Języki i Metody Programowania II 2018 – KOLOKWIUM(γ)

NAZWISKO: ..... IMIĘ: ..... DATA: ..... GRUPA: .....

1. Mamy dane typy danych:
   ```
   data Student = Student {id :: Int, name :: String, dateOfBirth :: String}
   data Score = Score {studentId :: Int, course1Score :: Int,
    course2Score :: Int, course3Score :: Int}
   ```
   Zdefiniuj typ `StudentWithScores` który będzie posiadał pola takie jak w typie danych Student i takie jak w typie danych Score (bez powtórzenia id). Uwaga na konflikt nazw w module! Jak można rozwiązać problem nazw?

2. Zdefiniuj instancje dla klasy `Show` dla typów danych z poprzedniego podpunktu

3. Zdefiniuj instancje dla klasy `Eq` dla typów danych z poprzedniego podpunktu

4. Dla przykładowych danych `students :: [Student]`
   ```
   students = [Student 26453 "Kristalee Copperwaite" "2000",
    Student 33596 "Roeberta Naden" "1997",...]
   scores :: [Score]
   scores = [Score 26453 93 97 80, Score 40241 86 85 87,
    Score 33596 82 60 80,...]
   ```
   zdefiniuj funkcję sortującą studentów po dacie urodzenia rosnąco.
   zdefiniuj funkcję sortującą studentów po imieniu i nazwisku urodzenia malejąco.
   zdefiniuj funkcję sortującą rekordy wyników tak, żeby rekordy były posortowane najpierw po wyniku z pierwszego kursu, a następnie (dla remisów) po id studneta
   zdefiniuj funkcję sortującą rekordy wyników tak, żeby rekordy z większą sumaryczną ilością punktów były wcześniej od tych z mniejszą sumaryczną ilością punktów
   ```
   sortBy :: (a -> a -> Ordering) -> [a] -> [a]
   data Ordering = LT | EQ | GT
   ```

5. Zdefiniuj fukcję o sygnaturze
   ```
   toStudentWithScores :: Student -> Score
      -> Maybe StudentWithScores
   ```
   która zwróci połączony rekord studenta z jego wynikami, jeśli studentId jest równe id a Nothing w przeciwnym wypadku.

6. Zdefinuj funkcję o sygnatrzue `findById :: [Score] -> Int -> [Score]` która ma za zadnie zwrócić wyniki dla studenta o zadanym id (drugi parametr) przy użyciu funkcji `filter`

7. Jeśli zdefinujemy funkcję `findBy f scrs = filter f scrs` to jaką sygnaturę będzie miała ta funkcja, jeśli scrs to `[Score]`?

8. Jeśli zdefinujemy funkcję `mapBy f scrs = map f scrs` to jaką sygnaturę będzie miała ta funkcja, jeśli scrs to `[Score]`?

9. Jeśli zdefinujemy funkcję `reduceBy f scrs = foldl f [] scrs` to jaką sygnaturę będzie miała ta funkcja, jeśli scrs to `[Score]`?

10. Przy użyciu funkcji map zdefinuj funkcję o sygnaturze
    `mapToJoin :: Student -> [Score] -> [Maybe StudentWithScores]` Wykorzystaj już zdefiniwane wcześniej funkcje

11. Zdefiniuj funkcję `joinStep1 :: [Student] -> [Score] -> [(Student, [Score])]`

12. Zdefiniuj funkcję `joinStep2 :: [(Student, [Score])] -> [[Maybe StudentWithScores]]`
13. Zdefiniuj funkcję `joinStep3 :: [[Maybe StudentWithScores]] -> [StudentWithScores]`
14. Zdefiniuj funkcję `join :: [Student] -> [Score] -> [StudentWithScores]`
15. Zdefiniuj klasę `Id` która rozszerza klasę `Ord` i zawiera pojedyncza metodę `toInt` konwertującą daną implementację na typ `Int`
16. Zdefiniuj klasę `HasId` zawiera pojedyncza metodę `getId` konwertującą daną implementację na dane typu `Id`
17. Zdefiniuj klasę `Repository` zawierające operacje insert, delete, get, update i count. insert pobiera element implementujący HasId i implementacje repozytorium i zwraca nowe repozytorium. delete pobiera id należące do klasy Id i implementacje repozytorium i zwraca nowe repozytorium get pobiera id należące do klasy Id i implementacje repozytorium i zwraca Maybe element należący do klasy HasId. update pobiera id należące do klasy Id, element implementujący HasId i implementacje repozytorium i zwraca nowe repozytorium. count pobierające implementacje repozytorium i zwraca Int
18. Zdefiniuj typ InMemoryRepository który posiada jedno pole typu `Map key value` i należy do klasy Repository

```
insert :: Ord k => k -> a -> Map k a -> Map k a
Data.Map.delete :: Ord k => k -> Map k a -> Map k a
Data.Map.alter :: Ord k => (Maybe a -> Maybe a) -> k -> Map k a -> Map k a
Data.Map.lookup :: Ord k => k -> Map k a -> Maybe a
Data.Map.size :: Map k a -> Int
```

19. Jak wygląda definicja instacji dla klasy Applicative, Functor i Monad dla typu danych Maybe

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  {-# MINIMAL fmap #-}

class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
  {-# MINIMAL pure, (<*>) #-}

class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a
  {-# MINIMAL (>>=) #-}
```

20. napisz program typu hello world wczytujący imię ze standardowego wejścia i wypisujący na standardowe wyjście to imie kapitalikami + hello a pozniej goodbye + to imie w odwroconej kolejnosci