Assignment 9

due at 23:59 on Wed Nov 29 (80 points)

This assignment is all about parsing using the parsec package. Write your code into a file named `a09.hs` and submit it to this dropbox: https://www.dropbox.com/request/UETJcbSHbKDCoUmjxwrS

You'll probably need these language extensions and imports:

```
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE NoMonomorphismRestriction #-}
import Control.Monad.State
import Data.Maybe
import Text.Parsec
```

If the parsec import fails, then you need to run `stack install parsec` at your command prompt (not inside GHCi).

Matching delimiters

In class, we wrote a parser to accept matched pairs of brackets, like this:

```
matchParens =
  many space >>
  many (many space >>
        char '[' >> delims >> char ']' >>
        many space) >> many space
```

But I was disappointed with the really ad-hoc handling of white-space. What I learned since then is that the recommendation is to eat up extra white space only *after* reading a particular character or string – not before. (This stackoverflow Q&A is relevant.)

So here are a couple of helper functions for doing just that:

```
trimChar c = char c <* spaces
trimString s = string s <* spaces
```

They also take advantage of the built-in function `spaces`, which is mostly equivalent to `many space` that I was typing previously. One difference though is that `many space` accumulates all the spaces into a string and returns it, whereas `spaces` just discards them.

Your task is to define `matching`, a parser that will accept matching pairs of square brackets `[]`, curly braces `{}`, or round parentheses `()`. It should be mutually recursive with this definition:

```
delims = many matching
```

in order to allow nested and side-by-side parentheses. Then, define this as the top-level parser, which takes care of *initial* spaces and end-of-input:

```
runDelims = parse (spaces >> delims <* eof) ""
```

The test code given below includes lots of test cases using `runDelims`, with strings that should be accepted (`isRight`) as well as rejected (`isLeft`).

Java keyword parsing

Next we worked on parsing sequences of keyword attributes as used in Java, and adding some rules to reject duplicate or conflicting keywords. The rules were that at most one visibility can be specified (`public`, `private`, or `protected`). And at most one of `transient` or `volatile`. And then the remaining three keywords were independent Booleans (either specified or not): `final`, `static`, and `synchronized`.

Here are the data structures for tracking the attribute values:

```
data JAttributes = JAttributes
  { isStatic :: Bool
  , isFinal :: Bool
  , isSynchronized :: Bool
  , visibility :: Maybe JVisibility
  , volatility :: Maybe JVolatility
  }
  deriving (Eq, Show)

data JVisibility
  = JVPublic
  | JVProtected
  | JVPrivate
  deriving (Eq, Show)

data JVolatility
  = JVTransient
  | JVVolatile
  deriving (Eq, Show)
```

```
defaultAttributes :: JAttributes
defaultAttributes =
  JAttributes{ visibility = Nothing
             , volatility = Nothing
             , isStatic = False
             , isFinal = False
             , isSynchronized = False
             }
```

And below I'll give the starting point below that I did in class. **Your task** is to finish the job so it passes the tests. Even better, maybe you can find a way to factor out the commonalities among these definitions.

```
parseFinal = do
  string "final"
  attrs <- getState
  if isFinal attrs then fail "duplicate final"
  else putState attrs{isFinal=True}

parseTransient = do
  string "transient"
  attrs <- getState
  case volatility attrs of
    Nothing -> putState attrs{volatility=Just JVTransient}
    Just JVTransient -> fail "duplicate 'transient'"
    Just JVVolatile -> fail "'transient' conflicts with 'volatile'"

parseVolatile = do
  string "volatile"
  attrs <- getState
  case volatility attrs of
    Nothing -> putState attrs{volatility=Just JVVolatile}
    Just JVVolatile -> fail "duplicate 'volatile'"
    Just JVTransient -> fail "'volatile' conflicts with 'transient'"

parseAnyAttribute =
  many space >>
  (parseFinal <|> parseTransient <|> parseVolatile)

parseAttributes =
  many space >> many parseAnyAttribute >> getState

runAttributes =
  runParser parseAttributes defaultAttributes ""
```

Numeric parsing

A parser for a programming or configuration language will need to be able to recognize numeric values. In class, we constructed a parser for integers that looked like this:

```
parseInteger = do
  neg <- optionMaybe (char '-')
  digits <- many1 (oneOf "0123456789")
  case neg of
    Nothing -> return digits
    Just _ -> return ("-" ++ digits)
```

Your task is to create a parser `parseFloat` that can handle floating-point values that optionally include decimal parts and exponents. See the test driver for examples of what it should accept and reject. When it accepts a floating-point string, it should return the entire string.

In my solution, I made use of `parseInteger` to do the integral parts of the float, which include both the part before the decimal point, and the exponent value.

The other library function I used is `fromMaybe` (imported from `Data.Maybe`). This removes the `Maybe` type around a parsed result by supplying a default value to use instead of `Nothing`:

```
ghci> fromMaybe "default" (Just "actual")
"actual"
ghci> fromMaybe "default" Nothing
"default"
```

I created a helper function like this, that optionally parses something according to given parser p, but if that fails it just returns the empty string:

```
optionMaybeEmpty parser =
  fromMaybe "" <$> optionMaybe parser
```

For example:

```
ghci> parse (optionMaybeEmpty (string "X")) "" "X"
Right "X"
ghci> parse (optionMaybeEmpty (string "X")) "" "Y"
Right ""
```

Test driver

```
main = flip execStateT (0,0) $ do

    -- Parsing matched delimiters
    isRight "1.01" $ runDelims "" -- empty ok
    isRight "1.02" $ runDelims "[]" -- single pair
    isRight "1.03" $ runDelims "()"
    isRight "1.04" $ runDelims "{}"
    isRight "1.05" $ runDelims "[]()" -- side-by-side
    isRight "1.06" $ runDelims "[]{}()"
    isRight "1.07" $ runDelims "  [ ]{   }   (  )   " -- with spaces
    isRight "1.08" $ runDelims "[()]"  -- nested
    isRight "1.09" $ runDelims "[(())]"
    isRight "1.10" $ runDelims "[{()}]"
    isRight "1.11" $ runDelims " [ (  ) ]"  -- nested with spaces
    isRight "1.12" $ runDelims "[( ()) ] "
    isRight "1.13" $ runDelims "[{() } ]"
    isLeft  "2.01" $ runDelims "]"  -- mismatched
    isLeft  "2.02" $ runDelims "[](){)"
    isLeft  "2.03" $ runDelims "[({))]"
    isLeft  "2.04" $ runDelims "(" -- unclosed
    isLeft  "2.05" $ runDelims "[()"
    isLeft  "2.06" $ runDelims "{{{}}"

    -- Parsing Java keyword sequences
    verify  "3.01" (Right defaultAttributes) $ runAttributes ""
    verify  "3.02" (Right defaultAttributes{isStatic=True})
      $ runAttributes "static"
    verify  "3.03" (Right defaultAttributes{isFinal=True})
      $ runAttributes "final"
    verify  "3.04" (Right defaultAttributes{isSynchronized=True})
      $ runAttributes "synchronized"
    verify  "3.05"
      (Right defaultAttributes{isSynchronized=True, isStatic=True})
      $ runAttributes "synchronized  static"
    verify  "3.06"
      (Right defaultAttributes{isSynchronized=True, isStatic=True})
      $ runAttributes "static synchronized"
    verify  "3.07"
      (Right defaultAttributes{isSynchronized=True, isStatic=True,
                               isFinal=True})
      $ runAttributes "final static synchronized"
    verify  "3.08"
      (Right defaultAttributes{volatility=Just JVTransient})
```

```
  $ runAttributes "transient"
verify  "3.09"
  (Right defaultAttributes{volatility=Just JVVolatile})
  $ runAttributes "volatile"
verify  "3.10"
  (Right defaultAttributes{visibility=Just JVPublic})
  $ runAttributes "public"
verify  "3.11"
  (Right defaultAttributes{visibility=Just JVPrivate})
  $ runAttributes "private"
verify  "3.12"
  (Right defaultAttributes{visibility=Just JVProtected})
  $ runAttributes "protected"
verify  "3.13"
  (Right defaultAttributes{visibility=Just JVPrivate, isFinal=True})
  $ runAttributes "private final"
verify  "3.14"
  (Right defaultAttributes{visibility=Just JVPrivate, isFinal=True})
  $ runAttributes "final private "
verify  "3.15"
  (Right defaultAttributes{visibility=Just JVPrivate, isStatic=True,
                           volatility=Just JVTransient})
  $ runAttributes "transient static private "

-- Errors in Java keyword sequences
isLeft  "4.01" $ runAttributes "final final"
isLeft  "4.02" $ runAttributes "static static"
isLeft  "4.03" $ runAttributes "synchronized public synchronized"
isLeft  "4.04" $ runAttributes "public final static final"
isLeft  "4.05" $ runAttributes "public public"
isLeft  "4.06" $ runAttributes "public private"
isLeft  "4.07" $ runAttributes "final transient static transient"
isLeft  "4.08" $ runAttributes "final transient static volatile"

-- Numeric parsing
let runParseFloat = parse (spaces >> parseFloat <* eof) ""
    checkFloat tag s = verify tag (Right s) $ runParseFloat s
checkFloat "5.01" "38281" -- Integer is a float
checkFloat "5.02" "-2848" -- Negative integer
checkFloat "5.03" "1."    -- decimal point without further digits
checkFloat "5.04" "1.001" -- decimal digits
checkFloat "5.05" "-1.9922" -- negative decimal
checkFloat "5.06" "1e100"   -- exponent without decimal
checkFloat "5.07" "1e-99"   -- negative exponent
checkFloat "5.08" "-1e-99"  -- negative with negative exponent
```

```
    checkFloat "5.09" "1.332e33" -- decimals and exponent
    checkFloat "5.10" "48384.23213e-234" -- larger decimals and exponent
    isLeft     "5.11" $ runParseFloat "1.-33" -- negative in middle
    isLeft     "5.12" $ runParseFloat ".1" -- nothing before decimal
                   -- (though some languages allow ".1" for floats?)
    isLeft     "5.13" $ runParseFloat "1.33e" -- missing exponent

  where
    say = liftIO . putStrLn
    correct (k, n) = (k+1, n+1)
    incorrect (k, n) = (k, n+1)
    sayOk tag = do
      modify correct
      say $ " OK " ++ tag
    sayErr tag expected actual = do
      modify incorrect
      say $ "ERR " ++ tag ++ ": expected " ++ expected
        ++ ", got " ++ show actual
    isLeft tag (Left _) = sayOk tag
    isLeft tag result = sayErr tag "Left" result
    isRight tag (Right _) = sayOk tag
    isRight tag result = sayErr tag "Right" result
    verify = verify' (==)
    verify' eq tag expected actual
      | eq expected actual = sayOk tag
      | otherwise = sayErr tag (show expected) actual
-- End of test driver
```