

Programkonstruktion

Moment 5 Mera om rekursion

Former av rekursion

- *enkel* rekursion – ett rekursivt anrop, argumenten (eg. varianten) minskar ett steg i taget.
- *fullständig* rekursion – argumenten (varianten) kan minska olika många steg i det rekursiva anropet.
- *multipel* rekursion – flera rekursiva anrop.
- *ömsesidig* rekursion – flera funktioner anropar varandra rekursivt.
- *kapslad* rekursion – rekursiva anrop i argumenten till andra rekursiva anrop.

Fakultetsfunktionen

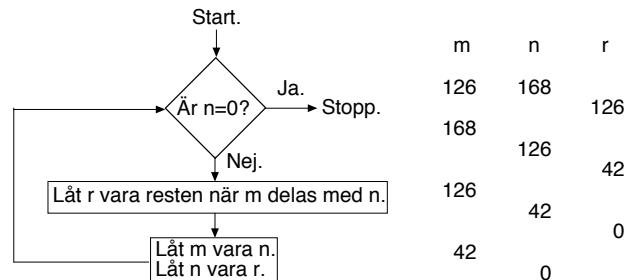
```
(* fact n
Type: int->int
Pre: n >= 0
Post: n-fakultet (n!)
Ex.: fact 4 = 24
      fact 6 = 720
      fact 12 = 479001600 *)
(* Variant: n *)
fun fact 0 = 1
   | fact n = n*fact(n-1)
```

fact växer fort – fact 13 ger overflow på institutionens maskiner.

Denna typ av rekursion när man har ett rekursivt anrop och varianten minskar ett steg i taget kallas *enkel rekursion*.

Största gemensamma delare (repris!)

Största gemensamma delaren (gcd) till 126 och 168

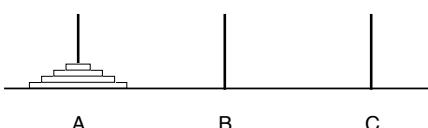


gcd i ML (repris!)

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

Varianten kan minska olika mycket i varje steg – *fullständig* rekursion.

Tornen i Hanoi



Problem: Flytta alla skivor från A till C.

Regler: En skiva i taget får flyttas

Endast den översta skivan får flyttas.

En skiva får bara ligga ovanpå större skivor.

Uppgift: Skriv ett program för att beräkna hur flyttningen skall gå till (givet ett visst antal brickor).

(Ett exempel på ett *planeringsproblem*)

Lösning av Hanoi-problemet

Vi skall flytta n brickor.

Antag att vi vet hur man skall flytta $n-1$ brickor.

- Flytta $n-1$ brickor från A till B (via C).
- Flytta den sista brickan från A till C.
- Flytta $n-1$ brickor från B till C (via A).

Klart!!

När man löser delproblemen är det inte säkert att "till" eller "via"-pinnarna är tomma, men i så fall ligger det större brickor på dem.

Denna lösning ger direkt ett naturligt program i ML.

Observera att lösningen kräver två rekursiva anrop (*multipel rekursion*).

PK 2007/08 moment 5

Sida 7

Uppdaterad 2005-09-21

Hanoi-problemet i ML

```
(* hanoi(n,from,via,to)
  Type: int*string*string*string->string
  Pre: n>=0. from, via och to är olika strängar.
  Post: En sträng som beskriver hur man skall
        flytta n brickor i Hanoi-spelet från
        pinne from till pinne to med hjälp av
        pinne via.
  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 2007/08 moment 5

Sida 8

Uppdaterad 2005-09-21

Mönstermatchning

Problem:

Man vill se om en sträng passar mot ett teckenmönster (*matchar*).

I detta exempel är ett mönster en följd av tecken där

- ? betyder "godtyckligt tecken"
- * betyder "en följd av noll eller flera godtyckliga tecken".

Exempel:

"K*a", "K?ll? Ank?" och "Kalle*" matchar "Kalle Anka"
"K*k", "K?alle Anka" och "???" matchar *inte*.

Skriv en funktion som kan avgöra om ett mönster matchar strängen
eller ej!

PK 2007/08 moment 5

Sida 9

Uppdaterad 2005-09-21

PK 2007/08 moment 5

Sida 10

Uppdaterad 2005-09-21

Problemuppdelning för matchning

Är mönstret tomt eller ej?

- Ja – då måste strängen också vara tomt.
- Nej – dela upp mönstret i första tecknen och resten
Välj alternativ beroende på första tecknen
a) "?" – matcha resten av strängen med resten av mönstret.
b) "*" – matcha stjärnan (hur?!?)
c) Annat tecken – kolla att strängen börjar med det tecknet och
matcha resten av strängen mot resten av mönstret.

Matcha stjärnan

Stjärnan matchar ett godtyckligt (inkl. 0) antal tecken. Ett mönster som börjar med en stjärna matchar alltså en sträng om *resten* av mönstret matchar strängen med *början* på godtycklig plats.

Man får prova i början av strängen och sedan prova ett tecken framåt i taget (*sökning*).

T.ex. sträng: "abcd", mönster "*cd".

Matchar "abcd" mönstret "cd"? Nej.

Matchar "bcd" mönstret "cd"? Nej.

Matchar "cd" mönstret "cd"? Ja!

(Här matchade alltså stjärnan de två tecknen "ab".)

PK 2007/08 moment 5

Sida 11

Uppdaterad 2005-09-21

PK 2007/08 moment 5

Sida 12

Uppdaterad 2005-09-21

Problemuppdelning för matchning av stjärna

Har vi provat hela strängen?

- Ja – Mönstret matchar inte
- Nej. Matchar strängen vid given position resten av mönstret?
a) Ja. Klart.
b) Nej. Prova med nästa position.

Nu kan vi skriva kod direkt från problemuppdelningarna!

Vi får en huvudfunktion och en hjälpfunktion.

Funktionen match

```
(* match(s,pattern)
Type: string*string->bool
Pre: (ingen)
Post: true om s matchar pattern, false annars.
Ex: match("abc","a?c") = true
    match("abc","*c") = true
    match("abc","?c") = false *)
(* Variant: size pattern *)
fun match(s,"") = s = ""
| match(s,pattern) =
  case String.sub(pattern,0) of
    "#*" =>
      matchStar(s,String.substring(pattern,1,
                                   size pattern-1),
                 0)
    | "?" => s <> "" andalso
      match(String.substring(s,1,size s-1),
            String.substring(pattern,1,size pattern-1))
    | c => s <> "" andalso
      String.sub(s,0) = c andalso
      match(String.substring(s,1,size s-1),
            String.substring(pattern,1,size pattern-1))
```

PK 2007/08 moment 5

Sida 13

Uppdaterad 2005-09-21

Funktionen matchStar

```
(* matchStar(s,pattern,pos)
Type: string*string*int->bool
Pre: 0<=pos<=size s
Post: sant om s - bortsett från pos (eller flera) tecken i början - matchar pattern, false annars.
Ex: matchStar("abcd","cd",0) = true
    matchStar("abcd","cd",3) = false *)
(* Variant: size s - pos *)
and matchStar(s,pattern,pos) =
  pos <= size s andalso
  (match(String.substring(s,pos,size s-pos),
         pattern) orelse
   matchStar(s,pattern,pos+1))
```

PK 2007/08 moment 5

Sida 14

Uppdaterad 2005-09-21

Testning av match....

Exemplet tidigare ger "typiska fall".

Dessutom, för kodtäckning:

```
match("", "") = true
match("a", "") = false
match("abc", "*bc") = true
match(" ", "?bc") = false
match("abc", "?bc") = true
match(" ", "abc") = false
match("xbc", "abc") = false
match("abc", "abc") = true

matchStar("abc", "c", 3) = false
match("bc", "c") = false
matchStar("abc", "c", 1) = true
matchStar("abc", "c", 2) = true
```

PK 2007/08 moment 5

Sida 15

Uppdaterad 2005-09-21

Testning av match (forts.)....

Dessutom, för gränsfall...

```
match("", "?") = false
match("a", "*") = true
match("a", "?a") = true
match("a", "a?") = false
match("a", "a") = true
match("a", "ba") = false
match("ab", "*") = true
match("ab", "?b") = false
match("ab", "a?") = true
match("ab", "?a") = false
match("ab", "aa") = false
match("ab", "ab") = true
```

PK 2007/08 moment 5

Sida 16

Uppdaterad 2005-09-21

Ömsesidig rekursion

Använd rekursion för att visa om ett naturligt tal är udda eller jämnt.

```
(* even n
Type: int->bool
Pre: n>=0
Post: true om n är jämnt, false annars.
Ex: even 2 = true
    even 3 = false *)
fun even 0 = true
| even n = odd (n-1);

(* odd n
Type: int->bool
Pre: n>=0
Post: true om n är udda, false annars.
Ex: odd 2 = false
    odd 3 = true *)
fun odd 0 = false
| odd n = even (n-1);
```

PK 2007/08 moment 5

Sida 17

Uppdaterad 2005-09-21

Ömsesidigt rekursion – problem

Programmet ovan fungerar inte direkt i ML – odd används (i even) innan den är definierad. Ömsesidigt rekursiva funktioner måste definieras *tillsammans* med hjälp av and-konstruktionen:

```
- fun even 0 = true
  | even n = odd (n-1)
  and odd 0 = false
  | odd n = even (n-1);
> val even = fn : int -> bool
val odd = fn : int -> bool
- even 5;
> val it = false : bool
- even 4;
> val it = true : bool
```

Ömsesidigt rekursiva funktioner måste ha *gemensam* variant. I detta fall är n en variant.

PK 2007/08 moment 5

Sida 18

Uppdaterad 2005-09-21

Kapslad rekursion – välgrundade ordningar

```
(* acker(n,m)
Type: int*int->int
Pre: n,m >= 0
Post: Ackermanns funktion av n och m
Ex: acker(2,2) = 7, acker(3,4) = 125
fun acker(0,m) = m+1
| acker(n,0) = acker(n-1, 1)
| acker(n,m) = acker(n-1, acker(n,m-1))
```

Ackermanns funktion terminerar, men det kan inte visas med vår variantteknik. Det går inte att förutse hur många anrop som behövs.

Titta på båda argumenten samtidigt! Under den *välgrundade ordningen* $(n_1, m_1) < (n_2, m_2)$ omm (om och endast om) $n_1 < n_2$ eller $n_1 = n_2$ och $m_1 < m_2$ så minskar argumenten i varje anrop – varje anrop är "enklare" så till slut måste man nå basfallet.

PK 2007/08 moment 5

Sida 19

Uppdaterad 2005-09-21

En gåta

Terminerar foo n för alla positiva heltal n?

```
(* foo n
Type: int->int
Pre: n>0 (?!!?!!?)
Post: 1
Ex: foo 4 = 1
     foo 5 = 1 *)
fun foo 1 = 1
| foo n = if n mod 2 = 0 then
            foo(n div 2)
        else
            foo(3*n+1)
foo 4? 4, 2, 1
foo 5? 5, 16, 8, 4, 2, 1
foo 13? 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
```

Här finns inte ens någon välgrundad ordning (som man känner till)!

PK 2007/08 moment 5

Sida 20

Uppdaterad 2005-09-21

Rekursion är inte alltid direkt tillämplig

Problem: Avgör om n är ett primtal!

```
(* prime n
Type: int->bool
Pre: n>0
Post: true om n är ett primtal, false annars
Ex: prime 4 = false
     prime 5 = true
*)
```

Problemuppdelning?? Kan man avgöra om n är ett primtal
...genom att veta om $n-1$ är ett primtal (enkel rekursion)?
...genom att veta att alla $n' < n$ är primtal (fullständig rekursion)?
n är tydligent *inte rätt* variabel att göra rekursion över, men vad är rätt i så fall?

PK 2007/08 moment 5

Sida 21

Uppdaterad 2005-09-21

Generalisering

Ibland kan man inte lösa ett problem direkt, utan man måste lösa ett *mera allmänt* problem först.

Problem: avgör om n inte är delbart med något tal från low till high.

```
nodivisors(n,low,high)
Type: int*int*int->bool
Pre: n>0, low>0, low<=high+1
Post: true om n inte är delbart med något tal
      från low till high, false annars.
Ex: nodivisors(4,2,3) = false
    nodivisors(5,2,4) = true
```

nodivisors kan skrivas med rekursion över t.ex. low.

n är ett primtal omm (om och endast om) n inte är delbart med något tal utom 1 och sig själv – alltså från 2 till n-1.

PK 2007/08 moment 5

Sida 22

Uppdaterad 2005-09-21

Programmet för primtalsbestämning

```
(* nodivisors(n,low,high)
Type: int*int*int->bool
Pre: n>0, low>0, low<=high+1
Post: true om n inte är delbart med något tal
      från low till high, false annars.
Ex: nodivisors(4,2,3) = false
    nodivisors(5,2,4) = true *)
(* Variant: high-low+1 *)
fun nodivisors(n,low,high) =
  low > high orelse
  (n mod low) <> 0 andalso
  nodivisors(n,low+1,high)

(* prime n *)
fun prime n = nodivisors(n,2,n-1)
```

PK 2007/08 moment 5

Sida 23

Uppdaterad 2005-09-21

En konstruktionsmetodik

Uppgift: Konstruera ett ML-program som beräknar en funktion f(a,...) givet en funktionsspecifikation (type, pre, post, ex)

- 1) Bestäm hur funktionen kan beräknas med hjälp av värdet för "enklare" indata. Välj argumentet att göra rekursion över.
Det kommer också att behövas fallanalys över detta.
- 2) Bestäm rekursionsvarianten och dokumentera den.
- 3) Ev. skriv defensiv kod för fall när rekursionsargumentet inte uppfyller förvillkoret.
- 4) Skriv kod för basfallet (fallen) *utan* att använda rekursion.
- 5) Skriv kod för det allmäna fallet. Använd resultatet från rekursiva anrop (med enklare indata) för att beräkna funktionsvärdet.
Kontrollera för de rekursiva anropen att varianten minskar och att förvillkoret är uppfyllt!

PK 2007/08 moment 5

Sida 24

Uppdaterad 2005-09-21

Introduktion till komplexitet

Antalet beräkningssteg för rekursiva program varierar med indata. I princip krävs flera steg när indata är "större", men *hur många* fler?

Ex: `sumUpTo n` adderar heltalet från 0 till n. Antalet steg är direkt proportionellt mot n – blir n dubbelt så stor blir antal beräkningssteg dubbelt så stort. (Tids)komplexitet: $O(n)$. *Linjär tid*.

Ex: `hanoi(n, from, via, to)` som löser Hanoi-spelet. För att beräkna `hanoi` för ett bestämt n, så måste `hanoi` anropas *två gånger* för $n-1$. En ökning av n med 1 ger alltså en *fördubbling* av antalet beräkningssteg. (Tids)komplexitet $O(2^n)$. *Exponentiell tid*.

Skriv inte program så att komplexiteten blir onödigt stor! Program med exponentiell tidskomplexitet är *oftast* praktiskt oanvändbara.

PK 2007/08 moment 5

Sida 25

Uppdaterad 2005-09-21

Minneskomplexitet

Även minnesåtgången varierar i allmänhet med indata.

Ex: iterativ `sumUpTo n` har en minnesåtgång som är konstant (oberoende av n). Minneskomplexitet $O(1)$. *Konstant minne*.

Ex: (icke svans-)rekursiv `sumUpTo n` har en minnesåtgång som är linjär (i n). Minneskomplexitet $O(n)$. *Linjärt minne*.

Minneskomplexiteten kan aldrig bli större än tidskomplexiteten? Varför?

Sida 26

Uppdaterad 2005-09-21

Exponentiering

Problem: Beräkna x^n , där x är ett flyttal och n ett naturligt tal.

```
(* expo(x,n)
Type: real*int->real
Pre: n>=0
Post: x upphöjd till n
Ex: expo(2.0,4) = 16.0
*)
(* Variant: n *)
fun expo(x,0) = 1.0
| expo(x,n) = x*expo(x,n-1)
```

Komplexiteten är linjär – $O(n)$.

PK 2007/08 moment 5

Sida 27

Uppdaterad 2005-09-21

Val av algoritm påverkar komplexiteten

$$\begin{aligned}x^n &= x^{\frac{n}{2}} \cdot x^{\frac{n}{2}} && \text{om } n \text{ är jämn} \\x^n &= x^{\frac{n}{2}} \cdot x^{\frac{n}{2}} \cdot x && \text{om } n \text{ är udda.}\end{aligned}$$

```
fun square x:real = x*x;
fun fastexpo(x,0) = 1.0
| fastexpo(x,n) =
  square(fastexpo(x, n div 2)) *
  (if n mod 2 = 0 then 1.0 else x)
```

`fastexpo` har samma funktionsspecifikation (bortsett från funktionsnamnet) och rekursionsvariant som `expo` har.

Vid varje rekursivt anrop blir n ungefär hälften så stor.
Om antalet anrop blir i , så var från början $n \approx 2^i$, alltså $i \approx \log_2 n$
Komplexiteten är *logaritmisk* – $O(\log n)$.

PK 2007/08 moment 5

Sida 28

Uppdaterad 2005-09-21

Lokala deklarationer

Man kan låta deklarationer gälla endast inuti ett uttryck.

```
let
  deklarationer...
in
  uttryck
end
```

är självst ett uttryck där räckvidden för de angivna deklarationerna endast går till slutet av `let`-uttrycket.

```
let
  val x = 3
  val y = 2
in
  x+y
end
```

...beräknas till 5. Definitionerna av x och y är *lokala* i `let`-uttrycket.

PK 2007/08 moment 5

Sida 29

Uppdaterad 2005-09-21

En användning av lokala deklarationer

```
fun fastexpo(x,0) = 1.0
| fastexpo(x,n) =
  square(fastexpo(x, n div 2)) *
  (if n mod 2 = 0 then 1.0 else x)
```

Hjälpfunktionen `square` kan ersättas av en lokal `val`-deklaration som sparar värdet av det rekursiva anropet.

```
fun fastexpo(x,0) = 1.0
| fastexpo(x,n) =
  let
    val xn2 = fastexpo(x, n div 2)
  in
    xn2 * xn2 *
    (if n mod 2 = 0 then 1.0 else x)
  end
```

• `fastexpo(x, n div 2)` räknas bara ut en gång fast dess värde används två gånger.

PK 2007/08 moment 5

Sida 30

Uppdaterad 2005-09-21

Minns ni countChar2

```
(* 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);
```

PK 2007/08 moment 5

Sida 31

Uppdaterad 2005-09-21

Uppdaterad 2005-09-21

En annan användning av lokala deklarationer

```
fun countChar2(s,c) =
  let
    val ss = size s;
    (* countCharAux2(s,c,pos) ..... *)
    fun countCharAux2(pos) =
      if pos >= ss then
        0
      else if String.sub(s,pos)=c then
        countCharAux2(pos+1)+1
      else
        countCharAux2(pos+1)
  in
    countCharAux2(0)
  end
• size s räknas bara ut en gång.
• countCharAux2 kan inte användas utanför countChar2
• s, c och ss behöver inte ges som argument till countCharAux2
(de är fria i countCharAux2).
```

PK 2007/08 moment 5

Sida 32

Uppdaterad 2005-09-21

Matchning i deklarationer

I en val-deklaration är identifierarnamnet *egentligen* ett mönster!

```
- val (x,y) = (1,2);
> val x = 1 : int
  val y = 2 : int
```

I detta fall *måste* uttrycket matcha mönstret, annars exekveringsfel!

I en lokal deklaration kan matchning användas för att ta fram delarna av ett sammansatt funktionsvärdet:

```
fun qaddSimp(x,y) =
  let
    val (p,q) = qadd(x,y);
    val gcdpq = gcd(p,q)
  in
    (p div gcdpq,q div gcdpq)
  end
```

(qadd var en exempelfunktion som adderade rationella tal.)

PK 2007/08 moment 5

Sida 33

Uppdaterad 2005-09-21

Uppdaterad 2005-09-21

Funktionsdefinitioner är syntaktiskt socker!

Funktionsdefinitioner är funktionsabstraktion+definitionsabstraktion. Detta kan man göra tydligt genom att göra funktionsabstraktionen med fn och definitionsabstraktionen med val!

```
val rec addUpTo = fn 0 => 0
  | n => addUpTo(n-1)+n;
```

Detta är precis detsamma som:

```
fun addUpTo 0 = 0
  | addUpTo n = addUpTo(n-1)+n;
```

nyckelordet rec tillåter att man använder det bundna namnet (addUpTo) i sin egen definition – normalt är det inte tillåtet eftersom namn måste bindas innan de används.

(Även nyckelordet rec är syntaktiskt socker, men att visa hur man kan ersätta det med annan ML-kod är mycket komplicerat.)

PK 2007/08 moment 5

Sida 34

Uppdaterad 2005-09-21

Lokala deklarationer är syntaktiskt socker!

```
let
  val v1 = uttryck1
  val v2 = uttryck2
in
  uttryck
end
```

Är *exakt samma sak* som

```
(fn v1 => (fn v2 => uttryck) uttryck2) uttryck1
```

PK 2007/08 moment 5

Sida 35

Uppdaterad 2005-09-21