# Homework 6

## Due Friday, 10/17/08

Please turn in your homework solutions by the beginning of class to the dropoff folder on `vpn.cs.pomona.edu` in directory `/common/cs/cs131/dropbox`. Make sure that all of your files include your name and the problem number in a comment.

1. (30 points) **Parsing Lists**

   Given the following BNF:

   ```
   <exp> ::= ( <list> ) | a
   <list> ::= <list>, <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 datatype:

   ```
   datatype tokens = AToken | LParen | RParen | Comma | EOF;
   ```

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

   ```
   fun lexstr s = getTokens (explode s);
   ```

   You will need to provide `getTokens`.

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

   ```
   lexstr "((a,a),a,(a,a))"; val it =
   [LParen,LParen,AToken,Comma,AToken,RParen,Comma,AToken,Comma,
   LParen,AToken,Comma,AToken,RParen,RParen,EOF] : tokens list
   ```

   *You may need to insert the following lines at the beginning of your program in order to see the entire output:*

   ```
   Control.Print.printDepth:=100;
   Control.Print.printLength:= 200;
   ```

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

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

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

   We can rewrite this as BNF as

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

where `e` stands for the empty string.

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 datatype definition can be used to represent the expressions generated by the grammar above:

```
datatype exp = A | List of exp list | ASTError of string;
```

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

Write a predictive recursive descent parser for the grammar in part 1c. It should generate abstract syntax trees for the datatype `listExp`.

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

```
val parseExp = fn : tokens list -> exp * tokens list
val parseList = fn : tokens list -> exp list * tokens list
val parseExpTail = fn : exp list * tokens list -> exp list * tokens list
val parse = fn : string -> exp
```

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

```
- parse "((a,a),a,(a,a))";
val it = List [List [A,A],A,List [A,A]] : exp
```

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 16 (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.

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

   Please do problem 7.11 from Mitchell, page 198.