PLC: Homework 3 [100 points]

Due date: **Monday**, March 11th
4 extra-credit points if you turn it in by Sunday, March 10th

About This Homework

For this homework, you will practice programming with monads, and also try out Haskell's parsing tools – though I still have to post the problems for those.

How to Turn In Your Solution

Similarly to previous homeworks, push your files to a directory called `hw3` in your github.uiowa.edu repo for the class.

Partners Allowed

You may work by yourself or with one partner (no more). See instructions for hw1 on the protocol you should use if you do work with a partner.

How To Get Help

You can post questions in the `hw3` section on Piazza.

You are also welcome to come to our office hours. See the course's Google Calendar, linked from the Resources tab of the Resources page on Piazza, for the locations and times for office hours.

1 Reading

Read Chapter 10 of *Programming in Haskell* (probably a good idea to review Chapter 12, too).

2 Basic problems using monads [40 points]

You will modify functions in `Basics.hs`. You can compile `Main.hs` to run some tests. These are worth 8 points each:

1. Rewrite `prob1` so that it uses `>>=` instead of `do` notation. Please use `>>=`, not `>>`.
2. Rewrite `prob2` so it uses `do` notation.
3. Modify `prob3` so it does the same computations as `prob2`, except with explicit state-passing.

4. Fill in `reverseArgs` so that it gets the command-line arguments (you can Google to figure out how to do this), reverses them, and returns them.

5. Fill in `getFirstArgIf` so it returns `Just x` if there are command-line arguments and `x` is the first of these, and `Nothing` otherwise.

2.1 Parsing [20 points]

The following problems are 10 points each. They are designed so that you need to read lexical specifications (`Tokens.x`) and grammars (`Grammar.y`), and do some tests with generated lexers and parsers.

1. In the `lexing1` subdirectory, you will find a program which lexes input expressions like `test.expr` and `test2.expr`. Do `cabal build` to compile this. You have to create a file `result.expr` which, when lexed by running `dist/build/Main/Main result.expr` , will produce the following output (except it will be on one big line; I wrapped the line here for typographical reasons):

   ```
   [TokenSym "a",TokenPlus,TokenSym "b",TokenTimes,
    TokenLParen,TokenSym "c",TokenPlus,TokenNum 7,TokenRParen]
   ```

   Hint: you can run `Main` on different test inputs to try to figure out which input can generate the above. Of course you can (and should) also consult the lexical specification in `Tokens.x` to try to figure this out.

2. In the `parsing1` subdirectory, you will find a program which parses input expressions like `test.expr`. Do `cabal build` to compile this. You have to create a file `result.expr` which, when lexed by running `dist/build/Main/Main result.expr` , will produce the following output:

   ```
   B (B (A (A (V "x") (V "x")) (A (V "x") (V "z"))) "x") "y"
   ```

2.2 Generating abstract syntax trees [40 points]

Your goal in this problem is to write a program that can read parenthesized expressions from an input file and print them in GraphViz format to an output file. We did something similar in lecture Feb. 7th. The format of the parenthesized expressions is:

- An expression is either a variable, which is a sequence of characters that begins with an upper- or lowercase capital letter (of the English alphabet) and then has a sequence of zero or more such letters and numeric digits (0-9); or
- a parenthesized expression that starts with a variable, like `(f (g x))` (the variable it starts with is `f`; so you do **not** have to support expressions like `((f g) x)`).

To help you out a little, I am providing you with `Graphviz.hs`, which generalizes some of the code we wrote in class Feb. 7th from binary trees to trees with unbounded (finite) branching. This code is concerned with generating strings in Graphviz format. It is up to you to generalize the other code we wrote in class Feb. 7th from binary to arbitrary-branching trees, and then to write

a lexical specification (`Tokens.x`) and a grammar (`Grammar.y`) to parse in examples like `test.ast` and `wide.ast`. Your lexer should discard whitespace and comments starting with `--` to the end of the line.

You are responsible for writing a file similar to `Main.hs` in `parsing1/`, though please note you will probably need to call it something else, like `AstGen.hs`, to avoid conflicting with the existing `Main.hs` file in the `hw3` directory. You also will need a `.cabal` file (again, similar to what is in `parsing1/`) so that we can compile your program with `cabal build` and get an executable called `dist/build/AstGen/AstGen`. The executable produced should be called `AstGen`. When we run

```
dist/build/AstGen/AstGen test.ast
```

your program is supposed to produce an output file called `test.ast.gv` which can be rendered by Graphviz (you can use, for example, `http://www.webgraphviz.com`). So in general if the input file is `YYY` then the output file should be named `YYY.gv`. Sample inputs and outputs are included, though you do not need to match the Graphviz output exactly, as long as rendering your generated Graphviz files produces the correct trees.