CSCE 314 [Section 502] Programming Languages – Fall 2016

Anandi Dutta

Assignment 3

Assigned on Friday, September 23, 2016

Electronic submission to eCampus due at **23:59, Wednesday, Oct. 05, 2016**

By electronically submitting this assignment to eCampus by logging in to your account, you are signing electronically on the following Aggie Honor Code:

"On my honor, as an Aggie, I have neither given nor received any unauthorized aid on any portion of the academic work included in this assignment."

In this assignment, you will practice (i) list comprehension, (ii) functional programming using *higher-order functions*, (iii) programmer defined data types in Haskell, and (iv) representing programs as their *abstract syntax trees* and evaluating them.

Below, you will find problem descriptions with specific requirements (for example, "Using `foldr`, define ..." means that using the `foldr` function in the definition is required). Read the descriptions and requirements carefully! There may be significant penalties for not fulfilling the requirements. You will earn total 120 points.

Note 1: This homework set is *individual* homework, not a team-based effort. Discussion of the concept is encouraged, but actual write-up of the solutions must be done individually.

Note 2: Submit electronically exactly one file, namely, *yourLastName-yourFirstName*-a3.hs, and nothing else, on eCampus.tamu.edu.

Note 3: Please make sure that the Haskell script (the .hs file) you submit compiles without any error when compiled using the Glasgow Haskell Compiler (ghc), version 7.4.2 that is installed in the departmental servers (linux.cse.tamu.edu and compute.cse.tamu.edu). If your program does not compile, there is a chance that you receive zero points for this assignment.

Note 4: Remember to put the head comment in your file, including your name, UIN, and *acknowledgements of any help received* in doing this assignment. You will get points deducted if you do not put the head comment. Again, remember the honor code.

Keep the name and type of each function exactly as given.

Part 1. List comprehensions/Higher Order Function

Problem 1. (10 points) Write a function that will delete leading white space from a string.
cutWhitespace [" x","y"," z"] ans: ["x","y","z"]
You are allowed to use any higher order function/List Comprehension(if you want to use) to solve this problem.

Problem 2. (10 points) Write a function that will take two list of lists, and multiply one with another. multList [[1,1,1],[3,4,6],[1,2,3]] [[3,2,2],[3,4,5],[5,4,3]] ans: [[3,2,2],[9,16,30],[5,8,9]]
You are allowed to use any higher order function/List Comprehension(if you want to use) to solve this problem.

You can use your own test case for Part 1(problem 1 and 2).
Part 2. Recursive functions, higher order functions

Problem 3. (8 points)

1. (8 points) Define a recursive function `mergeBy` that merges two sorted lists so that the resulting list is also sorted. Function `mergeBy`

   ```
   mergeBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]
   ```

   is different from `merge :: Ord a => [a] -> [a] -> [a]` in Exercise 4, in that it accepts three arguments, the first of which is a comparison function (of type `(a -> a -> Bool)`) that allows the resulting list to be sorted according to different criteria, for example, in an ascending order or in a descending order. Such a comparison function that returns a Boolean value (true or false) is called a *predicate*.

2. (8 points) Using `mergeBy` that you wrote above and `halves` that is discussed in class, define a recursive function `msortBy`. The problem specification stays the same as that for *msort* in Exercise 5, except the additional requirement of the first argument being a predicate. Thus, the type of `msortBy` is:

   ```
   msortBy :: (a -> a -> Bool) -> [a] -> [a]
   ```

3. (4 points) Using `msortBy`, define a merge sort function that sorts a list in an *ascending* order. The name and type of your function should be:

   ```
   mergeSort :: Ord a => [a] -> [a]
   ```

Problem 4. (5 points) Using `foldr`, define a function that multiplies all elements of a list. Multiplying the empty list should return 1.

```
multiply :: [Int] -> Int
```

Problem 5. (5points) Using `foldl`, define a function that concatenates all strings that are elements of a list.

```
concatenate :: [String] -> String
```

Problem 6. (10 points) Using `map`, `filter`, and `.` (function composition operator), define a function that examines a list of strings, keeping only those whose length is odd, converts them to upper case letters, and concatenates the results to produce a single string.

```
concatenateAndUpcaseOddLengthStrings :: [String] -> String
```

You need to `import Data.Char` in order to use the `toUpper` function (see the skeleton code).

Problem 7. (10 points) This problem has two parts:

1. (7 points) Using guards, implement a function myInsert that behaves the same way as the `insert` function defined in Data.List package. The Data.List.insert function takes an element and a list and inserts the element into the list at the first position where it is less than or equal to the next element. In particular, if the list is sorted before the call, the result will also be sorted.

   ```
   myInsert :: Ord a => a -> [a] -> [a]
   ```

2. (3 points) Using `myInsert` and `foldr`, implement insertion sort.

   ```
   insertionSort :: Ord a => [a] -> [a]
   ```

Problem 8. ($5 + 5 = 10$ points) Using `foldr1`, define `maxElem` that finds a maximal element in a list.

```
maxElem :: Ord a => [a] -> a
```

Explain why we are using `foldr1` instead of `foldr` (do some research on different fold functions). Put your explanation as a comment right before the definition of `maxElem`.

Part 3: Data types, type classes
Consider the following data type.

```
data Tree a b = Branch b (Tree a b) (Tree a b)
              | Leaf a
```

Problem 9. (10 points) Make `Tree` an instance of `Show`. Do not use `deriving`; define the instance yourself. Make the output look somewhat nice (e.g., indent nested branches).

Problem 10. (10 points) Implement the two functions that traverse the tree in the given order collecting the values from the tree nodes into a list:

```
preorder  :: (a -> c) -> (b -> c) -> Tree a b -> [c]
inorder   :: (a -> c) -> (b -> c) -> Tree a b -> [c]
```

Notice that the data type `Tree` can store different types of values in the leaves than on the branching nodes. Thus, each of these functions takes two functions as arguments: The first function maps the values stored in the leaves to some common type `c`, and the second function maps the values stored in the branching nodes to type `c`, thus, resulting in a list of type `[c]`.

Part 4: A tiny language

Let E (for *expression*) be a tiny programming language that supports the declaration of arithmetic expressions involving only addition and multiplication, and equality comparisons on integers. Here is an example program in E:

```
1 + 9 == 5 * ( 1 + 1 )
```

When evaluated, this program should evaluate to the truth value true.
In this exercise, we will not write E programs as strings, but as values of a Haskell data type `E`, that can represent E programs as their abstract syntax trees (ASTs). Given the following data type `E`,

```
data E = IntLit Int
       | BoolLit Bool
       | Plus E E    -- for addition
       | Mult E E    -- for multiplication
       | Equals E E
         deriving (Eq, Show)
```

The above example program is represented as its AST:

```
program = Equals
           (Plus (IntLit 1) (IntLit 9))
           (Mult
             (IntLit 5)
             (Plus (IntLit 1) (IntLit 1)))
```

Problem 11. (20 points) Define an evaluator for the language E. Its name and type should be

```
eval :: E -> E
```

The result of `eval` should not contain any operations or comparisons, just a value constructed either with `IntLit` or `BoolLit` constructors. The result of the example program above should be `BoolLit True`.
Note that E allows nonsensical programs, such as `Plus (BoolLit True) (IntLit 1)`. For such programs, the evaluator can abort.