Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue 10:00-11:50 @SRTC 155, Labs: Tue 14:00-15:50 & Wed 12:00-13:50 @FAB 88-10

Homework Assignment 2: Parser (First Version)

(Due 11/6/14 @ 11:59pm)

In this assignment, you are going to implement a first version of a top-down parser for the language, miniJava. In this version, the parser performs only one task — validating the syntax of the input program. It does not generate a parse tree or any other form of program representation. 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. Your parser is to be implemented with the parser-generator, JavaCC. This assignment carries a total of 100 points.

Preparation

Download a copy of "`hw2.zip`" from D2L. After unzipping, you should see a `hw2` directory with the following items:

- `hw2.pdf` — this document
- `mjRawGrammar.pdf` — miniJava's raw grammar
- `mjRawGrammar.jj` — a JavaCC program based on the raw grammar (It would not compile!)
- `Makefile` — for building the parser
- `runp0` — a script for running your parser with tests
- `tst/` — a directory containing sample miniJava programs

Raw Grammar vs LL Grammar

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
       |  "(" Expr ")"
       |  ExtId "(" [Args] ")"          // method call
       |  Lvalue
       |  Literal

Lvalue -> ExtId "[" Expr "]"            // array element
       |  ExtId

ExtId  -> ["this" "."] <ID> {"." <ID>}  // object field or just ID
```

It describes the expression syntax of the language. People can understand it. But to a compiler, this grammar is ambiguous, has left-recursions, and has productions with common-prefix (of unbounded length).

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 to see what happens.) It needs to be converted into an LL form before a top-down parser can be developed.

Syntax Errors

Any violation of miniJava's syntax rules should result in a syntax error. There are numerous places in a program a syntax error may occur. The following is an incomplete list of examples:

- *missing or misspelled keyword* — *e.g.* `class` is missing in a class declaration.
- *incorrect order of components* — *e.g.* a variable declarations appears after method declarations in a class; the four keywords `public static void main` in a `main` method declaration are out of order.
- *wrong syntax for a statement* — *e.g.* `return` statement with multiple expressions; method call on the left-hand-side of an assignment; etc.
- *wrong syntax for an expression* — *e.g.* multiple `[]` in an array reference; a string literal in an expression; etc.

Your Tasks

1. **Transforming the Grammar.** (60 points)

   The first task of this assignment is to use the techniques discussed in classes and labs to transform the grammar to an equivalent LL grammar. Specifically, you need to:

   - Use the operator precedence information (included at the end of the document `mjRawGrammar.pdf`) to rewrite the expression section of the grammar to eliminate the ambiguity in it.
   - 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 three 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. (It will generate a warning message, though, which you can ignore.)

2. **Converting the Transformed Grammar to JavaCC Code.** (30 points)

   Name your parser program `mjParser0.jj`. Copy and paste the `main` method and the token specifications form the provided file `mjRawGrammar.jj` to your `mjParser0.jj` file. (You need to change any reference of `mjRawGrammar` to `mjParser0`.)

   The mapping from a context-free grammar to a JavaCC program should be straightforward. Add a comment block in front of each parsing routine to show the corresponding grammar rule. (See `mjRawGrammar.jj` for examples.) This will help you to verify that your parsing routine is a faithful implementation of the grammar rule.

   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 where you see

the need. (*Hint:* If you do the grammar transformations right, you should need only one or two of `LOOKAHEAD(2)` insertions.)

3. **Developing Test Cases for Syntax Errors.** (10 points)

   Develop ten test inputs for testing your parser's detection of syntax errors. Name these test files `syntaxerror01.java` — `syntaxerror10.java`. Try to cover a wide spectrum of syntax error types. But note that your parser most likely will stop after detecting the first syntax error, so there is no point to include multiple syntax errors in a single test.

   For the purpose of validating your parser, you may want to develop more then ten test inputs. But you are only required to submit ten such tests.

Running and Testing

You can use the given `Makefile` to compile your program, or you can do it manually:

```
linux> javacc mjParser0.jj
linux> javac mjParser0.java
```

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

```
linux> java mjParser0 tst/test01.java
```

A shell script, `runp0`, can be used to run a batch of test programs with one command:

```
linux> ./runp0 tst/test*.java
```

For each program in the test suite `tst/test*.java`, you should expect to see the message `"Program's syntax is valid."`

For your error-testing inputs, you should expect to see a message reporting the nature and location of the first error in each case. This is done automatically by JavaCC.

Requirements

This assignment will be graded mostly on the correctness of your LL grammar, through the execution of your JavaCC program. You should extensively test your parser before submitting.

Minimum Requirement for Passing The minimum requirement for receiving a non-F grade on this assignment is that your JavaCC program compiles without error, and it validates at least one of the test programs.

Submission

Submit a single zip file, `hw2sol.zip`, containing, `mjParser0.jj`, and `syntaxerror*.java` through the "Dropbox" on the D2L class website. You don't need to encode your name in the zip file name, *but don't forget to include your name in your parser program.*.