WRITE ONLY ON ONE SIDE OF THE PAPER - the exams will be scanned in and only the front/odd pages will be read.

DO NOT WRITE WITH OTHER COLOUR THAN BLACK - coloured text may disappear during scanning

PUT YOUR ID AND PAGE NUMBER ON EACH PAGE YOU SUBMIT - make sure that the amount of pages is equal to the amount you note on the front information page

WRITE CLEARLY - if we cannot read you we cannot grade you.

PRELIMINARY AMOUNT OF POINTS: 6 (1,5 + 1 + 1 + 1 + 1,5)
(--A list of selected functions from the Haskell modules:
  Prelude
  Data.List
  Data.Maybe
  Data.Char--)
-----------------------------------------------------

-- standard type classes
class Eq a where
  show :: a -> String
class Show a where
  show :: a -> String

class (Eq a) => Ord a where
  max, min :: a -> a -> a

class (Eq a) => Num a where
  negate :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a

class (Num a, Ord a) => Real a where
  toRational :: a -> Rational

class (Real a, Enum a) => Integral a where
  quot, rem :: a -> a -> a
  div, mod :: a -> a -> a
  divMod :: a -> a -> (a, a)
toInteger :: a -> Integer

class (Num a) => Fractional a where
  (/) :: a -> a -> a
  fromRational :: Rational -> a

class (Fractional a) => Floating a where
  exp, log, sqrt :: a -> a
  sin, cos, tan :: a -> a

class (Real a, Fractional a) => RealFrac a where
  truncate, round :: (Integral b) -> a -> b
  ceiling, floor :: (Integral b) -> a -> b

-- numerical functions
even, odd :: (Integral a) -> a -> Bool
even n = n `rem` 2 == 0
odd = not . even

-- monadic functions
sequence :: Monad m => [m a] -> m []
sequence [] = return []
null [ ] :: [a] -> Maybe a
null [] = Nothing
null (a :) = Just a

-- a hidden goodie

instance Monad [] where
    return x = [x]
xs >>= f = concat (map f xs)

-- functions on pairs

fst :: (a, b) -> a
fst (x,y) = x

snd :: (a, b) -> b
snd (x,y) = y

curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f p = f (fst p) (snd p)

-- functions on lists

map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]

(+++) :: [a] -> [a] -> [a]
xs ++ ys = foldr (:) ys xs

filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]

concat :: [[a]] -> [a]
concat xs = foldr (++) [] xs

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f

head, last :: [a] -> a
head (x:_) = x
last [x] = x
last (_:xs) = last xs

tail, init :: [a] -> [a]
tail (_:xs) = xs
init [x] = []
init (x:xs) = x : init xs

null [ ] :: [a] -> Bool
null [] = True
null (a :) = False

length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

(+++) :: [a] -> Int -> a
(+++) !! 0 = x
(+++) !! n = xs !! (n-1)

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs

iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

repeat :: a -> [a]
repeat x = xs where xs = x:xs

replicate :: Int -> a -> [a]
replicate n x = take n (repeat x)

cycle :: [a] -> [a]
cycle [] = error "Prelude.cycle: empty list"
cycle xs = xs' where xs' = xs++xs'

take, drop :: Int -> [a] -> [a]
take n | n <= 0 = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs

drop n xs | n <= 0 = xs
drop _ [] = []
drop n (_:xs) = drop (n-1) xs

splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)

takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p = []
takeWhile p (x:xs) = p x = x : takeWhile p xs
    | otherwise = []
dropWhile p [] = []
dropWhile p xs@(x:xs')
    | p x = dropWhile p xs'
    | otherwise = xs
delete y (x:xs) = if x == y then xs else x : delete y xs

(*) :: Eq a => [a] -> [a] -> [a]
(*) = foldl (flip delete)

union :: Eq a => [a] -> [a] -> [a]
union xs ys = xs ++ (ys \ xs)

intersect :: Eq a => [a] -> [a] -> [a]
intersect xs ys = [x | x <- xs, x `elem` ys]

intersperse :: a -> [a] -> [a]
intersperse @ [1,2,3,4] = [1,0,2,0,3,0,4]

transpose :: [[a]] -> [[a]]
transpose [[a],[b],[c]] = [[a],[b],[c]]

partition :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs = (filter p xs, filter (not . p) xs)

group :: (a -> [a]) -> [a] -> [[a]]
group "apaabbbeee" = ["apa", "b", "apa", "b", "b", "e", "ee"]

isPrefixOf, isSuffixOf :: Eq a => [a] -> [a] -> Bool
isPrefixOf [] _ = True
isPrefixOf _ [] = False
isPrefixOf (x:xs) (y:ys) = x == y && isPrefixOf xs ys
isSuffixOf x y = reverse x `isPrefixOf` reverse y

sort :: (Ord a) => [a] -> [a]
sort = foldr insert []

insert :: (Ord a) => a -> [a] -> [a]
insert x [] = [x]
insert x (y:xs) = if x <= y then x:y:xs else y:insert x xs

-----------------------------------------------------------------

-- functions on Char

toupper, toLower :: Char -> Char
toupper 'a' = 'A'
toupper 'z' = 'Z'
toLower 'B' = 8

digitToInt :: Char -> Int
digitToInt 'B' = 8

intToDigit :: Int -> Char
intToDigit 3 = '3'

ord :: Char -> Int
chr :: Int -> Char
Exam

1. Type derivation

(a) Assume that the type of reduce is
\[ \text{reduce :: } a \rightarrow a \]
Find the type of
\[ \text{prepare = reduce . words . map toLower . filter} \]
\[ \quad (\text{not . flip elem ".,;:*!#$%&|"}) \]

(b) Given that
\[ \text{map2 :: (a -> b, c -> d) -> (a, c) -> (b, d)} \]
find the destination type \( b \) of the following function:
\[ \text{rulesCompile :: [(String, [String])] -> b} \]
\[ \quad \text{rulesCompile = (map . map2) (words . map toLower, map words)} \]

(c) Given that
\[ \text{transformationApply :: Eq a => a -> ([a] -> [a]) -> [a] -> ([a], [a])} \]
\[ \quad \text{orElse :: Maybe a -> Maybe a -> Maybe a} \]
find the type of
\[ \text{foldr1 orElse (map (transformationApply wildcard f x) pats)} \]

2. Let \( k \) be defined as follows:
\[ k = 0 : 1 : \text{zipWith (+) k (tail k)} \]

(a) What is the type of \( k \)?

(b) What are the first ten elements of \( k \)?

3. Define a function \( \text{altMap} \)
\[ \text{altMap :: (a -> b) -> (a -> b) -> [a] -> [b]} \]
that alternately applies its two functional arguments to successive elements of a list, in turn about order. For example:
\[ \text{altMap (+10) (+100) [0, 1, 2, 3, 4]} = [10, 101, 12, 103, 14] \]

Using \( \text{if} \) in your solution will cause a substantial deduction of available points.
4. Consider the following two versions of similarity score computations. The difference is in the expression defining value for \( \text{simEntry } i \ j \).

(a) Which of the versions is much faster than the other?

(b) Why?

Answering (a) but not (b) does not give much credit. Wrong answer is worth less than “I don’t know”.

**VERSION 1:**

```haskell
similScore :: String -> String -> Int
similScore xs ys = simScore (length xs) (length ys)
where
  simScore i j = simTable!!i!!j
  simTable = [[ simEntry i j | j<-[0..]] | i<-[0..] ]

simEntry :: Int -> Int -> Int
simEntry 0 0 = 0
simEntry i 0 = (i * scoreSpace)
simEntry 0 j = (scoreSpace * j)
simEntry i j = maximum [((simScore (i-1) (j-1)) + (score x y)),
    ((simScore (i-1) j) + (score x '-')),
    ((simScore i (j-1)) + (score '-' y))]
    where
      x = xs!!(i-1)
      y = ys!!(j-1)
```

**VERSION 2:**

```haskell
similScore :: String -> String -> Int
similScore xs ys = simScore (length xs) (length ys)
where
  simScore i j = simTable!!i!!j
  simTable = [[ simEntry i j | j<-[0..]] | i<-[0..] ]

simEntry :: Int -> Int -> Int
simEntry 0 0 = 0
simEntry i 0 = (i * scoreSpace)
simEntry 0 j = (scoreSpace * j)
simEntry i j = maximum [((simEntry (i-1) (j-1)) + (score x y)),
    ((simEntry (i-1) j) + (score x '-')),
    ((simEntry i (j-1)) + (score '-' y))]
    where
      x = xs!!(i-1)
      y = ys!!(j-1)
```
5. Define a type `CircList` (or `CL` for short, if you prefer) defining a circular list of arbitrary length (and holding arbitrary elements). Our examples below will use elements of type `Int`. You have to make sure that the current position is well-defined and accessible for the operations defined for this type. The picture below illustrates the concept and its possible representation using a standard list (with the assumption that the first element defines the current position and that the last position in the list is glued to the first one in a circular fashion):

Please note that it is your task to define the appropriate type constructor! Define then for this type the following functions:

- `perimeter :: CircList a -> Int`
- `currentelem :: CircList a -> Maybe a`
- `nextelem :: CircList a -> Maybe a`
- `previouselem :: CircList a -> Maybe a`
- `insert :: a -> CircList a -> CircList a`
- `delete :: Int -> CircList a -> CircList a`
- `takefromCL :: Int -> CircList a -> [a]`

returning the number of elements (positions) in the list; returning the current element in the list; returning the next element in the list; returning the previous element in the list; inserting an element between the current and the previous element in the list but keeping the current element intact; deleting `n` first elements from the list; and taking `n` first elements of the circular list (possibly circling if necessary), respectively. You may, and are actually encouraged to, define any helper functions you deem appropriate.

Examples of the intended functionality:

- `perimeter (CircList [1, 2, 3, 4, 5]) = 5`
- `currentelem (CircList [1, 2, 3, 4, 5]) = Just 1`
- `currentelem (CircList []) = Nothing`
- `nextelem (CircList [1, 2, 3, 4, 5]) = Just 2`
- `nextelem (CircList [1]) = Just 1`
- `previouselem (CircList [1, 2, 3, 4, 5]) = Just 5`
- `insert 6 (CircList [1, 2, 3, 4, 5]) = CircList [1, 2, 3, 4, 5, 6]`
- `delete 2 (CircList [1, 2, 3, 4, 5]) = CircList [3, 4, 5]`
- `takefromCL 4 (CircList [1, 2, 3]) = [1, 2, 3, 1]`

Finally, define a predicate

- `equalCL :: CircList a -> CircList a -> Bool`
yielding True if and only if both lists contain the same elements in the same order, but not necessarily with the same current position. E.g., if we used the standard list representation for circular lists (assuming the end is glued to the beginning) then

\[
\begin{align*}
\text{equalCL (CircList [1, 2, 3, 4, 5]) (CircList [3, 4, 5, 1, 2])} &= \text{True} \\
\text{equalCL (CircList [1, 2, 3, 4, 5]) (CircList [3, 4, 5, 2, 1])} &= \text{False} \\
\text{equalCL (CircList [1, 2, 3, 4, 5]) (CircList [3, 4, 5, 1, 2, 3])} &= \text{False} \\
\text{equalCL (CircList [1, 2, 3]) (CircList [2, 3, 1, 2, 3, 1])} &= \text{False}
\end{align*}
\]

Good Luck!