PLC: Workout 6 [80 points]

Due date: Friday, April 1, by midnight

About This Homework

This assignment is about writing parsers in Haskell using the `alex` and `happy` tools, which we saw in the recorded lecture for March 24th:

`https://uicapture.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=6cf63efa-a8de-4025-b659-ae610173e7e3`

Depending on how you installed Haskell, you may have `alex` and `happy` already on your computer, or else you may need to install them. To install, just run these commands in a terminal:

```
cabal v2-install alex
cabal v2-install happy
```

How to Turn In Your Solution

Follow this link to create your repo on Github Classrooms for workout4:

`https://classroom.github.com/a/_2NvQ9k3`

If you work with a partner, please add to your repo a file called `partners.txt`, and list your name and your partner's name there. Both of you should submit your solution: each partner pushes it to their github repo.

How To Get Help

You can post questions in the Discussions section of ICON.

You are also welcome to come to our office hours. See the course's Google Calendar, linked from the github page for the class, for times and Zoom links for office hours. You can also find these on the "Zoom links, office hours" page under "Pages" on ICON:

`https://uiowa.instructure.com/courses/179776/pages/office-hours`

1 Reading

No reading this week.

2 Regular expressions [50 points]

The subdirectory `RegexProblems` contains exercises for writing a lexical specification, to be processed by `alex`. Within that subdirectory, run

```
cabal v1-build
```

in a terminal or command shell to generate and compile the lexer. (Cabal takes care of invoking `alex` for you.) You can run the lexer on sample input like this:

```
cabal v1-run p0-yes.txt
```

Your job is to add regular expressions to the file `Tokens.x`. A solution is provided for you already as an example: `p0` matches one 'a' followed by zero or more 'b's. You will just add your regular expressions in a similar way as the example one, to `Tokens.x`. (All your solutions can be added to just this one `Tokens.x` file and it should work correctly.) Positive testcases are provided in files with names like `p0-yes.txt`, and a couple of negative ones (that should cause your lexer to report an error) in files like `p4-no.txt`. Note that lexers are expected to return a sequence of tokens, and so input files that have matching strings one after the other will return lists of length greater than 1 of matches. You need to implement the following [8 points each]:

- **p1:** match zero or more 'c's followed by one or more 'd's.
- **p2:** match one or more 'e' or 'f' characters.
- **p3:** match a single lowercase character ('a' through 'z'), then match zero or more characters that are either lowercase characters or numeric digits ('0' through '9').
- **p4:** match strings quoted by either single or double quotes, containing zero or more characters that are either upper or lowercase letters or space (you have to write $\boxed{\backslash\ }$ to indicate a space in alex's format) .
- **p5:** match any nonempty sequence of X and Y characters where there is an even number (including 0) of X characters.

You will find the expected output for all the `yes` tests in `expected-output.txt`.

3 Grammars [40 points]

Each of the following problems has its own subdirectory. You will only need to modify the productions part of the `Grammar.y` file in that directory (`Main.hs`, `Tokens.x`, and other files do not need to be modified). Again, compile using `cabal v1-build`, and test using `cabal v1-run yes1.tp`. Note that for both grammars, the lexer I am providing for you (in `Tokens.x` in each directory) already deals with whitespace. So your `Grammar.y` files only need to deal with the tokens listed at the top of the `Grammar.y` file. [20 points each]

- **Grammar1:** add productions to `Grammar1/Grammar.y` to recognize simple Haskell type expressions built from the unit type `()`, parentheses, and right-associative arrow types. Positive testcases are given in `yes1.tp`, etc. Expected output is in `expected-output.txt`.

- **Grammar2:** add productions to `Grammar2/Grammar.y` to recognize a simple fragment of an imperative programming language, as follows:
  - a `Prog` is a list of `FunCalls`, where each `FunCall` is followed by a semicolon
  - a `FunCall` is either a simple identifier (i.e., variable name like "x"), or else a function call, which consists of an identifier, then a left parenthesis, then a comma-separated list of zero or more `FunCalls`, and then a right parenthesis.

  For example, a simple `Prog` is:

  ```
  a;
  f(b,g(c));
  ```

  The `FunCalls` are `a` and `f(b,g(c))`. Notice that the second `FunCall` is nested. Positive testcases are again `yes1.prog`, etc., and expected output is again in `expected-output.txt`.