Advanced Programming
Homework Assignment 5

Due Wednesday, February 20, at 3PM

Preminimaries

1. Read chapter 18 in SOE.

2. Find a partner, if you don't already have one. As with the last assignment, this one should be done "shoulder to shoulder," with both of you working at the same screen at the same time.

   For this assignment, the amount of actual code you'll need to write is fairly small; the key point is understanding what it means! Use your partner for this: Make sure that each of you can explain every detail of the code to the other.

Main assignment

1. **A refined `Exception` monad.**

   (a) Here is a slight variant of the `Maybe` type constructor:

   ```
   data Exn a =
       OK a
     | Exception String
   ```

   Write an instance declaration that makes `Exn` a monad. (Do this without peeking at the `Monad` instance for `Maybe`, if possible.)

   Write a small demo that uses `do` syntax to plug together two or three functions that might return either normal values or exceptions.

2. **A refined `Parser` monad.**

   (a) Type in the definitions of the monadic parser combinators from lecture 8. (Don't worry about the part of the lecture that we didn't get to in class — it is not needed for this assignment. Concentrate on the part up through the section "A Complete Parser.")

   Make sure that the definitions compile and the examples work as advertised in the lecture.

   (b) Rewrite the implementation of the arithmetic expression parser so that, instead of returning just a number, it constructs an abstract syntax tree. Use the following datatype for ASTs:

   ```
   data Exp =
       Const Int
     | Plus Exp Exp
     | Times Exp Exp
     deriving Show
   ```

   For example... parsing the string `"3+4*(5+6)"` should yield this tree:

```
Plus (Const 3) (Times (Const 4) (Plus (Const 5) (Const 6)))
```

(c) Change your implementation so that it ignores whitespace between tokens. For example, parsing the string `"3 + 4   *(5+  6) "` should yield the same tree as above.

(For this part of the exercise, do not change the structure of the existing expression parser. Just add extra code to deal with whitespace.)

(d) You probably agree that dealing with whitespace in the same code that is parsing expressions is pretty ugly. This is why most compilers use two passes: a *lexical analysis* phase that groups sequences of characters into *tokens* and deals with issues like whitespace and comments, followed by a proper *parsing* phase that transforms a list of tokens into an abstract syntax tree.

To do the same thing here, we need to generalize the type of parsers so that instead of working over strings

```
newtype Parser a = Parser (String -> [(a,String)])
```

a parser can work over lists of any type (characters, tokens, etc.):

```
newtype Parser s a = Parser ([s] -> [(a,[s])])
```

Make this change to the definition for `Parser` in your code. Now make appropriate changes throughout the rest of the file so that everything works again.

(e) Declare a new datatype of tokens like this:

```
data Token =
    TNum Int
  | TPlus
  | TTimes
  | TOpen
  | TClose
  deriving (Eq, Show)
```

Define a new parser `tokenize`

```
tokenize :: Parser Char [Token]
```

that maps strings to lists of tokens, ignoring whitespace between tokens. For example, tokenizing the string `"3 + 4   *(5+  6) "` should yield:

```
[([TNum 3,TPlus,TNum 4,TTimes,TOpen,TNum 5,TPlus,TNum 6,TClose],""),
 ([TNum 3,TPlus,TNum 4,TTimes,TOpen,TNum 5,TPlus,TNum 6,TClose]," "),
 ([TNum 3,TPlus,TNum 4,TTimes,TOpen,TNum 5,TPlus,TNum 6],") "),
 ([TNum 3,TPlus,TNum 4,TTimes,TOpen,TNum 5,TPlus],"6) "),
 ([TNum 3,TPlus,TNum 4,TTimes,TOpen,TNum 5,TPlus]," 6) "),
 ([TNum 3,TPlus,TNum 4,TTimes,TOpen,TNum 5,TPlus],"  6) "),
 ([TNum 3,TPlus,TNum 4,TTimes,TOpen,TNum 5],"+  6) "),
 ([TNum 3,TPlus,TNum 4,TTimes,TOpen],"5+  6) "),
 ([TNum 3,TPlus,TNum 4,TTimes],"(5+  6) "),
 ([TNum 3,TPlus,TNum 4],"*(5+  6) "),
 ([TNum 3,TPlus,TNum 4]," *(5+  6) "),
 ([TNum 3,TPlus,TNum 4],"  *(5+  6) "),
 ([TNum 3,TPlus,TNum 4],"   *(5+  6) "),
 ([TNum 3,TPlus],"4   *(5+  6) "),
 ([TNum 3,TPlus]," 4   *(5+  6) "),
 ([TNum 3],"+ 4   *(5+  6) "),
 ([TNum 3]," + 4   *(5+  6) "),
 ([],"3 + 4   *(5+  6) ")]
```

(f) Write parsers

```
expression,eterm,efactor :: Parser Token Exp
```

that parse lists of tokens into abstract syntax trees. The structure of these parsers will be exactly the same as the parsers `exp`, `term`, and `factor` in the code you've been working with above, but they will work with tokens instead of characters.

Test that your new parser gives the same results as the old one.

Submission instructions

- Put your code in a file `YourNames5.hs`. (You don't need to include earlier versions of your parser—just the final version at the end of the problem.)

  Put both of your names in a comment at the top of the file.

  Also, please put the approximate number of hours that you spent on this assignment. Give separate numbers for time spent reading (individually) and time spent programming (together).

- The file `YourNames5.hs` should define a module `Main` that includes an action `main` demonstrating your code.

- Email the file to both `jschorr@seas.upenn.edu` and `bcpierce@cis.upenn.edu`.