CS321 Winter'14 Assignment 3: Parser (First Version)

(Due 2/4/14 @ 12pm)

In this assignment, you are going to implement a first version of a parser for a small subset of the Java programming language, called miniJava. In this version, the parser performs only one task — validating the syntax of the input program. For a syntactically correct input program, the parser simply prints out a message "Program's syntax is valid." For any program containing syntax errors, the parser detects and reports the nature and location of the first error. You are going to implement the parser in JavaCC.

Preparation

Download a copy of `assignment3.zip` from the D2L website. After unzipping, you should see an `assignment3` directory with the following items:

- `mjGrammarRaw.txt` — miniJava's raw grammar
- `mjGrammarRaw.jj` — a JavaCC program based on the raw grammar (Not valid and would not compile!)
- `tst` — a directory containing sample miniJava programs
- `Makefile` — for building the parser
- `run` — a script for running tests

For this assignment, you'll need to use Java and JavaCC. Linux lab machines already have them installed. If you plan to use your own computer, you need to install them first. For Java, the latest version from the official site is fine. For JavaCC, the version to install is 5.0.[1] `javacc-5.0.zip` is available on the D2L website. After downloading and unzipping it, you'll see a `javacc-5.0` directory and a `bin` sub-directory within it. Add this `bin` directory to your path, and you are done.

MiniJava and Its Grammars

A version of the miniJava language's grammar is included in the zip file. This grammar is targeted for people. It is called a "raw" grammar, since it needs to be *refined* to be suitable for compilers to use.

A section of this raw grammar is shown below:

```
Expr  -> Expr BinOp Expr
      |  UnOp Expr
      |  ExtId "(" [Args] ")"
      |  ExtId "[" Expr "]"
      |  ExtId
      |  "(" Expr ")"
      |  <IntLit>
      |  <BoolLit>
ExtId -> ["this" "."] <Id> {"." <Id>}
```

It describes the expression syntax of the language. People can understand it, possibly with additional comments for clarifying operators' precedence order. To a compiler, however, this grammar is ambiguous, has left-recursion, and has common-prefix of unbounded size. A direct implementation of this grammar would not work. (A JavaCC program based on this version is included in the zip file. You can try to compile it yourself.)

Therefore, the first task of this assignment is to use the techniques discussed in this week's class and lab to transform the grammar to an equivalent LL grammar. Specifically, you need to:

- Use the operator precedence information to rewrite the expression section of the grammar to eliminate the ambiguity in it.

[1] If you search online you may see a newer version, JavaCC 6.0. Don't download that version! It's buggy and unstable.

- Use the standard production-rewriting technique to eliminate all left-recursions in the grammar. These left-recursions also occur in the expression section.
- Use the left-factoring technique to reduce the need for multiple-token lookahead as much as possible. It may be difficult to factor out all common prefixes and produce an LL(1) grammar. So for this assignment, an LL(2) grammar is acceptable. However, using 3 or more lookahead tokens is *not* acceptable.

Note that you don't need to deal with the "dangling else" ambiguity problem. You may leave the grammar for the if statement as is; JavaCC has the default resolution of matching an else clause with the innermost if statement.

JavaCC Implementation

Name your parser program `mjGrammarLL.jj`. Copy and paste the `main` method and the token specifications form the provided file `mjGrammarRaw.jj` to your `mjGrammarLL.jj` file. (You need to change any reference to `mjGrammarRaw` to `mjGrammarLL`.)

The mapping from a CFG grammar to a JavaCC program should be straightforward. Do not change JavaCC's default lookahead setting (which is "single-token"). You should only use two-token lookahead occasionally, by inserting the directive `LOOKAHEAD(2)` at places as needed. (*Hint:* If you do the grammar transformations right, then you should need only one or two of `LOOKAHEAD(2)` insertions.)

Add a comment block in front of each parsing routine to show the corresponding grammar rule. (See `mjGrammarRaw.jj` for examples.) This will help you to verify that your parsing routine is a faithful implementation of the grammar rule.

Running and Testing

You can use the given `Makefile` to compile your parser to `mjGrammarLL.class`, or you can do it manually:

```
linux> javacc mjGrammarLL.jj
linux> javac mjGrammarLL.java
```

To run a test, just run the compiled parser program:

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

A shell script, `run`, can be used to run a batch of test programs with one command:

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

For each syntactically correct program (as all `tst/test*.java` are), you should expect to see the message `"Program's syntax is valid."` For programs with syntax errors (as all `tst/errp*.java` are), the script will place your parser's error message in `*.perr` files and use `diff` to compare them one-by-one with the reference version in `*.perr.ref`. If your parser implements a correct grammar for miniJava, it should detect all the errors in these programs. You may see differences between the your parser's error messages and those in the reference files. As long as the nature and the location of the reported error are the same, it is fine.

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 for receiving a non-F grade is that your JavaCC program compiles without error, and it validates at least one of the test programs.

What to Turn in

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