Regne/flervalgsoppgaver

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

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

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

d) Hva er typen til uttrykket `(\x -> 3 : tail x)`?

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

1. `(Maybe Integer , String)`
2. `(Maybe (Integer, String))`
3. `(Integer,Maybe String)`
4. `(Integer,Just String)`

f) Hva er riktig type til uttrykket `Just (Left Nothing)`?

1. `Maybe (Either (Maybe a) b)`
2. `Maybe (Either a (Maybe b))`
3. `Maybe (Maybe a, Maybe b)`
4. `Either (Maybe a) (Maybe b)`

g) Hvilken kind har `Either String`?

1. `Monad`
2. `* -> *`
3. `* -> * -> *`
4. `Functor`

h) Hvilken kind har `Integer -> Integer`?

1. `Num`
2. `* -> *`
3. `*`
4. `Eq`

Hvilke av typene er riktige typinger av funksjonen nedenfor:

```
f x y = if x then y else y*3
```

1. `f :: Integer -> Integer -> Integer`
2. `f :: Bool -> Bool -> Integer`
3. `f :: Bool -> Integer -> Integer`
4. `f :: (Num a) => Bool -> a -> a`
5. `f :: (Eq a) => a -> Integer -> Integer`

(OBS: Flere av alternativene kan være riktige)

Enkel IO

Skriv et program som leser inn et navn på formen “Fornavn Etternavn” og returner “Etternavn, Fornavn”.

Eksempel kjøring:

```
name: Haskell Curry
Curry, Haskell
```

Du kan ha bruk for funksjonen:

```
words :: String -> [String]
```

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 funskjon `isConsonant :: Char -> Bool` som sjekker om en char er en konsonant på norsk.

Her er en funksjon som oversetter til røverspråket:

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

Eksempel: `translate "haskell" = "hohasoskokelollol"`

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

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

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 og 3-1 = 2 og 3-2 = 1.

e) Skriv en funskjon, `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]`.

Map

(På eksamen ville vi her inkludert dokumentasjonen for Map.lookup, Map.insert og Map.insertWith - her får du slå opp disse selv.)

I denne oppgaven skal vi se på hvordan vi kan bruke maps til å representere et 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 en kant med en viss merkelapp (label).

Her en en graf med tre noder hvor merkelappene er bokstaver:

```
data N = A | B | C

graph0 :: Grap 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 :: Graph label node -> node -> node -> label -> Graph label node

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 :: Graph label node -> node -> label -> Maybe node goNext graph start label = _

c) Skriv goNext ved hjelp av (»=)-operatoren istedet for do-notasjon.

Dersom vi starter i en node, kan vi følge en liste med labels og kanskje ende opp i en sluttnode.

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

   followPath :: Graph label node -> node -> [label] -> Maybe node

For eksempel `followPath graph0 A "rotte" = Just A`

foldl vs foldr

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
```

Som vi kan visualisere med bildene nedenfor.

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

Nå skal vi se hvordan foldr og foldl interagerer med uendelige lister.

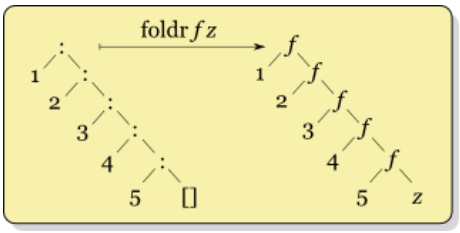

Figure 1: foldr

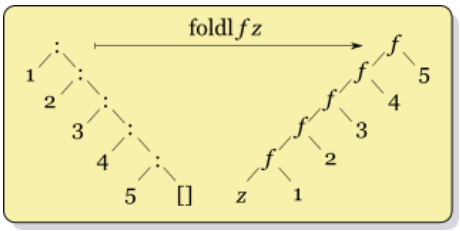

Figure 2: foldl

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

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

F.eks. repeat 1 = [1,1..]

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)`