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+0.5+1+0.5+1+2)
(A list of selected functions from the Haskell modules:

Prelude
Data.List
Data.Maybe
Data.Char)

-- standard type classes
Class Show a where
  show :: a -> String

Class Eq a where
  (==), (/=) :: a -> a -> Bool

Class (Eq a) => Ord a where
  (<), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a

Class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  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
  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 [a]
sequence = foldr mcons (return [])

where mcons p q = do x <- p; xs <- q; return (x:xs)

sequence_ :: Monad m => [m a] -> m ()
sequence_ xs = do sequence xs; return ()

-- functions on type classes

id        :: a -> a
id x      = x

const     :: a -> b -> a
const x y = x

(,)       :: (b -> c) -> (a -> b) -> a -> c
(f, g)    = \x -> f (g x)

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

($)       :: (a -> b) -> a -> b
f $ x      = f x

-- functions on Maybe

data Bool = False | True

(&&, (||)) :: Bool -> Bool -> Bool
True && x  = x
False && _ = False
True || _  = True
False || x = x
not        :: Bool -> Bool
not True   = False
not False  = True

-- functions on functions

isJust    :: Maybe a -> Bool
isJust (Just a) = True
isJust Nothing = False

isNothing :: Maybe a -> Bool
isNothing = not . isJust

fromJust :: Maybe a -> a
fromJust (Just a) = a

maybeToList :: Maybe a -> [a]
maybeToList Nothing = []
maybeToList (Just a) = [a]
listToMaybe :: [a] -> Maybe a
listToMaybe [] = Nothing
listToMaybe (a:_ ) = Just a

-- a hidden goodie

instance Monad [] where
  return x = [x]
xss >>= 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 (_:_ ) = False
length :: [a] -> Int
length [] = 0
length (_:l) = 1 + length l

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

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

repeat n x = take n (repeat x)

-- Prelude

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]
(\\) = 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 i [1,2,3,4] = [1,i,2,i,3,i,4]

transpose :: [[a]] -> [[a]]
transpose [[a1,a2,a3],[b1,b2,b3]] = [[a1,b1],[a2,b2],[a3,b3]]

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

group :: Eq a => [a] -> [[a]]
group "aapaabbbeee" = ["aaa","p","aaa","bbb","eee"]

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

isPrefixOf [a] [b] = a == b

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:ys) = if x <= y then x:y:ys else y:insert x ys

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

functions on Char

chr :: Int -> String
chr x = chr x

ord :: Char -> Int
ord (char) = ord char

toUpper, toLower :: Char -> Char
-- toUpper 'a' == 'A'
-- toLower 'Z' == 'z'
toUpper x = upper x
toLower x = lower x

digitToInt :: Char -> Int
digitToInt '0' = 0

digitToInt '9' = 9

toIntDigit :: Int -> Char
-- toIntDigit 3 == '3'
toIntDigit x = if x < 10 then chr (ord (chr $ x :: Char)) else take 1 (show x)

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

lines, words :: String -> [String]
-- lines "apa\nbepa\ncepa\n" == "apa\n\nbepa\n\ncepa\n"
-- words "apa\nbepa\n\ncepa\n" == "apa\n\nbepa\n\ncepa\n"

unlines, unwords :: [String] -> String
-- unlines ["apa\n\nbepa\n\ncepa\n"] == "apa\nbepa\ncepa\n"
-- unwords ["apa\n\nbepa\n\ncepa\n"] == "apa\nbepa\ncepa\n"

and, or :: [Bool] -> Bool
and = foldr (&&) True
or = foldr (||) False

any, all :: (a -> Bool) -> [a] -> Bool
any p = or . map p
all p = and . map p

elem, notElem :: (Eq a) => a -> [a] -> Bool
elem x = any (== x)
notElem x = all (/= x)

lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup key = Nothing
lookup key ((x,y):xs) = if key == x then Just y
                            else lookup key xs

sum, product :: (Num a) => [a] -> a
sum = foldl (+) 0
product = foldl (*) 1

maximum, minimum :: (Ord a) => [a] -> a
maximum [] = error "Prelude.maximum: empty list"
maximum xs = foldl1 max xs
minimum [] = error "Prelude.minimum: empty list"
minimum xs = foldl1 min xs

zip :: [a] -> [b] -> [(a,b)]
zip = zipWith (,)

zipWith :: (a->b->c) -> [a]->[b]->[c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _ _ = []

unzip :: [(a,b)] -> ([a],[b])
unzip = foldl (\(a,b) -> (as,bs) -> (a:as,b:bs)) ([],[])

nub :: [Eq a] => [a] -> [a]
nub [] = []
nub (x:xs) = x : nub [y | y <- xs, x /= y]

delete :: Eq a => a -> [a] -> [a]
delete y [] = []
delete y (x:xs) = if x == y then xs else x : delete y xs

foldr (\z a b -> z) a []
Exam

1. **Point-free notation** (1p)
   Rewrite the following two definitions into a point-free form (i.e., \( f = \ldots \), \( g = \ldots \)), using neither lambda-expressions nor list comprehensions nor enumeration nor `where` clause nor `let` clause:

   \[
   f \ x \ y = (3 - y) / x \\
   g \ x \ y = [x \ z \mid z \leftarrow [1,3..y]]
   \]

2. **Type derivation** (0.5p)
   Which type has the function \( g \) defined as

   \[
   g \ xs = [f \ x \mid x \leftarrow \text{xs}, x > 3]
   \]
   where \( f \ n = \text{replicate} \ n \pmb{+} \)?

3. **Type declarations** (1p)
   Explain the difference between `type`, `newtype` and `data` type declarations in Haskell.

4. **Programming** (0.5p)
   Give an example of a function with type

   \[
   (a \rightarrow b, [a]) \rightarrow [b]
   \]

5. **List comprehension** (1p)
   A Pythagorean triad is a triple of integers \((a,b,c)\) such that

   \[
   a^2 + b^2 = c^2
   \]

   Define a function of \( n \) that will find all Pythagorean triads with \( a \leq b \leq c \leq n \). Express it using list comprehension.
6. **Programming** (2p)

A proposition is a boolean formula of one of the following forms:

- a variable name (a string)
- \( p \land q \) (and)
- \( p \lor q \) (or)
- \( \neg p \) (not)

where \( p \) and \( q \) are propositions (note recursiveness). For example, \( p \lor \neg p \) is a proposition.

(a) Design a data type `Proposition` to represent propositions.

(b) Define a function

\[
\text{vars :: Proposition } \rightarrow \ [\text{String}]
\]

which returns a list of the variables in a proposition. Make sure each variable appears only once in the list you return. E.g., for (your representation of) \( p \lor \neg p \) it should return \( p \).

(c) Suppose you are given a list of variable names, and their values of type `Bool`, for example, \[ ("p", True), ("q", False) \]. Define a function

\[
\text{truthValue :: Proposition } \rightarrow \ [(\text{String}, \text{Bool})] \rightarrow \text{Bool}
\]

which determines whether the proposition is true when the variables have the values given as second argument. E.g., for (your representation of) \( p \lor \neg p \) and \[ ("p", \text{True}) \] it should return \text{True}.

(d) Define a function

\[
\text{tautology :: Proposition } \rightarrow \ \text{Bool}
\]

which returns true if the proposition holds for all values of the variables appearing in it. E.g., for (your representation of) \( p \lor \neg p \) it should return \text{True}.

Good Luck!