Funktionell programmering

Marcus Dicander

2016-02-25

Strukturen på funktionell programmering

1. Introduktion
2. Typer och Typklasser
3. Högre ordningens funktioner
4. Att använda Monader
5. Sammanfattning och fördjupning

Strukturen idag - Introduktion

1. Vad är funktionell programmering
2. Enkel matematik i GHCI
3. Enkla operationer på listor och strängar i Haskell
4. Funktioner
5. Pattern matching, Guards
6. Rekursion och svansrekursion
7. Lat evaluering, oändliga listor

Vad är funktionell programmering

- En programmeringsstil där den grundläggande operationen är funktionsanrop
- En vidareutveckling av lambdakalkylen från 1930-talet (Alonzo Church)
- Funktionella språk: Lisp, Scheme, Erlang, Haskell, F♯, Clojure

Less is more..

- Inga variabler, ingen initialisering, ingen uppdatering.
- Inga datastrukturer som möjliggör förändringsbart state. Inga föränderingsbara arrayer (Finns i monader, föreläsning 4).
- Inga for-loopar - iteration med rekursion (idag) och högre ordingens funktioner (föreläsning 3)

More is more...

- Full referential transparency - i en fil eller ett metodanrop betyder ett namn alltid samma sak
- Inga sidoeffekter - En funktion som anropas med samma värde två gånger ger alltid samma resultat
- Enkel parallelliserbarhet

Inslag av funktionell programmering i de vanligaste språken

- Java8 har Streams med forEach (map), filter, reduce (foldl) och lambda (föreläsning 3)
- Javascript (ES6)
- C++11 (parallellprogrammeringen, lambdas), C♯
- Python2/Python3 modulerna itertools och functional
- Perl6, pugs - första implementationen av Perl6 skrevs i Haskell (men Rakudo är ledande idag)

Hello World

```
main::IO()
main =
    putStrLn "Hello World!"
```

Glasgow Haskell Compiler

```
$ghc hello.hs
[1 of 1] Compiling Main                ( hello.hs, hello.o )
Linking hello ...
$./hello
Hello World!
```

ghci - Glasgow Haskell Compiler Interactive

```
Prelude> 1+2*3
7
Prelude> 16/3
5.333333333333333
Prelude> 16 `div` 3
5
Prelude> div 16 3
5
```

Vad du kan göra med GHCI

- Beräkna aritmetiska uttryck med +,-,*,/,‘div‘,‘mod‘
- Skapa listor och strängar
- Öppna filer med funktioner för att testa dessa

Skapa listor

```
Prelude> let decimals = [1,4,1,5,9,2,6,5]
Prelude> let firstTen = [1..10]
Prelude> firstTen
[1,2,3,4,5,6,7,8,9,10]
Prelude> let notSeven = [1..6]++[8..10]
```

Skapa fler listor

```
Prelude> let surprise = "Jag är en lista med Chars"
Prelude> surprise
"Jag \228r en lista med Chars"
Prelude> putStrLn surprise
Jag är en lista med Chars
Prelude> let grades = ['A'..'E']
Prelude> let name = "Haskell" ++ " " ++ "Curry"
Prelude>
```

Delar av listor och strängar

- head - första elementet, tail - resten av listan
- init - allt utom sista elementet, last - sista elementet
- take n - de n första elementen, drop n - allt efter de första n elementen
- !! n - Det n:te elementet med nollindexering. Skrivs efter listan.
- alla utom sista skrivs före listan: head [1,2,3]

ghci - Nu med en fil

```
Prelude> :load mathematics.hs
Prelude> :l mathematics.hs
[1 of 1] Compiling Main                ( mathematics.hs, interpr
Ok, modules loaded: Main.
*Main> hypothenuse 3 4
5.0
```

Pattern matching

- Undvik guards och if-satser. Om ett beteende följer direkt av ett konkret värde på en parameter, gör ett pattern för det
- Patterns scannas uppifrån och ned. Gotta catch'em all.
- Om du inte bryr dig om ett parametervärde, använd _

mathematics.hs

```
{- Returns the hypotenuse in a triangle with catheti a and b
hypotenuse::Double->Double->Double
hypotenuse a b =
    sqrt (a^2 + b^2)
```

Collatz Conjecture

$$n_{i+1} = \begin{cases} n_i/2 | even(n_i) \\ 3 * n_i + 1 | odd(n_i) \end{cases}$$

Påstående: Serien når 1 $\forall n \in Z^+$

Obevisad, men verifierad upp till $5.4 * 10^{18}$

Rekursion

- Börja med basfall. Vid vilket parametervärde är du klar?
- Fortsätt med att låta funktionen anropa sig själv så att den steg för steg går mot basfallet och bygger upp svaret

Svansrekursion

- Om det rekursiva anropet returnerar direkt utan några kvarhängade operationer från tidigare anrop så är metoden svansrekursiv
- Svansrekursiva anrop kan optimeras bättre av GHC, men se upp med '++'

Oändliga listor, lat evaluering

```
Prelude> let infinite = [1..]
Prelude> take 10 infinite
[1,2,3,4,5,6,7,8,9,10]
Prelude> take 10 oddCollection
[1,3,5,7,9,11,13,15,17,19]
Prelude> let primes = [2,3,5,7..]

<interactive>:6:22: parse error on input `..'
```

Funktionell programmering

DD1361

Polymorfism

- Polymorf funktion: Kan tillämpas på flera typer.
- Polymorf typ: ”Kan anta flera former” (Hutton).
- Vilken typ har `length`? Kika!

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

- `a` är en typvariabel
- Terminologi: parametrisk polymorfism

Fler polymorfa exempel

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

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

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

Typklasser

- Hur ställer vi krav på typer?
- Om `sum :: [a] -> a`, hur garanterar vi att addition är definierat på a?
- Om `sort :: [a] -> [a]`, hur vet vi att <= finns på a?

```
Prelude> :t sum
sum :: Num a => [a] -> a
Prelude> :t sort
sort :: Ord a => [a] -> [a]
```

Några typklasser

- Num Alla numeriska typer
- Ord Alla ordnade typer (<=)
- Eq Alla jämförbara typer (==)
- Show Typer med skrivbara element
- Read Typer med läsbara element
- Integral Heltalstyper: `Int, Integer`
- Fractional `Float` tex, de som stödjer division.

Överlagring (overloading)

- Samma funktion definierad på olika argument
- T.ex.: Addition (+) på `Int` är inte samma som addition på `Float`.
- ”ad hoc” polymorfism
- C++:

```
int Klass::kamp(int x, int y) {
return x * y; }
int Klass::kamp(String x, String y) {
return x.length() * y.length(); }
```

Överlagring i Haskell

- Explicit överlagring
- Skapa en klass som beskriver gemensamma egenskaper
- Visa hur överlagringen hanteras för olika typer

Överlagring i Haskell

- Från Prelude.hs:

```
class Eq a where
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

- Också från Prelude.hs:

```
instance Eq Integer where
x == y = integerEq x y
```

- Tala om att din klass tillhör Eq:

```
instance Eq MinKlass where
x == y = minEq x y
```

Arv hos typklasser

- Definiera ordnade typer:

```
class Eq a => Ord a where
(<), (<=) :: a -> a -> Bool
(>=), (>) :: a -> a -> Bool
```

- ”Alla `Ord`-typer måste vara `Eq`-typer också.”

Tillämpning

```
class Incrementable a where
      inc :: a -> a
instance Incrementable Char where
      inc c = chr ((ord c) + 1)
instance Incrementable a =>
   Incrementable [a] where
        inc l = map inc l
Prelude> inc ’a’
’b’
Prelude> inc "HAL"
"IBM"
```

Klassfunktioner

En klass kan definiera gemensamma funktioner:

```
class Incrementable a where
      inc :: a -> a
      inc2 :: a -> a
      inc2 x = inc (inc x)
instance Incrementable Char where
      inc c = chr ((ord c) + 1)
Prelude> inc ’a’
’b’
Prelude> inc2 ’a’
’c’
```

Omdefinition

```
class Incrementable a where
      inc :: a -> a
      inc2 :: a -> a
      inc2 x = inc (inc x)
instance Incrementable Integer where
      inc x = x + 1
instance Incrementable Char where
      inc c = chr ((ord c) + 1)
      inc2 c = chr ((ord c) - 32)
```

Omdefinition

```
Prelude> inc2 1
3
Prelude> inc ’a’
’b’
Prelude> inc2 ’a’
’A’
```

Jämför med objektorientering

Likheter

- Klasser och arv
- Klasser samlar funktioner
- Standardmetoder

Skillnad

- Inga objekt, inga attribut
- Två sorts polymorfism: Parametrisk och ad hoc
- Alla metoder är statiskt bestämda, ingen klassinformation i data
- Åtkomstkontroll (public, private) indirekt via modulsystem

Typkonvertering

I många språk:

```
Pseudo> length(lista) / 10
⇒5.5
```

I Haskell:

```
Prelude> length(lista) / 10
<interactive>:1:0:
No instance for (Fractional Int)
arising from a use of ‘/’ at
Possible fix: add an instance
declaration for (Fractional Int)
```

Explicit typkonvertering

```
Prelude> fromIntegral(length(lista))/ 10
5.5
Prelude> :t fromIntegral
fromIntegral :: (Integral a, Num b)
=> a -> b
```

Typkonverteringar i Prelude

- fromInteger
- fromRational
- toInteger
- toRational
- fromIntegral
- fromRealFrac
- fromIntegral
- fromRealFrac

Definiera typer

God programmering kräver

- god organisering
- god abstraktion

Abstrahera och organisera dina data!

I Haskell

- typer
- datastrukturer

Definera egna typnamn

```
type Distance = Float
```

gör att `Float` och `Distance` kan användas omväxlande.

Liknande med

```
type Coord = (Float, Float)
```

Definera egna typnamn

Även med polymorfa typer:

```
type AssocList a b = [(a, b)]
```

Användning:

```
getElem :: AssocList String Int ->
String -> Int
```

Fördelar:

Praktiskt! Tydliggörande!

Definiera egna datastrukturer

Vi vill kunna aggregera information.

Pascal, C, etc: record eller struct

Java, C++, etc: Klasser

Haskell: `data`

Exempel:

```
data Bool = True | False
data Address = None | Addr String
```

Konstruktorer: `True, False, None, Addr`

Exempel: Datatyp för färger

```
data Color = RGB Int Int Int
black = RGB 0 0 0
white = RGB 256 256 256
Main*> :t black
black :: Color
Main*> black
```

Ger felmeddelande! Detta händer även när man jämför de två objekten, `black == white`

Exempel: Datatyp för färger

```
data Color = RGB Int Int Int
    deriving (Show, Eq) -- mer?

black = RGB 0 0 0
white = RGB 256 256 256
```

Med hjälp av deriving kommer vi åt klasserna Show och Eq

Exempel: Datatyp för färger

```
Main*> black
RGB 0 0 0
Main*> white
RGB 256 256 256
Main*> black == white
False
```

Polymorfa datatyper

Välj din egen färgrepresentation:

```
data Color f = RGB f f f
                 deriving (Show, Eq)

black = RGB 0 0 0
white = RGB 1.0 1.0 1.0
```

Polymorfa datatyper

...fast begränsa till nummer!

```
data Num f=>Color f = RGB f f f
                    deriving (Show, Eq)

black = RGB 0 0 0
white = RGB 1.0 1.0 1.0
```

Datatyper och mönstermatchning

```
data Color = RGB Float Float Float
             deriving (Show, Eq)
red (RGB r g b) = r
green (RGB r g b) = g
blue (RGB r g b) = b
brightness :: Color -> Float
brightness (RGB r g b) =
           sqrt((r^2+g^2+b^2) / 3)
```

Datatyper med åtkomstfunktioner

Alternativ definition av Color:

```
data Color =
     RGB {red, green, blue :: Float}
                    deriving (Show, Eq)
brightness :: Color -> Float
brightness c =
    sqrt(((red c)^2+(green c)^2+
          (blue c)^2)/3)
```

Algebraiska datatyper

- En typ ”komponerad” av andra typer, var

och en med en konstruktor, kallas algebraisk.

- Obs: En produkttyp har endast en konstruktor.

Ex: `data Color = RGB Int Int Int`

- Obs: En enumererad typ listar konstruktorer utan argument.

Ex: `data Bool = True | False`

Minns färgmodulen: Color

```
data Color = RGB Int Int Int
```

Hur göra om vi vill ändra representationen?

```
data Color = HSV Int Int Int
```

Förstör annan kod!

```
brightness :: Color -> Float
brightness (RGB r g b) =
                    sqrt((r^2+g^2+b^2) / 3)
```

Exempel: En ADT för färger

```
data Color = RGB Int Int Int

makeColorRGB r g b = RGB r g b
makeColorHSV h s v = ...
```

Typer i olika språk

Statisk typning: Typen begränsar variabeln:

```
int i := 17
```

Används i Java, Haskell, etc.

Dynamisk typning: Beskriver värdet. Tänk:

```
i := 17
```

Används i Lisp, MatLab, Perl, etc

Typkonvertering

Implicit konvertering:

```
float x = 3.5;
int i = x;
```

Explicit konvertering:

```
float x = 3.5;
int i = (int) x;
```

Konvertering och omtolkning

- Vad får i för värde?

```
char *s = "Hej!";
int i = (int) s;
```

- Vad händer här?

```
long int stort =0x7FFFFFFFFFFFFFFF;
int litet = (int) stort;
printf("stort = %ld, litet = %d\n",
        stort, litet);
```

Utdata:

```
stort = 9223372036854775807, litet = -1
```

Stark och svag typning

- Oklara begrepp. Avser hur starkt språket begränsar tolkningen av värden.
- **Sebesta:** ”strongly typed if all type errors are always detected.”
- ”Ada, Java, and C# are nearly strongly typed”

Stark och svag typning: exempel

- Tcl: Alla variabler kan tolkas som strängar.
- Perl: Listvariabel kan tolkas som listlängd, beroende på sammanhang. En lista kan tolkas som en associativ array (hashtabell).
- C: Automatisk typkonvertering (casting) mellan flera inbyggda typer. Union kan ej typkontrolleras.
- C++: Skriv dina egna automatiska konverteringsregler.
- Haskell: Ingen automatisk typkonvertering.

Säkra typsystem

- Def: Ett typsystem är säkert om det garanterar att inga uttryck kan anta värden som inte är definierade av typen.
- Osäkert språk: C
- Säkra språk: Haskell, Ada(?)

Funktionell programmering

DD1361

Högre ordningens funktion

En funktion är en högre ordningens funktion om:
- den tar en eller flera funktioner som argument

eller
- returnerar en funktion

eller
- båda ovanstående

(Sebesta s. 695)

Funktionskomposition

- I matte: $f \circ g$ ”sammanfogar” f och g.
  Dvs $f \circ g(x) = f\,(g(x))$
- I Haskell: `(f . g) x`
- Kräver alltså att

```
g :: a -> b
f :: b -> c
```

Funktionskomposition, exempel

Dubbel tillämpning:

```
twice :: (a -> a) -> (a -> a)
twice f = f . f

kvadrat x = x*x

*Main> twice kvadrat 2
16
```

Lambda-kalkyl

Alonzo Church på 30-talet: formaliserade beräkningar med speciell notation kallad λ-kalkyl.

Delar av lambda-kalkylen har inspirerat den funktionella paradigmen. I scheme används lambda.

Allmänt: $\lambda x \rightarrow \text{expr}$ $\lambda x\, y \rightarrow \text{expr}$

Exempel: $\lambda\, x\, y \rightarrow 1/(x + y)$ blir i Haskell:

```
\ x y -> 1 / (x + y)
```

```
Prelude> (\x y -> 1 / (x + y)) 2 3
0.2
```

Lambda-kalkyl i Haskell

Man kan betrakta funktioner i Haskell som uppbyggda mhja λ-notation dvs

```
addera = \x y -> x + y

*Main> :t addera
addera :: Integer -> Integer -> Integer
```

Vilken är typen om man bara skickar med ett värde till addera?

```
*Main> :t (addera 2)
(addera 2) :: Integer -> Integer
```

Skapa funktioner av funktioner

På detta vis kan man bygga upp nya funktioner och som i detta fall kan man utgå från en generalisering och gå till en specialisering.

```
add2 = (addera 2)
```

Currying

Funktioner som tar ett argument och returnerar en ny funktion kallas Curried functions

Currying är en princip i Haskell:

Alla funktioner tar alltid en parameter och returnerar ett värde eller en ny funktion.

Haskell har fått namnet från Haskell B. Curry som är en av upphovsmännen till lambda-kalkylen. (Thompson s.236)

Syntaktiskt socker döljer λ-notation!

Med socker:

```
inc x = x + 1
```

eller

```
inc = (+) 1
```

Utan socker:

```
inc = \ x -> x + 1
```

Anonyma funktioner

Kan använda lambda-notationen för att skapa icke namngivna funktioner.

Exempel:

```
addOneList = map (\x -> x + 1) [1, 2, 3]
```

Kan man ange en typsignatur för en anonym funktion?

```
((\x -> x + 1) :: Int -> Int)
```

Några typiska grupper för funktioner

1) Mappning - applicera en funktion på alla element
2) Filtrera - välja element
3) Folding - Kombinera element
4) Bryta upp listor
5) Kombinationer
6) Primitiv rekursion och folding
7) Övrigt

Ett återkommande mönster

Scheme, schematiskt

```
(define funktionsnamn
  (lambda (lista indata)
    (if (listan är tom)
        returnera ett passande värde
        ((gör något med första elementet)
          sammanfogning
         (undersök resten av listan)))))
```

Ett återkommande mönster

Haskell, schematiskt

```
funknamn indata [] =
   returnera ett passande värde
funknamn indata
  (förstaElementet : restenAvListan) =
       ((gör något med förstaElementet)
        sammanfogning
        (undersök resten av listan))
```

map idé

```
map1 f [] = []
map1 f (x:xs) = (f x : (map1 f xs))

*Main> map1 kvadrat [1,2,3]
[1,4,9]

convertChars :: [Char] -> [Int]
convertChars str = map1 fromEnum str

*Main> convertChars "Ann"
```

Filter

Signatur:

```
filter :: (a -> Bool) -> [a] -> [a]
```

Exempel 1:

```
Prelude> [1,2..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> filter odd [1,2..10]
[1,3,5,7,9]
```

Filter, forts

Exempel 2:

```
Prelude>filter even (map length
                ["Chalmers, "LiTH, "KTH"])
[8,4]
```

Exempel 3:

```
Prelude> filter ((\ x -> x <= 3).length)
             ["Tom","Oskar","Ann","Lisa"]
["Tom","Ann"]
```

Filter, idé

```
filter1 predikat [] = []
filter1 predikat (x:xs)
  | predikat x = x : filter1 predikat xs
  | otherwise = filter1 predikat xs


*Main> filter1 (>0) [-1,1,-2,2]
[1,2]

-- med listomfattning
filter2 predikat xs =
                [ x | x<-xs,   predikat x ]
```

Generalisering

```
length1 [] = 0
length1 (x:xs) = 1 + (length1 xs)

*Main> length1 [1,2,3]
3

sum1 [] = 0
sum1 (x:xs) = x + (sum1 xs)

*Main> sum1 [1,2,3]
6
```

Hitta det gemensamma mönstret!

Folding - kombinera element

Tanken är att man viker in en operator mellan listans element. Generaliserar mhja folding.

”Veckla ut” definitionen:

$(f\ e_1\ (f\ e_2\ (f\ e_3\ \ldots(f\ e_x\ \text{startvärde})\ \ldots)))$

```
foldr _ start [] = start
foldr f start (e : rest) =
              (f e (foldr f start rest))
```

Användning av foldr

Summera alla tal i en lista

```
summa = foldr (+) 0 [1,2,3]
```

Listlängd:

```
length lst =
       foldr (\el acc -> acc + 1) 0 lst
```

Listlängd med Currying:

```
length = foldr (\el acc -> acc + 1) 0
```

Map:

```
map f =
    foldr (\el rest -> (f el : rest)) []
```

Hur kan vi vända en lista, reverse ?

```
reverse1 xs = foldr snoc [] xs

-- snoc; cons  baklänges!
snoc x xs = xs ++ [x]

*Main> reverse1 [1,2,3]
[3,2,1]
```

Inte alltid rätt att jobba ”från höger”.

- Inte alltid effektivt att jobba ”från höger”.
- Upprepade ”append” med ++!
- Varför kan vi inte använda (:)?

```
foldl _ start [] = start
foldl op start (e : rest) =
          (foldl op (op start e) rest))

reverse = foldl (\xs x -> x : xs) []
```

Funktioner

- Har samma ”rättigheter” som andra värden.
- Behöver inte namnges, anonym
- Nya funktioner från gamla, currying
- Kan ges som argument
- Kan returneras som värde
- Kan lagras i datastrukturer

Funktioner i lista

```
fkn = [(+),(-)]

*Main> (head fkn) 2 1
3

*Main> (head (tail fkn)) 2 1
1
```

Strikt evaluering

Def: Strikt evaluering innebär att alla parametrars värde är kända när en operator eller funktion anropas.

Ex: Funktionsanrop i Java, C, etc

Icke-strikt evaluering

Hur beräknas

- `if jobbigt(17) && tungt(4711) then ...`?

Vanligen: Man ”kortsluter” jämförelsen.

Implementeras för `and` och `or`

Vad händer här?

```
int main(int argc, char** argv) {
    if (hej() || hopp()) {
       printf("du glade\n");}
    else {
       printf("nix pix!\n");}}
int hej() {
    printf("hej ");
    return 1;}
int hopp() {
    printf("hopp ");
    return 1;}
```

Lat evaluering

Hur beräknas

- `head [1..10]`?
- `head (qsort longlist)`?
- Lat evaluering i Haskell:

  ”Beräkna bara det som behövs”
- Varje uttryck lagras som ett löfte om evaluering — vid behov.

Se Hutton eller be en kursare förklara sina anteckningar.

Testa latheten!

Vad händer med

`head (qsort [4,5,4,1,2,9])?`

- Instrumentera koden:

```
import Hugs.Observe
-- Ej standardmodul
qsort [] = []
qsort (x : xs) =
 qsort [(observe ’’e1’’ e) | e<-xs, e<x]
 ++ [(observe ’’x’’ x)]
 ++ qsort [(observe ’’e2’’ e) |
                              e<-xs, e>=x]
```

Testa latheten!

```
QS> head (qsort [4, 5, 4, 3, 1, 2,9])
1
>>>>>>> Observations <<<<<<
e1
1
3
2
2
1
x
1
QS>
```

Inga värden på `e2` — det uttrycket har aldrig behövt evalueras

Vad händer med...

- ones = (1 : ones)?

```
Prelude> ones
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...∞
```

- take 10 ones ?

```
Prelude> take 10 ones
[1,1,1,1,1,1,1,1,1,1]
```

- ex = (1 : map ((+) 1) ex)?

```
Prelude> take 10 ex
[1,2,3,4,5,6,7,8,9,10]
```

Strömmar: Oändliga listor

Typexemplet: Fibonaccitalen

Antag att `fibs` är listan av Fibonaccital.

```
      1 1 2 3  5  8 13 21 ... fibs
    + 1 2 3 5  8 13 21 34 ... tail fibs
      2 3 5 8 13 21 34 55 ... tail(tail fibs)
```

```
fibs = 1:1:(elementWiseAdd fibs
                           (tail fibs))
    where elementWiseAdd = zipWith (+)
Prelude> take 10 fibs
[1,1,2,3,5,8,13,21,34,55]
```

Plus och minus med lat evaluering

Nackdelar:
- Kan slöa ner ett program
  (använd då explicit strikt evaluering!)
- Kan slösa på minne: ”löftet” tar plats
- Överraskande för den ovetande?

Fördelar:
- Kan snabba upp ett program
- Undviker onödiga beräkningar.
- Automatiskt!
- Erbjuder smarta uttryckssätt.

Valfri strikthet

- Operatorn (`$!`). Ersätta `f x` med `f $! x`
  tvingar `x` att evalueras först.
- Funktionen `seq :: a -> b -> b`

```
  let x = fkn1
      y = fkn2
  in y ‘seq‘ fkn3 x y
```

- Anpassade funktioner, t.ex. `foldl’`.
- Strikta datatyper, t.ex.

```
  data StrictColor = SRGB !r !g !b
```

  lagrar inga löften

Funktionell programmering

DD1361

Generella egenskaper (funk prog)

- Variabler Nej!
- Tilldelning Nej!
- Sidoeffekter Nej!
- Pekare, referenser Nej!
- Skräpsamling (garbage collection) Ja!

Det funktionella dogmat

Genom att programmera funktionellt blir det:

- mindre att skriva
- färre buggar
- god modularisering
- god kodåtervinning

Haskell: egenskaper

- Strikt funktionellt. Inget ''fusk'' med tilldelning.
- Starkt typsystem. Hittar många fel tidigt, stödjer kodning.
- Typklasser. Överlagrade funktioner kodåtervinning, god abstraktion.
- Lat evaluering. Beräkna bara det som behövs.
- Funktioner är ''first class''. Behandlas som vilka värden som helst.
- Moduler. Storskalig programutveckling, goda bibliotek.

Utvikning: idiom

Wikipedia: An idiom is a low-level pattern that addresses a problem common in a particular programming language. Ett typiskt funktionellt idiom: Loopa över lista/sträng, inte över ett index.

Utvikning: Design patterns

'' Idiom på högnivå'' Wikipedia: A design pattern is a general reusable solution to a commonly occurring problem in software design.

Problemet med I/O

- I/O ej funktionellt
- getChar plockar bort ett tecken från en buffert
- putChar skriver in ett tecken i en buffert.
- Hur åstadkomma I/O utan sideffekter?
- **Lisp mfl:** Fuska! Använd sidoeffekter.

I/O Generellt

```
Pseudokod:
main =
    printStr("Rev: ")
    printStr(reverse(getLine))
```

- **I imperativt program:** Ordning och direkt tillgång till omvärlden.
- **I Haskell:** Ordning oklar, alla funktioner har värden, tillåter ej sidoeffekter (pure functions).

I/O i Haskell

- **Lat I/O:** Låtsas läsa in allt i början.
- **Monadisk I/O:** “Kapsla in världen på ett säkert sätt”
- Speciell notation för I/O.

interact: en god Unix-medborgare

```
interact :: (String -> String) -> IO ()

import Data.Char
main = interact (map toUpper)
```

Läs från `stdin`

Skriv till `stdout`

interact: en god Unix-medborgare

Vad gör detta?

```
module Main where
import Data.List
main =
   interact (concat . sort . lines)
```

interact: en god Unix-medborgare

Vad gör detta?

```
module Main where
import Data.List

newline str = str ++ "\n"

main =
 interact (newline . show .
           length . words)
```

De 10 vanligaste orden

```
module Main where
import Data.List (sortBy, sort, group)
import Data.Char (toLower)

countElems = map (\x ->
                    (head x, length x))

sortBySnd = sortBy (\x y ->
                  snd y ‘compare‘ snd x)

lower = map toLower
```

De 10 vanligaste orden

```
rankWords = sortBySnd . countElems .
            group . sort . words . lower

formatOutput = unlines . map
    (\(str, i) -> str ++ "\t" ++ show i)

main = interact (formatOutput .
                 (take 10) . rankWords)
```

Monadisk I/O

- Särskild notation som döljer problemen
- Monader: funktionellt idiom för
  - sekvensiella beroenden
  - att dölja parametrar
  - förenkla kod

Designprincip för I/O i Haskell

Världen	**Monadisk IO**	**Din kod**
Filer	getChar	myPreparations
Portar	getLine	computeItAll
stdin/stdout	openFile	myFilter
Grafik	readFile	
	isEOF	
	m.m.	

Ansats: Kapsla in världen

Vad vi vill ha:

```
type IO a = World -> (a, World)
```

IO-typer är handlingar: Eng: actions Exempel:

```
getChar :: IO Char
getChar :: World -> (Char, World)

putChar :: Char -> IO ()
putChar :: Char -> World -> ((), World)

isEOF :: IO Bool
getLine :: IO String
```

Till versaler igen

Låtsas läsa hela filen:

```
module Main where
import IO
import Data.Char
main = do {
           str <- getContents;
           putStrLn (map toUpper str);
}
```

Räkna ord igen

```
module Main where
import IO

main =do {
          input <- getContents;
          ws <- return (length
                        (words input));
          putStrLn (show ws);
}
```

Lat I/O: räkna ord i fil

```
module Main where
import IO
main = do { ih <- openFile
                    "input.txt" ReadMode;
            input <- hGetContents ih;
            ws <- return (length
                          (words input));
            putStrLn(show ws);
            hClose(ih)
}
```

Två nya operatorer i monadisk I/O

- `<-` plockar ut ett värde från IO-monaden. Kan skickas till “rena” funktioner utan IO-signatur.
- `return` betyder “sätt in ett värde i IO-monaden”. `return 'A'` skapar värde av typen `IO Char`.
- **Viktigt:** `return` avslutar ej ett `do`-uttryck!

Förenkla koden

blir

```
main = do {
     ih <- openFile "input.txt" ReadMode;
     input <- hGetContents ih;
     putStrLn (show (length
                          (words input)));
     hClose(ih);
}
```

Lat I/O: räkna ord i given fil

```
module Main where
import System.Environment (getArgs)
import IO
main = do { args <- getArgs;
            ih <- openFile (head args)
                                ReadMode;
            input <- hGetContents ih;
            putStrLn(show (length
                           (words input)));
            hClose(ih);
}
```

”Lat I/O farligt”, varför?

```
main = do {
     ih <- openFile "input.txt" ReadMode;
     input <- hGetContents ih;
     hClose(ih);
     putStrLn(show (length
                    (words input)));
}
```

Egen kod i IO-monaden: getLine

```
getLine :: IO [Char]
getLine = do { c <- getChar;
               if c == '\n' then
                  return []
               else
                  do { cs <- getLine;
                       return (c : cs)}
}
```

Paketera resultatet med return

Exempel: Räkna rader och tecken

Indata: Läs från stdin

Utdata: Skriv antalet rader och tecken till stdout

```
module Main where
import IO
main = do {
          (nLines, nChars) <- wc 0 0;
           putStrLn (show nLines ++
           "\t” ++ show nChars)
}
```

Exempel: Räkna tecken och rader

```
wc :: Int -> Int -> IO (Int, Int)
wc nLines nChars=
     do flag <- isEOF
        if flag then
           return (nLines, nChars)
        else
           consumeAndCount nLines nChars
```

Exempel: Räkna tecken och rader

```
consumeAndCount :: Int -> Int ->
                                IO (Int, Int)

consumeAndCount nl nc =
     do { c <- getChar;
          if (c == '\n') then
              wc (1 + nl) (1 + nc)
          else wc nl (1+nc)
}
```

Testa programmet

```
WC> wc 0 0
hubba
^D
WC> main
hubba
^D
1 6
WC>
```

I terminalen:

```
$ ghc -o minwc minwc.hs
$ ./minwc
hubba
bubba
^D
2 12
$
```

Lura Haskell... med unsafePerformIO

- Ett trick för att komma runt monadreglerna: `unsafePerformIO :: IO a -> a`
- Använd inte för F4 eller på tenta...
- Peyton-Jones:
  - Riktigt obekväm I/O, ”Once-per-run I/O”
  - Debugging:

```
visa :: String -> a -> a
visa s x =
    unsafePerformIO (putStrLns s >>
                                  return x)
```

”Haskell i verkligheten”

- `Data.ByteString` för strängar: 1 byte/bokstav istället för ca 12 byte/bokstav
- `Data.Map` för associativa listor. Operationer är O(log n).
- `Data.Array`, en oföränderlig array
- Vanlig (?) `Array` som monad: `Data.Array.ST`