Programming in Haskell – Homework Assignment 6

UNIZG FER, 2013/2014

Handed out: December 11, 2013. Due: December 15, 2013 at 23:59

Note: Define each function with the exact name and type specified. You can (and in most cases should) define each function using a number of simpler functions. Provide a type signature above each function definition and comment the function above the type signature. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values. Use the `error` function for cases in which a function should terminate with an error message, and return the exact error message specified. Problems marked with a star ($\star$) are optional.

1. Define three versions of a function for testing whether a number is prime.

   (a) Using `foldr`.

   ```
   fprime :: Integral a => a -> Bool
   fprime 23 ⇒ True
   fprime 12 ⇒ False
   ```

   (b) Using `Prelude.all` or similar functions on lists (e.g. `map, filter, and`).

   ```
   lprime :: Integral a => a -> Bool
   ```

   (c) Using explicit recursion.

   ```
   rprime :: Integral a => a -> Bool
   ```

2. Using `foldr` define `cycle'` that behaves exactly like `Prelude.cycle`.

   ```
   cycle' :: [a] -> [a]
   cycle' [] ⇒ error "cycle':  empty list"
   cycle' "abc" ⇒ "abcabcabcabcabc...
   ```

3. Define `mapMasked` that works like `Prelude.map` but applies different functions to different elements based on the index returned by the indexing function (first argument of `mapMasked`). In other words, we apply the i-th function on all elements for which the indexing function returns the index i. You don't have to account for index-out-of-bounds errors. (Also, figure out what the functions `Prelude.const` and `Prelude.id` do.)

   ```
   mapMasked :: (a -> Int) -> [a -> b] -> [a] -> [b]
   mapMasked (`mod` 3) [id,(+2),(+1)] [1..] ⇒ [3,3,3,6,6,6,9,9,9,...]
   mapMasked (\x -> if x >= 0 then 1 else 0) [const (-1),const 1] [-2,-3,5,0]
   ⇒ [-1,-1,1,1]
   mapMasked (min 3 .  length) [const "0", const "1", const "2", const ">=3"]
   ⇒ ["","elephant","ox"] => ["0",">=3","2"]
   ```

4. Using `foldr`, define `mean` to calculate the arithmetic mean of a list of elements.

```
mean :: Fractional a => [a] -> a
mean [] ⇒ NaN
mean [6,3,6] ⇒ 5
```

5. (a) Using `foldr`, define `ncomp` to compose a list of functions into a single function.

```
ncomp :: [a -> a] -> (a -> a)
ncomp [(/6),(+5),(*(-1))] ⇒ ((/6).(+5).(*(-1)))
ncomp [(+2)] ⇒ (+2)
ncomp [] ⇒ id
```

   (b) Define `fsort` that sorts functions in ascending order by the average value they return for a given domain. Use `Data.List.sortBy` and `Data.Ord.comparing`.

```
fsort :: (Fractional a, Ord a) => [a -> a] -> [a] -> [a -> a]
fsort [(/6),(+5),(*(-1))] [1,2,3,4,5] ⇒ [(*(-1)),(/6),(+5)]
```

   (c) Define `compsort` that sorts functions in ascending order by the average value they return for a given domain and then returns their composition.

```
compsort :: (Fractional a, Ord a) => [a -> a] -> [a] -> (a -> a)
compsort [(/6),(+5),(*(-1))] [1,2,3,4,5] 3 ⇒ (*(-1)).(/6).(+5) $
3 => (-1.333)
```

6. Let's implement a set structure using functions. We define a set to be equal to an indicator function that tests whether an element is a member of that set:

```
type Set a = a -> Bool
```

   Building on such a definition, define the following functions. Try to define each in a single line of code (very doable!). Use eta reduction wherever you think it simplifies the function definition.

   (a) A function that creates an empty set. (Hint: what does `const` do?)

```
empty :: Set a
```

   (b) A function that creates a set containing a single element.

```
single :: Eq a => a -> Set a
```

   (c) A function that returns the union of two sets.

```
union' :: Set a -> Set a -> Set a
```

   (d) A function that returns the difference between two sets (all elements contained within the first, but not contained within the second set).

```
difference :: Set a -> Set a -> Set a
```

   (e) A function that inserts an element into the set.

```
insert' :: Eq a => a -> Set a -> Set a
```

   (f) A function that removes an element from the set. (Hint: the order of operations matters.)

```
remove :: Eq a => a -> Set a -> Set a
```

7. You are given a non-empty two-level nested list of elements and two binary functions `f1` and `f2` that operate on them. Write a funcion `reduce` that reduces the given list using `f1` and `f2` in the following manner:

```
reduce :: [[a]] -> (a -> a -> a) -> (a -> a -> a) -> a
reduce [ [a,b,c], [d,e] ] f1 f2 -> (a `f2` b `f2` c) `f1` (d `f2` e)
```

For example:

```
reduce [ [1,2,3], [4,5] ] (*) (+) ⇒ (1+2+3) * (4+5) = 6 * 9 = 54
reduce [["one","Two"],["two","One"]] (\a b -> a ++ "-" ++ b) (++)
⇒ "oneTwo-twoOne"
reduce [[]] f1 f2
⇒ error "List must not be empty - don't know what to return!"
```

8. (a) Using a fold function (e.g., `foldr`), define `insert''` that behaves exactly like `Data.List.insert`. The insert function takes an element and a list and inserts the element into the list at the first position where the newly inserted element will be less than or equal to the next element. In particular, if the list is sorted before the insertion, the result will also be a sorted list.

   ```
   insert'' :: Ord a => a -> [a] -> [a]
   insert'' 2.5 [1,2,3] ⇒ [1.0,2.0,2.5,3.0]
   insert'' 2.5 [3,2,1] ⇒ [2.5,3.0,2.0,1.0]
   insert'' 2 [1..] ⇒ [1,2,2,3...
   ```

   (b) Implement a simple variant of the insertion sort algorithm using fold and the previously defined `insert''` function.

   ```
   insertionSort :: Ord a => [a] -> [a]
   insertionSort ⇒ Data.List.sort
   ```

Corrections

```
v2:  5: Added (Fractional a) to signature.
     7: Fixed order of arguments in first example.
v3:  5: Add Ord to signatures.
     6: Rename insert to insert' and union to union'.
     8: Rename insert' to insert''.
v4:  5: Remove [1..5] from tests, to not require Enum instance.
v5:  5: Replace mod with (/).
```