Funktionell programmering

DD1361

Tupler

- Två eller fler (men ändligt) antal element. Elementen kan vara av olika typer.

  Ex: `(1,2) :: (Integer, Integer)`
  `(('2', "hejsan") , True) :: ?`

- Tupel med två element -> ett par. Kan använda funktionerna `fst` och `snd`. `fst(1,2) -> 1`
- Varför inte en tupel med ett element, t.ex. `(+)`?

Listor allmänt

I imperativa språk används t.ex. pekare och poster för att explicit bygga upp en önskad liststruktur. Dvs man allokerar minnesutrymme för posten, länkar samman den med listan och bygger på så sätt ut listan.

En fin bild på hur det kan se ut!

Diskontinuerlig lista, ~C

delProgram (listPekare, värde)

    allokera minnesutrymme för post
    om allokering ok
        tilldela posten dess värde
        länka posten till listan
        återsänd den nya listan
    annars
        signalera fel

Finns det sidoeffekter?

Listor i scheme

- Byggs upp av par, där det sista elementet är den tomma listan, `()`.
- `cons` sätter ihop två komponenter till ett par.
- `car, cdr` plockar ut den första respektive den andra komponenten ur ett par.

Anrop	Resultat
`(cons 2 ())`	`-> (2)`
`(cons 2 (cons 3 ()))`	`-> (2 3)`
`(car (cons 2 (cons 3 ())))`	`-> 2`
`(cdr (cons 2 (cons 3 ())))`	`-> (3)`

Listor i Haskell

- Godtyckligt antal element av samma typ.
- Ett element sätts in i en lista med hjälp av :
- En äkta lista har alltid den tomma listan, `[]` sist.

Exempel:

```
Prelude> 2:[]
[2]
Prelude> 3:[2]
[3,2]
```

```
Prelude> [True, False]
[True, False]
Prelude> [”Hej”, ”då”]
[”Hej”, ”då”]
```

Arbeta med listor i Haskell

```
Prelude> 2:(3:(6:[]))
[2,3,6]
Prelude> 8:[2,3,6]
[8,2,3,6]
Prelude> [1,2]++[4,3]
[1,2,4,3]
Prelude> null []
True
Prelude> null [2,3,4]
False
```

```
Prelude> head [2,3,6]
2
Prelude> tail [2,3,6]
[3,6]
Prelude> elem 2 [2,3]
True
Prelude> length [2,3]
2
Prelude> sum [2,3]
5
```

Polymorfism

- Vilken typ har `length`?

```
Prelude> :t length
length :: [a] -> Int
```

`a` är en typvariabel

- Polymorf funktion: Kan tillämpas på flera typer. Polymorf typ: ”Kan anta flera former” (Hutton)

Fler polymorfa exempel

```
Prelude> :type head
head :: [a] -> a

Prelude> :t tail
tail :: [a] -> [a]

Prelude> :t fst
fst :: (a,b) -> a
```

Listlängd

1. Är listan tom är längden noll
2. Addera ett till längden av resten av listan

- Scheme

```
(define length
  (lambda (lista)
    (if (null? lista)
          0
          (+ 1 (length (cdr lista))))))
```

`map`, en viktig listfunktion

```
Typsignatur:
map :: (a -> b) -> [a] -> [b]

Prelude> map length ["Asp","Gran","Ek"]
[3,4,2]
```

I detta exempel gäller:

```
a: String
b: Int
```

Mösterpassning på listor

Huvudet på en lista:

```
head [] =
    error ”No head on an empty list!”
head (e:restOfList) = e
```

eller

```
head [] =
    error ”No head on an empty list!”
head (e:_) = e
```

Strängar

En sträng är en lista med tecken, dvs `String` är synonymt med `[Char]`

```
'a':'b':'c':[] :: String
['a','b','c'] :: String
"abc" :: String

Prelude> head "hejsan"
'h'
Prelude> tail "rot" ++ 'a':[]
"ota"
```

Räkna blanka i en sträng, 1

```
countSpaces :: String -> Int
countSpaces [] = 0
countSpaces (c : restOfString)
    = d + countSpaces restOfString
       where d = if c == ’ ’ then 1
                             else 0

Prelude> countSpaces ”X Y Z ”
3
```

Vad händer med `countSpaces "X Y Z "`?

Rekursionssteg

```
countSpaces ”X Y Z ”
0 + countSpaces ” Y Z ”
0 + 1 + countSpaces ”Y Z ”
0 + 1 + 0 + countSpaces ” Z ”
0 + 1 + 0 + (1 + countSpaces ”Z ”)
0 + 1 + 0 + (1 + 0 + countSpaces ” ”)
0 + 1 + 0 + (1 + 0 + (1 + countSpaces ””))
0 + 1 + 0 + (1 + 0 + (1 + 0))
0 + 1 + 0 + (1 + 0 + 1)
……
1 + 2
3
```

Det belastar stacken!

Räkna blanka i en sträng, 2

```
countSpaces :: String -> Int
countSpaces [] = 0
countSpaces (' ' : restOfString)
        = 1 + countSpaces restOfString
countSpaces (_ : restOfString)
        = 0 + countSpaces restOfString
```

Räkna blanka i en sträng, 3

```
countSpaces :: String -> Int
countSpaces [] = 0
countSpaces (c : restOfStr)
  | c == ' ' = 1 + countSpaces restOfStr
  | otherwise = countSpaces restOfStr
```

Räkna blanka i en sträng, 4

```
countSpaces :: String -> Int
countSpaces s = count 0 s
count n [] = n
count n (c : restOfStr)
    | c == ' ' = count (n+1) restOfStr
    | otherwise = count n restOfStr
```

Ackumulering av resultatet i parametern n.

Räkna blanka i en sträng, 5

```
countSpaces :: String -> Int
countSpaces s = count 0 s
   where
     count n [] = n
     count n (c : restOfStr)
        | c == ' ' = count (n+1) restOfStr
        | otherwise = count n restOfStr
```

`count` är lokal för `countSpaces` och syns inte för andra funktioner! Eng: local scope.

Vad händer i count?

```
count 0 ”X Y Z ”
count 0 ” Y Z ”
count 1 ”Y Z ”
count 1 ” Z ”
count 2 ”Z ”
count 2 ” ”
count 3 ””
3
```

Från loop till rekursion

Räkna vokaler och konsonater

```
function count(string s){
  int nCons = 0;
  int nVow = 0;
  for (int i=0; i++; i<length(s)) {
     if (isCons(s[i])) then {
        nCons++;
     } else if isVow(s[i]) then {
        nVow++;     }  }
  return (nCons, nVow);}
```

Recept: loop till rekursion

- En loop ersätts med en inre funktion.
- Varje aktiv variabel i loopen blir en parameter i rekursionen.
- Stoppvillkoret blir basfall.
- Starta rekursionen med loopens starttillstånd.

Tillämpa receptet

- `for`-loopen ersätts med funktionen `loop`.
- `loop` har 3 parametrar: `i, nCons och nVow`.
- Basfallet blir `i>=length s`.
- Startanrop: `loop 0 0 0`
  (dvs `i=0, nCons=0, nVow=0`)

Räkna vokaler och konsonanter

```
count :: String -> (Int, Int)
count s = loop s 0 0
  where
   loop [] nCons nVow = (nCons, nVow)
   loop (c:str) nCons nVow
    | isCons c = loop str (nCons+1) nVow
    | isVow c = loop str nCons (nVow+1)
    | otherwise = loop str nCons nVow
```

Ömsesidig rekursion

Rensa bort LaTex-kommentarer i en given sträng

Från ’%’ till radslut.

```
remCom :: String -> String
remCom []           = []   -- Basfall!
remCom (’%’ : str) = stripCom str
remCom (c : str)    = c : remCom str

stripCom :: String -> String
stripCom []              = []
stripCom (’\n’ : str)  = ’\n’ : remCom str
stripCom (_ : str)       = stripCom str
```

Listomfattning

- Hur kan vi dubblera alla udda tal större än 3 i en lista?

1. Gå igenom listan rekursivt ett element i taget och bygg upp en ny lista med de element som uppfyller kraven har dubblerats.
2. Filtrera bort allt man inte vill ha, resten dubbleras.
3. Tänk matematiskt. $M = \{2*x;\ x>3,\ x \text{ udda},\ x \in \{1..6\}\}$

   Använd listomfattning, (listan [1..6] är hårdkodad):

   ```
   [2*n| n <- [1,2 .. 6], odd n, n>3]
   ```

3. Ytterligare idéer?

Kompakt quicksort

- Välj ett pivotelement x.
- Sortera alla element mindre än x.
- Sortera alla element större än x.
- Sätt samman resultatet.

Med listomfattning:

```
qsort [] = []
qsort (x:xs) = qsort [e | e <- xs, e<x]
               ++ [x]
               ++ qsort [e | e <- xs, e>=x]
```

Listomfattning

Pythagoras tal, $a^2 + b^2$:

```
pythagoras n =
      [a^2+b^2| a <- [1..n], b <-[a..n]]
Prelude> pythagoras 5
[2,5,10,17,  osv ]   Testa själva!!!
```

Returnera tupler med mer info:

```
pythagoras n =
     [(a^2+b^2,a,b)| a<- [1..n], b <-[a..n]]
Prelude> pythagoras 5
[(2,1,1),(5,1,2),  osv ]   Testa själva!!!
```