Homework 4

Due Midnight, Thursday, 10/2/2014

Please turn in your homework solutions online at `http://www.dci.pomona.edu/tools-bin/cs131upload.php` before the time it is due.

Note that some of the problems on this assignment were "borrowed" from a similar course at Stanford taught by Kathleen Fisher and others.

1. (10 points) **Parsing** Please do problem 4.2 from Mitchell, page 83.

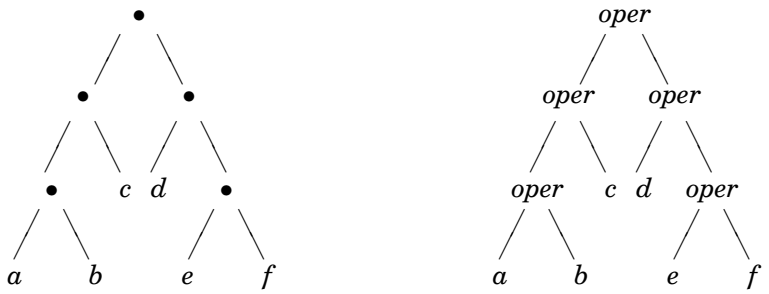
   Example 4.2 specifies that multiplication and division have higher precedence than addition and subtraction, and that operators of the same precedence are left associative, e.g., 6 - 2 - 1 is interpreted as being equivalent to (6 - 2) - 1 and *not* 6 - (2 - 1).

2. (10 points) **Haskell Reduce for Trees**

   The binary tree datatype

   ```
   data Tree a = Leaf 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 (Leaf 1) (Leaf 2)) (Leaf 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`?

3. (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.

4. (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
```

You should have no problem translating the rest of that part to Haskell. Answer the problem using Haskell rather than ML.

5. (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`.

6. (10 points) **Custom Haskell Control Structures**

You will find it extremely helpful to read Sections 1 and 2 (pp. 1 – 16) of Simon Peyton Jones' paper, "Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell" before tackling this problem. You will find a link to it next to the Lecture 6 notes.

One of the claimed advantages of first-class functions is the ability to write custom control structures. This question will explore this by asking you to write some familiar control structures.

Below, we provide code for the `whileIO` and `ifIO` control structures implemented in the IO monad. We also provide an example of its usage which prints the integers between 0 and 3 inclusive.

```
import Control.Monad.ST
import Data.IORef

ifIO :: IO Bool -> IO a -> IO a -> IO a
ifIO b tv fv = do { bv <- b;
                    if bv then tv else fv}

whileIO :: IO Bool -> IO () -> IO ()
whileIO b m = ifIO b
              (do {m; whileIO b m})
              (return ())

whileTest = do {v <- newIORef 0;
                whileIO (do{ x <- readIORef v;
                             return (x<4)})
                        (do{ x<-readIORef v;
                             print x;
                             writeIORef v (1+x) })}
```

IORef is in the standard Haskell library and supports mutable variables in the IO monad. It is described at http://www.haskell.org/ghc/docs/7.6.2/html/libraries/base/Data-IORef.html.

`untilIO` is a very similar control structure to `whileIO`, except the loop condition test 1) executes after the loop body and 2) causes loop exit if true instead of false, as is the case for `whileIO`. Please implement `untilIO` with the provided type signature and also create `untilTest` to print the integers between 0 and 3 inclusive.

```
untilIO:: IO () -> IO Bool -> IO ()
```

For this problem, please turn in the code given here along with the functions you write so that the TA's will be able to grade it without cutting and pasting code.