EDAN40

examination

4 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+2+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) => 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)
listToMaybe :: [a] -> Maybe a
listToMaybe [] = Nothing
listToMaybe (a:_)= Just a

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

--- a hidden goodie

instance Monad (Maybe a) where
  return x = Just x
  Nothing >>= f = Nothing
  Just x >>= f = f x

instance Monad (p a) 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 (_:xs) = 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 (x:_)= False

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

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

\(\setminus\) \quad \text{Eq a \implies [a] \implies [a]}\)
\(\setminus\) \quad \text{foldl (flip delete)}

union \quad \text{Eq a \implies [a] \implies [a]}\)
union xs ys \quad \text{= xs \++ ( ys \\setminus xs )}

\(\cap\) \quad \text{Eq a \implies [a] \implies [a]}\)
\(\cap\) \quad \text{intersperse \Theta \([1,2,3,4]\) \implies \([1,0,2,0,3,0,4]\)}\)

transpose \quad \text{[[a]] \implies [[a]]}\)
-- transpose **\([1,2,3],[4,5,6]\)** \implies **\([1,4],[2,5],[3,6]\)**

partition \quad \text{\(\{[a]\} \implies [a]\)}\)
-- partition **\([1,2,3],[4,5,6]\)** \implies **\([1,4],[2,5],[3,6]\)**

transpose \quad \text{[[a]] \implies [[a]]}\)
-- transpose **\([1,2,3],[4,5,6]\)** \implies **\([1,4],[2,5],[3,6]\)**

partition p xs \quad \text{= \{filter p xs, filter (not \cdot p) xs\}}\)

\(\{[a]\} \implies [[a]]\)
-- group "aapaabbeee" \implies ["aa",",p",",aa",",bb",",ee"]

isPrefixOf, isSuffixOf \quad \text{Eq a \implies [a] \implies Bool}
isPrefixOf [] \quad \text{= True}
isPrefixOf [] [] \quad \text{= False}
isPrefixOf x:ys (y:ys) \quad \text{x == y \\& isPrefixOf xs ys}
isSuffixOf x y \quad \text{= reverse x `isSuffixOf` reverse y}
sort \quad \text{\{Ord a\} \implies [a] \implies [a]}\)
sort \quad \text{foldr insert []}
insert \quad \text{\{Ord a\} \implies a \implies [a] \implies [a]}\)
insert x [] \quad \text{= [x]}
insert x (y:xs) \quad \text{= if x <= y then x:y:xs else y:insert x xs}

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

functions on Char

toupper, toLower :: Char \implies Char
-- toupper 'a' \implies 'A'
-- tolower 'Z' \implies 'z'
.isdigitToInt :: Char \implies Int
-- digitToInt '8' \implies 8

toIntDigit :: Int \implies Char
-- toIntDigit 3 \implies '3'

ord :: Char \implies Int
chr :: Int \implies Char
Exam

1. Proving program properties (1p)
   The `Functor` class is defined as follows:

   ```haskell
   class Functor f where
     fmap :: (a -> b) -> f a -> f b
   ```

   It is mandatory that all instances of `Functor` should obey:

   ```haskell
   fmap id = id
   fmap (p . q) = (fmap p) . (fmap q)
   ```

   Assume the following definition of lists as a functor instance:

   ```haskell
   instance Functor [] where
     fmap g [] = []
     fmap g (x:xs) = [g x] ++ (fmap g xs)
   ```

   Is this a correct definition of a functor instance? Why or why not? Prove your claim.

2. Types and type classes (2p)
   - (0.3p) Define a tree data structure so that the trees are ternary (i.e., each node has either exactly three children or is a leaf) and store strings in each node.
   - (0.3p) Generalize your definition so that your ternary trees can contain objects of an arbitrary predetermined type in a node.
   - (0.7p) Assuming your polymorphic trees type is denoted by `Tree3 a`, write all necessary code so that the following function is correct:

     ```haskell
     myLength :: Tree3 String -> Tree3 Integer
     myLength = fmap length
     ```

     and yields an (obviously ternary) tree with nodes containing lengths of the strings placed in the respective nodes of the argument tree. Your solution must contain the word `instance` to get full credit.
   - (0.7p) write all necessary code so that you can compare two ternary trees for equality using the `==` operator.

3. 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

\[
g x y = \text{map } x \mathbin{\&} \text{filter } (<3) y
\]

Hint: you may find \texttt{flip} useful.

4. **Sparks (1p)**

Consider the following code parallelizing the quicksort algorithm.

```hs
-- file: ch24/Sorting.hs
import Control.Parallel (par, pseq)
parSort (x:xs) = greater 'par' (lesser 'pseq'
  (lesser ++ x:greater))
  where lesser = parSort [y | y <- xs, y < x]
    greater = parSort [y | y <- xs, y >= x]
parSort _ = []
```

It will be almost as (in)efficient as the sequential version:

```hs
-- file: ch24/Sorting.hs
sort :: (Ord a) => [a] -> [a]
sort (x:xs) = lesser ++ x:greater
  where lesser = sort [y | y <- xs, y < x]
    greater = sort [y | y <- xs, y >= x]
sort _ = []
```

Why?

5. **Parsing (1p)**

This is an excerpt from the assignment N3 code:

```hs
assignment = word #~ accept ":=" # Expr.parse
  #~ require ":;" >-> buildAss
buildAss (v, e) = Assignment v e
```

- (0,5p) Provide types for all functions and operators named here.
- (0,5p) Rewrite \texttt{assignment} using \texttt{do}-notation.

Good Luck!