Programming in Haskell – Homework Assignment 4

UNIZG FER, 2017/2018

Handed out: November 7, 2017. Due: November 14, 2017 at 23:00

1 Instructions

1. To submit your homework you need to have a folder named after your JMBAG. In that folder there should be two files, `Homework.hs` for homework problems and `Exercises.hs` for all in-class exercises (yes, you need to submit those as well). You should ZIP that whole folder and submit it through Ferko.

   Example folder structure:

   - 0036461143
     - Homework.hs
     - Exercises.hs

   You can download the homework template file from the FER web repository.

2. If you need some help with your homework or have any questions, ask them on our Google group.

3. Define each function with the exact name and type specified. You can (and in most cases you should) define each function using a number of simpler functions.

4. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values (must be total). Use the `error` function for cases in which a function should terminate with an error message.

5. Problems marked with a star (⋆) are optional.

2 Grading

Each problem is worth a certain number of points. The points are given at the beginning of each problem or sub-problem (if they are scored independently).

These points are scaled, together with a score for the in-class exercises, if any, to 10.

Problems marked with a star (⋆) are scored on top of the mandatory problems, before scaling. The score is capped at 10, but this allows for a perfect score even with some problems remaining unsolved.

3 Problems

1. *(5 points)* Let's take a look at the `fix` higher-order function from the `Data.Function` module.

```
fix :: (a -> a) -> a
fix f = let x = f x in x
```

It looks strange, but if you think about it for a moment, it seems like it would just result in an infinite number of function applications:

```
fix f = x = f x = f (f x) = f (f (f x))) = ...
```

But because Haskell is non-strict, this doesn't necessarily happen. For example: `fix (1+)` diverges, because it expands to:

```
fix (1+) = x = 1+x = 1+(1+x) = ...  = 1+(1+(1+...(1+x)...))  = ...
```

But `fix (const 1)` doesn't, because:

```
fix (const 1) = x = const 1 x
```

and then because `const` ignores its second argument, it returns `1`. What about `fix (1:)`? Does it diverge, or does it produce something?

`fix` is introduced into typed lambda calculus as a primitive which allows you to define recursive functions. Let's see how that works on an example:

```
sumTo n = if n == 0 then 0 else n + sumTo (n - 1)
sumTo' n = fix (\rec x -> if x == 0 then 0 else x + rec (x - 1)) n
```

The first parameter of the lambda function in `sumTo'` is the lambda function itself, given by `fix`. So the function gets to recursively call itself even though it's an anonymous function!

Try to define a few recursive functions and structures, with just `fix` and lambda functions. You'll be awarded 0.7 points for every definition.

You can find more information on this link.

(a) (0.7 pts) define non accumulator style `factorial`:

```
factorial ::  (Num a, Eq a) => a -> a
```

(b) (0.7 pts) define non accumulator style `sum'`

```
sum' ::  Num a => [a] -> a
```

(c) (0.7 pts) define accumulator style `factorial'`

```
factorial' :: (Num a, Eq a) => a -> a
```

(d) (0.7 pts) define accumulator style `sum''`

```
sum'' :: Num a => [a] -> a
```

(e) (0.7 pts) define list of natural numbers

```
nats :: [Integer]
```

(f) (0.7 pts) define `map'`

```
map' :: (a -> b) -> [a] -> [b]
```

(g) (0.7 pts) define `zip'`

```
zip' :: [a] -> [b] -> [(a, b)]
```

2. *(3 point)* In this problem we interpret lists with distinct elements as sets. The order of elements within lists is irrelevant.

   (a) Write a function:

   ```
   subsets ::  Int -> [a] -> [[a]]
   ```

   which generates a list of all k-sized subsets of the given set.
   Some examples:

   ```
   subsets 0 [1,2,3] ⇒ [[]]
   subsets 1 [1,2,3] ⇒ [[1],[2],[3]]
   subsets 1 [1,2,2,3] ⇒ [[1],[2],[3]]
   subsets 2 [1,2,3] ⇒ [[1,2],[1,3],[2,3]]
   subsets 2 [1,2,2,3,3] ⇒ [[1,2],[1,3],[2,3]]
   subsets 3 [1,2,3] ⇒ [[1,2,3]]
   subsets 100 [1,2,3] ⇒ []
   ```

   (b) A partition of a non-empty set S is a family $A_i : i \in I$ of its non-empty subsets such that the sets A_i are pairwise disjoint and their union is equal to S.
   Write a function:

   ```
   partitions ::  [a] -> [[[a]]]
   ```

   which generates a list of all partitions of the given set. Each partition is represented as a list of lists, the sub lists being the disjoint subsets of the given set.
   Some examples:

   ```
   partitions [] ⇒ error
   partitions [1] ⇒ [[[1]]]
   partitions [1,2] ⇒ [[[1,2]],[[1],[2]]]
   partitions [1,2,3]
   ⇒ [[[1,2,3]],[[1],[2,3]],[[1,2],[3]],[[2],[1,3]],[[1],[2],[3]]]
   ```

   To help you with checking the correctness of your functions, the number of partitions of a n-sized set is called the n-th Bell number.

3. *(2 points($\star$))* Define a function `permutations'` that, given a list, returns a list of all its permutations. Implement it using explicit recursion. The ordering of the list of permutations is irrelevant!