1 Forelesning 7

1.1 Plan for forelesningen

- Let-bindinger
- Introduksjon til rekursjon

1.1.1 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))
```

1.1.2 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)
```

1.1.3 Flere let features

Let uttrykk kan brukes

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

1.2 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))
```

◇

1.2.1 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
```

□

1.3 Forskjellige typer rekursjon

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

- Produktiv rekursjon
- Generell rekursjon

1.3.1 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.

1.4 Primitiv vs generell rekursjon (Ackermann)

Men det finnnes 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.

1.5 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

◇