Assignment 9 solutions

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

Matching delimiters

```
trimChar c = char c <* spaces

matching = brackets <|> braces <|> parens
  where
    brackets = trimChar '[' >> delims >> trimChar ']'
    braces   = trimChar '{' >> delims >> trimChar '}'
    parens   = trimChar '(' >> delims >> trimChar ')'

delims = many matching

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

Java keyword parsing

```
trimString s = string s <* spaces

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
             }
```

Here's a nice way to capture the commonality of all the Boolean keywords. This function takes a string kw for the keyword, and then functions to select (sel) and update (upd) the attributes record.

```
parseBool kw sel upd = do
  trimString kw
  attrs <- getState
  if sel attrs then fail ("duplicate '" ++ kw ++ "'")
  else putState (upd attrs)
```

We can use it to implement synchronized, final, and static.

```
parseSynchronized =
  parseBool "synchronized" isSynchronized (\a -> a{isSynchronized=True})

parseFinal =
  parseBool "final" isFinal (\a -> a{isFinal=True})

parseStatic =
  parseBool "static" isStatic (\a -> a{isStatic=True})
```

Here is an abstraction of the volatility and visibility settings, which are stored as a Maybe of some other enumeration type. In this function, the parameters kw, sel, and upd play the same role as before, but we also have the parameter ctor for the constructor that applies to this keyword.

```
parseMaybe kw ctor sel upd = do
  trimString kw
  attrs <- getState
  case sel attrs of
```

```
    Nothing -> putState (upd attrs (Just ctor))
    Just x
      | x == ctor -> fail ("duplicate '" ++ kw ++ "'")
      | otherwise -> fail ("'" ++ kw ++ "' conflicts with " ++ show x)
```

With this helper function to update the volatility of the record, we can implement `transient` and `volatile` keywords.

```
setVolatility a v = a{volatility=v}

parseTransient =
  parseMaybe "transient" JVTransient volatility setVolatility

parseVolatile = do
  parseMaybe "volatile" JVVolatile volatility setVolatility
```

With this helper function to update the visibility of the record, we can implement the `public`, `private`, and `protected`.

```
setVisibility a v = a{visibility=v}

parsePublic = do
  parseMaybe "public" JVPublic visibility setVisibility

parsePrivate = do
  parseMaybe "private" JVPrivate visibility setVisibility

parseProtected = do
  parseMaybe "protected" JVProtected visibility setVisibility
```

List the alternatives for any particular keyword. I used `try` in places where there are subsequent keywords that start with the same letter, such as `synchronized` which precedes `static`.

```
parseAnyAttribute =
  (try parseSynchronized <|> parseFinal <|> parseStatic <|>
   parseTransient <|> parseVolatile <|>
   try parsePublic <|> try parsePrivate <|> try parseProtected)
```

Parse any sequence of attributes, and then return the state.

```
parseAttributes =
  many parseAnyAttribute >> getState
```

Run the parser, which permits leading spaces and expects end of string. Just provide the string of keywords to be parsed.

```
runAttributes =
  runParser (spaces >> parseAttributes <* eof) defaultAttributes ""
```

Numeric parsing

In class we used optionMaybe, which is like optional but it returns a Maybe type to indicate whether the parse was successful. Here I just wrap that with a fromMaybe (from the Data.Maybe module), which provides a default value of the empty string instead of Nothing.

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

Here's a composition pattern that I found very common-place in writing these numeric parsers: given two parsers p and q, run them in sequence, but then return the concatenation of the two resulting strings. So the parser appendParsers (string "!") (many (char "-")) when run on the string "!----" will succeed and return the entire string.

```
appendParsers p q = do
  s1 <- p
  s2 <- q
  return $ s1 ++ s2
```

So now, an integer has an optional negative sign, followed by one or more digits. (The digit parser is built into parsec, but is equivalent to oneOf "0123456789" that we used previously.)

```
parseInteger =
  optionMaybeEmpty (string "-") `appendParsers` many1 digit
```

A float starts like an integer, but then it has an optional dot followed by zero or more digits, then optionally followed by an exponent (which can be negative). Notice how I use parseInteger directly in this definition.

```
parseFloat =
  parseInteger `appendParsers` decPart `appendParsers` expPart
  where
    decPart = optionMaybeEmpty (appendParsers (string ".") (many digit))
    expPart = optionMaybeEmpty (appendParsers (string "e") (parseInteger))
```

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
```