# 1 Forelesning 0

## 1.1 Praktiske Detaljer

- Ukesoppgaver er obligatoriske
  - Automatisk rettet
  - Poeng for oppgavene, 24 poeng totalt
  - Må ha 18 poeng for å ta eksamen
  - 3% Bonus for å ta eksamen

Ukesoppgaver: Fredag hver uke, første frist er 2 september

MittUIB: Modules, videoer og oppgaver

**OBS:** Følg meg på kunngjøringer på MittUIB. Viktig å se diverse videoer før forelesning.

## 1.2 Hva gjør Haskell spesielt som programmeringspråk?

- Funksjonellt programmeringsspråk
- Algebraisk data typer
- Typeutledning og polymorfi
- Verdier er uforanderlige → Memoisering
- Lat og strikt evaluering av verdier

**Haskell Standarden** - Definerer haskell som språk, nyeste versjon fra 2010. **GHC** - Den dominerende haskellkompilatoren, har gjevnlig nye utgivelser, og tilbyr mange utvidelser av språket.

# 2 Forelesning 1

## 2.1 Plan for forelesningen

- Gjennomgang av forventinger
- Online ressurser
- Eksempler på programmer i Haskell
- Funksjoner i Haskell
- Strukturen til et Haskellprogram

## 2.2 Online Haskellressurser

- *Learn You a Haskell*: learnyouahaskell.com
- *Haskell wikibok*: en.m.wikibooks.orgs/wiki/haskell
- *Hoogle*: hoogle.haskell.ord
- *Mer*: haskell.ord/documentation

## 2.3 Eksempler på programmer laget i Haskell

- Pandoc
- Xmonad
- Darcs
- GF - Grammatical Framework
- GitHub's semantic tool

Og andre diverse selskaper som Standard Chartered og Klarna.

## 2.4 Funksjoner

Hva er en funksjon? Vi bruker en funksjon ved å få en verdi ved å gi den et argument.

I matematikken brukes $f(x)$ for å bruke en funksjon $f$ på en verdi $x$. Hvis funksjonen tar imot flere argumenter skriver man $f(x, y, z)$ for å gi dem.

I Haskell droppes parantesene, og man skriver bare `f x`, og dersom det er flere argumenter skrives det `f x y z`. For å sette sammen funksjoner, må vi likevel bruke paranteser: `f (g x)`. Dersom vi hadde skrevet `f g x` ville vi gitt to argumenter til funksjonen.

## 2.5 Haskellprogrammer

Filnavn i haskell slutter på `.hs` - ellers er hver fil ofte en *modul*, hvor filnavn ofte er det samme som modulnavn. Modulnavn kommer øverst i filen, og er på formen `module moduleName where`. Verdien `main` er en spesiell verdi som har typen `IO ()`. For å lage en kjørbar fil må main verdien ligge i modulen main.

### 2.5.1 Presidensregler

I Haskell binder funksjonene sterkes, det vil si at koden under tolkes på følgende måte.

# 3 Forelesning 2

## 3.1 Tall i Haskell

## 3.2 Funksjonsdefinisjoner

Matematisk skriver vi $f : A \to B$ for å si at funksjonen $f$ tar input av typen $A$ og returnerer $B$. Vanligvis skrives funksjoner som

$$f(x) = x^2 + 4$$

Her er det at implisitt at det er en funksjon av typen

$$f : \mathbb{R} \to \mathbb{R}$$

I haskell deklareres funksjoner på følgende måte

- Type deklarasjon: `myFun :: A -> B`

## 3.3 Polymorfi

Ordet *polymorfi* kommer fra de greske røttene polus og morphe, i.e. *mange* og *form*. Så kort oversatt til norsk: *flerformet*.

I haskell er det to typer polymorfi:

- Parametrisk polymorfi (aka. typevariabler)
- Ad-Hoc polymorfi (aka. typeklasser)

Idag skal vi se på den første av disse.

En parametrisk, polymorf funksjon i Haskell er en funksjon som bruker typevariabler til å defineres for alle typer samtidig.
OBS:

- Typevariables begynner alltid med liten forbokstav.
- Konkrete typer (ikke variabler) har stor forbokstav.
- Noen innebygde konkrete typer har speisell syntaks: lister, tupler osv.

**Eksempel.** Et veldig enkelt eksempel

```
id ::  t -> t
id a = a
```

Denne funksjonen tar et element inn og spytter samme element ut. Ofte bruker vi (forvirrende nok) samme bokstav for typen og elementene i typen

```
const ::  a -> b -> a
const a b = a
```

◇

**Eksempel.** Klarer vi å finne en type for funksjonen fra forrige forelesning?

```
h ::  ?
h z x = z (z x)
```

**Svar:**

```
h ::  (a -> a) -> a -> a
h z x = z (z x)
```

I lambdakalkyle er dette representasjonen av tallet 2. ◇

**Eksempel.** Selv de enkleste ting kan være polymorfe.

```
[] ::  [a]
```

Den tomme listen er en liste av alle typer. ◇

**Oppgave.** Lag en funksjon med typen

1. `f ::  a -> a -> a`
2. `g ::  a -> [[a]]`
3. `t ::  (a -> b) -> (b -> c) -> (a -> c)`
4. `s ::  (a -> b -> c) -> (a -> b) -> a -> c`

**Svar:**

1. `f x y = x`
2. `g a = [[a,a], [a,a]]`
3. `t f g a = g (f a)`
4. `s f g a = f a (g a)`

### 3.4 Uforanderlige verdier

En verdi er *uforanderlige* dersom den ikke kan endres etter at den er opprettet. I Haskell er *alle* verdier uforanderlige. Forandring uttrykkes istedet ved hjelp av funksjoner.

I begynnelsen kan det være litt forvirrende fordi noe som heter "aliasing", gjør at det ser ut som om verdier kan forandre seg i GHCi.

### 3.5 Neste tema: Vanlige typer

Vi skal gå gjennom følgende typer:

- Tupler
- Maybe
- Lister
- Either
- Map

## 4 Forelesning 3

#### 4.0.1 Plan for forelesningen

Vi begynner på lista fra forrige gang:

- Tuppler
- Maybe
- Lister
- Either
- Map

Vi kommer definitivt ikke lenger enn til lister idag.

## 4.1 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

### 4.1.1 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.

### 4.1.2 Kanonsike verdier

De kanoniske verdiene i tuppeltypene er de på formen:

### 4.1.3 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)`

#### 4.1.4 Projeksjoner

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

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

### 4.2 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
```

#### 4.2.1 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:

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

### 4.2.2 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
```

### 4.2.3 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]]
```

## 4.3 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`

◇

### 4.3.1 Konstruktører

Maybe typene har to konstruktører

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

### 4.3.2 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

## 4.4 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

### 4.4.1 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:[]`

### 4.4.2 Mønster

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

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

### 4.4.3 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
```

# 5 Forelesning 4

## 5.1 Plan for forelesning

Vi fortsetter fra forrige gang:

- Eksempler på listefunksjoner
- Assosiasjonslister
- Map
- Listekomprehensjon
- (Bonus: Isomofi av lister)

## 5.2 Rush

Med unntak av 1979, utga rockebandet "Rush" ett album hvert å i perioden 1976-1982.

## 5.3 Nyttige listefunksjoner

```
elem ::  (Eq a) => a -> [a] -> Bool
words ::  String -> [String]
unwords ::  [String] -> String
```

**Eksempel.** Skriv en funksjon som fjerner alle ord som inneholder bokstaven "e".

◇

```
removee ::  String -> String
removee text = unwords (filter (notElem 'e') (words text))
```

## 5.4 Zip

Zip er en funksjon med typing

```
zip ::  [a] -> [b] -> [(a,b)]
```

## 5.5 Currying: Forklaring

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

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

Dette kalles "currying"

### 5.6 Lookup

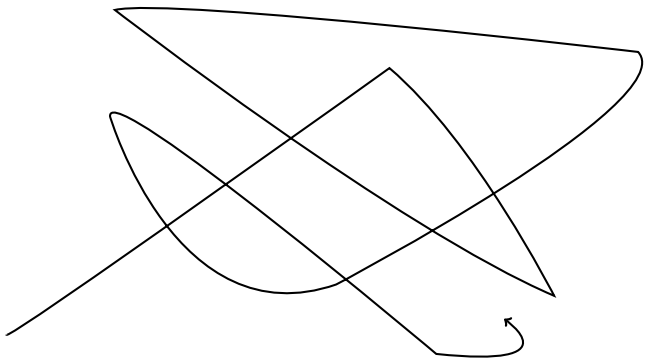
Den nyttigste funksjonen for assisiasjonslister:

```
lookup ::  a -> [(a,b)] -> Maybe b
```

### 5.7 Isomorfi av typer: Definisjon

Vi sier at to typer $A$ og $B$ er isomorfe dersom vi kan funne funksjoner

```
f ::  A -> B og
g ::  B -> A
```



## 6 Forelesning 5

### 6.1 Oblig 1

Obligene kommer til å handle om å lage ett program for å gjøre beregninger tilsvarende de beregningene en skrittteller ville brukt for å telle antall skritt vi går.

Kanskje det kan være lurt og interpolere?

### 6.2 Plan for forelesning

- Mer listekomprehensjon
- Lambda uttrykk ($\lambda$)

## 6.3 Pythagoreiske trippler

Pythagoras teorem sier at i en rettvinklet trekant med sidelengder $a$ og $b$ (kateter)) og $c$ (hyptenus) har forholdet $a^2 + b^2 = c^2$

Det er sjeldent at alle disse tre blir heltall, for eksempel, dersom $a = b = 1$ så er $c = \sqrt{2}$. Men det hender, for eksempel er $3^2 + 4^2 = 25 = 5^2$. Pythagoreiske trippler er definert som

$$\left\{(a, b, c) \in \mathbb{R}^3, a^2 + b^2 = c^2\right\}$$

## 6.4 $\lambda$-utrrykk

### 6.4.1 Historisk Opphav

$\lambda$-notasjon ble introdusert av Alonzo Chruch på 30-tallet, og først adoptert i et faktisk programmeringsspråk i McCarthus LISP på 50-tallet.

I Haskell ser $\lambda$-uttrykk ut som:

```
\x y z -> UTTRYKK MED x y og z I SEG
```

Dette lager en ny, anonym funksjon som tar `x y z` som variable.

**Eksempel.**

```
\x -> [x]
```

Dette uttrykket står for funksjonen som tar et element og lager en liste med kun ett element. ◇

- Et $\lambda$-uttrykk står for en funksjon og kan brukes alle steder hvor man ville hatt en funksjon.
- Funksjonen kan ta ett eller flere argumenter.
- Etter `->` kommer uttrykket som definerer funksjonen
- Et $\lambda$-uttrykk har ikke noe navn (med mindre du gir det et)

### 6.4.2 Hva er poenget?

En $\lambda$-funksjon kan brukes når vi egentlig hadde brukt en hjelpefunksjon, men vi ikke vil definere en ny global funksjon.

### 6.4.3 Når skal jeg bruke en $\lambda$-funksjon

Bruk et $\lambda$-uttrykk når:

- Du kun behøver funksjonen akkurat der du er.
- Funksjonen bruker variabler fra konteksten du er i
- Når uttrykket er lite nok til å bli leselig.
- Når det ikke finnes en enekel måte å skrive uttrykket som en kombinasjon av andre funksjoner

# 7 Forelesning 7

## 7.1 Plan for forelesningen

- Let-bindinger
- Introduksjon til rekursjon

### 7.1.1 Let-bindinger

Vi kan innføre midlertidige verdier i programmene våre ved å bruke nøkkelordet `let` og `in`:

```
solveQuadratic a b c
    = let discriminant = b*b - 4*a*c
        in ((-b + sqrt discriminant)/(2*a)
            , (-b - sqrt discriminant)/(2*a))
```

### 7.1.2 Mønster matching

Hva gjør denne funksjonen?

```
ordpair :: (Ord a) => a -> a -> (a,a)
ordpair x y = if x <= y then (x, y) else (y, x)
```

Den plasserer det minste elementet i input i første index i tuplen som funksjonen returnerer.

Hva gjør funksjonen `diff`

```
diff x y = let (a, b) = ordpair x y
    in (b - a)
```

### 7.1.3 Flere let features

Let uttrykk kan brukes

- til å definerer funksjoner
- i *do*-notasjon

## 7.2 Rekursjon

Fra engelsk: "recursion" fra

- "recur" på norsk "å skje igjen" eller "gjentatt"

(egentlig fra latin "recurro", som betyr "å løpe tilbake"), men hva er det som "gjentas" eller skjer igjen?

> **Definisjon 1.** En funksjon er rekursiv dersom den kaller seg selv igjen (muligens med nye parametere).

**Eksempel.** Her er en av ukesoppgavene fra uke 1, skrevet rekursivt.

```
triangle :: Integer -> Integer
triangle n = if n == 0
    then 0
    else n + (triangle (n - 1))
```

◇

### 7.2.1 Oppsett

Strukturen i en typisk rekursiv funksjon

- Grunntilfelle(r)
- Rekursive tilfeller

**Eksempel.** Vi kan la mønstrene i argumentet bestemme grunntilfellet og rekursive tilfeller

```
sum' :: [Integer] -> Integer
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

◇

**Eksempel.** Rekursive funskjoner er relativt enkle å bevise ting om

```
sum' [a,b,c] = a + b + c
```

◇

**Bevis.** Vi har at

```
sum' [a,b,c]
= sum (a:b:c:[])
= a + sum'(b:c:[])
= a + b + sum'(c:[])
= a + b + c + sum' []
= a + b + c + 0
= a + b + c
```

□

## 7.3 Forskjellige typer rekursjon

- Rekursjon på heltall
- Rekursjon på lister (og senere datatyper)

- Produktiv rekursjon
- Generell rekursjon

### 7.3.1 Rekursjon på heltall

```
countDown :: Integer -> [Integer]
countDown 0 = [0]
countDown x = x:(counDown (x - 1))
```

- Oftests er 0 eller 1 grunntilfellet.
- De rekursive kallene kaller funksjonen med et midnre parameter.

OBS: Dette gir ofte partielle funksjoner.

- Hva skjer med funksjonen over hvis man gir negativ input?
  - Jo den vil kalle på seg selv i all evighet ettersom den aldri når grunntilfellet.
- Hvordan burde vi ordne det slik at vi ikke får problemer med negative tall?
  - Enten ved å sjekke om tallet er negativt, og dersom det er det, legger vi til 1 istedenfor og trekke fra, eller så kan vi kalle funksjonen på absoluttverdien av $x$.

## 7.4 Primitiv vs generell rekursjon (Ackermann)

Men det finnes veldig kompliserte ting man kan gj;re hvis man vil:

```
ackermann :: Integer -> Integer -> Integer
ackermann 0 n = n + 1
ackermann m 0 = ackermann (m - 1) 1
ackermann m n = ackermann (m - 1) (ackermann m (n - 1))
```

Denne funksjonen vokser veldig fort! Men i teorien kommer den alltid frem til et svar.

## 7.5 Rekursjon på lister og andre datastrukturer

Vi har allerede sett et eksempel, la oss ta noen til

**Eksempel.** For å reversere en liste rekursivt

```
reverse :: [a] -> [a]
reverse [] = []
reverse (a : as) = reverse as ++ [a]
```

- Gjør mønster matching på alle konstruktørene.
- Det rekursive kallet gjøres direkte på variablene introdusert av mønsteret.

◇

**Eksempel.** For å definere zip

◇

# 8 Forelesning 7

## 8.1 Dypere mønstermatching

**Eksempel.** Skriv en funksjon som sjekker om en typer er sortert.

1. Hva er typen til en slik funksjon?
2. Hva må funksjonen gjøre for å sjekke at en liste er sortert?
3. Hvis vi vil bruke rekursjon, hvilke mønster kan vi bruke?

```
sorted :: (ord a) => [a] -> Bool
```

◇

**Eksempel.** Vi kan også matche hvert enkelt element i listen, basert på typen

```
filterEmpty:: [[a]] -> [[a]]
filterEmpty [] = []
filterEmpty ([]:xs) = xs
filterEmpty ([]:xs) = xs
filterEmpty (x:xs)
```

◇

## 8.2 Gjensidig rekursjon

To funksjoner kan v1're definert i termer av hverandre:

```
odds :: [a] -> [a]
odds [] = []
odds (x:xs) = evens xs

evens :: [a] -> [a]
evens [] = []
evens (x:xs) = x : odds xs
```

## 8.3 Egendefinerte datatyper

I haskell kan man innføre nye datatyper ved hjelp av nøkkelordet data:

```
data CelestialObject = Star String Integer
```

```
                    | Planet String String
                    | Moon String String
```

Deklarasjonen består av

- Navn på datatypen: `CelestialObject`
- Liste med konstruktører: `Star, Planet, Moon`
- Argumenter til konstruktøren (data som lagres i elementene)

Når en datatyper er definert kan vi lage elementer i den:

```
solarSystem :: [CelestialObject]
solarSystem = [Star "The Sun" 4600000000,
               Planet "Mercury" "The Sun",
               Planet "Venus" "The Sun",
               Planet "Earth" "The Sun", Moon "The Moon" "Earth"
               Planet "Mars" "The Sun", Moon "Phobos" "Mars"
                                      , Moon "Deimos" "Mars"]
```

Vi kan også definere funksjoner ved hjelp av mønster

```
displayInfo :: CelestialObject -> String
displayInfo (Star name age)
= "The star " ++ name ++ " is " ++ age ++ " years old."
```

# 9 Forelesning 8

## 9.1 Plan for forelesningen

- Fortsette med egendefinerte datatyper
- Bruke datatyper til å modellere et domene
- Trær

## 9.2 Modellering

Hvordan modellerer vi domenet vårt i Haskell?

- Vi lager nye datatyper som beskriver ting i domenet vårt
- Vi lager funksjoner som lar oss manipulere tingene i domenet vårt

Hvordan vet vi at programmet vårt er korrekt mtp domenet?

- Sunnhet: Alle velltypede elementer i programmet vårt er gyldie ting i domenet vårt
- Kompletthet: Alt vi vil modellere er typede elementer i programmet vårt.

Det er ikke enten eller, men heller en flytende overgang

### 9.2.1 Invarianter

Ofte må vi legge ekstra betingelser på elementene våre for å utelukke ugyldige elementer, verdier og tilstander. Disse egenskapene kalles *invarianter*, fordi de bevares (varierer ikke) av operasjonene i programmet vårt.

**Eksempel.** I Obligen modellerer vi:

- Signaler: `[Double]`
- Filtere på signaler: `[Double] -> Double`

◇

# 10 Forelesning 9

## 10.1 Typeklasser

Typeklassene ligner på "interface" i Java. Typeklasser er mer fleksible:

- Et Java interface må defineres før klassen.
- En typeklasse kan defineres når som helst.

**Idea.** Noen funknsjoner fungerer for alle typer (polymorfe), men noen fungerer bare på typer med en viss struktur. For eksempel likhet, rodning, eller + og −.

En typeklasse bestemmer typen til en liste med funksjoner. En hver type som kører til typeklassen skriver in egen implementasjon.

**Eksempel.** Klassen `Eq` er den enkleste innebygde klassen:

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

For eksempel Bool:

```
data Bool = True | False

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

For eksempel for lister:

```
instance (Eq a) => Eq [a] where
[] == [] = True
(a:as) == (b:bs) = (a == b) && (as == bs)
_ == _ = False
```

◇

**Eksempel.** Eksempel på rotering av lister

```
class Rotateable a where
    rotateLeft :: a -> a
    rotateRight :: a -> a

instance Rotateable [b] where
    rotateLeft [] = []
    rotateLeft (a:as) = as ++ [a]
    rotateLeft [] = []
    rotateLeft as = last as : init as
```

◇

## 10.2 Binære søkretrær

### 10.2.1 Antagelse

Vi antar at vi jobber med en type som implementerer Ord typen. (F. eks heltall, eller strenger)

### 10.2.2 Basis ide

Binære søketrær er basert på samme ide som QuickSort: Hvis vi har et element $x$ kan vi dele opp en hver mengde i tre deler:

- De som er mindre enn $x$ (til venstre)
- Elementet $x$ selv (i midten)
- De som er større enn $x$ (til høyre)