PROGRAMMING III (COMP 2209)
SEMESTER ONE, 2019

Home

Exercises for Assessment

Here is the collected list of exercises for assessment as they appear on the weekly sheets.

Your solutions are due by 4pm 5th Dec but may be submitted at any point before that.

Repeat multiple submissions are allowed.

You will be given feedback on how well your solutions perform against our test suites. However, marks will only be given to your final submission after the 5th Dec. These exercises comprise 20% of the overall assessment for the module.

Please read the Submission Instructions carefully and make use of the automated feedback system.

Exercise A1

Define a function `histogram :: Int -> [Int] -> [Int]` such that `histogram n xs`, say, returns a list that counts the frequency of values drawn from `xs` in the ranges `0..n-1`, `n..2n-1`, `2n..3n-1` and so on, until all values in `xs` have been counted. For example, `histogram 5 [1, 2, 10, 4, 7, 12] = [3, 1, 2]`.

(2 marks)

Exercise A2

Use the Babylonian method to define a function

```
approxSqrt :: Double -> Double -> Double
```

that takes two `Double` values, `d` and `epsilon`, and returns an approximation to the real square root of `d`. You should generate an infinite sequence of approximations starting with `x_(0) = 1` and where each subsequent element of the sequence is the next approximation given by

```
x_(n+1) = (x_(n) + d / x_(n)) / 2
```

You should return the first approximation in this sequence where the size of the difference between this approximation and the real square root of `d` is strictly less than `epsilon`.

(2 marks)

Exercise A3

Given a list `ss`, we say it is a subsequence of a list `ts` if and only if `ss` can result from deleting zero or more, not necessarily contiguous, elements of `ts`.

Define a function `longestCommonSubsequence :: Eq a => [[a]] -> [a]` that, given a list of lists `xss`, returns the longest list `ys` sucn that `ys` is a subsequence of `xs` for all `xs` that are contained in `xss`.

For example,

```
longestCommonSubsequence [[1,2,3], [0,1,3],
[1,3,4]] = [1,3]
```

Note that, according to the definition above, `[1,3]` is a subsequence of `[1,2,3]` because elements of the subsequence do not need to occur in neighbouring positions of the supersequence. Your implementation should satisfy the equation `longestCommonSubsequence [] = []`. This is valid because any list is a subsequence of all members of an empty list.

(2 marks)

Exercise A4

A metric is a function that takes two points in some space and returns a non-negative real value indicating the "distance" between them. This is subject to some properties (see e.g. Wikipedia).

We will represent metrics as functions of type

```
type Metric a = Point a -> Point a -> Double
```

where

```
type Point a = (a,a)
```

Define a function `neighbours :: Int -> Metric a -> Point a -> [Point a] -> [Point a]` such that `neighbours k d p xs` returns a list of the `k` nearest neighbours, in distance order, according to metric `d` to point `p` in the list `xs` of points. If there are fewer than `k` elements in the list `xs`, then all should be returned as suitable neighbours. If two points in the list `xs` are equally close to `p`, the point listed earlier in the list should be returned before the later.

(2 marks)

Exercise A5

Given a binary predicate **P** on a set **S** (that is, a subset of **S x S**) we say that a function **B** from **S** to itself is a **Bonding** if it is total, symmetric, and, for all **x**,**y** in **S**, **B(x) = y** implies that **y** does not equal **x** and furthermore **P(x,y)** holds. In other words, each element in **S** is paired with exactly one other element such that the pairs satisfy the predicate.

In this exercise you will need to write a function that finds Bondings. We will model the input predicate **P** in Haskell as a function of type `(a -> a -> Bool)`. We will model the set **S** as a list of type `[a]`. As Bondings do not always exist for a given **P** and **S**, we will return a Bonding as a `Maybe` type of a list of pairs `[(a,a)]`.

Define a function

```
findBonding :: Eq a => (a -> a -> Bool) -> [a] -
> Maybe [(a,a)]
```

such that `findBonding p xs` returns any Bonding of the list `xs` for predicate `p`. You may assume that `xs` does not contain duplicate elements.

For example, `findBonding (\x -> \y -> odd(x+y)) [2,3,4,5,6,7]` may return

```
Just [(2,3),(3,2),(4,5),(5,4),(6,7),(7,6)]
```

as a valid answer. Whereas `findBonding (\x -> \y -> even(x+y)) [2,3,4,5,6,7]` should return `Nothing` as no Bonding exists in this case.

(2 marks)

Exercise A6

Consider the tree data type

```
data VTree a = Leaf | Node (VTree a) a Int
(VTree a)
```

which is suitable for representing Binary Trees with the extra property that each node in the tree carries a counter of how many times that node has been visited.

Consider also the Zipper type given by

```
data Direction a = L a Int (VTree a) | R a Int
(VTree a)
type Trail a = [Direction a]
type Zipper a = (VTree a, Trail a)
```

that can be used to represent a tree with a *current node*. In zipper state `(t,trail)` the current node is the root node of subtree `t` and the rest of the tree (i.e. that part of the tree not below `t`) is represented in `trail`.

We are going to represent Sets as Binary Search Trees using this Zipper type. To do this implement a function

```
insertFromCurrentNode :: Ord a => a -> Zipper a
-> Zipper a
```

that takes a value and a zipper and inserts a new node containing the given value in to the tree by navigating **from the current node** to the appropriate point in the tree. If the value is already present in the tree you do not need to insert a new node. You should not navigate through more of the tree than is necessary to insert the given value. Recall that a Binary Search Tree has the property that an inorder traversal of the tree will result in a sorted list.

As part of the implementation of `insertFromCurrentNode` you will need to increment the counter in each node, each time you visit that node of the tree. Freshly created nodes should have a visit count of 1. Starting evaluation of `insertFromCurrentNode` at the current node does not count as a visit to the current node.

For example, starting with the empty zipper `(Leaf, [])` calling

```
insertFromCurrentNode 2 $ insertFromCurrentNode
4 $ insertFromCurrentNode 3 (Leaf,[])
```

should result in the tree `Node (Node Leaf 2 1 Leaf) 3 2 (Node Leaf 4 1 Leaf)` with the current node being the one holding value 2. Note the visit count in the root node is 2 as it has been visited twice, once upon creation of the node, and once while it was visited during the insertion of value 2 starting from the node containing value 4.

When testing your code you may find the following function useful:

```
mkTree :: Ord a => [a] -> Zipper a
mkTree = foldl (\z -> \x ->
insertFromCurrentNode x z) (Leaf,[])
```

This takes a list of values and inserts them, in list order, in to the empty zipper. I will be using this function in the automated test harness.

(3 Marks)

Exercise A7

We are going to write an evaluator for a simple stack-based machine called SM. Programs for SM are given as lists of instructions and the stack is represented as a list of `Int` where

```
data Instruction = Add | Mul | Dup | Pop
type Stack = [Int]
type SMProg = [Instruction]
```

The intended semantics of SM is as follows: `Add` pops the top two elements of the stack, adds them and pushes the result back on to the stack. `Mul` is similar but it multiplies the top two elements. `Dup` pushes another copy of the top element on to the stack and `Pop` removes the top element of the stack.

Write a function `evalInst :: Stack -> SMProg -> Stack` that takes a stack and an SM program and returns the resulting stack after evaluating the program on the given input stack.

(1 Mark)

We will use the stack machine SM from Exercise A7 in the next two questions.

Exercise A8

We say that a program for SM is a *reducer* for a given input stack S, if it only uses the instructions `Add`, `Mul` or `Pop` and, after evaluating the program on S, the resulting stack contains a single value.

A maximal reducer for a stack S is a reducer where the resulting single value is maximum among all possible reducers for S. That is, the reducer returns the highest possible value for that input stack.

Write a function called `findMaxReducers :: Stack -> [SMProg]` that returns all maximal reducers for the given input stack. As an example, `findMaxReducers [0,1,4,5]` should return `[[Add,Add,Mul],[Pop,Add,Mul]]` where each of these reducers leaves a stack of `[25]`, it being the largest possible value.

(3 Marks)

Exercise A9

Suppose the initial stack has a singleton initial value [x], then evaluating the instruction sequence `[Dup, Mul]` will leave the single value `x^2` on the stack. Likewise, `[Dup, Dup, Dup, Mul, Mul, Mul]` will produce `x^4`. However, for the latter example there is a shorter program that computes this result: `[Dup, Mul, Dup, Mul]`.

Write a function `optimalPower :: Int -> SMProg` that, given an input value of a positive integer N say, returns a SM program that will transform a stack `[x]` to the stack `[x^N]` for any value for `x`. This returned SM program must be as short as possible.

(3 Marks)

Page maintained by Julian Rathke.