EDAF40

examination

2.5 hp

3rd June 2019
14:00 - 19:00

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 OR DARK BLUE - lightly 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+1+1+1+1+1)
(-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) => RealFloat 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 functions
  id      :: a -> a
  id x    = x

  const   :: a -> b -> a
  const x _ = 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 Bools
  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 Maybe
  data Maybe a = Nothing | Just a

  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]
    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]
foldr 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:_)= last x

last [x]  = x
last (_:xs) = last xs

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

null [ ]   = True
null (_:_ ) = False

length :: [a] -> Int
length [] = 0
length (_:l) = 1 + length l

(!!) :: [a] -> Int -> a
(!!) xs i = x

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]
            = 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 | x = dropWhile p xs'
                             otherwise = xs
lines, words :: String -> [String]
-- lines "apa\nbe pa\ncepa\n" == ["apa","be pa","cepa"]
-- words "apa be pa\ncepa\n" == ["apa","be pa","cepa"]

unlines, unwords :: Text -> String
-- unlines ["apa","be pa","cepa"] == "apa\nbe pa\ncepa\n"
-- unwords ["apa","be pa","cepa"] == "apa be pa\ncepa\n"

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

any, all :: (a -> Bool) -> [a] -> Bool
any       = or . map p
all        = 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) |
| key == x    = Just y
| otherwise   = 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 z (a:as) (b:bs) = z a b : zipWith z as bs
zipWith _ _ _ = []

unzip :: [(a,b)] -> ([a],[b])
unzip = foldr (!\(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

(\\)
:: 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 0 [1,2,3,4] == [1,0,2,0,3,0,4]

transpose :: [[a]] -> [[a]]
transpose [[1,2,3],[4,5,6]] == [[1,4],[2,5],[3,6]]

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

group :: Eq a => [a] -> [[a]]
group "aapaabbeee" == ["aa","p","aa","bb","ee"]
isPrefixOf, isSuffixOf :: Eq a => a -> [a] -> Bool
isPrefixOf []            = True
isPrefixOf [t]           = 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]
in\sert x []  = [x]
in\sert x (y:xs) = if x <= y then y:x:xs else y:insert x xs

-- functions on Char

toUppercase, toLowercase :: Char -> Char
toUppercase 'a' = 'A'
toLowercase 'A' = 'a'
toUppercase 'Z' = 'Z'
toLowercase 'z' = 'z'
digitToInt :: Char -> Int
digitToInt '0' = 0
digitToInt '9' = 9

intToDigit :: Int -> Char
intToDigit 0 = '0'
intToDigit 9 = '9'

ord :: Char -> Int
chr :: Int -> Char
1. **Type derivation** (1p)
   Give the types of the following expressions:

   - `zipWith map`
   - `map zipWith`
   - `map.zipWith`

   and explain their meaning.

2. **Programming** (1p)
   Write a function

   ```haskell
   permutations :: [a] -> [[a]]
   ```

   that given an arbitrary list with non-repeating elements would produce all
   the permutations of this list. (A *permutation* of a list is a list containing
   exactly the same elements, but possibly in different order.)

   Examples:
   ```haskell
   Prelude> permutations []
   []
   Prelude> permutations [1]
   [[1]]
   Prelude> permutations [1,2]
   [[1,2],[2,1]]
   Prelude> permutations [1,2,3]
   [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
   ```

   Note that the order of individual permutations in the above examples is
   not important, just that they are to be found somewhere in the answer.

   You may use the assumption that elements in the input list do not repeat.
   For the repeating case the outcome may be arbitrary.
   Actually, a solution neglecting this fact may be simpler.

3. **List comprehension** (1p)
   Write, using list comprehension syntax, a single function definition (try
   to avoid *if*, *case* and similar constructs) with signature

   ```haskell
   g :: [[[Int]]] -> [[[Int]]],
   ```

   which, from a list of lists of *Int*, returns a list of the tails of those lists
   using, as filtering condition, that the head of each *Int* must be odd.
   Also, your function must not trigger an error when it meets an empty
   *Int*, but rather silently skip such an entry. Example:
Prelude> g [[1,2],[],[6,2,3],[3],[6,5,4,3],[6],[5,1,1]]
[[2],[],[1,1]]

Rewrite now this definition using \texttt{map} and \texttt{filter} instead of list comprehension.

4. Folding (1p)
   
   What does the following function do?

   \[
   \text{ok\_\_nd \ } xs = \text{foldr} \ (\text{\texttt{++}}) \ [] \ (\text{\texttt{map} } (\text{\texttt{\_\_y \ -> \ [\_\_y]}}) \ xs)
   \]

5. Pattern matching (1p)

   Define the following function using pattern matching:

   \[
   \text{oneOf :: Bool -> Bool -> Bool -> Bool}
   \]

   \[
   \text{oneOf a b c}
   \]

   \[
   | \text{not (a or b)} = c
   | \text{not (b or c)} = a
   | \text{not (a or c)} = b
   | \text{otherwise} = \text{False}
   \]

6. Bind (1p)

   The following function could have been part of your solution to Assignment 2:

   \[
   \text{eliminatem} \ n \ [] \ g = \text{Just} \ g
   \]

   \[
   \text{eliminatem} \ n \ (s:ss) \ g = \text{eliminate} \ n \ s \ g \ >>= \text{eliminatem} \ n \ ss
   \]

   (a) (0,4p) Given that the type of \(g\) is \texttt{Grid}, write the signature for this function. Assume the most generic type for \texttt{eliminate}.

   (b) (0,2p) How would your answer look like if the first line were changed to

   \[
   \text{eliminatem} \ n \ [] \ g = [g]
   \]

   (c) (0,4p) Would the second line be correct after this change? Answer \texttt{YES} or \texttt{NO}, and motivate.

Good Luck!