Föreläsning 1. 2D1370 Funktionell programmering v13 tisdag

Omkursen. Pardigmbegreppet. Funktionella paradigmen. Haskell. Kring Hudak kapitel 1.

Paradigmer.

Vad menas med paradigm?
När använder man olika paradigm i datalogi?

Programmeringspråk och kodning vs Design utfomning och paradigmer vs Algoritmer
Översikt datalogiska paradigm nästa sida.

Varför kunna olika paradigm?

Ej helt oberoende.
Perspektiv.
Kanske paradigmskifte? Ex OO, kalkylblad, DNA.

Om hårt typade språk:

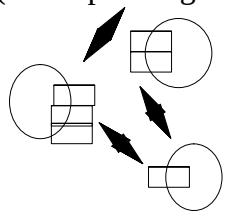
I princip oberoende av paradigm.
Hårt typade språk : Man "tvingas" tänka sig in i olika typer, kompilatorn hittar många fel.
Otypade språk: Full frihet

Om pekare (referenser):

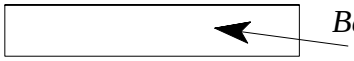
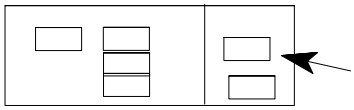
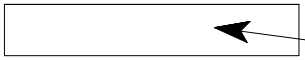
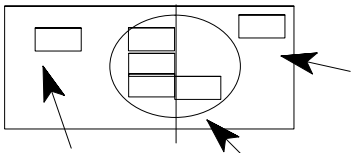
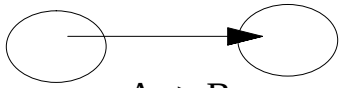
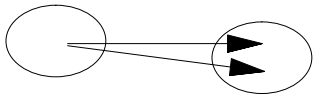
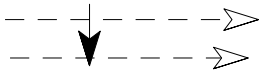
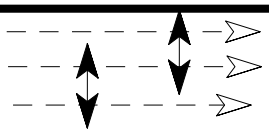
Oberoende av paradigm?
Osynligt?
Halvautomatiskt (Java)?
Explicit (Pascal, C)?

Vad är programmering? Tre aspekter (inspirerat av Sten Henriksson, Lund):

Programmering är en sorts matematik och logik.
Acceptabla språk börjar väl med bättre funktionella språk?
Körbara specifikation? Körbar Design? Bevis att programmen är korrekta.
Programmering är algoritmer.
Vad är beräkningsbart? Går allt att göra? O(n). Turing maskiner mm.
Programspråk (t o m paradigmer?) sekundärt? Numerisk analys.
Programmering är klassik ingenjörskonst.
Nedbrytning av moduler. Gränsnitt,
Design och implementation.
Implementation, programspråk (t o m paradigmer?) sekundärt?



Programmering är bara en del av ett system.
Design av systemet det viktiga. Design och implementation av programmet sekundärt. Att vara systemerare är "finare" (och mer välavlönat) än att vara programmerare. "ADB".
Användargränssnittet, samverkan med hela systemet det viktiga.

paradigm	ide' och semantik
Imperativ	Simulering av Von Neuman-dator CPU styrning ALU  *Beräkning (evaluering) av uttryck* Primärminne  *Kod (funktioner, procedurer, subrutiner..)* *Variabler,* *aggregat av variabler* *lokala / statiska /på heapen* Tilldelning := Flödeskontroll Sekvens ; Snurror Alternativ
OO	Imperativ med objekt och arv CPU styrning ALU  *Beräkning (evaluering) av uttryck* Primärminne  *Kod (funktioner, procedurer, subrutiner..)* *Variabler* *Objekt, aggregat* *med metoder* Tilldelning := Flödeskontroll Sekvens ; Snurror Alternativ
Funktionell	Matte (funktioner) som programmeringsspråk  A -> B *Beräkning (evaluering) av uttryck,* *ofta med funktioner* *Kod (def av funktioner och typer)*
Logisk	Logik (predikat, funktioner -> Bool) som programmeringsspråk  -> Bool *Beräkning (evaluering) av predikat,* *ofta med förslag hur logiska* *varaibler skall instasieras* *Kod (def av predikat)*
Händelsestyrd	 *Användarprocess som styr* *Programexekveringen (lyssanre i java)*
Jämnlöpande	 *Flera process eller trådar (Thread i Java)* *som styr varandra, synkronisering, moniorer mm*

Kort (personlig) karaktärisik :

Imperativ :

Så har man alltid gjort och alla andra gör så här! Dessutom kan jag redan en del!
"Sjävklart" att man instruerar datorn i vilken ordning saker skall göras.
Programmering som simulering. I verkliga livet går tiden, i datorn programmet.
Avspeglar datorns konstruktion.

OO (objektorienterad, objekt inriktad) :

Simuleringsynen ännu mer betonad.
Utveckling av imperativa pardigmen, men delvis inkompatibelt.
Nära hur "vanliga" männsikor ser på värden, värden består av objekt
av olika sorter (klasser) med både egenskaper (attribut) och
handlingsmönster (metoder).

Funktionell:

Ung = matte; Varför lära sig programmera när man redan kan matte / funktioner?
Matematik har mångahundraåriga traditioner.
Välgjorda språk, generella och ortogonala konsrtuktioner,
"Hög nivå", korta "program"
dvs svaret på frågan får man? / kan man? är i regel JA!
Men går det inte lite väl sakta? Används det?
Man börjar på detta sätt i Göteborg, Luleå, Umeå, Linköping, KTH före ca 1995,
SU (mat-datalogisk), Oxford, Västerås, Skottland, Yale m fl ställen

Logik:

Ung =logik; Varför lära sig programmera när man redan kan logik / relationer?
Logik har hundraåriga traditioner, "hög nivå"
Ren logik programmering / i praktiken kompromisser.
"Hög nivå", mycket korta "program"
Men fattar man vad som händer igentligen? Men går det inte lite väl sakta?
Prolog populärt i AI (artificiell intellegens).

Händelsestyrd:

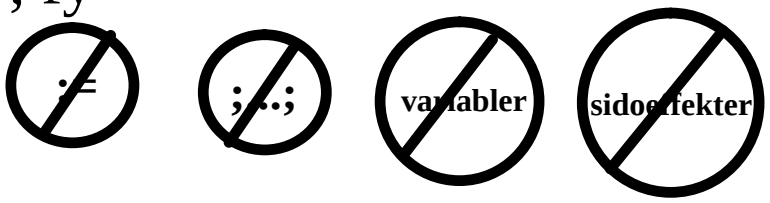
Många (de flesta)program skall ju göra nåt annat än bara "räkna ut nåt",
dvs samverka med verkligheten.
Egen paradigm ?

Jämlöpande :

"Naturligt" att saker och ting kan ske parallellt.
CPU kostar ju knappt nåt, varför inte använda hundratals, så man slipper vänta?
Egen paradigm ? Samma sak som händelsestyrd?
Synkronisering och utbyte av data, hur?
"Ju flera kockar ju sämre soppa"?

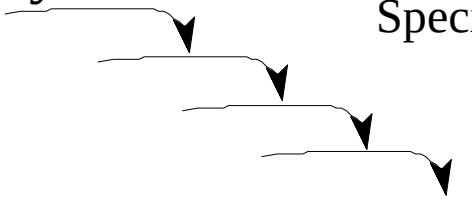
Funktionell programmering

Fy, fy



=> bugfree
förståeliga
korta

Enjoy Vattenfallsmodellen i funktionella språk



Specifikation ofta ekvationer

Design funktioner def med ekvationer

Kodning funktioner def med ekv

Högre ordningens funktioner Abstraktion

Lat evaluering, ger oändliga datastrukturer (Haskell bl a)

Historik, geografi

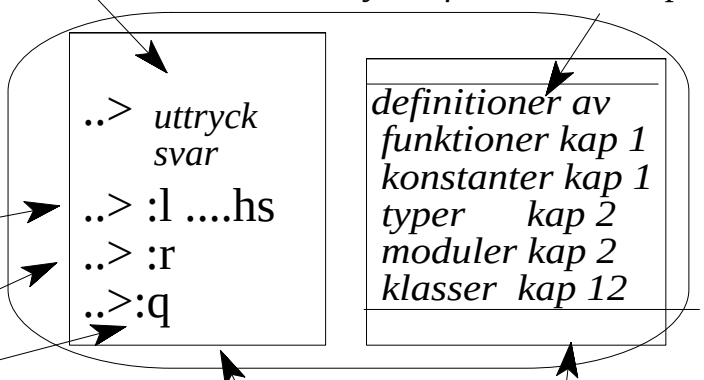
När man vill tänka imperativt (simulering) ?
Gräns mellan paradigmer Bra begrepp ?
Hudaks bok, "für alles" , nisher, denna kurs

Praktik : Räknedosa, Interpretern Hugs

Modulen Prelude laddas alltid automatiskt och innehåller många fördefinierade funkitoner, operatorer och typer

Räknedose- användning

Programmering av skript med nya definitoner som sparaas på fil



Inladdning av skript med ytterligare definitoner, load

reload av script

quit

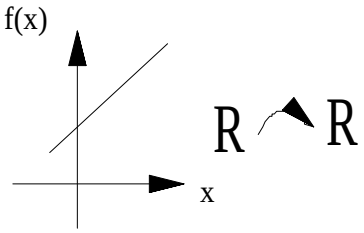
Att skriva funktioner . Elementära typer. Hudak kapitel 1.

Ett exempel :

Hur gör man om man vill definiera egna funktioner (Java : metoder som returnerar värden), och hur använder man dem, kanske gång på gång? Jo:

Java **Haskell** **Matte**

Definition av f:




```
class ...{                      f::Double->Double
 static double f(double x) {    f x = x + 3.0
 return x+3;
}                               (I emacs fönstret)
```

Vi definierar f(x) = x +3

Vi kallar x för *(formell) parameter*, i matte ofta variabel **som ju betyder något helt annat i programmering!**

Användning (anrop, applikation) av f

```
......main(.....) {

 utdata.println(
  "f(14.0) = " + f(14.0));      f 14
                                17.0
                                (I hugs fönstret)
```

Vad blir f(14) ?

Vi kallar `14.0` och `2.0` för *argument* (i vissa böcker *aktuell parameter).*

Ytterligare ett exempel. Herons formel :

Haskell

```
area1 ::(Double, Double, Double) -> Double
area1(a,b,c) = sqrt (p*(p-a)*(p-b)*(p-c))
  where  p = (a+b+c) / 2
```

```
area ::Double -> Double -> Double -> Double

area a b c  = sqrt( p*(p-a)*(p-b)*(p-c))
   where  p = (a+b+c) / 2
```

```
Main> area1(3.0, 4.0, 5.0)
6.0
Main> area 3.0 4.0 5.0
6.0
```

Matte

Ytan hos en triangel har sidor med sidlängder a, b och c beräknas med Herons formel:

area1 :: R^3 -> R

$$area1(a,b,c) = V\ p(p-a)(p-b)(p-c)$$

där p = (a+b+c) / 2

area :: R -> R -> R -> R

$$area\ a\ b\ c = V\ p(p-a)(p-b)(p-c)$$

där p = (a+b+c) / 2

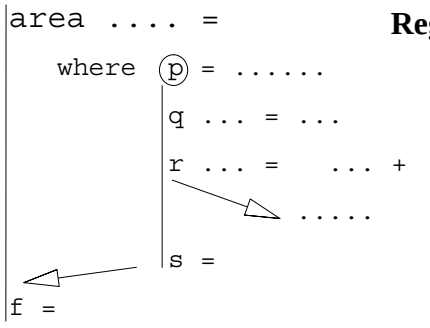
Beräkna några trianglars ytor, t ex den egyptiska triangeln med sidan 3, 4 och 5.

I stället för `where` kan man skriva så här:

```
area2(a,b,c) = let
                 p = (a+b+c) / 2
               in sqrt (p*(p-a)*(p-b)*(p-c))
```

Även hjälpfunktioner kan definieras lokalt med `let in` eller `where` och får då lokalt "skåp". Är man slarvig definerar man dom dock ofta "på högsta nivån", vilket gör dem mer lätt-testade. När allt funkar kan man sedan göra dem lokala.

Haskell kan skrivas layout-oberoende, men det absolut vanligaste är att använda layout på ett sådanant sättt att layouten får syntaktiskt betydelse (jfr Hudak sid 28).



Regel för `where let do of`

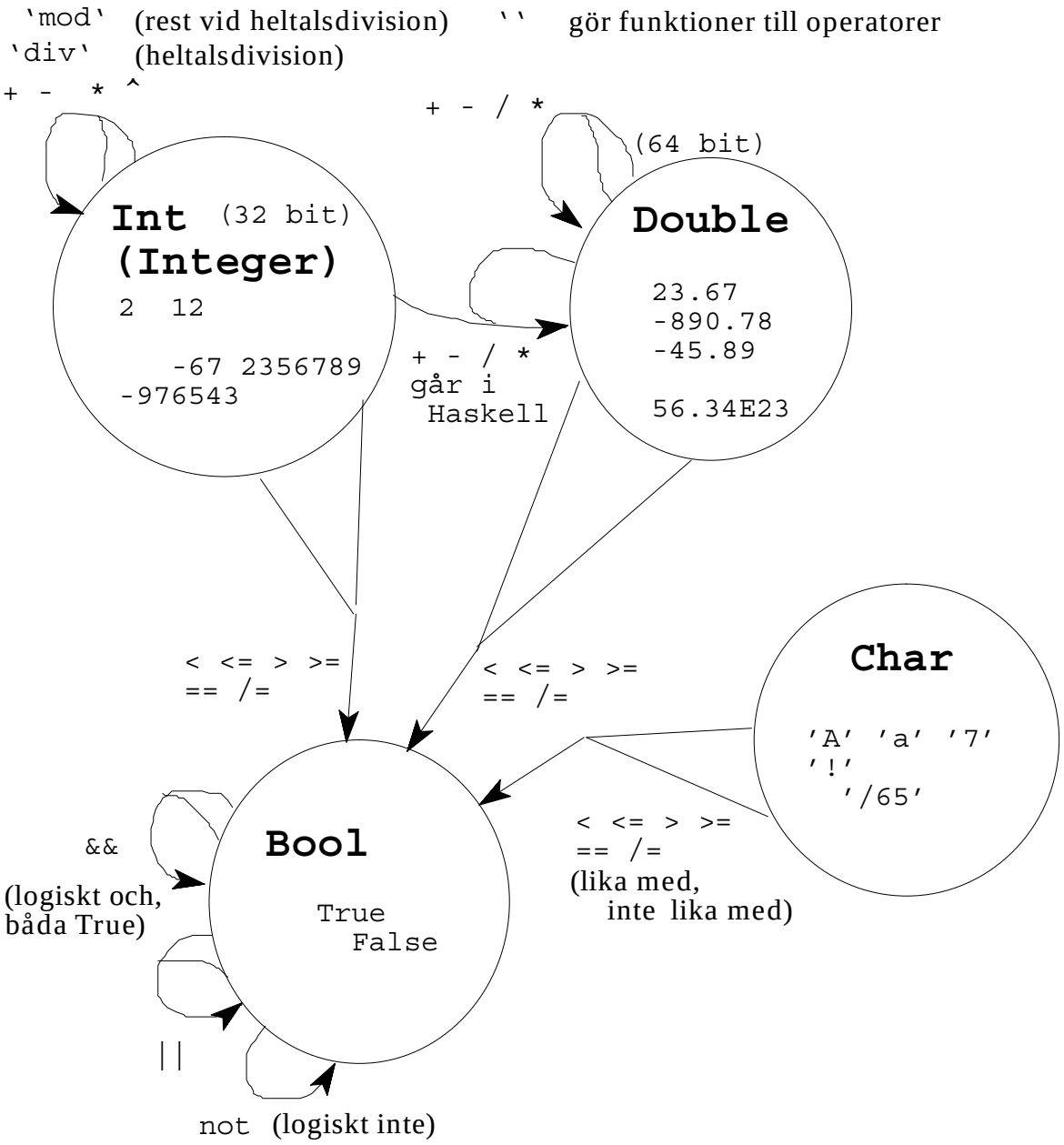
"syskondefinitioner" q r s rakt under "första barnet" p

till höger fortsättning på defav r i förgående rad

till vänster "slut på syskon" . f är syskon 2 på högsta nivån

Typer och operatorer.

Översikt av några fördefinierade (dvs de finns i Preluden) datatyperna och de viktigaste operatorerna med vars hjälp vi kan bilda uttryck :



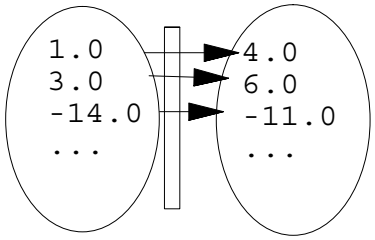
Det finns fler primitiva datatyper än `Int` för heltal, bl a `Integer` med "oändligt antal siffror". Det finns förutom `Double` ytterligare en primitiva datatyp `Float` för flyttal ("decimaltal"), det som skiljer är att bara 32 bit tilldelas sådan variabler i minnet i stället för 64 bitar och att antalet signifikanta siffror och största och minsta exponent är därför är mindre.

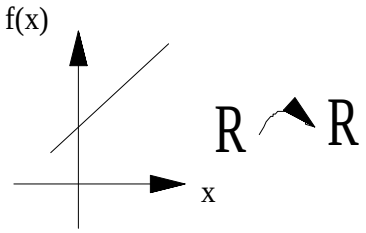
I Hudak kapitel 1 även listor och par och en funktion `listSum` med listargument.
Typen `String`: `type String = [Char]`

Att definiera en funktion: Studera definitionsområdet!

en ekvation med ett fall/
flera ekvationer/
en ekvation med flera fall (vakter/case/if)?

Haskell **Matte**

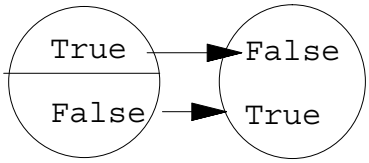


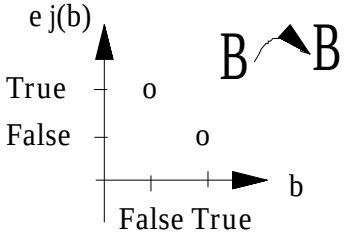



```
f::Double->Double
f  x = x + 3.0
```

En ekvation klarar alla värden

Vi definierar f(x) = x +3






```
ej::Bool-> Bool
ej True  = False
ej False = True
```

Två ekvation klarar båda värdena
Mönsterpassning
(pattern match)

False om b = True
ej(b) =
True om b = False

Sämre: En ekvation med flera fall:

med vakter (guards):

```
ej :: Bool -> Bool
ej b | b            = False
     | otherwise    = True
```

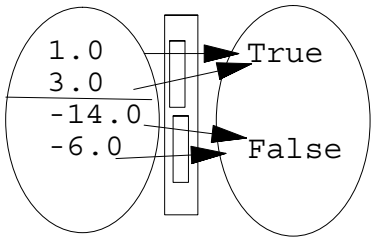
med if-uttryck

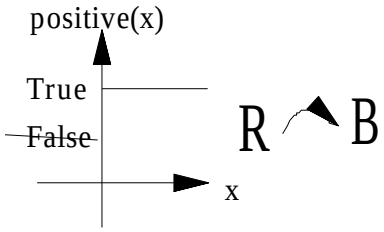
```
ej :: Bool -> Bool            -
ej b = if b
          then False
          else True
```

med case-uttryck

```
ej :: Bool -> Bool
ej b = case (b) of
          True -> False
           False -> True
```

Haskell **Matte**






```
positive::Double->Bool
positive x
      | x>= 0 = True
      |otherwise = False
```

En ekvation med vakter klarar alla värden

positive(x) = True om x >= 0
False i övriga fall

Mönster passning med flera ekvtioner fungerar ej.
Däremot möjligt med if eller case, (sämre)

Haskell har operatorn ^ för potenser (dvs den finns i"Prelude"), tex 3^4 (3^4) blir 81. Hur skriver vi den själv om den inte funnits? Vi kallar den power som funktion och ^* som operator:

```
module Power where   --  Exempel inför laboration efter kap 1
-- Körs som t ex : power 3 6 eller  3 ^* 6 (svar 729) eller 3 ^* (-6) (svar error)

(^*) = power  -- operator names Hudak p 9
              -- () makes funktion of operator Hudak p 9

{-                                    -- comments Hudak p 88

power :: Int -> Int -> Int            -- Om man kan take, repeat (i Prelude)
power m n = product (take n (repeat m))

power :: Int -> Int -> Int      -- rekursiva lösningar för n>0
power _ 0 = 1
power m n = m * power m (n-1)

power :: Int -> Int -> Int     -- if för n< 0 (varför ej 1 / power m (-n) ?)
power _ 0 = 1                  -- vakter bättre
power m n = if n<0 then error " second argument till power  >= 0"
            else m * power m (n-1)

power :: Int -> Int -> Int       -- vakter för n< 0 Bäst!
power _ 0 = 1                    -- Fokker 1.4.2
power m n
   | n<0       = error " second argument till power  >= 0"
   | otherwise = m * power m (n-1)

power :: Int -> Int -> Int       -- case för n< 0
power _ 0 = 1                    -- används sällan
power m n =
   case n<0 of
         True   -> error " second argument till power  >= 0"
         _      -> m * power m (n-1)
```

```
power :: Int -> Int -> Int       -- accumulerande lösningar
power _ 0 = 1   -- behövs ej
power m n
   | n<0       = error " second argument till power  >= 0"
   | otherwise = pw m n 0 1

pw :: Int->Int->Int->Int-> Int
pw m n i acc
     | i==n        = acc
     | otherwise   = pw m n (i+1) (acc*m)

power :: Int -> Int -> Int       -- accumulerande lösningar  med where
power _ 0 = 1
power m n
   | n<0       = error " second argument till power  >= 0"
   | otherwise = pw m n 0 1
   where                         -- layout  Fokker 1.4.5
     pw :: Int->Int->Int->Int-> Int
     pw m n i acc
        | i==n        = acc
        | otherwise   = pw m n (i+1) (acc*m)

power :: Int -> Int -> Int       -- accumulerande lösningar  med let .. in
power _ 0 = 1
power m n
   | n<0       = error " second argument till power  >= 0"
   | otherwise =
    let
      pw :: Int->Int->Int->Int-> Int
      pw m n i acc
        | i==n        = acc
        | otherwise   = pw m n (i+1) (acc*m)
    in pw m n 0 1
-}
--      Mer avancerat

{-
power :: Int -> Int -> Int       -- accumulerande lösningar  med until
power  m n                       -- Fokker 2.3.2
   | n<0       = error " second argument till power  >= 0"
   | otherwise = snd (until test f (0, 1))
    where
       test :: (Int, Int) -> Bool
       test (i, _)  = ( i == n )
       f :: (Int, Int) -> (Int, Int)
       f (i, acc) = (i+1, acc*m)

power :: Int -> Int -> Int       -- accumulerande lösningar  med until, lamda
power  m n                       -- Fokker 2.3.2, Fokker 2.3.4
   | n<0       = error " second argument till power  >= 0"
   | otherwise = snd (until (\(i, _) -> ( i == n ))
                            (\(i, acc) -> (i+1, acc*m) )
                            (0, 1))
-}
--
{-
power :: Int -> Int -> Int       --  Om man kan map ,[ .. ] (i Prelude)
power  m n  = product (map (\_ -> m) [1..n]) -- Fokker 2.3.1,  n>0 ger  1


power :: Int -> Int -> Int       --  Om man kan listomfattning
power  m n  = product [m  | x <- [1..n]] -- Fokker 3.2.7


power :: Int -> Int -> Int       --  Om man kan foldr (i Prelude)
power  m n  = foldr (*) 1  [m  | x <- [1..n]] -- Fokker 3.2.7
```

```
power :: Int -> Int -> Int       --  Om man kan foldl (i Prelude)
power  m n  = foldl (*) 1  [m  | x <- [1..n]] -- Fokker 3.2.7

-}

power :: Int -> Int -> Int       --  Prelude fiffiga def av (^)
power  _ 0 = 1
power  m n
        | n > 0     = f m (n-1) m
           where  f _ 0 y = y
                  f m n y = g m n
                    where g x n
                            | even n    = g (x*x) (n `quot` 2)
                            | otherwise = f x (n-1) (x*y)
power _ _  =  error " second argument till power  >= 0"
```

Modulen `Power` finns på filen `Power.hs` i katalogen

`/info/funpro02/LKykringHudak1`

Där finns också ytterligare program. På katalogen med namn

`/info/funpro02/LKykringHudak7`

finns förståss program som har med Hudak kapitel 7 osv.

Se också "att definiera en funktion i Haskell i "Kompletterande material".

Ett exempel på hur man skriver funktioner med listparameter, typ `[a] ->...`:

```
sumList :: [Int] -> Int
sumList []     = 0
sumList (i:is) = i + sumList is


sumList1 :: [Int] -> Int
sumList1 list = sumListAcc list 0


sumListAcc :: [Int] -> Int ->Int
sumListAcc []     acc = acc
sumListAcc (i:is) acc = sumListAcc is (acc+i)
```

Hur fungerar `power` ?

```
power :: Int -> Int -> Int      -- rekursiva lösning för n>0
power _ 0 = 1                 --pattern match recursion
power m n = m * power m (n-1)
```

Haskell-system förenklar helt enkelt uttryck (med kontroll av typerna coh använding av definitionerna). Jfr imperativa språk där man för att förstå semantiken måste tänka sig en vonNeuman-maskin.

```
power 3 6                                          blir
3 * power 3 5                                      blir
3 * (3 * power 3 4)                                blir
3 * (3 * (3 * power 3 3))                          blir
3 * (3 * (3 * (3 * power 3 2)))                    blir
3 * (3 * (3 * (3 * (3 * power 3 1))))              blir
3 * (3 * (3 * (3 * (3 * (3 * power 3 0)))))        blir
3 * (3 * (3 * (3 * (3 * (3 * 1)))))                blir
3 * (3 * (3 * (3 * (3 * 3))))                      blir
3 * (3 * (3 * (3 * 9)))                            blir
3 * (3 * (3 * 27))                                 blir
3 * (3 * 81)                                       blir
3 * 243                                            blir
729
```

Minnesåtgång kan bli stor. (En rättfram rekursiv lösning på labbens Fibonacchi-uppgift blir mycket ineffektiv, varför?). En bättre lösning är :

```
power :: Int -> Int -> Int   -- accumulerande lösningar  med where
power _ 0 = 1
power m n
  | n<0       = error " second argument till power  >= 0"
  | otherwise = pw m n 0 1
  where                          -- layout  Fokker 1.4.5
    pw :: Int->Int->Int->Int-> Int
    pw m n i acc
       | i==n        = acc
       | otherwise   = pw m n (i+1) (acc*m)

    power 3 6           blir   -- Körning, dvs förenkling eller reduktion
    pw 3 6 0   1        blir   -- Kallas även evaluering, eng  evaluation
    pw 3 6 1   3        blir   -- Hudak: Computation by Calculation  blir =>
    pw 3 6 2   9        blir
    pw 3 6 3  27        blir
    pw 3 6 4  81        blir
    pw 3 6 5 243        blir
    pw 3 6 6 729        blir
    729
```

Minneseffektiv. Även i funktionella språk måste man (förstås) tänka ibland. Att denna lösning "inte sväller ut" beror på att det rekursiva anropet av `pw` ej ingår i något uttryck. Man säger att `pw` är *svansrekursiv*."Imperativa programmerare" lär sig att skirva "loopar" i dessa fall och att bara tillgripa rekursion när det är svårt att lösa iterativt. (Mer om effektivitet i Bvs, fast svårläst än så länge) Lösningar med accumulatorer (och, kommer senare, `until` och `foldl` men inte `foldr`) är svansrekursiva.