Intro til Haskell

Plan

- Generelt
- Mye syntaks
- Type classes

Hva er Haskell?

- Rent funksjonelt programmeringsspråk
- Lazy
- Statisk typet med global type inference
- Mye til felles med ML-språk
  - Elm
  - F#
  - SML

Historie

- Laget av en komite for å samle forskning på lazy programmmeringsspråk
  - Fra Miranda
- 1.0 : 1990
  - Eldre enn Java og Python
- Viktige folk:
  - Simon Peyton Jones, John Hughes, Erik Meijer, Phillip Wadler

Syntaks

- Mye likt som Elm
  - Elm-syntaks er ca et subset av Haskell-syntaks, med noen små endringer

```
funksjonsNavn :: a -> Maybe a
funksjonsNavn aVal = Just aVal
```

- en funksjon med navn funksjonsNavn
- øverste linjen er en typedefinisjon, som sier hvilken type det er
  - trengs som regel ikke pga type inference, men er ofte hjelpsomt
- Generisk type `a`
  - Generiske typer har små bokstaver. Konkrete typer store
- `->` sier at det er en funksjon
  - tar inn en `a` og gir tilbake en `Maybe a`
- `aVal` er navnet på argumentet
- `Just aVal` er returverdien

Pattern matching : Funksjon med flere definisjoner

Man kan definere funksjoner en gang per case

```
not :: Bool -> Bool
not True = False
not False = True
```

Man har også case (som i Elm)

```
not :: Bool -> Bool
not b = case b of
    True -> False
    False -> True
```

Guards

Istedenfor chaining av if-else, så har man guards

```
describe :: Int -> String
describe num
    | num > 100 = "Big"
    | num < 0 = "Negative"
    | otherwise = "Normal"

--definert i Prelude
otherwise = True
```

Kan kombineres med pattern matching

```
firstElementAsStringIfOdd [x] | odd x = show x
firstElementAsStringIfOdd _ = ""
```

Currying - by default

Alle funksjoner tar inn et argument om gangen

```
add :: Int -> Int -> Int
add x y = x + y

-- samme som
add :: Int -> (Int -> Int)
add = \x -> (\y -> x + y)

> add 2 3
5
> (add 2) 3
5

f = add 2
> f 5
7
```

Funksjonsapplikasjon

Binder til venstre

```
f g h x
-- blir parset som
((f g) h) x

> head [negate,abs] 5
-5

> (head [negate,abs]) 5
> negate 5
> -5
```

Dollar : $

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

Brukes for å slippe paranteser

```
> map show (reverse (tail [1,2,3]))
["3","2"]

-- sammme som

> map show $ reverse $ tail [1,2,3]
["3","2"]
```

Compose : .

Det er vanlig i Haskell å bygge opp funksjoner ved å compose andre

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)

add1 x = x+1

times10 x = x*10

> (times10 . add1) 5
60

> times10 (add1 5)
60

mapTailEvens = map (*2) . filter even . tail
-- vs
mapTailEvens x = map (*2) (filter even (tail x))
```

Sections

```
> :t (:)
(:) :: a -> [a] -> [a]
> (:) 1 [2]
[1,2]
> :t (:) 1
(:) 1 :: Num a => [a] -> [a]

> :t (1:)
(1:) :: Num a => [a] -> [a]
> :t (1:) [2]
(1:) [2] :: Num a => [a]
> (1:) [2]
[1,2]

> (:[2]) 1
[1,2]
> (\x -> x:[2]) 1
[1,2]

> map (:[0]) [1,2,3,4]
[[1,0],[2,0],[3,0],[4,0]]
```

Bruke vanlige funksjoner infix

```
> elem 1 [2,3,1,4]
True
> 1 `elem` [2,3,1,4]
True
```

Placeholders / errors

```
undefined :: a
error :: String -> a

f :: Int -> b
f = undefined

oneToString 1 = "One"
oneToString _ = error "not one"

> error "hei"
*** Exception: hei
CallStack (from HasCallStack):
  error, called at <interactive>:6:1 in interactive:Ghci3
```

Feilmeldinger

```
> 1 + [2]

<interactive>:12:1: error:
    • Non type-variable argument in the constraint: Num [a]
      (Use FlexibleContexts to permit this)
    • When checking the inferred type
        it :: forall a. (Num a, Num [a]) => [a]

> map reverse "hei"

<interactive>:14:13: error:
    • Couldn't match type ‘Char’ with ‘[a]’
      Expected type: [[a]]
        Actual type: [Char]
    • In the second argument of ‘map’, namely ‘"hei"’
      In the expression: map reverse "hei"
      In an equation for ‘it’: it = map reverse "hei"
    • Relevant bindings include
        it :: [[a]] (bound at <interactive>:14:1)
```

Typed holes, where og let

Demo

Imports

```
-- Importer alt fra en modul
> import Data.List
> union [1,2,3] [3,4,5]
[1,2,3,4,5]
-- Importer spesifikke funksjoner eller typer
> import Data.Maybe (maybe)
> maybe 0 (+1) (Just 5)
6
-- Importer med namespace som modulnavnet
> import qualified Data.Either
> Data.Either.isLeft (Left 2)
True
-- Importer med eget namespace
> import qualified Data.Set as Set
> Set.fromList [1,2,3]
fromList [1,2,3]
-- Importer alt utenom spesifikke ting
> import Data.List hiding (union)
```

Lister

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

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


xs = [1,2,3]
xs' = 1:(2:(3:[]))
xs'' = 1:2:3:[]
```

Lazy

```
doSomething bool = if bool then trueCase else falseCalse
    where
        trueCase = print True
        falseCase = print False

> take 2 [1,2,error "STOP", 3]
[1,2]

> map (\x -> div x 0) [1,2,0]
[*** Exception: divide by zero

> length $ map (\x -> div x 0) [1,2,0]
3

-- Lister er da "streams"
ones = 1 : ones

> take 5 ones
[1,1,1,1,1]
```

Felles oppgaver -- Oppgaver.md

Type classes

- Haskell sin måte å ha forskjellige implementasjoner for forskjellige typer
  - Ad-hoc polymorfisme
- Ikke som en Class i feks Java
- Ganske nærme et interface
- Noe som ikke finnes i Elm

Eksempel på en Type Class - Show

- Show er en type class for alle typer som har verdier kan gjøres om til en String
  - En slags serialisering
  - Litt som toString, bare ikke for alle typer (feks funksjoner)

```
class Show a where
  show :: a -> String

-- Bool og Int implementerer Show
> show True
"True"
> show 1
"1"
```

Type classes - typesignaturer

- At en funksjon krever at en type implementerer en type class har en egen syntax

```
show :: Show a => a -> String
```

- Alt før => viser constraints
- her : Hvis en type implementer Show, kan man bruke show-funksjonen for å gjøre den om en verdi av den typen til en String

Bruke type class-funksjoner i egne funksjoner

```
showBoth :: (Show a,Show b) => a -> b -> String
showBoth a b = show a ++ show b

> showBoth 1 True
"1True"
```

Type classes - Instances

Hvordan implementere type classes

```haskell
--Implementere Show For Bool
instance Show Bool where
    show True = "True"
    show False = "False"

--For Maybe a
--Hvorfor trenger vi Show a her?
instance Show a => Show (Maybe a) where
    show Nothing = "Nothing"
    show (Just a) = "Just " ++ show a
```

Eksempel 2 : Eq - for å sjekke likhet

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

instance Eq Bool where
    True == True = True
    False == False = True
    _ == _ = False

instance Eq a => Eq [a] where
    [] == [] = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _ == _ = False
```

Superklasser

- Type classes kan ha superklasser
- Det betyr at man må ha en instance av superklassen for å kunne lage en instance av subklassen
- Så alle typer som er Ord er også Eq

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

--Det funker pga Bool har Eq instance
instance Ord Bool where
    True <= False = False
    _ <= _ = True

--Dette funker siden Ord a impliserer Eq a, siden Eq er superklasse
ordEq :: Ord a => a -> a -> Bool
ordEq a b = a == b
```