Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue&Thu 12:00-13:15 @SH 212, Lab: Fri 10:00-11:15/11:30-12:45 @EB 103

Homework Assignment 2: Grammar Transformations

(Due 11/5/15 @ 11:59pm)

In this assignment, you are going to transform a grammar for the miniJava language into a form that is suitable for top-down parsing; and to validate the new grammar with a JavaCC implementation. This assignment carries a total of 10 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
- `RawGrammar.txt` — miniJava's raw grammar
- `RawGrammar.jj` — a JavaCC program based on the raw grammar (It would not compile!)
- `Makefile` — for building the parser
- `run` — a script for running your parser with tests
- `tst/` — a directory containing sample miniJava programs

The Raw Grammar

The miniJava "raw" grammar provided with this assignment is a grammar intended for people to read. It is not suitable as the base for parser implementation. As an example, consider the grammar's expression section 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
```

People can understand it without much difficulty. But to a compiler writer or a compiler-generator program, this grammar section has many issues. It is ambiguous, has left-recursions, and has productions with common-prefix of unbounded length. It would not be possible to implement a parer directly based on this grammar. (A JavaCC program based on the raw grammar is included in the zip file. You can try to compile it yourself to see what happens.) For our goal of implementing a top-down parser (in the next assignment), the grammar needs to be converted into a suitable form, namely, an *LL grammar*. An LL grammar is a grammar that is unambiguous and does not have left-recursions.

Grammar Transformations (5 points)

The first task of this assignment is to use the techniques discussed in classes and labs to transform the raw grammar into an equivalent LL grammar. There is a further requirement on productions' common-prefix length: In the transformed grammar, productions should not have a common-prefix of more than one terminal symbol. Using compiler terminology, the new grammar must be LL(1) or LL(2). (The JavaCC implementation will be able to verify that your grammar satisfies this requirement.)

Specifically, you need to:

- Use the operator precedence information (included at the end of the file `RawGrammar.txt`) to rewrite the expression section of the grammar to eliminate its ambiguity.
- 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 eliminate common-prefix among productions for the same non-terminal, or to reduce the size of common-prefix to a single terminal. (*Note:* While possible, it is a challenging task to factor out all common prefixes.)

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. This particular form of ambiguity can be handled by an *ad hoc* technique inside the parser (by matching an else clause with the innermost if statement). Most parser generators, including JavaCC, has this technique built-in.

Also note that JavaCC allows Extended BNF form, so you should consider taking advantage of it in expressing the resulting grammar of these transformations.

Put your transformed grammar in a *plain text* file, `LLGrammar.txt`.

Grammar Validation with JavaCC (5 points)

Once you have developed an LL grammar, your second task is to convert it into a JavaCC program to validate it. Name your program `LLGrammar.jj`. Copy and paste the token specifications and the `main` method form the provided file `RawGrammar.jj` to your program file. (You need to change any reference of `RawGrammar` to `LLGrammar`.)

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 `RawGrammar.jj` for examples.) This will help you to verify that your parsing routines are a faithful implementation of the grammar rules.

Do not change JavaCC's default lookahead setting (which is 1, meaning that the generated parser is LL(1)). 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.)

If your transformed grammar is indeed LL(1) (or LL(2) with the `LOOKAHEAD(2)` insertions), your corresponding `LLGrammar.jj` program should compile without error or warning. (The warning related to the ambiguous if statement does not count, and can be ignored.)

Test Cases for Syntax Error (0 points)

Any violation of miniJava's grammar should result in a syntax error. There are numerous places in a program a syntax error may occur. The following are a few examples:

- *"Missing or misspelled keyword"* — *e.g.* the keyword `class` is missing in a class declaration.

- *"Incorrect order of components"* — *e.g.* a variable declaration appears after method declarations in a class; or the four keywords `public static void main` in a `main` method declaration are out of the required order.
- *"Wrong syntax for a statement"* — *e.g.* a `return` statement has multiple return values; or a method call appearing on the left-hand-side of an assignment statement.
- *"Wrong syntax for an expression"* — *e.g.* multiple brackets (`[]`s) in an array reference; or a string literal in an expression.

Your third task is to develop a set of test inputs for testing your JavaCC program's detection of syntax errors. JavaCC catches syntax errors automatically, so there is no need to add any special code to deal with error detection. There is no need to include more than one syntax error in a single test input, since your program will most likely stop after detecting the first error.

This part carries no point, and nothing needs to be submitted. However, since your program will be tested for syntax-error detection during grading, it is to your benefit that you validate your program as much as possible beforehand.

Running and Testing

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

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

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

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

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

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

For each program in the provided test suite, 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.

Minimum Requirement for Passing The minimum requirement for receiving a non-F grade on this assignment is that your transformed grammar is 40% or more correct, *or* your JavaCC program compiles without error and validates at least one of the test programs.

Submission

Submit a single zip file, `hw2sol.zip`, containing, `LLGrammar.txt` and `LLGrammar.jj` through the "Dropbox" on the D2L class website.