Napredno funkcionalno programiranje

Zadaća 3

Amer Hasanović

Problem 1

U modulu `MySet` data je početna definicija klase `ActsAsSet`, koja predstavlja generički interfejs za tipove koji se mogu ponašati kao setovi:

```
{-# OPTIONS_GHC -Wall #-}

module MySet where

class ActsAsSet f where
  emp :: f a

  singleton :: a -> f a

  insert :: Ord a => a -> f a -> f a

  getElement :: Ord a => a -> f a -> Maybe a

  fold :: (a -> b -> b) -> b -> f a -> b

  delete :: Ord a => a -> f a -> f a
  delete x  = fold (\y s -> if x == y then s else insert y s) emp

  union :: Ord a => f a -> f a -> f a

  isEmpty :: f a -> Bool

  size :: f a -> Int

  isIn :: Ord a => a -> f a -> Bool

  toSortedList :: f a -> [a]
```

Zadatak 1

U klasi `ActsAsSet` postoji osnovna implementacija funkcije `delete`. Modificirajte klasu tako da dodate osnovne implementacije funkcija `union`, `isEmpty`, `size`, `isIn` i `toSortedList`.

Zadatak 2

Unutar modula `MySet` implementirajte slijedeće funkcije:

```
fromList :: (ActsAsSet f, Ord a) => [a] -> f a
fromList = undefined

difference :: (ActsAsSet f, Ord a) => f a -> f a -> f a
difference xs ys = undefined

subset :: (ActsAsSet f, Ord a) => f a -> f a -> Bool
subset xs ys = undefined
```

Gdje `difference xs ys` vraća kontejner sa elementima iz `xs` koji nisu u `ys`, a `subset xs ys` predstavlja test da li su svi elementi iz `xs` ujedno i elementi u `ys`.

Zadatak 3

Sa stanovišta algoritamske kompleksnosti unutar modula `MySet` napravite što bolje instance klase `ActsAsSet` za Haskell liste, kao i za tip `Tree`, gdje je tip `Tree` definiran kao:

```
data Tree a = Empty | Node (Tree a) a (Tree a) deriving (Show, Eq)
```

Zadatak 4

Neka je unutar modula `MySet` dat sljedeći kod:

```
newtype Pair k v = Pair { getTuple :: (k,v) }

instance Eq k => Eq (Pair k v) where
  (Pair kv1) == (Pair kv2) = fst kv1 == fst kv2

instance Ord k => Ord (Pair k v) where
  compare (Pair kv1) (Pair kv2) = compare (fst kv1) (fst kv2)

instance (Show k, Show v) => Show (Pair k v) where
  show (Pair (k,v)) = show k ++ " |-> " ++ show v

type Map f k v = f (Pair k v)
type ListMap k v = Map [] k v
type TreeMap k v = Map Tree k v

emptyMap :: ActsAsSet f => Map f k v
emptyMap = undefined

extend :: (ActsAsSet f, Ord k) => k -> v -> Map f k v -> Map f k v
extend k v = undefined

toSList :: ActsAsSet f => Map f k v -> [(k,v)]
toSList = undefined

lookup :: (ActsAsSet f, Ord k) => k -> Map f k v -> Maybe v
lookup k  = undefined

remove :: (ActsAsSet f, Ord k) => k -> Map f k v -> Map f k v
remove k = undefined
```

Implementirajte funkcije `emptyMap`, `extend` i `toSList`.

Ukoliko je moguće implementirajte funkcije `lookup` i `remove`, u suprotnom, predložite i obrazložite neophodne promjene u kodu, a zatim implementirajte `lookup` i `remove`.

Problem 2

U modulu `Math` dat je sljedeći kod:

```
{-# OPTIONS_GHC -Wall #-}

module Math where

import Text.ParserCombinators.ReadP
import Data.Char
import Control.Applicative hiding (optional)

data MExp a =
    Num a
  | Add (MExp a) (MExp a)
  | Multiply (MExp a) (MExp a)
  | Subtract (MExp a) (MExp a) deriving (Show)

compute :: (Num a) => MExp a -> a
compute (Num a) = undefined

pNum :: String -> Integer
pNum = read

number :: ReadP (MExp Integer)
number = do
  a <- option "" (string "-")
  b <- munch1 isDigit
  pure $ Num $ pNum (a++b)

parsePlusOrMinus :: ReadP (MExp Integer)
parsePlusOrMinus = do
  a <- term
  op <- char '+' <|> char '-'
  b <- mexpr
  case op of
    '+' -> pure $ Add a b
    _ -> pure $ Subtract a b

mexpr :: ReadP (MExp Integer)
mexpr = parsePlusOrMinus <|> term

term :: ReadP (MExp Integer)
term = parseTimes <|> number

parseTimes :: ReadP (MExp Integer)
parseTimes = do
  a <- number
  _ <- char '*'
  b <- term
  pure $ Multiply a b

parseMExp :: String -> Maybe (MExp Integer)
parseMExp input = case readP_to_S mexpr input of
    [] -> Nothing
    xs -> (Just . fst . last) xs
```

Tip `MExp` predstavlja `Haskell` reprezentaciju matematičkog izraza sa jednostavnim operacijama sabiranja, oduzimanja i množenja. Funkcija `parseMExp` omogućava konvertovanje (parsiranje) `String`-a u `MExp`, pod uslovom da je sadržaj stringa validni matematički izraz, npr:

```
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude> :load Math.hs
[1 of 1] Compiling Math             ( Math.hs, interpreted )
Ok, one module loaded.
*Math> parseMExp "2"
Just (Num 2)
*Math> parseMExp "2+3"
Just (Add (Num 2) (Num 3))
*Math> parseMExp "-4*2+3*5*4"
Just (Add (Multiply (Num (-4)) (Num 2)) (Multiply (Num 3) (Multiply
(Num 5) (Num 4))))
*Math> parseMExp ""8"
Nothing
*Math> parseMExp "'dskfj"
Nothing
```

Zadatak 1

Implementirajte funkciju `compute` koja vraća rezultat proračuna matematičkog izraza predstavljenog putem `MExpr`, npr:

```
*Math> compute <$> parseMExp "2"
Just 2
*Math> compute <$> parseMExp "2*2"
Just 4
*Math> compute <$> parseMExp "2+4*3-8*25"
Just (-186)
```

Zadatak 2

Implementirajte **minimalni** neophodni kod kako bi sljedeće linije koda bile ispravne i proizvodile rezultate kao u primjeru:

```
*Math> parseMExp "2" == parseMExp "1+1+0"
True
*Math> parseMExp "2*4" > parseMExp "2+2+2"
True
*Math> parseMExp "2*4" < parseMExp "100-40"
True
*Math> (+) <$> parseMExp "2*4" <*> parseMExp "100-40"
Just (Num 68)
```