sumUpTo(1-1)+n [n-->1])+n        [n-->2]
—> (sumUpTo 0 +n   [n-->1])+n        [n-->2]
—> (0+n [n-->1])+n                   [n-->2]
—> (0+1 [n-->1])+n                   [n-->2]
—> (1 [n-->1])+n                     [n-->2]
—> 1+n                               [n-->2]
—> 1+2                               [n-->2]
—> 3                                 [n-->2]
—> 3
```


Minnesbehov

Iteration behöver *konstant minne.*
Rekursion i allmänhet behöver minne i proportion till antalet rekursiva anrop eftersom en del av beräkningen ”väntar”.
Detta påverkar också beteendet vid icke-terminering:

```
sumUpTo ~1
—> sumUpTo(~1-1)+ ~1
—> sumUpTo ~2 + ~1
—> (sumUpTo(~2-1)+ ~2)+ ~1
—> (sumUpTo ~3 + ~2)+ ~1
—> (((sumUpTo(~3-1)+ ~3)+ ~2)+ ~1
—> (((sumUpTo ~4 + ~3)+ ~2)+ ~1
—> .....
```

Uttrycket blir hela tiden större... Minnet tar efterhand slut.
Detta *kan* också drabba ett korrekt terminerande program om de rekursiva anropen är många och kräver mycket minnesutrymme.

Beräkningsordning i rekursion

```
fun fooAux("",ack) = ack
  | fooAux(s,ack)  =
       fooAux(String.substring(s,1,size s -1),
              String.substring(s,0,1) ^ ack)
fun foo s = fooAux(s,"")

fun foo' "" = ""
  | foo' s  = String.substring(s,0,1) ^
              foo'(String.substring(s,1,size s -1)

foo("abc") = "cba"
foo'("abc") = "abc"
```

Användning av ackumulator ger annan ordning i resultatet!

Exempel: räkna uppåt

```
(* sumRange(i,n)
   Type: int*int->int
   Pre: n >= i-1
   Post: Summan av talen från i till n.
   Ex.: sumRange(0,4) = 10
        sumRange(2,4) = 9
        sumRange(5,4) = 0 *)
(* Variant: n-i+1 *)
fun sumRange(i,n) = if i>n then
                      0
                    else
                      sumRange(i+1,n)+i;
(* sumUpTo n
   Type: int->int
   Pre: n >= 0
   Post: Summan av talen från 0 till n
   Ex.: sumUpTo 4 = 10
        sumUpTo 0 = 0 *)
fun sumUpTo n = sumRange(0,n);
```


Undvik onödiga begränsningar

Varför förvillkor `n>=i-1`? Varför inte `n>=i` eller t.o.m. `n>i`?
Varför någon alls?

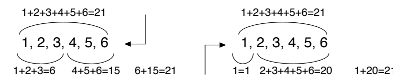
`sumRange(i,n)` summerar tal från `i` till `n`.
Antalet tal är `n-i+1`. Vad händer om antalet är 1, 0 eller –1?
Kan man tala om ”summan” av ett tal, noll tal, –1 tal?!?

Varför är basfallet i `sumRange` skrivet som det är?
`if i>n then 0…` varför inte `if i>=n then i…`
eller t.o.m. `if i>=n+1 then i+n…`

Man bör försöka hitta en naturlig tolkning av ”summa” som är så generell som möjligt utan att krångla till programmet. Då får man ett program som är flexibelt och kan användas i många situationer.

Tolkning av konstiga ”summor”

Summerar man två delar av en talserie och adderar delsummorna får man summan av hela talserien.



Låt nu en delserie innehålla *ett* tal. Regeln gäller fortfarande om man antar att summan av *ett* tal är talet självt.

Låt en delserie innehålla vara tom – dvs innehålla noll tal. Regeln gäller fortfarande om man antar att summan av *noll* tal är 0.

En talserie bestäms av sitt första och sista tal. En serie med noll tal får ett ”sista” tal (`n`) som är ett mindre än det ”första” talet (`i`). Därför bör `sumRange` kunna hantera alla fall där `n>=i-1`.

Exempel: räkna tecken 1

```
(* countChar(s,c)
   Type: string*char->int
   Pre: (ingen)
   Post: Antalet förekomster av c i s
   Ex.: countChar("Hej, du glade",#"d")=2
        countChar("Hej, du glade",#"q")=0
*)
(* Variant: size s *)
fun countChar("",_) = 0
  | countChar(s,c) =
      countChar(String.substring(s,1,size s -1),c)
      + (if String.sub(s,0)=c then
           1
         else
           0)
```


Exempel: räkna tecken 2

```
(* countCharAux2(s,c,pos)
   Type: string*char*int->int
   Pre: 0<=pos<=size s
   Post: Antalet förekomster av c i s fr.o.m. position pos.
   Ex.: countCharAux2("Hej, du glade",#"d",7)=1
        countCharAux2("Hej, du glade",#"d",3)=2 *)
(* Variant: size s - pos *)
fun countCharAux2(s,c,pos) =
      if pos >= size s then
        0
      else if String.sub(s,pos)=c then
        countCharAux2(s,c,pos+1)+1
      else
        countCharAux2(s,c,pos+1);
(* countChar2(s,c)
   Type: string*char->int
   Pre: (ingen)
   Post: Antalet förekomster av c i s
   Ex.: countChar2("Hej, du glade",#"d")=2
        countChar2("Hej, du glade",#"q")=0 *)
fun countChar2(s,c) = countCharAux2(s,c,0);
```


Exempel: räkna tecken baklänges

```
(* countCharAux3(s,c,pos)
   Type: string*char*int->int
   Pre: 0<=pos<=size s
   Post: Antalet förekomster av c i s före position pos.
   Ex.: countCharAux3("Hej, du glade",#"d",7)=1
        countCharAux3("Hej, du glade",#"d",3)=0 *)
(* Variant: pos *)
fun countCharAux3(s,c,0) = 0
  | countCharAux3(s,c,pos) =
      if String.sub(s,pos-1)=c then
        countCharAux3(s,c,pos-1)+1
      else
        countCharAux3(s,c,pos-1);
(* countChar3(s,c)
   Type: string*char->int
   Pre: (ingen)
   Post: Antalet förekomster av c i s
   Ex.: countChar3("Hej, du glade",#"d")=2
        countChar3("Hej, du glade",#"q")=0 *)
fun countChar3(s,c) = countCharAux3(s,c,size s);
```


Flera basfall

Vänd på tecknen i en sträng! Problemuppdelning:
• Byt plats på första och sista tecknet
• Byt plats på tecknen emellan
Vad händer om strängen har jämnt antal tecken? Udda?
Två basfall – för variantvärden 0 resp. 1.

```
(* Variant: size s *)
fun revString "" = ""
  | revString s =
      if size s = 1 then
        s
      else
        String.substring(s,size s -1,1) ^
         revString(String.substring(s,1,size s-2))
         ^ String.substring(s,0,1)
```


Testning av rekursion

Grundprinciperna gäller:
• *All* kod.
• *Gränsfall* (inklusive *triviala* fall)
• *Typiska* (icke-triviala) fall som täcker *hela specifikationen*

Gränsfall är t.ex. vid testning av `sumRange`:
• inga tal / ett tal / två tal (kanske)

...vid testning av `revString`:
• tom sträng / sträng med ett tecken /
sträng med två tecken / sträng med tre tecken (kanske)

...vid testning av `countChar`:
• tom sträng / sträng med ett tecken /
längre sträng med 0, 1 resp. 2 tecken som stämmer /
tecken som (inte) stämmer på plats 0 resp. 1 i strängen...