1 Forelesning 13

1.1 Plan for forelesningen

- `foldr, foldl, og foldl'`
- Eliminatorer
- Punktfri notasjon

1.2 Fold funksjonen

1.2.1 foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (a:as) = f a (foldr f z as)
```

Eksempel. Sum via foldr

```
sum = foldr (+) 0
```

- 0 blir lagt til sist.
- Vi må vente til hele listen er brettet ut før vi kan begynne å summere.

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

◇

Oppgave. Definer følgende funksjoner ved hjelp av `foldr`

```
factorial :: (Num a) => a -> a
head' :: [a] -> Maybe a
```

1.3 foldl

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z [] = z
foldl f z (a:as) = foldl f (f z a) as
```

Eksempel. Sum via foldl

```
sum = foldl (+) 0
```

```
sum [1,2,3] = foldl (+) 0 (1:2:3:[])
            = foldl (+) (0+1) (2:3:[])
            = foldl (+) ((0+1)+2) (3:[])
            = foldl (+) (((0+1)+2)+3) []
            = (((0+1)+2)+3)
            = 6
```

◇

- 0 kommer nå først
- Vi kunne begynt å regne ut summen før vi kom til siste linje (tail call recursion)

MEN: haskell er lazy, og regner derfor ikke ut denne mens vi går

1.4 Stack Overflow

Med både foldl og foldr er det fare for stack overflow

```
foldr (+) 0 [1 .. 10000000000]
foldl (+) 0 [1 .. 10000000000]
foldr' (+) 0 [1 .. 10000000000]
```

1.5 Eliminatorer

1.5.1 Foldr er spesiell

En hver vellfundert rekursjon på lister kan erstattes med `foldr`

Eksempel. Map via foldr

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (a:as) = f a : map f as
```

Kunne vært

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

◇

En slik funksjon som fanger opp vellfundert rekursjon kalles en **eliminator**.
Andre datatyper har egne eliminatorer.

Eksempel. Lister har:

```
foldr ::  (a -> b -> b) -> b -> [a] -> b
```

Maybe har:

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

Either har

```
either ::  (a -> c) -> (b -> c) -> Either a b -> c
```

◇

1.6 Punktfri notasjon

Det er ikke alltid nødvendig å skrive ut argumentene:

```
applyFilter fil signal = map fil (iterate tail (extend signal))
```

kan skrives som

```
applyfilter fil = map fil .  iterate tail .  extend
```

1.7 Unødvendige lambdaer

Spesielt nyttig når man lager en funksjon ved hjelp av høyereordens funksjoner

```
lessThan3 :: [Integer] -> [Integer]
lessThan3 list = filter (\a -> a < 3) list
```

kunne vært skrevet

```
lessThan3 :: [Integer] -> [Integer]
lessThan3 list = filter (<3) list
```