1 Øvingsoppgaver

1.1 Regne / Flervalgsoppgaver

a) Hva er verdien til uttrykket `map (+3) [1,2,3]`? → `[4,5,6]`

b) Hva er verdien til uttrykket `sum [x + 3 | x <- [1,2,3]]`? → `15`

c) Hva er verdien til uttrykket `(\x y -> x - y) 7 3`? → `4`

d) Hva er typen til uttrykket `(\x -> 3 : tail x)`? → `Num a => [Int] -> [Int]`

e) Hva er riktig type til uttrykket `(3, Just "Haskell")`? → `(Integer, Maybe String)`

f) Hva er riktig type til uttrykket `Just (Left Nothing)`? → `Maybe (Either (Maybe a) b)`

g) Hvilken kind har `Either String`? → `* -> * -> *`

h) Hvilken Kind har `Integer -> Integer` → `*`

i) Hva er typen til funksjonen `f x y = if x then y else y * 3`?

- `f :: Bool -> Integer -> Integer`
- `f :: (Num a) -> Bool -> a -> a`

1.2 Enkel IO

Skriv et program som leser inn et navn på formen "Fornavn Etternavn" og returnerer "Etternavn, Fornavn"

Svar.

```haskell
main = do
  name <- parse <$> getLine
  print name

parse :: String -> String
parse name = last (words name) ++ ", " ++ head (words name)
```

1.3 Listeoperasjoner

Husk at `concat :: [[a]] ->[a]` kan brukes til å sette sammen en liste med strenger til en streng. Konsonantene i språket vårt er `"bcdfghjklmnpqrstvwxz"`.

a) Skriv en funksjon `isConsonant :: Char -> Bool` som sjekker om en char er en konsonant på norsk.

```haskell
isConsonant :: Char -> Bool
isConsonant c = c `elem` "bcdfghjklmnpqrstvwxz"
```

Her er en funksjon som oversetter til røverspråket:

```haskell
translate :: String -> String
translate word = concat [if isConsonant x then [x] ++ "o" ++ [x] else [x] | x <- word]
```

b) Funksjonen translate bruker listekomprehensjon, skriv den slik at den bruker map istedet.

```
translate :: String -> String
translate = concatMap (\x -> if isConsonant x then [x] ++ "o" ++ [x] else [x])
```

c) Skriv funksjonen `translate` slik at den bruker do-notasjon for lister

```
translate :: String -> IO String
translate (x : xs) = do
  if isConsonant x
    then return $ [x] ++ "o" ++ [x] ++ translate xs
    else return $ x : translate xs
```

d) Skriv en funksjon `differences :: [Integer] -> [Integer]` som regner ut alle positive differanser mellom elementene i en liste, ved hjelp av listekomprehensjon. Eksempel: `differences [1,2,3] = [1,2,1]` fordi $2-1=1$, $3-1=2$ og $3-2=1$.

e) Skriv en funksjon `everyOther :: [a] -> [a]`, som fjerner annethvert element fra en liste. Behold det første elementet i listen, fjern det andre, behold det tredje osv. Eksempel: `everyOther [1,2,3,4] = [1,3]`

```
everyOther :: [a] -> [a]
everyOther [] = []
everyOther [x] = [x]
everyOther (x : y : ys) = x : everyOther ys
```

1.4 Map

I denne oppgaven skal vi se på hvordan vi kan bruke maps til å representere en graf hvor vi har merket kantene.

```
type Graph label node = Map node (Map label node)
```

Hver node mappes til et map som forteller hvilken node som ligger i enden til en kan med en viss merkelapp (`label`). Her er en graf med tre noder hvor merkelappene er bokstaver

```
data N = A | B | C

graph0 :: Graph Char N
graph0 = Map.fromList [(A,Map.fromList [('r',B)])
                      ,(B,Map.fromList [('o',B),('t',C)])
                      ,(C,Map.fromList [('e',A),('t',C)])]
```

a) Skriv en funksjon som setter inn en kant med en gitt merkelapp mellom to noder i en graf.

```
insertLabeledEdge :: (Ord node) => Graph label node -> node -> node -> label -> Graph
    label node
insertLabeledEdge g n1 n2 l = Map.insert n1 (Map.singleton l n1) g
```

b) Bruk do-notasjon for Maybe til å skrive en funksjon som slår opp en node og en label i en graf og gir den neste noden

```
goNext :: (Ord node, Ord label) => Graph label node -> node -> label -> Maybe node
goNext graph start label = do
```

```
    labelMap <- Map.lookup start graph
    Map.lookup label labelMap
```

c) Skriv `goNext` ved hjelp av `>>=` operatoren istedet for do-notasjon.

```
goNext' :: (Ord node, Ord label) => Graph label node -> node -> label -> Maybe node
goNext' graph start label = Map.lookup start graph >>= Map.lookup label
```

d) Skriv en rekursiv funksjon som følger en liste med label fra en start node til en sluttnode

```
followPath :: (Ord node, Ord label, Ord N) => Graph label node -> node -> [label] ->
    Maybe node
followPath g n (x : xs) =
  case goNext' g n x of
    (Just node) -> followPath g node xs
    Nothing -> Just n
```

1.5 foldr vs foldl

I denne oppgaven skal vi se på forskjellen mellom foldr og foldl. Husk at definisjonene til `foldr` og `foldl` er som følger.

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

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

a) Bruk definisjonen til å regne ut `foldr (:)  [] [1,2,3]`

Svar.

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

b) Bruk definisjonen til å regne ut `foldl (\ -> a:l [] [1,2,3]`

Svar.

```
foldl (\l a -> a:l) [] [1,2,3] = foldl (\l a -> a:l) 1:[] [2,3]
                               = foldl (\l a -> a:l) 2:1:[] [3]
                               = foldl (\l a -> a:l) 3:2:1:[] []
                               = 3:2:1:[]
                               = [3,2,1]
```

Funksjonen `repeat :: a -> [a]` er funksjonen som gir en uendelig liste som repeterer et enkelt element.

```
repeat :: a -> [a]
repeat x = x : repeat x
```

Funksjonen `and :: [Bool] -> Bool` kan skrives både ved hjelp av `foldr` og `foldl`.
`and = foldr (&&) True`
eller:
`and' = foldl (&&) True`
c) Forklar hva som skjer hvis man evaluerer følgende uttrykk:

- and (repeat False)
- and' (repeat False)

and - Dersom man tar en liste med bools (`[False, False, False, False, False]`), vil `and` gjøre følgende.

```
and (repeat False) = foldr (&&) True [False, False, False, ..]
                   = True && False foldr (&&) True [False, False, ..]
                   = False foldr (&&) True [False, False, ..]
                   = False foldr (&&) True [False, False, ..]
                   = ...
                   = False
```

Som vi ser vil resultatet alltid evalueres til `False`, og derfor vil det returneres `False`.

and' - Hvis vi tar samme liste, vil følgende skje

```
and' (repeat False) = foldl (&&) True [False, False, False, ..]
                    = foldl (&&) ((&&) True False) [False, False, ..]
                    = ...
```

Ettersom foldl evalueres lazy, vil denne funksjonen henge for alltid og aldri komme til noe resultat.