CIS 194: Homework 11

Due Monday, April 8

- Files you should submit: `SExpr.hs`. You should take the version that we have provided and add your solutions. Note that we have also provided `AParser.hs`—you are welcome to use your own `AParser.hs` from last week's homework or ours, whichever you prefer.

Parsing S-expressions

In `AParser.hs` from last week's homework, we now have the following:

- the definition of a basic `Parser` type
- a few primitive parsers such as `satisfy`, `char`, and `posInt`
- `Functor`, `Applicative`, and `Alternative` instances for `Parser`

So, what can we do with this? It may not seem like we have much to go on, but it turns out we can actually do quite a lot.

Remember, for this week's homework you should only need to write code on top of the interface provided by the `Functor`, `Applicative`, and `Alternative` instances. In particular, you should not write any code that depends on the details of the `Parser` implementation. (To help with this, the version of `AParser.hs` we provided this week does not even export the `Parser` constructor, so it is literally impossible to depend on the details!)

Exercise 1

First, let's see how to take a parser for (say) widgets and turn it into a parser for *lists* of widgets. In particular, there are two functions you should implement: `zeroOrMore` takes a parser as input and runs it consecutively as many times as possible (which could be none, if it fails right away), returning a list of the results. `zeroOrMore` always succeeds. `oneOrMore` is similar, except that it requires the input parser to succeed at least once. If the input parser fails right away then `oneOrMore` also fails.

For example, below we use `zeroOrMore` and `oneOrMore` to parse a sequence of uppercase characters. The longest possible sequence of uppercase characters is returned as a list. In this case, `zeroOrMore` and `oneOrMore` behave identically:

```
*AParser> runParser (zeroOrMore (satisfy isUpper)) "ABCdEfgH"
Just ("ABC","dEfgH")
*AParser> runParser (oneOrMore (satisfy isUpper)) "ABCdEfgH"
Just ("ABC","dEfgH")
```

The difference between them can be seen when there is not an uppercase character at the beginning of the input. `zeroOrMore` succeeds and returns the empty list without consuming any input; `oneOrMore` fails.

```
*AParser> runParser (zeroOrMore (satisfy isUpper)) "abcdeFGh"
Just ("","abcdeFGh")
*AParser> runParser (oneOrMore (satisfy isUpper)) "abcdeFGh"
Nothing
```

Implement `zeroOrMore` and `oneOrMore` with the following type signatures:

```
zeroOrMore :: Parser a -> Parser [a]
oneOrMore  :: Parser a -> Parser [a]
```

Hint: To parse one or more occurrences of `p`, run `p` once and then parse zero or more occurrences of `p`. To parse zero or more occurrences of `p`, try parsing one or more; if that fails, return the empty list.

Exercise 2

There are a few more utility parsers needed before we can accomplish the final parsing task. First, `spaces` should parse a consecutive list of zero or more whitespace characters (use the `isSpace` function from the standard `Data.Char` module).

```
spaces :: Parser String
```

Next, `ident` should parse an *identifier*, which for our purposes will be an alphabetic character (use `isAlpha`) followed by zero or more alphanumeric characters (use `isAlphaNum`). In other words, an identifier can be any nonempty sequence of letters and digits, except that it may not start with a digit.

```
ident :: Parser String
```

For example:

```
*AParser> runParser ident "foobar baz"
Just ("foobar"," baz")
*AParser> runParser ident "foo33fA"
Just ("foo33fA","")
*AParser> runParser ident "2bad"
Nothing
*AParser> runParser ident ""
Nothing
```

Exercise 3

S-expressions are a simple syntactic format for tree-structured data, originally developed as a syntax for Lisp programs. We'll close out our demonstration of parser combinators by writing a simple S-expression parser.

An *identifier* is represented as just a `String`; the format for valid identifiers is represented by the `ident` parser you wrote in the previous exercise.

```
type Ident = String
```

An "atom" is either an integer value (which can be parsed with `posInt`) or an identifier.

```
data Atom = N Integer | I Ident
  deriving Show
```

Finally, an S-expression is either an atom, or a list of S-expressions.[1]

[1] Actually, this is slightly different than the usual definition of S-expressions in Lisp, which also includes binary "cons" cells; but it's good enough for our purposes.

```
data SExpr = A Atom
           | Comb [SExpr]
  deriving Show
```

Textually, S-expressions can optionally begin and end with any number of spaces; after *throwing away leading and trailing spaces* they consist of either an atom, *or* an open parenthesis followed by one or more S-expressions followed by a close parenthesis.

$$\begin{aligned} \mathit{atom} &::= \mathit{int} \\ &\mid \mathit{ident} \end{aligned}$$

$$\begin{aligned} S &::= \mathit{atom} \\ &\mid \texttt{(}S^*\texttt{)} \end{aligned}$$

For example, the following are all valid S-expressions:

```
5
foo3
(bar (foo) 3 5 874)
(((lambda x (lambda y (plus x y))) 3) 5)
(   lots  of   (  spaces   in  )  this ( one ) )
```

We have provided Haskell data types representing S-expressions in `SExpr.hs`. Write a parser for S-expressions, that is, something of type

```
parseSExpr :: Parser SExpr
```

Hints: To parse something but ignore its output, you can use the `(*>)` and `(<*)` operators, which have the types

```
(*>) :: Applicative f => f a -> f b -> f b
(<*) :: Applicative f => f a -> f b -> f a
```

`p1 *> p2` runs `p1` and `p2` in sequence, but ignores the result of `p1` and just returns the result of `p2`. `p1 <* p2` also runs `p1` and `p2` in sequence, but returns the result of `p1` (ignoring `p2`'s result) instead.

For example:

```
*AParser> runParser (spaces *> posInt) "     345"
Just (345,"")
```