Homework 6

Due Friday, 10/15/10

Please turn in your homework solutions online at `http://www.dci.pomona.edu/tools-bin/cs131upload.php` before the beginning of class on the due date.

1. (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 parentheses 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]]`.

Write a predictive recursive descent parser for the grammar in part 1c. 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]]
```

2. (20 points) **Table-driven parsing**

Early in the semester we discussed the fact that `if-then-else` expressions in C, C++, Java, and Pascal are ambiguous. Suppose we use the following grammar for a very simple language with conditional statements:

```
    <stmt> ::= if (bool) <stmt> <possElse> | assn
<possElse> ::= else <stmt> | e
```

where `e` stands for the empty string.

In this grammar, "`if`", "`bool`", "`assn`", and "`else`" are tokens, and hence terminals of the grammar, while `<stmt>` and `<possElse>` represent non-terminals. (In a more realistic language,

"`bool`" would be a non-terminal that would generate Boolean valued expressions, while "`assn`" would generate assignment statement. However, it makes the problem easier if we just let them be terminals.

(a) Show that the string "`if (bool) if (bool) assn else assn`" has two distinct parse trees.

(b) Compute the **first** and **follow** sets for the grammar, and build a parse table like that in Lecture 15 (see online notes).

(c) Explain how the parse table suggests that the grammar will be problematic to parse using predictive parsing like that explained in class.

(d) How could we manually manipulate the parse table so that, if used as the basis for a recursive descent or stack-based top down parser, it always chooses a parse according to the C/C++/Java/Pascal rules (i.e., always associate an "`else`" with the nearest "`if`")?

3. (10 points) **Static and Dynamic Scope**

   Please do problem 7.8 from Mitchell, page 196.

4. (10 points) **Eval and Scope**

   Please do problem 7.10 from Mitchell, page 197. You only need to do parts b and c, but please read part a so that the rest makes sense.

5. (18 points) **Lambda Calculus and Scope**

   Please do problem 7.11 from Mitchell, page 198.