Homework 04

Due 2/20/2019

Purpose:

The purpose of this homework assignment is to help you become a better Haskell programmer and to implement applications of regular languages in Haskell..

Like last week I want you to turn in a file Hmwk4.pdf with all the answers to the homework, but also to pull out your Haskell programs and turn them in a text file named Hmwk4.hs. The first line of that file should be

```
module Hmwk4 where
```

The next line should be a comment with your name. The rest of the file should be your Haskell code. We will take that file and automatically test it by running your code on our test data. As a result, you should be sure that your code compiles properly (i.e., running "ghci Hmwk4.hs" should compile your code without errors and it should be possible to run your code on my test cases (ideally without it blowing up when executed!).

The Hmwk4.hs file will be used for automated testing of your code. Likely, no human will read it. Thus if you don't include that code in the Hmwk4.pdf file, you will likely not get any credit for it. Please remember this!

While your Hmwk4.pdf file will be turned in to gradeScope as usual, you will need to turn in the file Hmwk4.hs via `https://submit.cs.pomona.edu/2019sp/cs101`.

Be sure to test your programs one last time before submitting. I've seen students mess up when commenting code in a way that causes everything to break. Code that doesnt compile will get very little credit. We will be using some automatic testing, and code that doesnt compile brings everything to a halt. With a large class we will not have time to go in to manually tweak your code to make it work.

IMPORTANT: When you write the functions requested below, please make sure that they have the exact names and types specified in the question. If you make an error, your program will crash on our test suite and you will get very little credit.

Be sure that all of my sample code works, and worry about edge cases that might cause your program to give the wrong answer as I will test it more thoroughly.

space2 ex

Tasks:

**Problems from the texts are given in the form c.n(k) where c is the chapter and n is the problem number, and k is the sub-problem number. Thus problem 2.7(d) is problem 7(d) from Chapter 2.*

1. (0 points) **Academic Honesty Statement**
2. (15 points) **Basic Functions**

   (a) **Zipping and Unzipping**

   The function `zip` to compute the pairwise interleaving of two lists of arbitrary length is predefined, but I'd like you to write it from scratch anyway (calling it `zip'`). You should use pattern matching to define this function. The function should have type:

   ```
   zip':: [t] -> [t1] -> [(t, t1)]
   ```

   ```
   -> *Main> zip' [1,3,5,7] ["a","b","c","de"]
   [(1,"a"),(3,"b"),(5,"c"),(7,"de")]
   it :: [(Integer, [Char])]
   ```

Note: If the lists don't have the same length, you may decide how you would like the function to behave. If you don't specify any behavior at all you will get a warning from the compiler that you have not taken care of all possible patterns— this is fine.

Write the inverse function, `unzip'`, which behaves as follows:

```
unzip' :: [(s, t)] -> ([s], [t])

*Main> unzip' [(1,"a"),(3,"b"),(5,"c"),(7,"de")]
([1,3,5,7],["a","b","c","de"])
it :: ([Integer], [[Char]])
```

Again, `unzip` is built-in, but you will write your own `unzip'`.

Write `zip3'`, to zip three lists.

```
zip3' :: [t] -> [t1] -> [t2] -> [(t, t1, t2)]

*Main> zip3' [1,3,5,7] ["a","b","c","de"] [1,2,3,4]
[(1,"a",1),(3,"b",2),(5,"c",3),(7,"de",4)]
it :: [(Integer, [Char], Integer)]
```

Once again, `zip3` is built-in, but you will write your own `zip3'`.

Why can't you write a function `zip_any` that takes a list of any number of lists and zips them into tuples? From the first part of this question it should be pretty clear that for any fixed n, one can write a function `zipn`. The difficulty here is to write a single function that works for all n! I.e., can we write a single function `zip_any` such that `zip_any [list1,list2,...,listk]` returns a list of k-tuples no matter what k is?

(b) **find**

Write a function find that takes a pair of an element and a list and returns the location of the first occurrence of the element in the list (or -1 if it doesn't occur).

```
find :: (Eq a, Eq a1, Num a1) => (a, [a]) -> a1
```

The prefix `(Eq a, Eq a1, Num a1) =>` indicates that the type a of elements of the list must support equality (after all you must check the first argument to see if it is equal to any of the elements of the list). The return type must be some kind of a numeric type as it indicates where in the list the element is found.

For example:

```
*Main> find(3, [1, 2, 3, 4, 5])
2
*Main> find("cow", ["cow", "dog"])
0
*Main> find("rabbit", ["cow", "dog"])
-1
```

First write a definition for find where the element is guaranteed to be in the list. Then, modify your definition so that it returns `-1` if the element is not in the list.

3. (10 points) **Trees**

Here is the datatype definition for a binary tree storing integers only at the leaves (it was also discussed in class):

```
data IntTree = Leaf Integer | Interior (IntTree,IntTree) deriving Show
```

Write a function `treeSum:IntTree` → `Integer` that adds up the values in the leaves of a tree:

```
*Main> treeSum(Leaf 3)
3
*Main> treeSum(Interior (Leaf 2, Leaf 3))
5
*Main> treeSum(Interior(Leaf 2, Interior(Leaf 1, Leaf 1)))
4
```

Write a function `height : IntTree` → `Integer` that returns the height of a tree:

```
*Main> height(Leaf 3);
0
*Main> height(Interior(Leaf 2, Leaf 3));
1
*Main> height(Interior(Leaf 2, Interior(Leaf 1, Leaf 1)));
2
```

(Again the system gives me a more general type for my function: `(Num a, Ord a) => IntTree -> a`.)

Write a function `balanced: IntTree` → `Bool` that returns true if a tree is balanced (i.e., both subtrees are balanced and differ in height by at most one). You may use your height function above.

```
*Main> balanced(Leaf 3);
True
*Main> balanced(Interior(Leaf 2, Leaf 3));
True
*Main> balanced(Interior(Leaf 2, Interior(Leaf 3, Interior(Leaf 1, Leaf 1))));
False
```

Is your implementation as efficient as possible? What is wrong with using the `height` function in the definition of `balanced`? How would you write `balanced` to be more efficient? (You need not write code, but describe how you would do this.)

4. (10 points) **Stack Operations**

Certain programming languages (and calculators) evaluate expressions using a stack. As some of you may know, PostScript is a programming language of this ilk for describing images when sending them to a printer. We are going to implement a simple evaluator for such a language. Computation is expressed as a sequence of operations, which are drawn from the following data type:

```
data OpCode = Push Float | Add | Mult | Sub | Div | Swap deriving Show
```

The operations have the following effect on the operand stack. (The top of the stack is shown on the left.)

OpCode	Initial Stack	Resulting Stack
`Push(r)`	...	r ...
`Add`	a b ...	(b + a) ...
`Mult`	a b ...	(b * a) ...
`Sub`	a b ...	(b - a) ...
`Div`	a b ...	(b / a) ...
`Swap`	a b ...	b a ...

The stack may be represented using a list for this example, although we could also define a stack data type for it.

```
type Stack = [Float]
```

Write a recursive evaluation function with the signature

```
eval :: ([OpCode], Stack) -> Float
```

It takes a list of operations and a stack. The function should perform each operation in order and return what is left in the top of the stack when no operations are left. For example,

```
eval([Push 2.0, Push 1.0, Sub],[])
```

returns 1.0. The `eval` function will have the following basic form:

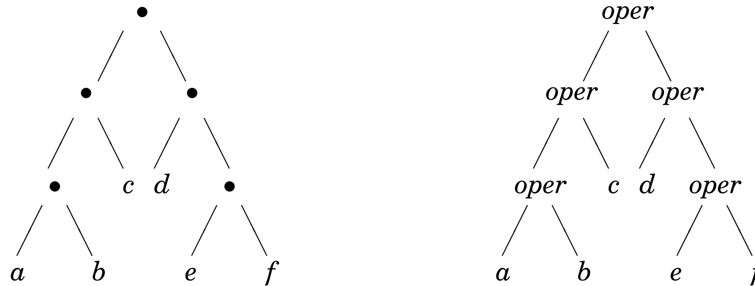
```
eval ([],          a:rest)   = --
eval ((Push n):ops, rest)    = --
--
eval (_,             _)      = 0.0;
```

You need to fill in the blanks and add cases for the other opcodes.

The last rule handles illegal cases by matching any operation list and stack not handled by the cases you write. These illegal cases include ending with an empty stack, performing addition when fewer than two elements are on the stack, and so on. You may ignore divide-by-zero errors for now (or look at exception handling in one of the tutorials – we will cover that topic in a few weeks).

5. (10 points) **Haskell Reduce for Trees** In an earlier problem we defined a datatype IntTree for trees whose leaves hold Integers. In this questions we define trees that can hold leaves of any fixed type:

```
data Tree a = ALeaf a | Node (Tree a) (Tree a)
```



(a) Write a function

```
reduceTree :: (a -> a -> a) -> Tree a -> a
```

that combines all the values of the leaves using the binary operation passed as a parameter. In more detail, if `oper :  a -> a -> a` and `t` is the nonempty tree on the left in this picture, then reduce `oper t` should be the result obtained by evaluating the tree on the right. For example, if f is the function

```
f :: Int -> Int -> Int
f x y = x + y
```

then `reduceTree f (Node (Node (ALeaf 1) (ALeaf 2)) (ALeaf 3)) = (1 + 2) + 3 = 6`. Explain your definition of `reduce` in one or two sentences. (Notice that this is slight generalization of a function you wrote earlier for integer trees.)

(b) Write a function `treeToList :: Tree a -> [a]` that returns a list of the elements in the argument tree in the same order left-to-right order they occur in the tree. Thus for the sample tree, the result would be `[a,b,c,d,e,f]`.

(c) Write a function `reduceList :: (a -> a -> a) -> [a] -> a` that reduces a non-empty list according to the supplied function argument. Notice that the results of reduceList (-) [8,3,1] should be (8 - 3) - 1 == 4, not 8 - (3 - 1) = 6. You may assume that the list parameter has at least length 1 and that, when applied to a singleton list [x], simply returns x.

(d) For what kind of arithmetic operations, `f`, would you expect `reduceList f (treeToList tree) == reducetree f tree`?

6. (15 points) **Higher-Order Functions**

One of the advantages of functional languages is the ability to write high-level functions which capture general patterns. For instance, in class we defined the "`listify`" function which could be used to make a binary operation apply to an entire list.

(a) Your assignment is to write a high-level function to support list abstractions. The languages Miranda, Haskell, and Python allow the user to write list abstractions of the form:

```
[f(x) | x <- startlist; cond(x)]
```

where `startlist` is a list of type `a`, `f: a -> b` (for some type `b`), and `cond: a -> bool`. This expression results in a list containing all elements of the form f(x), where x is an element in the list "`startlist`", and expression "`cond(x)`" is true. For example, if sqr(x) = x*x and odd(x) is true iff x is an odd integer, then

```
[sqr(x) | x <- [1,2,5,4,3], odd(x)]
```

returns the list `[1,25,9]` (that is, the squares of the odd elements of the list - 1,5,3). Note that the list returned preserves the order of `startlist`.

This function could have been defined from first principles in Haskell, using built-in Haskell functions like map, and filter. Do not use the list comprehension syntax of Haskell, as that makes the problem totally trivial! You are to write a function

```
listcomp :: (a -> b) -> [a] -> (a -> Bool) -> [b]
```

so that

```
listcomp f startlist cond = [f(x) | x <- startlist; cond(x)].
```

(Hint: One way to do this is to divide the function up into two pieces, the first of which calculates the list, `[x | x <- startlist; cond(x)]`, and then think how the "`map`" function can be used to compute the final answer. It's also pretty straightforward to do it all at once.)

(b) Test your function by writing a function getEmps which extracts the list of all names of managerial employees over the age of 60 from a list of employee records, each of which has the following fields: "`name`" which is a string, "`age`" which is an integer, and "`status`" which has a value of managerial, clerical, or manual. You will need to look up how records are used in Haskell, as I didn't talk about them in class. (Be sure to define this function correctly. I'm always amazed at the number of people who miss this problem by carelessness!)

(c) Generalize your function in part a to

```
listcomp2 g slist1 slist2 cond =
                    [g x y | x <- list1; y <- list2; cond x y]
```

Here `g` is to be applied to *all* combinations of elements from `list1` and `list2` that satisfy the condition given by `cond`.

7. (10 points) **Implementing DFSM** In the bottom half of file SimpleExampleFSM.hs (after the definition of decide) there are definitions of a data type FSM and a function gAccept that simulate an arbitrary DFSM. In this model, the states of the machine are numbers. The constructor for an FSM requires the programmer to specify the starting state, the set of transitions, and a set of accepting states. The transitions are represented by triple of the form (s, i, t), which is interpreted as if the machine is in state s with i as the next input, then it should transition to state t. See the definition of sampleFSM at the bottom of the file. You can run the simulator on machine sampleFSM and input "abaabbb" by writing

```
gAccept sampleFSM "abbababb"
```

Please write an encoding of a deterministic finite automata that accepts all strings that contain an even number of a's or an odd number of b's. [*Originally I asked you to write an encoding of a different DFSM, but I accidentally put up the solution. Hence the new problem! I left the solution to the original in the file SimpleExampleFSM.hs.*] I suggest you design the DFSM graph

first and then encode it. Please name your DFSM eaobFSM. My testing code will invoke `gAccept eaobFSM w` for a variety of strings w to make sure it accepts the desired language.

[You may find the function Set.fromList to be helpful – look it up in the Haskell libraries on-line.]

Do not copy over all of my code in SimpleExampleFSM.hs. Instead have your program file start with

```
import SimpleExampleFSM
```

If you then copy the SimpleExampleFSM.hs file into the same directory as your program, your code should be able to access my types and functions.

Criteria:

Your assignment will be graded based on the correctness and the completeness of your solutions. Please make sure that you use the correct terminologies when you write proofs or describe a structure, such as CFG, PDA, etc.

Submission Guideline:

Please edit your HW following the editing guideline, especially the instructions on the number of pages each question should occupy. The text in blue are instructions; they should be either removed, or replaced with proper contents and turned into black. Please submit your homework solutions online via gradescope.