CS 115: Functional Programming, Spring 2016

Assignment 6: Fun with parsing

Due: *Monday, May 30, 02:00:00*

Coverage

This assignment covers the material up to lecture 21.

Submitting your assignment

Your code will consist of three files, corresponding to the three parts of this assignment. The first file is called `State.hs` and contains the part A and part B solutions. The second file is called `Sexpr.hs` and contains the part C solutions. The third file is called `XML.hs` and contains the part D solutions. Submit your assignment to csman as usual.

As this assignment is the last of the term, there will be no min-grading, so give it your best effort! We will allow one rework only.

Writing and testing your code

As usual, all of your Haskell functions, even internal ones, should have explicit type signatures.

All of your code should be compiled with the `-W` (warnings enabled) command-line option (whether using `ghc` for standalone programs or `ghci` for interactive testing).

Your code (especially the code for the standalone programs) should be commented appropriately and neatly and reasonably formatted.

Don't forget to use the `parseTest` function to test parsers interactively. This was described in lecture 21. Import the module `Text.Parsec` to get access to this function. (Actually, if you load up your code which imports this module, you will have access to it already.) Look it up in Hoogle if you need more information about how to use it.

Part A: `IORef`s and state monads

In this section, you will be writing functions in an imperative style using either the `IO` monad or a state monad. Here are the definitions of while loops for each monad, which you should include in your assignment submission:

```
-- While loop in the IO monad.
whileIO :: IO Bool -> IO () -> IO ()
whileIO test block =
  do b <- test
     when b (block >> whileIO test block)

-- While loop in the state monad.
whileState :: (s -> Bool) -> State s () -> State s ()
whileState test body =
  do s0 <- get
     when (test s0)
          (do modify (execState body)
              whileState test body)
```

Note that somewhere in your code you will need to import the modules `Control.Monad`, `Control.Monad.State`, and `Data.IORef` for this code to work. If `Control.Monad.State` isn't available, you will need to install it. Open up a terminal and type:

```
$ cabal update
$ cabal install mtl
```

(where the `$` is a prompt, of course). `mtl` stands for "Monad Transformer Library" and it includes the `Control.Monad.State` module.

You should use either `whileIO` or `whileState` in all of the functions in this section (as opposed to *e.g.* using explicit recursion). Obviously, use `whileIO` for the functions using the `IO` monad and `whileState` for the functions using the `State` monad.

1. **[20]** Write a function called `factIO` which computes factorials and works in the `IO` monad.

   The type signature of this function will be:

   ```
   factIO :: Integer -> IO Integer
   ```

   Use `IORef`s to store all the local variables of the function. There should be two local variables, one representing a counter and the other the running total. The counter is initialized with the input to `factIO` and is decremented by 1 at each iteration until it hits zero. The running total is initialized with 1 and is multiplied by the counter value at each iteration. At the end, the running total will have the desired factorial. This is returned (using the `return` method of the `IO` monad) as the result.

   Signal an error if the input is invalid (*i.e.* less than zero). This applies to all of the functions in this section.

2. **[20]** Write a function called `factState` that computes factorials and uses a state monad.

   The type signature of the function will be:

   ```
   factState :: Integer -> Integer
   ```

   The state type will be `(Integer, Integer)`; the first `Integer` will represent the counter and the second the running total. This function is actually shorter (less than 10 lines) and easier to write than the `factIO` function. You will need to define a state transformer "helper value" which will actually embody the computation.

3. **[20]** Write a function called `fibIO` that computes fibonacci numbers and uses the `IO` monad. This fibonacci function should have `fib(0) == 0` and `fib(1) == 1` and so on.

   The type signature of the function will be:

   ```
   fibIO :: Integer -> IO Integer
   ```

   Again, use `IORef`s to hold all the local variables for the function. This time, there are three local variables; one represents the count, and two represent the last two fibonacci numbers computed. Initialize the count with the input to the `fibIO` function and decrement it by 1 at each iteration; when it hits zero, the desired fibonacci number is in the smaller of the other two variables. The non-counter variables get increased on each iteration according to the usual fibonacci rule. Don't forget to check the input for errors.

4. **[20]** Write a function called `fibState` that computes fibonacci numbers and uses a state monad. The fibonacci function is as described above.

   The type signature of the function will be:

   ```
   fibState :: Integer -> Integer
   ```

   The state type will be `(Integer, Integer, Integer)`; the first two `Integer`s represent the last two fibonacci numbers counted and the final `Integer` represents a counter which starts equal to the input to `fibState` and is decremented by 1 at each iteration until it hits zero. The desired fibonacci number will then be in the smaller of the other two `Integer` values in the state. Again, this function is quite short (less than 10 lines).

Part B: Deriving the reader monad

We have discussed state monads in class and derived them from first principles. There is another kind of state-like monad called the "reader" monad; this is used in stateful computations that only require read-only access to the state. In this section you will derive the reader monad from first principles. Note that the datatypes we will use are simpler than the ones actually found in the GHC libraries, because they are less general. Nevertheless, the principles are the same.

The reader monad represents computations that can access read-only data. Represented conceptually, functions in this monad would look like:

```
  a -------[read from a stored value]------> b
```

In analogy to the State monad, we can write these functions as:

```
  (a, r) -> b
```

for some state type `r`. We can curry this to get:

```
  a -> r -> b
```

which is isomorphic. We can parenthesize it as follows:

```
  a -> (r -> b)
```

We therefore define the `Reader` datatype as follows:

```
  data Reader r b = Reader (r -> b)
```

so we can write the characteristic monadic functions in the reader monad as:

```
  a -> Reader r b
```

and the only difference is that values of the type `(r -> b)` are now values of the type `(Reader r b)` which is just a `Reader` wrapper around the same function type.

This is equivalent to the monadic function form:

```
  a -> m b
```

with `(Reader r)` as the `Monad` instance.

We will also define this helper function:

```
  runReader :: Reader r a -> r -> a
  runReader (Reader f) = f
```

We will use this definition of the `Monad` instance for the `Reader` monad:

```
instance Monad (Reader r) where
  return x = Reader (\r -> x)

  mx >>= f = Reader (\r ->
               let x = runReader mx r in
                 runReader (f x) r)
```

The `>>=` operator can also be written as:

```
  mx >>= f = Reader (\r ->
               let (Reader g) = mx
                   x = g r
                   (Reader h) = f x
               in h r)
```

[90] Derive the definitions of `return` and `>>=` that we have given above from first principles. Recall that `return` must conceptually be the identity function in the monad and the `>>=` operator must be defined so that monadic function composition (the `>=>` operator from the `Control.Monad` module) works correctly. Use the derivation of the `Monad` instance for state monads as a model for your derivation.

Note that you do *not* have to prove that the `Monad` instance for the reader monad satisfies the monad laws.

Hint: Start by deriving the `>>=` operator and then derive the `return` method.

Note: Please write your answer in a multi-line Haskell comment (between `{-` and `-}` symbols).

Part C: Extending the S-expression parser

In lecture 21 we created a very simple S-expression parser for a Scheme-like language. In this section you will extend the S-expression parser in several ways to get some practice working with parser combinators.

While doing this, you should be aware that there is nothing about an S-expression parser that necessarily connects it to the Scheme programming language. Scheme does use S-expressions as the basis for its syntax (though with lots of extensions, such as the dot notation for improper lists), but you could use S-expressions in other contexts as well, and people do.

Template code

The code used in the lecture is available here. You will be modifying various parts of this code in this section. There is also a file of sample Scheme-like code here called `test.scm`. Use this to test your parser. You can do this by starting `ghci`, loading the file `Sexpr.hs`, and at the prompt, typing:

```
Prelude> test
```

which will attempt to load the file `test.scm`, parse all the code and output a "pretty-printed" version of the code. (You may not think it's particularly pretty, but it will show the structure of the code very explicitly.) If the parser gives errors on any part of this code, it isn't working properly. Do **not** change the pretty-printer code itself except where specifically indicated below.

Exercises

1. **[30]** The parser given to you in the original version of the file `Sexpr.hs` has these datatypes as the basic S-expression datatypes:

```
data Atom =
    BoolA  Bool
  | IntA   Integer
  | FloatA Double
  | IdA    String
  deriving (Show)

data Sexpr =
    AtomS Atom
  | ListS [Sexpr]
  | QuoteS Sexpr
  deriving (Show)
```

The `Atom` type is fairly standard (though we'll be extending it below), but the `Sexpr` type itself is a bit odd. There is actually no need for the `QuoteS` constructor for quoted expressions. If you recall from CS 4, quoted expression like these ones:

```
'a
'(foo bar baz)
```

are actually syntactic sugar for these expressions:

```
(quote a)
(quote (foo bar baz))
```

These S-expressions can be represented as `Sexpr` expressions without the `QuoteS` constructor as follows:

```
ListS [AtomS (IdA "quote"), AtomS (IdA "a")]
ListS [AtomS (IdA "quote"),
       ListS [AtomS (IdA "foo"),
              AtomS (IdA "bar"),
              AtomS (IdA "baz")]
```

Therefore, your first exercise is to remove all the code dealing with the `QuoteS` constructor (including the code in the pretty-printer *i.e*. the `ppSexpr` function). Nevertheless, your parser must be able to parse quoted expressions, converting them into S-expressions of the form given above. The `Sexpr` datatype will become simply:

```
data Sexpr =
    AtomS Atom
  | ListS [Sexpr]
  deriving (Show)
```

This exercise is mostly to get you familiar with the code base. You won't be changing the way parsing is done, but you will have to modify the function `parseSexpr` and change the way quoted expressions are converted into `Sexpr`s.

2. **[30]** Many if not most Schemes allow you to use square brackets or curly braces to delimit S-expressions instead of parentheses. This is convenient in deeply-nested expressions so you can easily tell visually where an expression starts and ends. A closing square bracket can only match an opening square bracket, and not an opening parenthesis (and similarly for curly braces). In this exercise, you will extend the S-expression parser to allow it to parse S-expressions using square brackets or curly braces. Some examples:

```
(this is an expression using parentheses)
[this is an expresssion using square brackets]
{this uses curly braces}
(this [weird expression] {uses all [kinds (of delimiters)]})
```

Modify the `parseList` function so that it can parse S-expressions with any of the three types of delimiters. The best way to do this is to define a helper function which takes the two delimiter characters as arguments and outputs a parser, and then use three such parsers in the body of `parseList`. Use the `<|>` (alternation) operator to combine the three parsers into one big parser.

Note that the fact that a given S-expression uses parentheses or square brackets or curly braces is *not* encoded into the datatype; the datatype doesn't care about which delimiters were used for a given S-expression.

3. **[15]** Note that you don't need to use the `try` combinator in the `parseList` function in the event that you tried to parse one kind of delimiter and failed. Why not? (Write the answer in a comment.)

4. **[30]** The floating-point number parser in the template code is limited in that it can't handle floating-point numbers with exponents. So these are valid floating-point numbers according to the parser:

```
1.2
-3.4
```

```
42.12334
```

but these aren't:

```
1.2e10
-3.4e-10
42.12334E+10
```

Modify the function `parseFloat` to make it parse floating-point numbers with exponents. Exponents are optional, but if present they have this syntax: exponent letter (`e` or `E`), optional sign (`+` or `-` (or no sign)), and one or more digits (`0` to `9`).

There are a couple of things you should know about the floating-point number parser in the template code:

1. Floating-point numbers are required to have a decimal point, and at least one digit after the decimal point. Most computer languages will accept floating-point numbers without decimal points if *e.g.* an exponent is present, but requiring the decimal point simplifies the parsing. Requiring a digit after the decimal point is my personal preference; I think that floating-point numbers without digits after the decimal point look ugly :-) You don't need to change any of this; it's just for your information.

2. The parser works by creating a string version of the floating-point number and then using the Haskell `read` function to convert the string to a floating-point number. This approach will still work with exponents, but in this case the string version of the floating-point number may have an exponent (represented as a string concatenated to the end of the rest of the floating-point number).

5. **[30]** Scheme includes strings as a basic data type, but the `Atom` datatype in the parser doesn't support strings. Change it to this type:

```
data Atom =
    BoolA   Bool
  | IntA    Integer
  | FloatA  Double
  | IdA     String
  | StringA String
  deriving (Show)
```

As you can see, now strings are a special kind of atom which use the `StringA` constructor. Extend the atom parser `parseAtom` so that it can parse strings as well. Do this by defining a parser called `parseString` which parses strings, and then call that inside of `parseAtom`. *Hint*: Inside `parseAtom`, handle strings after integers but before identifiers. The string syntax we'll use is extremely simple: a string is a sequence of characters between two double-quote (`"`) characters. The double-quote character cannot occur in a string (there is no backslash-escaping, for instance), but you can put newline characters in a string directly (you don't have to write *e.g* `\n` for a newline, and in fact that will give you two characters, not one). Here are some example strings the parser can recognize:

```
"I am a string"
"This string
spans
multiple lines"
```

```
"I can have 'single quotes' inside but not double quotes"
```

It isn't be too hard to improve the string parser to handle backslash escapes, but it isn't required.

Part D: Writing a simple XML parser

In this section, you will write an extremely simplified XML parser using parser combinators. In case you don't know, XML is a structured data format which is a generalization of the HTML (HyperText Markup Language) format commonly used to write web pages. Unlike HTML, XML is extremely regular and is used for much more than just formatting documents. It is a general-purpose data format which is heavily used in the real world. XML is not terribly complex, but writing a full XML parser would take too long, so the parser you will write here will be missing many XML features. Some of these missing features include:

- XML comments, which consist of text between the delimiters `<!--` and `-->`.
- Initial `DOCTYPE` declarations and similar declarations.
- Forms without specific end tags *e.g.* `<hr />`.
- Forms with attributes *e.g.* `<a href="http://csman.cms.caltech.edu/">csman</a>`.

If none of this means anything to you, don't worry; we'll explain exactly what syntax your parser will have to handle below.

Template code

As in part A, we are supplying you with a file of template code here. You should download it and modify it according to the instructions given below. The template code defines the XML data structures and the pretty-printer, so you can concentrate on writing the parser. There is also a test file called `test.xml` which you should use to test the parser; it contains all the forms that your parser should be able to parse. Again, to test your code, just run `ghci`, load up `XML.hs`, and type:

```
Prelude> test
```

As before, this will attempt to load the test file, parse its contents and output a "pretty-printed" version of the XML. Again, if the parser gives errors on any part of this code, it isn't working properly. Do **not** change the pretty-printer code at all for this section.

The template code as given doesn't work; you need to fill in the parts labeled `{- TODO -}` with real code, and please remove the `{- TODO -}` comments as you fill in the code. Note that there are lines containing the `<?>` operator following the `{- TODO -}` comments; these should be left in, as they will affect what kind of error messages your code generates.

Also, there exist some Haskell modules that include XML parsing code. You are not allowed to use these for this assignment, for what should be obvious reasons. If you do, you will get no credit for this section.

Simplified XML syntax

The XML parser you will write will handle only two kinds of forms:

1. Tagged forms, which look like this: `<TAGNAME>TEXT...</TAGNAME>`. The tag name (in the place marked TAGNAME) consists of one or more characters, all of which are either letters (upper- or lowercase, from `a` (or `A`) to `z` (or `Z`)) or digits from `0` to `9` (but no symbolic characters). Tags can contain a mixture of uppercase characters, lowercase characters, and digits. The TEXT part can consist of any number of characters (or no characters at all) except for the characters `<`, `>`, or `&`. Notice that the end tag is distinct from the start tag because it contains the forward-slash character (`/`) before the tag name.

2. Special character entities, which look like this: `&NAME;`. Entities always start with the `&` character, followed by the entity name (which consists of lower-case letter characters only). In fact, only a few character entities will be recognized by the parser; anything else that has the correct entity syntax but isn't one of the specified entities is an error.

Here are some examples of tagged forms:

```
<p>This is a test.</p>

<a><b><c>Forms</c> can be <b>nested</b> as long as their <emphasis>start tags match
their end tags!</emphasis></b></a>
```

Here are the character entities that the parser will recognize:

```
&lt;  (means the < (less-than) character)
&gt;  (means the > (greater-than) character)
&amp; (means the & (ampersand) character)
```

Note that the limitation on text (that it can't contain the literal characters `<`, `>`, amd `&`) is not a real limitation because we could just use the character entities `&lt;`, `&gt;`, and `&amp;` instead.

Datatypes

Here are the datatypes you will use when writing your XML parser:

```
type Tag = String

data Entity = LT_E | GT_E | AMP_E
   deriving (Show)

data Elem =
    TextE String     -- raw text
  | EntE Entity      -- entity
  | FormE Tag [Elem] -- tagged data
  deriving (Show)
```

We use the type name `Tag` as an abbreviation for `String` in cases where we intend the string to be used as a tag. `Entity` refers to the character entities, which are either `&lt;` (less-than sign), `&gt;` (greater-than sign), or `&amp;` (ampersand symbol), as described above. They are represented by the constructors `LT_E`, `GT_E`, and `AMP_E` respectively. The `Elem` type is the type of all XML elements; these are either a text string, an entity or a tagged form. The contents of a tagged form are a list of XML elements, which is a bit like the lists in S-expressions (this is what allows tags to be nested).

Code to write

[120] As mentioned above, you need to download the template file `XML.hs`, and remove the `{- TODO -}` comments and replace them with your own code. You should test the sub-parsers individually using `parseTest` as described above and make sure they each work before continuing. When you are done, run `test` at the `ghci` prompt to test the parser. You will need to have downloaded the `test.xml` file into the current directory before you do this.

Hints and suggestions

- There is not a lot of code to write for this section (about 30 lines), so if you find yourself writing a large amount of code, you are doing something wrong (or at least not doing it in the most efficient way).
- **Hoogle is your friend!** Before you write a parser, check to see if there is already a parser defined in the Parsec libraries that does what you need (or most of what you need). Some of the sub-parsers you will write can be defined in one line in terms of parsers in the Parsec libraries. Check out the documentation for the following modules:

  ```
  Text.Parsec
  Text.Parsec.Combinator
  Text.Parsec.Char
  Text.Parsec.Prim
  ```

- Somewhere in the code you are likely to run into a situation where you want a parser to backtrack when it fails. In that case, you need to use the `try` combinator. We only needed to use `try` once in all of our code, but it was critical in that location, so watch out for it.
