Homework 3: Lectures 5 & 6

CS 440: Programming Languages and Translators, Spring 2020

Due Fri Feb 7, 11:59 pm

2/6 p.1, 2/7 pp.1,2

What to submit

Put everything into a Haskell source file (e.g., `Smith_Jones_440_hw3.hs` and submit that on Blackboard. Include every participant's names in a comment in the `*.hs` file. For the regular expression problems, add your answers as comments to the `*.hs` file.

- [2/7] **New Requirement 1**: The people in the group who don't turn in their group's solution should turn in a page listing the members of the group (including themselves). So if N people are in the group, there are N things turned in: One solution and N-1 notes, with all N items listing all N people in the group. The reason for doing this is so that I can look at Blackboard and figure out who didn't turn in a homework without having to wait for them to be graded. Submit a note for homework 3 even if your group has already turned it in or it's past homework 3's due date.
- **New Requirement 2**: If you don't work with anyone on a homework, add a note saying that to your submission. That way, if we get a submission that says nothing about who worked on it, we'll know right away that it's a mistake. Right now, non-submitters don't know their names were left off until the homework's been graded. (And they look at blackboard.) If you've already turned in homework 3, it's okay, just start doing it with homework 4.

Problems [50 points]

Lecture 5: Tail recursion and Datatypes

1. (8 points) Write a function `common :: Eq a => [a] -> [a] -> ([a], [a], [a])` where `common x y` = `(cp, x', y')`, where `cp` is the longest common prefix of `x` and `y`, and `x'` and `y'` are `x` and `y` with `cp` removed. (I.e., `x` = `cp` ++ `x'`, `y` = `cp` ++ `y'`, and `cp` cannot be extended.) **Example**: if `x` is `[1,3,5,7,9]`, and `y` is `[1,3,5,8,9]`, then `common x y` is `([1,3,5],[7,9],[8,9])`. **Restriction**: `common` should call an assistant tail recursive routine that implements a loop. *Hint*: You'll also want `reverse` somewhere.

For Problem 2, we'll use a `List` type constructor **modified from lecture**: It uses `Lnode` and `LNil` as constructors (to avoid name clashes with trees later) and omits `Show` from any `deriving` clause. (We'll be implementing `show` ourselves.)

[2/6] `data List a = Lnode a (List a) | LNil deriving (Eq)` -- *No deriving (*`Show`*)*

2. (6 points) Mimic the missing `show` routine by writing a `listShow x` routine that returns the exact same string `show x` would return. (For testing purposes, add `deriving` (..., `Show`) and verify that `listShow x`

`==` `Show x`.) You can assume that the type of `x` is "simple" in the sense that it's not a `data` type. (So no `x :: List(List Int)` values.) [2/4]

For Problems 3 and 4, use the following binary tree definition **modified from lecture**: It takes two type arguments, so nodes and leafs can contain different types of values.

```
data Tree a b = Leaf b | Node a (Tree a b) (Tree a b) deriving (Read, Show, Eq)
```

3. (6 points) Write a `isFull :: Tree a b -> Bool` function that tests for a full tree (every node has two leafs or two trees; a tree that's just a leaf is also full). Note: 2 of the 6 points are for using just pattern matching to check for a leaf or a node (no defining `isNode` or `isLeaf` functions to figure out what the argument looks like). [2/7]: Yes, it's not the standard definition of "is full" because we don't have empty trees. Let's just go with it.

4. (12 points) For this problem, let's call an "expression tree" a `Tree String b` tree where all of the node data are strings from the set `"+"`, `"-"`, `"*"`, and `"/"`, and the leafs hold numbers. Write an `eval` function routine that evaluates an expression tree. Division requires fractional numbers, so the type of `eval` is `Fractional t => Tree String t -> t`. [2/7]

   Examples: Let `e1` = `Node "+" (Leaf 2) (Leaf 4)`, `e2` = `Node "-" (Leaf 11) (Leaf 8)`, and `e3` = `Node "/" (Node "*" e1 e2) (Leaf 36)`. Then `eval e1` = `6.0`; `eval e2` = `3.0`; and `eval e3` = `0.5`.

Lecture 6: Compilers, Languages, and Regular Expressions

For Problems 5-7, give a regular expression for each description. Use `^...$` to get an expression that matches a whole line of input. You don't have to find the shortest possible expression.

5. [6 points] The line consists of a natural number (0 and up), with no leading zero unless the whole thing is a single zero, and going right-to-left, groups of 3 digits are separated by commas. **Examples**: `0`; `1`; `12`; `123`; `1,234`; `12,345`; `1,000,000`. Not examples: `01`; `1000`; `123,4`; `12,34` .

6. [6 points] The line doesn't begin or end with whitespace and all whitespace is one character long. **Example**: `"So this is ok"`. Not examples: `"this  is bad"`; `" this too"`; `"and this "`. For whitespace, use either space or tab (`\t`).

7. [6 points] The line is at most 4 lower-case letters and doesn't include lower case `w`. Feel free to use *exp*`{`*nbr*`}` and *exp*`{`*nbr*`,`*nbr*`}` which give an exact number of occurrences or a range of number occurrences of a regular expression. E.g., `a{3}` is short for `aaa`, and `a{2,3}` is short for `aa|aaa`. (This is a finite language, but don't try to solve the problem by listing all the strings :-)

