Haskell

Kjetil Ørbekk

Programvareverkstedet,
19. mars 2009

Imperativ programmering

- Tilstand
- Operasjoner

```
function uppercase(list) {
    x = 1

    while (x < length(list)) {
        upperCase(list[x])
        x = x + 1
    }
}
```

Funksjonell programmering

- Verdier
- Funksjoner
- Transformasjon

```
upperCase list = map toUpper list
```

Haskell

- Haskell 98
- Et **rent funksjonelt** programmeringsspråk
- **Statisk** og **implisitt** typesystem
- **Lat** evaluering

Funksjonell programmering

- `f x --` lik for lik x

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

«Sett variablene dine riktige første gangen, så slipper du å endre dem!»

Haskell

Freedom
from
state

Litt syntaks

```
-- Kommentar, eller:
{- Kommentar -}

-- En funksjon:
f x = 2 * x

-- Vi evaluerer den:
> f 10
20

-- Mer avansert:
g x y = x*3 + y^2

> g 2 3
15
```

Layout

```
funksjon x y z = do {
  foobar; blaz;
  boo;

} where {
  foobar = blaff x;
  blaz = boo x;
  boo = z foobar;
}
```

```
funksjon x y z = do
    foobar
    blaz
    boo

  where
    foobar = blaff x
    blaz   = boo x
    boo    = z foobar
```

GHC

http://www.haskell.org/ghc/

aptitude install ghc

Noen funksjoner

```
max x y =
  if x > y
    then x
    else y

> max 4 1
4
```

```
x ~> y = max x y

> 4 ~> 1
4
```

```
min x y
  | x < y     = x
  | otherwise = y
```

```
x <~##! y = min x y

> 4 <~##! 1
1
```

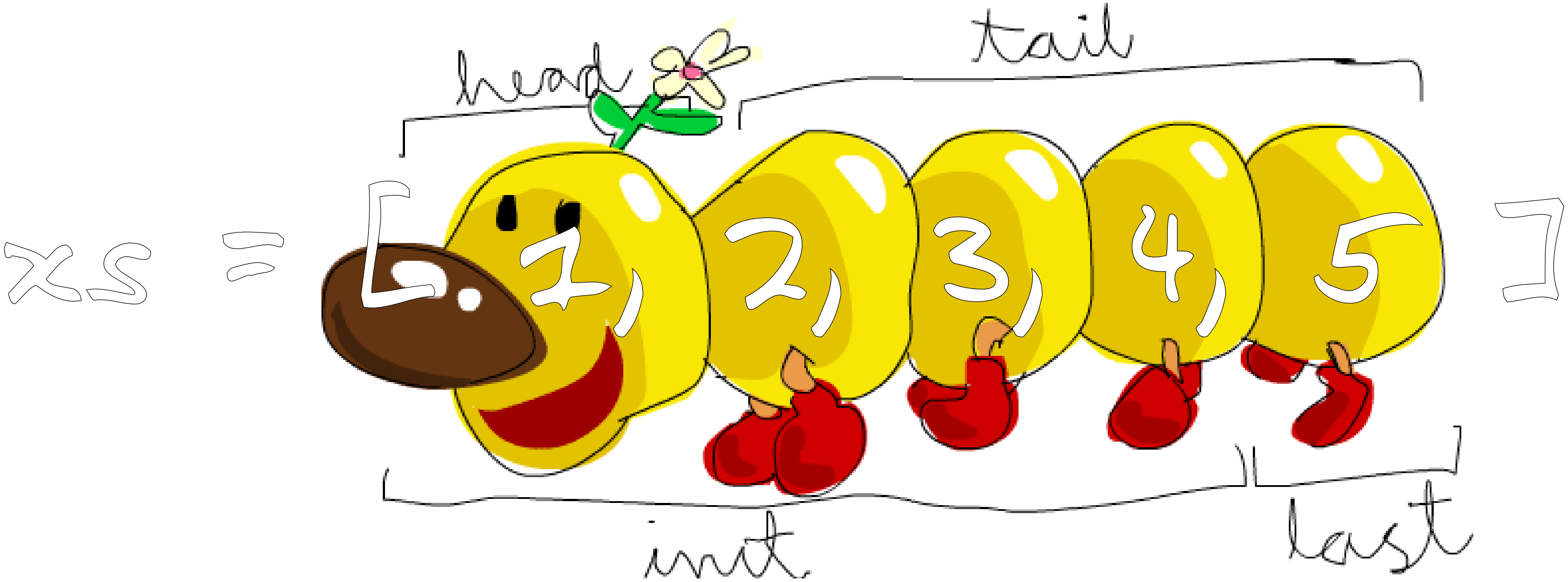
Lister (omg ♥)

- Mest (mis)brukte datastrukturen i FP

```
xs = [1, 2, 3]

ys = [“en liten katt”, “200 ekorn”,
      “fire marsvintonn”]

-- Eller:

zs = [(1, “liten katt”), (200, “ekorn”),
      (4, “marsvintonn”)]
```

Flere lister

```
xs = [1, 2]

-- er sukker for:

xs = 1 : 2 : []

-- les: 1 conset på 2 conset på tom liste
--
-- Altså (i pseudo-haskell):

data [a] =     [ ]
         | a : [a]
```


head
tail
xs = [1, 2, 3, 4, 5]
init
last

Eksempler

```
xs = [1, 2, 3, 4, 5]

> head xs               =>           1
> tail xs               => [2, 3, 4, 5]
```

Lister til hygge og moro

```
-- null: er denne lista tom lr??
null []      == True
null [1,2,3] == False

-- nå kan vi lage hva vi vil:
length xs =
  if null xs
     then 0                    -- tom liste
     else 1 + length (tail xs) -- recurshun!

-- og en operator:
xs !! n =
  if n == 0
     then head xs
     else tail xs !! (n-1)
```

Pattern matching

```
-- Constructors
data Bool = True | False

-- Pattern matching:
not True      = False
not False     = True

True  && True = True
_     && _    = False

True  || _    = True
False || x    = x
```

Listekos v. 2.0

```
-- ...med pattern matching på lister.
--
-- Constructors:
data [a] =     [ ]
         | a : [a]
-- Med variabler for a og [a]:

liste = []
         -- eller
liste = x : xs

-- null: er denne lista tom lr??
null []     = True
null (x:xs) = False
```

Listekos, 1.0 vs. 2.0

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


(x:xs) !! 0 = 0
(x:xs) !! n = xs !! (n-1)
```

```
length xs =
  if null xs
     then 0
     else 1 + length (tail xs)


xs !! n =
  if n == 0
     then head xs
     else tail xs !! (n-1)
```

Funksjoner vs. operatorer

```
--
-- I Haskell: operatorer er _ikke_ spesielle.
--

> mod 5 2 -- kjent som 5 % 2 i andre språk.


> 5 `mod` 2
> (`mod` 2) 5
> (2 `mod`) 5
2

1 + 2
(+) 1 2
(1+)  2
(+2)  1
```

List comprehensions

```
-- Doble alle elementer i en liste:
xs = [1..10]

ys = [ x * 2  |  x <- xs ]

> ys
[2, 4 .. 20]

-- Finne alle oddetall i en liste:

ys = [ y  |  y <- xs,  odd y ]

> ys
[1, 3 .. 10]
```

Hva er dette?

```
funky []     = []
funky (x:xs) = funky a ++ [x] ++ funky b

  where a = [y | y <- xs, y <  x]
        b = [z | z <- xs, z >= x]
```

(til hjelp

```
[1, 2, 3] ++ [4, 5, 6] => [1, 2, 3, 4, 5, 6]
```

)

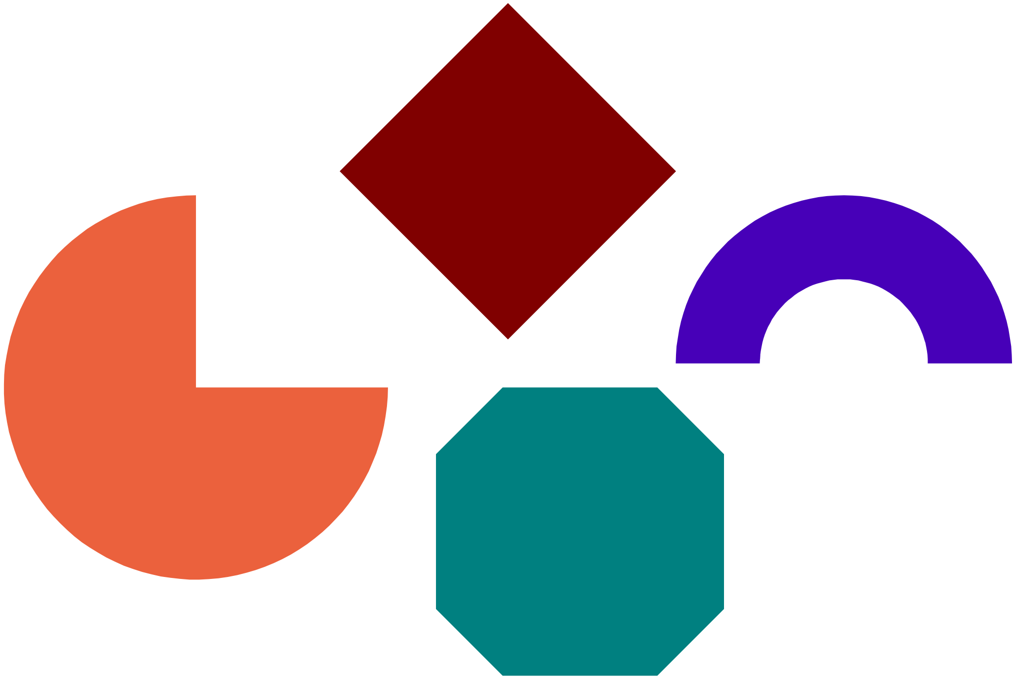
Oppgave

- Skriv en funksjon “drop n” som fjerner de n første elementene av en liste:

  drop 3 [1, 2, 3, 4] => [4]

```
-- Til hjelp: take n
take _ [] = []
take 0 [] = []
take n (x:xs) = x : take (n-1) xs

-- Dette er en god start:
drop _ []     = <???>
drop 0 xs     = <???>
drop n (x:xs) = <???>
```

Typer



Alt har en type

```
x :: Int
x =  1

a :: [Int]
a =  [1,2,3]

b :: [[Int]]
b =  [[1,2], [7,8]]

c :: [(Int, String)]
c =  [(10, “egg”), (5, “datamaskiner”)]
```

Ja, alt!

```
f :: Int -> Int
f x = 2 * x

length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs

drop :: Int -> [a] -> [a]
drop _ []     = []
drop 0 xs     = xs
drop n (x:xs) = drop (n-1) xs
```

‹‹Tips: Bruk :t i ghci for å spørre etter typen til et uttrykk.

Eks: :t drop››

Hva er typen til (==)?

```
-- Hvis vi gir den argumenter:

:t x == y  => Bool

-- Uten argumenter:

:t (==)  => (a -> a -> Bool) ?
```

Typeklasser

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)

data Babl = Abl | Fabl

> Abl == Abl
error: No instance for (Eq Babl).

instance Eq Babl where
  Abl  == Abl  = True
  Fabl == Fabl = True
  _    == _    = True

> Abl == Fabl
False
```

Demo

:t

($)

```
($) :: (a -> b) -> a -> b
f $ x = f x

infixr 0 $

map (+1) map (*2) [1,2,3]
  == (map (+1) map) (*2) [1,2,3]

map (+1) $ map (*2) [1,2,3]
  == map (+1) (map (*2) [1,2,3])
```

Funksjonskomposisjon

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)

(f . g) x = f (g x)

> sqrt (-2)
NaN

> (sqrt . abs) (-2)
1.4142...
```

map

```
f :: Int -> Int
f = x * 2

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

map f :: [Int] -> [Int]

> map f [1..3] :: [Int]
[f 1, f 2, f 3] -- ==
[  2,   4,   6]
```

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

> toLower 'A'
'a'

> map toLower “JG VILHA KAKE”
“jg vilha kake”

> map (`mod` 3) [0..]
[0,1,2,0,1,2,0,1,2,...]
```

λ

```
-- Lambda: funksjoner uten navn
> (λx -> x * 2) 2
4

-- Til map:
map (\x -> x / 2) [1..10]
> [0.5, 1.0, 1.5, .. 10]

-- Men vanligere:
map (/2) [1..10]

> (\x y z -> x^2 + y + z * 2)
```

Filter

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

filter _        []     = []
filter predikat (x:xs) =
  case predikat x of
      True  -> x : filter predikat xs
      False ->     filter predikat xs

-- eller

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

Eksempel

```
-- List comprehension:
xs = [1..10]
ys = [ y * 2 | y <- xs, odd y ]

-- Map og filter:

ys = map (*2) $ filter odd $ xs


qsort :: (Ord a) => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger

  where smaller = filter (< x) xs
        larger  = filter (>=x) xs
```

Fold

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f x []     = x
foldr f k (x:xs) = f x (foldr f k xs)

--              == f x $ foldr f xs, remember?

foldr (+) 0 [1,2,3]                  =
  (+) 1 (foldr (+) 0 [2,3])          =
  (+) 1 ((+) 2 (foldr 0 [3]))        =
  (+) 1 ((+) 2 ((+) 3 (foldr 0 []))) =
  (+) 1 ((+) 2 ((+) 3 0))            =
  (+) 1 ((+) 2 3)                    =
  (+) 1  5                           =
  6
```

Mer folding

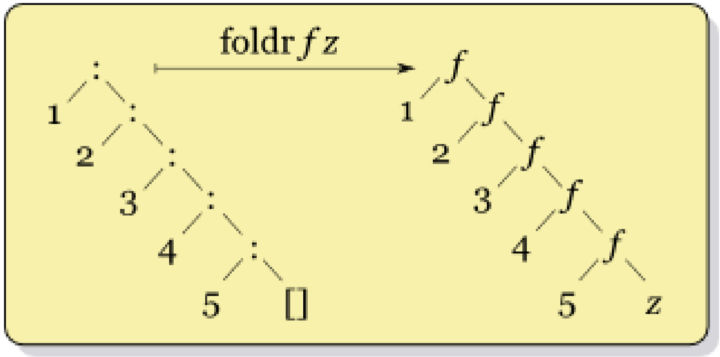
```
sum, product :: (Num a) => [a] -> a
sum xs     = foldr (+) 0 xs
pruduct xs = foldr (*) 1 xs

map :: (a -> b) -> [a] -> [b]
map f xs = foldr (\x xs -> f x : xs) [] xs

concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss

length :: [a] -> Int
length xs = foldr (\x n -> 1 + n) 0 xs
```

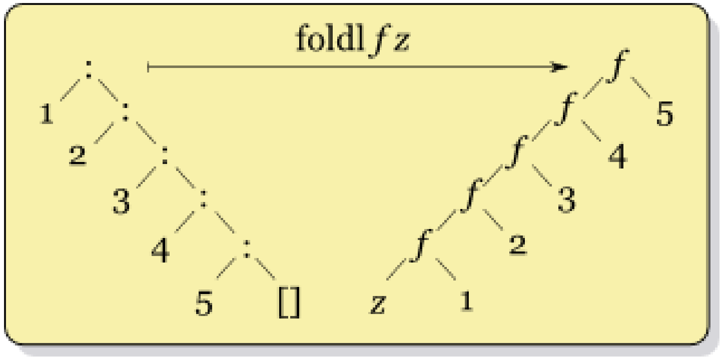
Eksempel: insertion sort

```
-- Hjelpefunksjon

insert :: (Ord a) => a -> [a] -> [a]
insert e []                 = [e]
insert e (x:xs) | x < e     = x : insert e xs
                | otherwise = e : (x:xs)


isort :: (Ord a) => [a] -> [a]
isort xs = foldr insert [] xs
```

foldl

- Tail recursive.
- http://www.haskell.org/haskellwiki/Fold





Oppgaver

- Definer `filter` med foldr.
- `takeWhile` er gitt nedenfor. Kan du skrive den med en foldr og en hjelpefunksjon?

```
takeWhile _ []     = []
takeWhile p (x:xs) | p x       = x : takeWhile p xs
                   | otherwise = []
```

Monads

Monad

```
-- Monad:

data m a = [...]

-- Består av:

-- return x; “pakker inn” x i monaden
return :: a -> m a

-- En funksjon som “binder sammen”
(>>=) :: m a -> (a -> m b) -> m b
```

Maybe

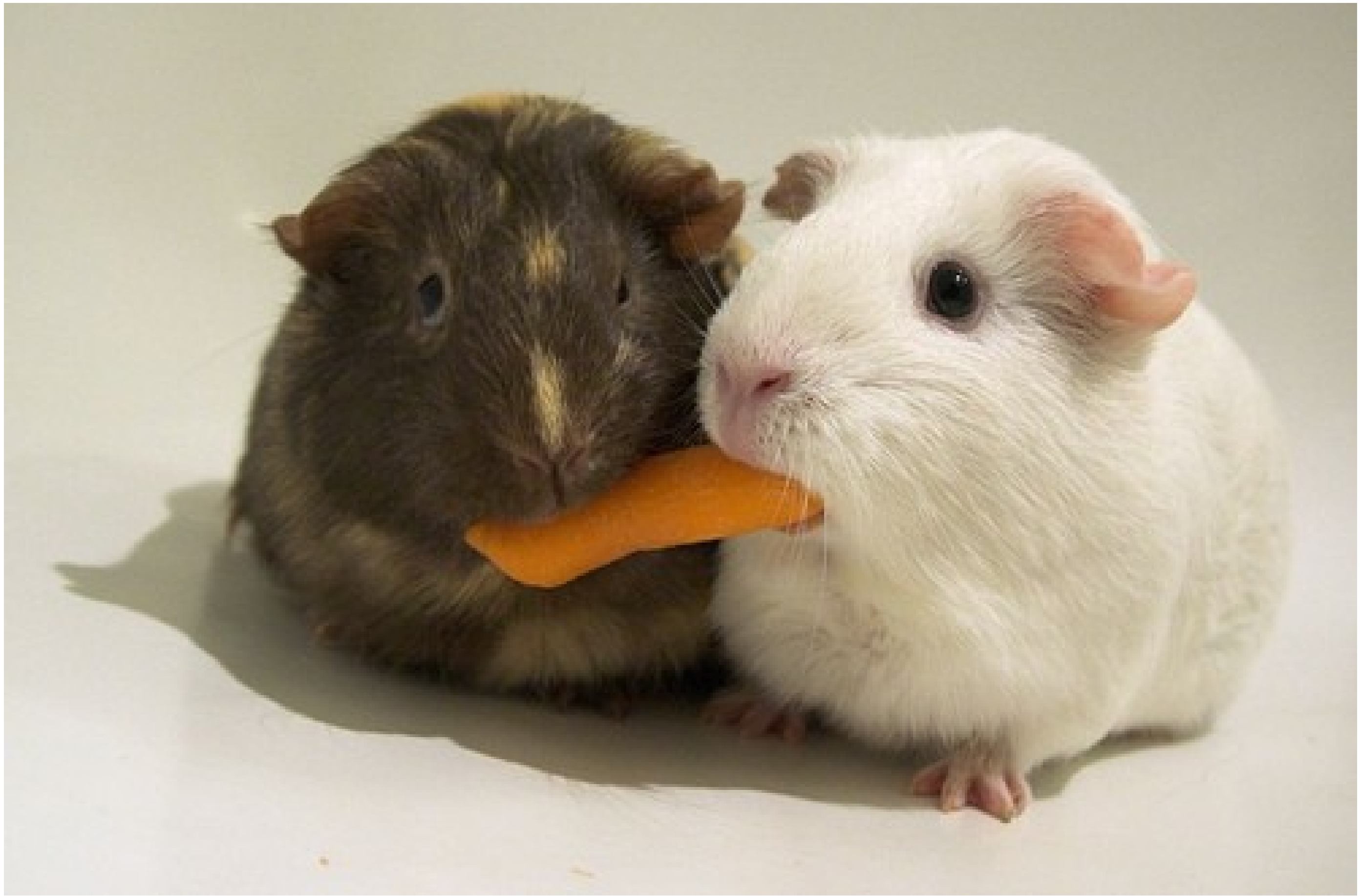
```
data Maybe a = Just a
             | Nothing


data Person = Person {
    mor  :: Maybe Person
  , far  :: Maybe Person
  , navn :: String
  }

mor   :: Person -> Maybe Person
far   :: Person -> Maybe Person
navn  :: String
```

Grandparents

```
mormor :: Person -> Maybe Person
mormor p = case mor p of
             Nothing -> Nothing
             Just m  -> mor m

farmormor :: Person -> Maybe Person
farmormor p = case mor p of
                Nothing -> Nothing
                Just m  -> case mor m of
                             Nothing -> Nothing
                             Just m' -> far m'

-- ...idioti!
```

Maybe Monad

```
data Maybe a = Just a | Nothing
     -- husk “data m a = ...”, m = Maybe

return   :: a -> Maybe a      -- a -> m a
return x = Just x

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
              --  m a -> (a -> m b) -> m b

Nothing >>= _   = Nothing
Just x  >>= f   = f x

-- Vi skriver farmormor på nytt:

farmormor p = p >>= mor >>= mor >>= far


-- Velkommen til Monads!
```



IO, en Monad

```
--
-- IO: Spesielt og innebygd (no shit)
--

data IO a = ...

return :: a -> IO a
return x = ...

(>>=) :: IO a -> (a -> IO b) -> IO b
i >>= f  = ...
```

Hello, World!

```
-- IO er da ikke så vanskelig? :-)

main :: IO ()
main = putStrLn “Hello, World!”


(>>=)    :: IO a -> (a -> IO b) -> IO b

putStrLn :: String -> IO ()
getStrLn :: IO String


getLine >>= putStrLn -- cat?
```

Eksempel: wc

```
-- words “en liten\ntest” => [“en”, “liten”, “test”]

getContents :: IO String          -- leser hele stdin
words       :: String -> [String]
length      :: [a] -> Int
show        :: (Show a) => a -> String
putStrLn    :: String -> IO ()

-- Gitt en string:
main = getContents >>= \content ->
       putStrLn $ show $ length $ words content

-- eller
main = getContents >>=
       putStrLn . show . length . words
```

do

```
main = getContents >>= \content ->
       let nWords = length (words content)
       in  putStrLn $ show nWords

-- vs

main = do content <- getContents
          let nWords = length (words content)
          putStrLn (show nWords)
```

Fra fil

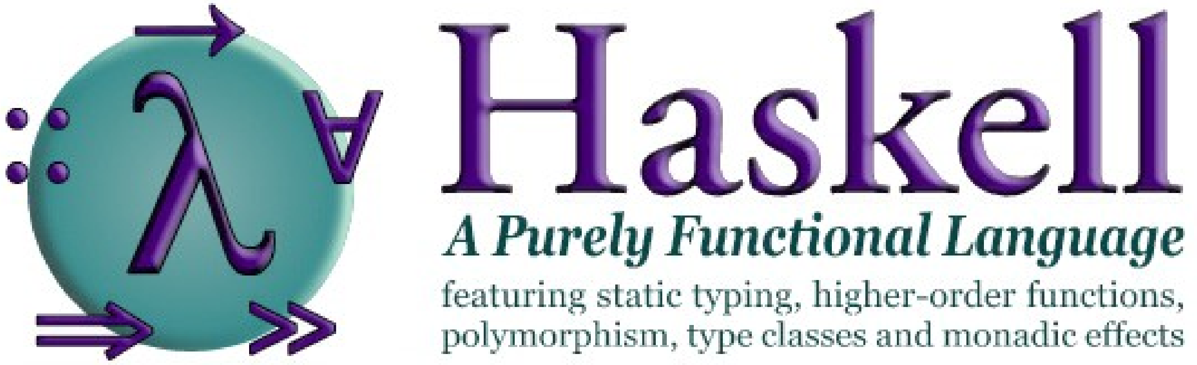
```
import System.IO

wc :: IO String -> IO Int
wc reader = do content <- reader
               let nWords = length (words content)
               return nWords

-- readFile :: String -> IO String
-- getArgs  :: IO [String]

main :: IO ()
main = do
  args <- getArgs
  wordCount <- if null args
                  then wc getContents
                  else wc (readFile $ head args)

  putStrLn (show wordCount)
```

Mer om Monads

- Monads: Mer enn bare IO
- State Monad ♥
- Sjekk “All About Monads”: http://www.haskell.org/all_about_monads/

Mer Haskell

- **Lat evaluering**
- **Currying**
- Parallellitet
- Haskell FFI
- Profiling
- Debugging
- Extensions



A Purely Functional Language

featuring static typing, higher-order functions,
polymorphism, type classes and monadic effects

Ressurser

- Real World Haskell
  http://www.realworldhaskell.org/
- Learn You A Haskell
  http://learnyouahaskell.com/
- #haskell på freenode
- Hoogle – API-søking
  http://www.haskell.org/hoogle/