# Homework 2

## Due Thursday, 2/7/2019

**Important**: For this assignment I would like you to turn in a zipped folder with two files. The first file should be a LaTeX document with the complete solutions to all problems (including code). The second file should have suffix "hs" and contain all of the Haskell programs you are writing for this assignment. All functions should be named exactly as specified in this assignment. We will be testing your code by loading this file and running our code on it. The first line of your Haskell file should be:

```
module Hmwk2 where
```

Putting this line in will allow us to automatically test your program. Leaving it out will result in all of our tests failing, and you receiving a correspondingly low score, making all of us sad! As usual, test your program by loading the "hs" file once you have fully commented your code.

Name your two files `Hmwk2.hs` and `Hmwk2.pdf`. Be sure to have your name in each of the two files so we know who produced each.

When you are ready to turn in your homework, compress your folder into a zip file. Then go to `https://submit.cs.pomona.edu/2019sp/cs131` and click on the assignment 2 on that page and follow the instructions to submit your work.

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.

1. **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.)

2. **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.

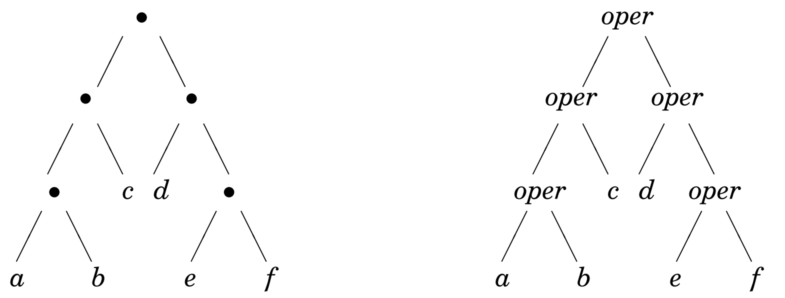
The last rule handles illegal cases by matching any operation list and stack not handled by the cases you write. These illegal cases include ending with an empty stack, performing addition when fewer than two elements are on the stack, and so on. You may ignore divide-by-zero errors for now (or look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

3. (10 points) **Haskell Reduce for Trees**

   The binary tree datatype

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

   describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node).



   (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 last week for integer trees.)

(b) Write a function `toList :: 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 (toList tree) == reducetree f tree`?

4. (20 points) **Currying**

This problem asks you to show that the Haskell types `(a, b) -> c` and `a -> b -> c` are essentially equivalent.

(a) Haskell has predefined functions `curry` and `uncurry`. Inside ghci, type `:t curry` and then `:t uncurry` to see their types. While these functions are predefined, you could have defined them yourself in Haskell. Write down definitions of functions `curry'` and `uncurry'` with the same types as their unprimed counterparts. (Remarkably, these are the only functions with these types!)

(b) Assuming you got the definitions right, you should be able to prove that for all functions `f :: ((a, b) -> c)` and `g :: (a -> b -> c)`:

```
uncurry(curry(f)) = f
curry(uncurry(g)) = g
```

or in other words, that the functions are inverses of each other.

Write down the proof that these functions (i.e., the left and right sides of each of the equations) are the same.

*Hint*: To show that two functions p and q with the same domain D are the same, it is enough to show that for all elements x in D, p(x) = q(x). So, for example, to prove the first statement, you must show that for all x of type a and y of type b, `uncurry(curry(f))(x,y) = f(x,y)` You can complete the proof by expanding the left side and showing that you get the right side. Do something similar for the second equation, though you will need to provide two arguments.

5. (10 points) **Disjoint Unions**

Please do problem 5.7 from Mitchell, page 125 (page 116 in the new chapter, but both problems are the same!).

A quick summary of C unions for those who have not used them before:

The declaration

```
union IntString {
   int i;
```

```
        char *s;
    } x;
```

declares a variable `x` with type `union IntString`. The variable `x` may contain either an integer or a string value. (You may think of the type `char *` as being like `string` for this question.) To store an integer into `x`, you would write `x.i = 10`. To store a string, you would write `x.s = "moo"`. Similarly, we can read the value stored in `x` as an integer or a string with `num = x.i` and `str = x.s`, respectively. The expression `x.i` interprets and returns whatever value is in `x` as an integer, regardless of what was last stored to `x`, and similarly for `x.s`.

For part b of this problem, use the Haskell declaration:

```
data Union a b = TagA a | TagB b

data IntString = TagInt Int | TagString String

        x = if ... then TagInt 3 else TagString "here, fido"

        answer = let TagInt m = x in m + 5
```

Answer the problem using Haskell rather than ML. Contrary to what is hinted at in the problem, if you run this (filling in the . . . by a truth value) you will not get a compile-time warning. (In the ML version you do get the warning). How might the system know statically this is potentially unsafe and hence be able to give you such a warning?

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, except that you may use the built-in Haskell functions 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 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`.