W20 21 - Prüfung

Funktionale Programmierung (Technische Universität München)

Path to Success

We consider the following data types modelling a simple file system and file paths. For simplicity, our files will be named using `Int`s:

```
type FileName = Int
```

A file path is then either a single file name or a file name and a trailing file path:

```
data Path = FileName :/: Path | Node FileName
  deriving (Eq, Show)

-- make `:/:` bind to the right, e.g. `0 :/: 2 :/: Node 3 = 0 :/: (2 :/: Node 3)`
infixr 5 :/:
```

Remember that `(:/:) :: FileName -> Path -> Path` is just an infix constructor. For example, `1 :/: Node 2` is equivalent to `(:/:) 1 (Node 2)`.

Now a file is either a plain file (i.e. not a directory) with a name and some content or a directory with a name and a list of files contained in the directory.

```
data File a = File FileName a | Directory FileName [File a]
  deriving (Eq, Show)
```

In all what follows, you can assume that a directory does not contain two direct children with the same file name. For example, the directory

```
Directory 0 [File 1 "FPV", File 1 "Rocks"]
```

is invalid and will not be passed to your functions.

1. [2P] Implement a function `allPaths :: File a -> [Path]` such that `allPaths f` returns all valid paths in `f` (in no particular order). For example:

```
allPaths (Directory 0 [File 2 "Haskell", File 1 "Love"]) = [Node 0, 0 :/: Node 2, 0 :/: Node 1]
allPaths (Directory 0 [Directory 2 []]) = [Node 0, 0 :/: Node 2]
```

2. [3.5P] Implement a function `replaceAt :: File a -> Path -> a -> Maybe (File a)` such that `replaceAt f p c` returns an updated version of `f` where the content of the plain file at path `p` is changed to `c`. If there is no plain file to be changed at the given path, `Nothing` should be returned. For example:

```
replaceAt (Directory 0 [Directory 2 [File 3 "FPV"], File 1 "Love"]) (0 :/: 2 :/: Node 3) "Turtle"
  = Just (Directory 0 [Directory 2 [File 3 "Turtle"], File 1 "Love"])
replaceAt (Directory 0 [Directory 2 [File 3 "FPV"], File 1 "Love"]) (0 :/: Node 2) "Oops"
  = Nothing
```

Sharing is Caring

Evaluate the following expression as far as possible using the Haskell-like evaluation strategy described in the lecture. If needed, indicate infinite reductions by "..." as soon as nontermination becomes apparent and leave a remark why the reduction does not terminate.

Given the functions

```
fp :: (a -> a) -> a
fp f = f (fp f)

superConst :: Int -> a -> a
superConst n m = fp aux n m
  where
    aux _ 0 = (\x -> x)
    aux f n = (\x y -> x (x y)) (f (n - 1))
```

Evaluate `superConst 1 2`.

Observational Equivalence

Decide if the following statement is true or false. If it is true, give a brief justification why. If it is false, give a counterexample and a brief justification why your counterexample is correct.

Consider the following reverse function:

```
rev :: [a] -> [a]
rev [] = []
rev (x:xs) = rev xs ++ [x]
```

Then the terms `rev (rev xs)` and `xs` are *observationally equivalent*. Two terms are said to be observationally equivalent if and only if they can be replaced by each other in any program `p` without changing the output of `p`.

Winner, Winner, Chicken Dinner

We have a big tournament going on. To make it managable, the participants are divided into groups. In each group, the participant with the highest score wins. We are interested in the highest score overall.

Your task is to write a `main` function that outputs (on separate lines) the winner of each group and the highest score overall.

The input is given in the following format:

- The first line contains an integer `n`. Then, `n` blocks each describing a group follow.
- The first line of a block is an integer `b`. The block then consists of `b` additional lines.
- Each line consists of the participant's score and their name by a space, e.g. `8 Niklas`.

You may assume that the scores lie between 1 and 100, that the names of a participants in a group are unique, and that each group has exactly one winner.

Here is an example interaction with the program (lines starting with `>>` are inputs from the user):

```
>> 2
>> 3
>> 8 Niklas
>> 3 Joshua
>> 10 Serge
Serge
>> 2
>> 7 Leroy
>> 5 Leon
Leroy
10
```

Hippety, Hoppety, what is the Property?

In this exercise, we consider run-length encoded lists for types `a` that are an instance of the typeclass `Eq`. The run-length encoding represents a list of that type as a list of pairs, i.e. a list of type `[(a, Int)]`. An element `(x, i)` of such a list encodes `i` consecutive occurences of `x`. As an example consider the run-length encoding `[('H', 1), ('e', 1), ('l', 2), ('o', 3)]` which represents the list `['H','e','l','l','o','o','o']`.

The run-length encoding is valid if there are no consecutive elements with the same first component and if all second components are strictly positive. To check this property, the template provides the function `validRLE`.

Your tasks is to write a QuickCheck test `prop_permutationsRLE` that tests the function `permutationsRLE`. The test should be complete, i.e. every correct implementation of `permutationsRLE` passes the test and for every incorrect implementation, there is at least one test that fails for suitable test parameters.

Similarly to the function `permutations` from `Data.List`, the function `permutationsRLE` should return all permutations of the input list with the distinction that it takes a run-length encoded list as the input and returns a list of run-length encoded permutations. The order does not matter and duplicates are allowed. A list `ys` is a permutation of a list `xs` if we can obtain `ys` by rearranging the elements of `xs`. For example, `"acb"`,`"bac"`, and `"abc"` are all permutations of `"abc"` while `"aab"` is not a permutation of `"abc"`.

Important: It is not required to implement the function `permutationsRLE`.

You are given the following definitions:

```
f1 :: (Integer -> Integer -> Integer) -> Integer -> [Integer] ->
Integer
f1 f z []     = z
f1 f z (x:xs) = f1 f (f z x) xs

f2 :: Integer -> [Integer] -> Integer
f2 a [] = a
f2 a (x:xs) = f2 (a+x) xs
```

Prove the following lemma using structural induction:

```
f2 a xs = f1 (+) a xs
```

Proof 2

You are given the following definitions:

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

reverse :: [a] -> [a]
reverse [] = []
reverse (x : xs) = reverse xs ++ [x]

fr :: Integer -> [Integer] -> Integer
fr z []       = z
fr z (x : xs) = x + fr z xs
```

Prove the following lemma using structural induction:

```
fr z (reverse xs) = fr z xs
```

If you need an auxiliary lemma, prove that lemma as well.

You can assume that `+` is associative. Thus you can just write `x + y + z`, without parantheses. You may also use the usual properties of `+` without justification but as an explicit proof step. For example, the equation `y + 1 + x = x + y + 1` is a correct proof step.

A shapely exercise

Your template contains a data type `Shape` which represents rectangles and right-angled triangles. A rectangle of height `h` and width `w` is constructed as `Rectangle h w`, and a triangle of height `h` and base length `b` is constructed as `Triangle h b`.

1. [1.5P] Instantiate the class `Area` with the `Shape` data type. The `area` function should compute the area of the given object, whereas `scale sh f` should multiply each component of `sh` (height and width in the case of rectangles, height and base length in the case of triangles) by `f`.
2. [2P] We now want to handle lists of objects from the `Area` class, e.g. shapes. Write an instance for `Area` that works for any list where the element type has an `Area` instance. In this case `area` should compute the sum of the areas of the list elements, and `scale` should scale each element in the list .

You may assume that any `Float` value given to the constructors of `Shape` or to the `scale` function is positive.

Do It Once, Do It Twice, Do It Thrice!

You are organizing a board game tournament, and need to write some sophisticated software to pair players against each other.

Your pairing function takes as its first argument a list of game results in the form of a triple `(player1, player2, result)`, where the players are strings, and `result` is either `1` if `player1` has won, `-1` if `player2` was victorious, or `0` if there was a tie.

The second argument is simply a list of player names, which are supposed to be paired against the winning players from the first list. The order of the pairings should be the same as in the original lists, i.e. the first winning player from the first argument should be paired against the first player from the second argument and so on. In case of a tie, neither player is paired.

To illustrate, here is an example:

```
pairings [("Jeff","Liz",1),("Dean","Ruth",0),("Tom","Kate",-1),("Rita","John",1)] ["Dave","Lucy","Nancy","Rick"] = [("Jeff","Dave"),("Kate","Lucy"),("Rita","Nancy")]
```

As you can see in the example, the second argument may contain more elements than are necessary. You may assume that the input is well-formed, i.e. that the result is always `0`,`1` or `-1` and that the second list always contains enough elements.

Implement the function in three different ways:

1. [2P] As a recursive function with the help of pattern matching. You are not allowed to use list comprehensions or higher-order functions. Call this function `pairingsA`.
2. [2P] As a list comprehension without using any higher-order functions or explicit recursion. Call this function `pairingsB`.
3. [2P] Use higher-order functions (e.g. `map`, `filter`, etc.) but no recursion or list comprehensions. Call this function `pairingsC`.