Assignment 11

due at 23:59 on Fri Dec 15 (80 points)

For this assignment, you will use test frameworks to compose tests for the parser in assignment 9. You can refer to the A9 solutions.

Before starting, make sure you have installed all the stack packages specified at the top of the recent notes page, as well as:

```
stack install tasty-th
```

Your file `a11.hs` will then begin with this header:

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE NoMonomorphismRestriction #-}
import Data.Maybe (fromMaybe)
import Data.Either (isLeft)
import Text.Parsec
import Test.QuickCheck (quickCheck)
import Test.Tasty.HUnit
import Test.Tasty.QuickCheck
import Test.Tasty.TH
```

The module `Test.Tasty.TH` uses a feature called `TemplateHaskell` to generate code for you. This sometimes goes by the name of "metaprogramming". It can accumulate all of the definitions from your file with names beginning with `case_` or `prop_`, incorporating them into a test tree structure. Use it like this:

```
case_simple_test = 1+1 @?= 2
case_another_test = 2+2 @?= 4

main = $(defaultMainGenerator)
```

The `$(...)` syntax is part of TemplateHaskell. The `defaultMainGenerator`, defined in `Test.Tasty.TH`, expands to a function like the `runTests` we wrote manually in the notes. Here's the result:

```
ghci> main
Main
  simple test:  OK
  another test: OK
```

```
All 2 tests passed (0.00s)
*** Exception: ExitSuccess
```

Now that you know how to create tests, it's your job to write and run tests of the `parseInt` and `parseFloat` functions from assignment 9. Here are some further requirements.

1. You should write at least three QuickCheck properties. One example would be to generate random integers, convert them to strings using `show`, and then try to parse them. The result should be a string that's the same as the `show` representation. You can repeat that for floats to get a second QuickCheck property.

2. Another QuickCheck idea is to generate integers (or floats) and add some leading or trailing garbage characters (not numbers) to them. Then you will verify that the parse fails! Remember, parsers return an `Either` type, where `Right` is successful and `Left` is a failure. You can compare to `Right` using == or === (or `@?=` in HUnit) but if you're expecting a `Left` result (a failed parse) then you may instead want to use the `isLeft` function:

   ```
   ghci> run parseInteger "  323  " == Right "323"
   True
   ghci> isLeft (run parseInteger " x119")
   True
   ```

3. Finally, you should include HUnit tests for all of the parsing-related test cases in the original test driver for assignment 9. That is, the tests labeled 5.01 through 5.13.

   Pulling all of these together with Tasty's `defaultMainGenerator`, here is the output of my program:

   ```
   ghci> main
   Main
     int parseable:              OK
       +++ OK, passed 100 tests.
     float parseable:            OK
       +++ OK, passed 100 tests.
     int leading spaces:         OK
       +++ OK, passed 100 tests.
     int trailing spaces:        OK
       +++ OK, passed 100 tests.
     int fails trailing garbage: OK
       +++ OK, passed 100 tests.
     int is float:               OK
     neg int:                    OK
   ```

```
  dec point:                OK
  decimal digits:           OK
  negative decimal:         OK
  exp wo decimal:           OK
  negative exp:             OK
  negative w negative exp:  OK
  decimals and exp:         OK
  larger decimals exp:      OK
  neg in middle:            OK
  empty before decimal:     OK
  missing exp:              OK

All 18 tests passed (0.01s)
*** Exception: ExitSuccess
```

Very nice! You can also run the tests *outside* of GHCi by just writing this on your regular command prompt:

```
stack runghc a11.hs
```

Run this way, your test program also can handle command-line arguments:

```
stack runghc a11.hs --help
```

And it should display results in color, depending on your terminal settings.

Submit your `a11.hs` to this dropbox: https://www.dropbox.com/request/WszOYjJO3g6rQZn1odIa