Homework 7

Due Thursday, 3/28/2019

Purpose:

The purpose of this homework assignment is to help you better understand how to create parsers in Haskell.

Please to turn in a file Hmwk7.pdf with all the answers to the homework, but also pull out your Haskell programs and turn them in a text file named Hmwk7.hs. The first line of that file should be

```
module Hmwk7 where
```

The next line should be a comment with your name. The rest of the file should be your Haskell code. We will take that file and automatically test it by running your code on our test data. As a result, you should be sure that your code compiles properly (i.e., running "ghci Hmwk7.hs" should compile your code without errors and it should be possible to run your code on my test cases (ideally without it blowing up when executed!).

The Hmwk7.hs file will be used for automated testing of your code. Aside from looking for your name in the contents, it is likely that no human will read it. Thus if you don't include that code in the Hmwk7.pdf file, you will likely not get any credit for it. Please remember this!

While your Hmwk7.pdf file will be turned in to gradeScope as usual, you will need to turn in the file Hmwk7.hs via `https://submit.cs.pomona.edu/2019sp/cs101`.

Be sure to test your programs one last time before submitting. I've seen students mess up when commenting code in a way that causes everything to break. Code that doesnt compile will get very little credit. We will be using some automatic testing, and code that doesnt compile brings everything to a halt. With a large class we will not have time to go in to manually tweak your code to make it work.

IMPORTANT: When you write the functions requested below, please make sure that they have the exact names and types specified in the question. If you make an error, your program will crash on our test suite and you will get very little credit.

Be sure that all of my sample code works, and worry about edge cases that might cause your program to give the wrong answer as I will test it more thoroughly.

1. (10 points) **Chomsky Normal Form**

   We consider the strings of balanced parentheses of two types over the alphabet is $\Sigma = \{(,), [,]\}$. The language PAREN_2 is the smallest set of strings satisfying the following three properties.

   (a) $\epsilon \in \text{PAREN}_2$;

   (b) if x is in PAREN_2, then so are (x) and $[x]$; and

   (c) if x and y are in PAREN_2, then so is xy.

   Give a context-free grammar in Chomsky normal form for $\text{PAREN}_2 - \{\epsilon\}$.

   I suggest you create a regular cfg for the language and then convert it to Chomsky Normal Form using the algorithm we covered in class. Show your work so that we can verify your grammar is correct.

2. (30 points) **Parsing Tuples**

   Given the following BNF:

```
<exp> ::= ( <tuple> ) | a
<tuple> ::= <tuple>, <exp> | <exp>
```

(a) Draw the parse tree for ((a,a),a,(a)).

(b) Write a lexer for terms of this form. The tokens are simply "a", "(", ")", and ",". The tokens generated by the lexer should be from the following type:

```
data Tokens = AToken | LParen | RParen | Comma | Error String |
              EOF deriving (Eq,Show)
```

where `EOF` marks the end of the token list. The main lexer function should be of the form:

```
getTokens :: [Char] -> [Tokens]
```

As an example, you should get the following results when testing `getTokens`:

```
*Main> getTokens "((a,a),a,(a,a))"
[LParen,LParen,AToken,Comma,AToken,RParen,Comma,AToken,Comma,
LParen,AToken,Comma,AToken,RParen,RParen,EOF]
```

(c) An alternative grammar for the above language in EBNF is

```
<exp> ::= ( <tuple> ) | a
<tuple> ::= <exp> {, <exp> }*
```

where the * means the items in braces may repeat 0 or more times.

We can rewrite this as BNF as

```
<exp> ::= ( <tuple> ) | a
<tuple> ::= <exp> <expTail>
<expTail> ::= e | , <exp> <expTail>
```

where `e` stands for the empty string.

Recall from class that we can build a table that will direct a parser as follows. The rows of the table will correspond to non-terminals, while the columns will correspond to terminals. The entries are productions from our grammar.

Put production `X` ::= α in entry (`X`,`b`) if either

- `b` $\in$ FIRST(α), or
- $\epsilon \in$ FIRST(α) and `b` $\in$ FOLLOW(`X`).

For any non-terminal `X` and terminal `b`, the production `X` ::= α will occur in the corresponding entry if applying this production can eventually lead to a string starting with `b`.

For this to give us an unambiguous parse, no table entry should contain two productions. (If so, we would have to rewrite the grammar!) Slots with no entries correspond to errors in the parse (e.g., that the string is not in the language generated by the grammar).

We can also write this restriction out as the following two laws for predictive parsing:

i. If `A ::=` α_1 `|` `...|` α_n then for all i $\neq$ j, First(α_i) $\cap$ First(α_j) = $\emptyset$.

ii. If `X` $\rightarrow^*$ ϵ, then First(`X`) $\cap$ Follow(`X`) = $\emptyset$.

Compute First and Follow for each non-terminal of this grammar and show that the grammar follows the first and second rules of predictive parsing.

(d) The following definition can be used to represent abstract syntax trees of the expressions generated by the grammar above:

```
data Exp = A | AST_Tuple [Exp] | AST_Error String deriving (Eq,Show)
```

[The last item is simply there to handle the case of errors.] Thus the tuple (a,a,a) would be represented as `AST_Tuple[A,A,A]`, while the term in part (a) would be represented as `AST_Tuple [AST_Tuple [A,A],A,AST_Tuple[A,A]]`.

Write a predictive recursive descent parser for the grammar in part 2c. It should generate abstract syntax trees of type `Exp`.

Hint: Watch the types of your parsing functions. You should have

```
parseExp :: [Tokens] -> (Exp, [Tokens])
parseTuple :: [Tokens] -> ([Exp], [Tokens])
parseExpTail :: ([Exp], [Tokens]) -> ([Exp], [Tokens])
parse :: [Char] -> Exp
```

where `parse` is the function invoked on the input string. E.g.,

```
*Main> parse "((a,a),a,(a,a))"
AST_Tuple [AST_Tuple [A,A],A,AST_Tuple [A,A]]
```