Prof. Jingke Li (FAB 120-06, li@cs.pdx.edu), Classes: MW 4:40-6:30pm, DLC304, Office Hours: MW 2-3pm.

Project 3: Abstract Syntax Tree

(Due Wednesday 11/9/11)

This project is to add semantic actions to your MINI parser to generate an abstract syntax tree (AST) for the input program.

MINI's AST Nodes

MINI's abstract syntax is defined in terms of a set of AST nodes, each implemented as a Java class. A listing of these nodes appears as Appendix B in *The MINI Programming Language* manual. The whole set of class definitions is included in the accompanied file, `proj3-code.tar`. You are required to use this set of definitions without any modification.

Generating an AST

- Copy your MINI parser program, `miniParser0.jj`, from Project 2 and rename it `miniParser.jj`. (*Note:* You also need to rename the heading inside the the file accordingly.) Add an "`import ast.*;`" line inside the `PARSER_BEGIN` `PARSER_END` brackets at the top of the file, so that the parser can access the AST node definitions. The new heading should look like

```
PARSER_BEGIN(miniParser)
package parser;
import ast.*;
public class miniParser {}
PARSER_END(miniParser)
```

- Modify the parsing routines so that each routine returns a proper AST node. You need to figure out, for each parsing routine, what the corresponding AST node (or list class object) should be, and where to insert semantic actions to generate it. While the AST for a given program should be unique, how to generate it is dependent on the concrete grammar that you have written for MINI.

  Note that a parsing routine can also take input parameters. This feature allows information to be passed from a parent node to its children. It is useful in handling grammar rules resulted from left-recursion-elimination transformations.

- The following are a few sample parsing routines:

```
// Program -> ClassDecl {ClassDecl}
//
Program Program() :
{ ClassDecl c; ClassDeclList cl = new ClassDeclList(); }
{
  c=ClassDecl() { cl.add(c); } (c=ClassDecl() { cl.add(c); })* <EOF>
  { return new Program(cl); }
}

// Expr -> AndExpr OrTail
//
Exp Expr() :
{ Exp e1, e2 = null; }
{
```

```
  e1=AndExpr() e2=OrTail(e1)
  { return e2==null ? e1 : e2; }
}

// OrTail -> ["||" AndExpr OrTail]
//
Exp OrTail(Exp e0) :
{ Exp e1 = null, e2 = null; }
{
  ["||" e1=AndExpr() { e1 = new Binop(Binop.OR, e0, e1); } e2=OrTail(e1)]
  { return e2==null ? e1 : e2; }
}
```

Requirements

Your AST-generating parser will be graded mostly on its correctness. For your convenience, a driver program, a set of test programs, and a shell script for running the parser and comparing results are provided to you (in the accompanied `proj3-code.tar` file). The driver program invokes the parser to generate an AST, and then invokes a dumping routine on the AST to print the nodes out.

For the given set of MINI test programs, your parser should generate *identical* AST outputs as those saved in the `.ast.ref` files.

Note that your parser should continue detect any and all forms of syntax errors.

Submission

Submit your parser file `miniParser.jj` through the "Dropbox" on the D2L class webpage.