DD1361
Programmeringsparadigm

Carina Edlund

carina@nada.kth.se

Funktionell programmering

Grundidéen med funktionell programmering är att härma matematiken och dess funktionsbegrepp.

Matematiskt funktionsbegrepp

En funktion från en mängd M till en mängd N är en regel som varje objekt i M på ett entydigt sätt ordnar ett objekt i N. (Persson, Böijers)

Haskells funktionsbegrepp

Är en avbildning som tar ett eller flera argument och producerar ett enda resultat, och är definierad med hjälp av en ekvation som ger funktionen ett namn, ett namn för varje argument och en kropp som anger hur resultatet ska beräknas utifrån argumenten.(Hutton, Sebesta)

Konsekvenser

- Ett språk som endast använder denna typ av funktioner kallas ett rent funktionellt språk.
- Funktionerna ger alltid samma resultat med samma indata.
- Funktionerna har inga sidoeffekter.
- Använder inte variabler eller tilldelningssatser vilket innebär att programmeraren inte behöver bry sig om minneshantering och tillstånd i programmet.

Funktioner i matematiken

- Ett klassiskt matte-exempel

  $$f(x) = x^2$$

  eller tydligare

  $$\mathrm{square}(x) = x^2$$

- Eller exempel från nummen:

  ```
  rot_hittar_metod(gissad_rot, tolerens)
      diverse initieringar
      differentialekvationslösare(intervall, tid)
  ```

Enkel funktion, scheme

```
(define kvadrat
   (lambda (x)
     (* x x)))
```

Enkel funktion Haskell

```
kvadrat x =
    x*x
```

OBS! Indenteringen är viktig!

Körning

- Funktionen skrivs
- Funktionen interpreteras eller kompileras
- Funktionen anropas,
  - Scheme: `(kvadrat 3)`
  - Haskell: `kvadrat 3`
- Ett resultat beräknas och returneras

Haskell

- Vi kommer att använda Glasgow Haskell Compiler, GHC med dess interaktiva interpretator ghci

- CSC: module add ghc
  - Starta ghci med `dator> ghci`
  - Kompilera med ghc

- Hugs98: en etablerad interaktiv interpretator

Enkla beräkningar med ghci

```
dator> ghci
Prelude> 2*3
6
Prelude> not True
False
Prelude> True || False
True
Prelude> not (2>3)
True
Prelude> 'a' == 'a'
True
```

```
Prelude> 42/17
2.47058823529412
Prelude> 42 div 17
ERROR – Cannot infer …
Prelude> div 42 17
2
Prelude> 42 'div' 17
2
Prelude> mod 9 2
2
```

Haskell. Typer och dess värden

Typ	Värde
Bool	True, False
Int	-(2^31) till (2^31)-1
Integer	Godtyckligt stora heltal
Float	32 bitar
Double	64 bitar
Char	’a’, ’9’, ’\n’
String	”Haskell”

Haskell: grunder i kort

- Kommentar resten av raden med:

  `-- bortkommenterar resten av raden`
- Kommentarer över flera rader med: {- -}

  `{- Det här är en kommentar -}`
- Namn (parametrar, funktioner) börjar med gemen Ex. `minFunktion`
- Värden (symboler) börjar med versaler. Ex: `True`
- Funktionsanrop utan parenteser

  Ex: $f(3)$ skrivs `f 3`

Herons formel

Arean, A för en triangel med sidorna a, b, c är:

$$A = \sqrt{p\,(p-a)\,(p-b)\,(p-c)}$$

där $p = (a+b+c)/2$

I Haskell:

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

where och let

Uttrycket

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

kan ersättas med

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

Upprepning

Hur kan vi beräkna summan av kvadraterna:

$$\text{kvadrat\_summa} = \sum_{k=1}^{N} k^2$$

dvs

$$\text{kvadrat\_summa} = 1^2 + 2^2 + \ \ldots + N^2$$

Iteration

Imperativ stil använder iteration liknande:

```
del_program kvadrat_summa (n)
    resultat= 0
    k = 1
    loopa så länge k<=n
        resultat = resultat + k*k
        k = k + 1
    returnera resultat
```

Iteration, fortsättning

Här utnyttjas minnesutrymmen som kallas `resultat` respektive `k` för att lagra delresultat från beräkningarna som utförs i loopens steg.

I iterationen har vi inte bara möjlighet att lagra ett värde i ett minnesutrymme utan även att tilldela det ett värde om och om igen.

Rekursion

- För att upprepa beräkningar kan man i matematiken använda rekursion.
- Grundidén: är vi klara med beräkningarna returnera resultatet i basfallet annars ta ett steg närmare lösningen med hjälp av ett rekursivt anrop.

- $$kS(n) = \begin{cases} n^2, \ n = 1 \\ kS(n-1) + n^2, \ annars \end{cases}$$

Rekursion, scheme

```
(define kS
   (lambda (n)
     (if (= n 1)
          (* n n)
          (+ (kS (- n 1)) (* n n)))))
```

Rekursion, Haskell

```
kS n =
     if (n == 1) then
          n*n
     else
          kS (n-1) + n*n
```

Funktionen anropas:

```
*Main> kS 3
14
*Main>
```

Rekursion, beräkningssteg

Beräkningsstegen för anropet kS(3):

kS 3

kS 2 + 3*3

(kS 1 + 2*2) + 3*3

(1*1 + 2*2) + 3*3

5+9

14

Observationer

- If-satsens konstruktion och vad den returnerar
- Parentes kring (n-1), funkar det utan?
- För vilka värden på n är funktionen giltig?
- Vilken typ har indata? Utdata?
- Kan man begränsa indata?
- Vad händer vid anropet?
- Övrigt??

Rekursion, Haskell

Information om funktion:

```
*Main> :i kS
kS :: (Num a) => a -> a  -- Defined at kS.hs
```

Lägg till typsignatur

```
kS :: Integer -> Integer
kS n =
     if (n == 1) then
          n*n
     else
          kS (n-1) + n*n
```

Svansrekursion

- Skriv om funktionen så att resultaten av beräkningarna som utförs i varje steg skickas med det rekursiva anropet.
- Kan vara nödvändigt att införa ytterligare en parameter till funktionen för att lagra resultat.
- Var noga med att anropa med rätt värden.

Svansrekursion, matematiskt

Man kan i matematiken använda det svansrekursiva tänkandet:

$$kSS(n, res) = \begin{cases} res, & n = 1 \\ kSS\ (n-1, res + n * n), & annars \end{cases}$$

Anropet $\mathrm{kSS}(3,1)$ kan vi räkna ut enligt:

$\mathrm{kSS}(3,1)$ -> $\mathrm{kSS}(2,10)$ -> $\mathrm{kSS}(1,14)$ -> 14

dvs $\mathrm{kSS}(3,1) = 14$

Svansrekursivt tänkande, Haskell

Lägg till typsignatur

```
kSS :: Integer -> Integer -> Integer
kSS n res =
     if (n == 1) then
          res
     else
          kSS (n-1)  (res + n*n)
```

Anropa funktionen, (varför dessa värden?):

```
*Main> kSS 3 1
14
```

Svansrekursion

- När resultatet av det rekursiva anropet INTE är en operand till en operation kan man tala om svansrekursion.
- Svans-/rekursion ersätter loop i t.ex. imperativa språk.
- Känns igen av moderna kompilatorer/interpretatorer

Funktion igen

Funktion som testar om ett tecken är en siffra:

```
isDigit :: Char -> Bool
isDigit c =
    c >= '0' && c <= '9'
```

Vakter (guarded expressions)

I matematiken

$$abs(x) = \begin{cases} x, & x > 0 \\ -x, & annars \end{cases}$$

I Haskell

```
abs x | x>0 = x
      | otherwise = -x
```

Vakt och mönstermatchning

Kvadratsummeberäkning med vakt:

```
kSV n | n == 1 = 1
      | otherwise = kSV (n-1) + n*n
```

Kvadratsummeberäkning med mönstermatchning:

```
kSM 1 = 1
kSM n = kSM (n-1) + n*n
```

Layout-regeln

”Syskon-uttryck” står under varandra

```
f a b c = a + d + e
  where d = b^3
        e = c^4
```

OBS! Indentering

Rekursion (upprepning)

Hur kan vi beräkna den harmoniska serien?

$$harm\_serie \quad = \sum_{k=1}^{\infty} 1/x^k$$