Functional Programming / Funkcinis programavimas

Exercise set 5

Solutions to be sent until December 15th

Exercise 1. We can define a data structure for generalised expressions as follows:

```
data Expr a = Lit a | EVar Var | Op (Ops a) [Expr a]
type Ops a = [a] -> a
type Var = Char
```

Thus, an expression can be either a literal (of the type `a`), a variable name, or an operator applied to a list of expressions. In turn, an operator is represented as a function that reduces a list of `a` values to a single result.

To evaluate an expression, we need to know all the current values (of the type `a`) associated to the expression variables. Define a new type `Valuation a`, relating variables (of the type `Var`) with the values of the type `a`. Then write a function:

```
eval ::  Valuation a -> Expr a -> a
```

that, for the given variable valuation and expression, evaluates (folds) the expression to a single value.

Exercise 2. Define a new data type

```
NumList a = Nlist [a],
```

for lists containing numbers.

Make this type an instance of the type classes `Eq` and `Ord` by defining that such lists are equal only if their average values are equal, and such lists can be compared only by comparing their respective average values. Add the needed context (type class dependencies) to make such instances work.

The average value for `[]` is 0.

Exercise 3. We can handle functions working on numbers as numbers themselves, i.e., add them, negate them, etc. For instance, adding two such functions gives us a function that first separately applies the functions on the given argument and then adds the respective results. Similarly, negating a function is a function that negates the function result for the given argument.

Make functions working on numbers as an instance of the type class `Num`, i.e.,

```
instance (Num a, Num b) => Num (a-> b) where ...
```

by defining the respective number operations `(+),(*), negate, abs, signum` and `fromInteger` for the type `a->b`.

Exercise 4. Instead of binary trees, we can define general trees with an arbitrary list of subtrees, e.g.,

```
data GTree a = Leaf a | Gnode [GTree a]
```

Define functions for this datatype, which

- return the depth of a `GTree`;
- find whether an element occurs in a `GTree`;
- map a given function over the elements at the leaves of a `GTree` (i.e., a variation of the `mapTree` function from the Lecture 11 slides for `GTree`).

Exercise 5. The `Maybe a` type (see the Lecture 11 slides) could be generalised to allow messages to be carried in the `Nothing` part, e.g.,

```
data Result a = OK a | Error String
```

Define the function

```
composeResult ::  (a -> Result b) -> (b -> Result c)
     -> (a -> Result c)
```

which composes together two (possibly error-raising) functions.

Exercise 6. An efficient Haskell implementation of sets (for ordered types) can be imported as, e.g.,

```
import qualified Data.Set as S
```

A data structure describing relations between elements of two types is often represented as sets of pairs of the related elements, i.e.,

```
type Relation a b = S.Set (a,b)
```

Define the following functions on relations:

```
dom :: Ord a => Relation a b -> S.Set a
ran :: Ord b => Relation a b -> S.Set b
image :: (Ord a,Ord b) => S.Set a -> Relation a b -> S.Set b
```

where the function `dom r` returns the relation domain (the set of all elements of `a` related to `b` by `r`), the function `ran r` returns the relation range (the set of all elements of `b` related to `a` by `r`), and the function `image s r` returns a set of all elements of `b` related to the elements from `s` by `r`.

Since all relations are special kinds of sets, all the set functions from `qualified Data.Set as S` can be used on them, e.g., `S.member, S.map, S.filter,` etc.

Exercise 7. The Goldbach conjecture for prime numbers (still unproven in the general case) states that any even number greater than 2 can be rewritten as a sum of two prime numbers. Write a function that checks that the conjecture is true up to the given upper bound. In other words, the function

```
goldbach :: Integer -> Bool
```

for the given integer `n`, should return `True` if all even numbers in `[4..n]` satisfy the conjecture.

Use the generated infinite list of primes (from the Lecture 12 slides) and list comprehensions to construct and compare the corresponding lists of numbers (e.g., those that can be expressed as sums of two primes and those that are even).

Exercise 8. Infinite data streams (see the Lecture 12 slides), i.e., the lists that must be infinite, can be defined in Haskell as follows:

```
data Stream a = Cons a (Stream a)
```

Define the described below functions for this datatype:

- `streamtoList :: Stream a -> [a]`
  creates an infinite list out of the given stream;
- `streamIterate :: (a -> a) -> a -> Stream a`
  creates a stream for the given iteration function and the starting stream element (seed);
- `streamInterleave :: Stream a -> Stream a -> Stream a`
  merges two streams into one so that their elements are interleaved. In other words, for two given streams $\langle e_{11}, e_{12}, e_{13}, ...\rangle$ and $\langle e_{21}, e_{22}, e_{23}, ...\rangle$, the result would be the stream $\langle e_{11}, e_{21}, e_{12}, e_{22}, ...\rangle$