1 Forelesning 3

1.0.1 Plan for forelesningen

Vi begynner på lista fra forrige gang:

- Tuppler
- Maybe
- Lister
- Either
- Map

Vi kommer definitivt ikke lenger enn til lister idag.

1.1 Tuppler: Eksempeltyper

Tuppeltyper skrives med paranteser og komma:

- `(Integer, String)` er typen av alle par av heltall og strenger.
- `(Double,Double,Double)` er typen av alle 3D koordinater, eller vektorer, med dobbel presisjons-flyttall.

⋮

osv

1.1.1 Sammenligning med lister

- Tuppler: Fiksert antall posisjoner for data.
- Hver posisjon har en type.
- Lister: Variabelt antall posisjoner for data (lengde)
- Lister: Alle elementer i listen har samme type.

1.1.2 Kanonsike verdier

De kanoniske verdiene i tuppeltypene er de på formen:

1.1.3 Funksjoner definert med møn

> **Eksempel.** La oss skrive en funksjon som normaliserer en vektor. (Det vil si å skalere den slik at den har lengde 1.)

```haskell
module NormalizeVector where Set module name to Main

normalise :: (Double, Double, Double) → (Double, Double, Double)
normalise (x, y, z) = (x / norm, y / norm, z / norm)
  where
    norm :: Double
    norm = sqrt (x ^ 2 + y ^ 2 + z ^ 2)

main :: IO ()
main = do
  putStrLn "Enter a three dimensional vector: "
  input ← getLine
  let result = normalise (read input)
  putStrLn $ "Normalised vector: " ++ show result
```

◇

Generelt definerer vi mønster matching hvordan funksjonen opererer på en kanonisk verdi av typen.

$(1, 2, 3)$ er en kanonisk verdi av typen `(Double, Double, Double)`

1.1.4 Projeksjoner

For par (tupler med to posisjoner), har vi to funksjoner for å hente ut data fra et par:

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

1.2 Currying

En funksjon som tar to verdier har vi sett kan ha type på formen:

```
A -> B -> C
```

Eksempel fra boken

```
add ::  (Integer, Integer) -> Integer
add (x, y) = x + y
add' ::  Integer -> Integer -> Integer
add' x y = x + y
```

1.2.1 Fordeler med begge

Fordelen med currying er partiell applikasjon

```
add' 3 ::  Integer -> Integer
```

Fordelen med uncurrying er at det er lett å mappe funksjonen inn i en struktur:

$$> mapadd3[(1,2),(2,1),(0,3)]$$

1.2.2 Funksjoner for å gå frem og tilbake

Gjøre en funksjon curried:

```
curry ::  ((a,b) -> c) -> a -> b -> c
```

Gjøre en funksjon uncurried:

```
uncurry ::  (a -> b -> c) -> (a,b) -> c
```

1.2.3 Hva er konvensjonen i Haskell

Standard i Haskell er å definere funksjonen som ferdig curry-et, og heller bruke uncurry eksplisitt der det er behov.

```
map (uncurry (+)) [(x,y) | x <- [0..3], y <- [x..3]]
```

1.3 Maybe

Vi lager *Maybe-typer* ved å skrive `Maybe` foran typen

- `Maybe Integer` er typen av kanskje heltall

Eksempel. Eksempler på elementer

```
Just 3 ::  Maybe Integer
Nothing ::  Maybe Integer

readMaybe ::  String -> Maybe Integer
```

Merk. `readMaybe` må importeres fra `Text.Read`

◇

1.3.1 Konstruktører

Maybe typene har to konstruktører

- `Nothing ::  Maybe a`
- `Just ::  a -> Maybe a`

1.3.2 Nyttige funksjoner

La oss se på noen nyttige funksjoner

- `maybe :: b -> (a -> b) -> Maybe a -> b`
- `fromMaybe :: a -> Maybe a -> a`
- `fmap :: (a -> b) -> Maybe a -> Maybe b`

Man kan også få ut boolske verdier fra en Maybe-verdi

1.4 Lister

- `[Integer]` er en liste av heltall
- `[Char] = String` er en streng / liste med characters
- `[[Integer]]` er en liste med lister av heltall
- `[Double -> Double]` er en liste med funksjoner på flyttal

1.4.1 Konstruktører

Lister har to konstruktører

- `[] :: [a]` - Den tomme listen
- `(:) :: a -> [a] -> [a]` - Legger et element foran i listen

Dermed er kanoniske element i en listetype `[A]` de som er på formen

- `[]` eller
- `a : as` hvor `a :: A` og `as :: [A]`

Syntaktisk sukker `[1,2,3,]` istedetfor `1:2:3:[]`

1.4.2 Mønster

Vi kan definere funksjoner på lister ved hjelp av mønster som matcher `(:)` og `[]`:

```
safeHead :: [a] -> Maybe a
safeHead
```

1.4.3 Rekursjon

Senere skal vi se hvordan vi kan definere alle mulige funksjoner på lister vha. rekusjon

```
duplicate :: [a] -> [a]
dublicate [] = []
duplicate (a : as) = a : a : duplicate as
```