CS321 Winter'14 Assignment 4: Parser with AST Generation

(Due 2/25/14 @ 12pm)

This assignment is a follow up to Assignment 3. In this assignment, you are going to insert semantic actions to the parsing routines of Assignment 3 parser program to generate an abstract syntax tree (AST) for the input program.

This assignment requires a correct LL1 or LL2 miniJava grammar. You are encouraged to use your own `mjGrammarLL.jj` program for this assignment as the base for this assignment. However, if for any reason, your `mjGrammarLL.jj` is not working perfectly at this time, you may convert the provided LL2 grammar in `mjGrammarLL2.txt` into JavaCC parsing routines and insert them into your new parser program. The parser program for this assignment should be called `mjParser.jj`. (If you choose to use your own grammar, you may proclaim in your program's comment block, "This parser is built with my own grammar.").

Preparation

Download the zip file "`hw4 materials`" from the D2L website. After unzipping, you should see an `hw4` directory with the following items:

`ast` — a directory containing AST node definitions

`mjGrammarLL2.txt` — an LL(2) grammar for miniJava

`mjParser0.jj` — a starting version of miniJava parser; it contains only the lexer portion

`tst` — a directory containing sample miniJava programs and their corresponding AST dump outputs

`Makefile` — for building the parser

`run` — a script for running tests

Details

The following are suggested steps to complete this assignment.

1. *Familiarize yourself with the AST class definitions, especially the constructors.* In your parser program, you'll call AST node constructors to create AST nodes. Knowing what your choices are and what exact forms you need to follow will be of great help. A list of AST node constructors with their expected parameters can be found at the end of this document. Full details are in the file `ast/Ast.java`.

2. *Decide a return type for each parsing routine.* In Assignment 3's parser program, parsing routines do not return anything. In the new version for this assignment, all parsing routines participate in the construction of an AST; each routine contributes its share by returning an AST object. Here is a rough guidance for return types (Details depend on your exact grammar.):

Parsing Routine	*Return Type*
Program()	Ast.Program
ClassDecl()	Ast.ClassDecl
MethodDecl()	Ast.MethodDecl
Param()	Ast.Param
VarDecl()	Ast.VarDecl
Any Type routine	Ast.Type
Any Stmt routine	Ast.Stmt
Any Expr routine	Ast.Exp
Id()	String
IntLit()	int
BoolLit()	boolean
StrLit()	String

Note that three of the return types, `Ast.Type`, `Ast.Stmt`, and `Ast.Exp`, are abstract classes. Any routine returning these types need to return an object of a concrete subclass of them (*e.g.* `Ast.IntType`).

3. *Insert semantic actions into the parsing routines.* You may want to start with the simpler ones first. Most statement routines have direct corresponding AST nodes, so they easy to handle. Binary and unary operation routines are also easy to handle. The more difficult routines are those involving `ExtId`, since they cover many different miniJava constructs, *e.g.* method calls, assignments, array elements, and object fields. For these routines, you need to analyze each carefully to decide what type of AST node to return. In some cases, a single routine may need to return different types of AST nodes for different sub cases.

4. *Compile and debug your program.* You can use the `Makefile` to compile your parser:

   ```
   linux> make
   ```

   Your program should be free of JavaCC or Java compiler warnings.

5. *Run tests.* To run a test, you could do

   ```
   linux> java mjParser tst/test01.java
   ```

   To run a batch of test programs together, use the `run` script:

   ```
   linux> ./run mjParser tst/test*.java
   ```

   It will place your parser's output in `*.ast` files and use `diff` to compare them one-by-one with the reference version in `.ast.ref` files. Since for each test program, there is an unique AST, your parser's output should match the reference copy exactly.

   Although semantic actions have been inserted into the parsing routines, your new parser program should still detect syntax errors exactly the same as the plain version in Assignment 3 did. Try running your program with `tst/errp*.java` programs; verify that it still catches all errors.

A Few Sample Routines

The following are a couple of sample parsing routines:

```
// Program -> {ClassDecl}
Ast.Program Program():
{ List<Ast.ClassDecl> cl = new ArrayList<Ast.ClassDecl>();
  Ast.ClassDecl c; }
{
  ( c=ClassDecl() {cl.add(c);} )* <EOF>
  { return new Ast.Program(cl); }
}
// Type -> BasicType ["[" "]"] | <Id>
Ast.Type Type():
{ Ast.Type t; String nm; }
{
  ( t=BasicType() [ "[" "]" {t=new Ast.ArrayType(t);} ]
  | nm=Id() {t=new Ast.ObjType(nm);} )
  { return t; }
}
// Expr -> AndExpr {"||" AndExpr}
Ast.Exp Expr():
{ Ast.Exp e1, e2; }
{
  e1=AndExpr() ( "||" e2=AndExpr() {e1 = new Ast.Binop(Ast.BOP.OR,e1,e2);} )*
  { return e1; }
}
```

Requirements and Grading

This assignment will be graded mostly on your parser's correctness. We may use additional miniJava programs to test. The minimum requirement is that your parser runs and generates at least one valid AST output.

What to Turn in

Submit a single file, `mjParser.jj`, through the "Dropbox" on the D2L class website.