Prof. Jingke Li (FAB120-06, lij@pdx.edu), Mon 8:15-10:05 @FAB 171, Lab: Wed 8:15-10:05 @FAB 88-10

Homework Assignment 1: Lexer

(Due 10/22/15 @ 11:59pm)

In this assignment, you are going to implement two versions of lexer for a small Java-like language: one through manual implementation and the other with the lexer-generation tool, JavaCC.

This assignment builds on top of Lab 2. If you have not attended Lab 2 for any reason, please go through its materials first. You may copy and use any code from Lab 2 for this assignment.

The *minimum requirement* for this assignment is that one of your lexer programs compiles on the CS Linux system without error and passes one of the provided test inputs. This assignment carries a total of 10 points.

Preparation

Download the zip file "`hw1.zip`" from D2L. After unzipping, you should see a `hw1` directory with the following items:

- `hw1.pdf` — this document
- `Lexer1.java` — a starter version of lexer1
- `RunLexer1.java`, `RunLexer2.java` — driver programs for the lexers
- `Makefile` — for compiling the lexers
- `run1`, `run2` — two scripts for running the tests
- `tst/` — a subdirectory containing a set of test inputs

The Token Definitions

We call the input langauge *miniJava*. Its token definitions follow Java's lexical rules with some exceptions for simplification purpose.

- miniJava is case sensitive — upper and lower-case letters are *not* considered equivalent.
- The following are miniJava's *keywords* — they must be written in the exact form as given:

  ```
  class extends static public main void boolean int double String true
  false new this if else while return System out println
  ```

  Note that some of these words are not keywords in Java (*e.g.* `main`, `true`, `false`, etc.). They are made reserved in miniJava to simplify syntax compatibility with Java. (For example, you can use `System.out.println` to print in miniJava even though it does not support packages.)
- An *identifier* starts with a letter, and is followed by an optional sequence of letters and/or digits. If such a sequence matches a keyword, then the sequence is considered a keyword, not an identifier. There is no limit on the length of an identifier.
- There are three forms of *integer* literals:
  - a *decimal* literal is either a singleton 0, or a non-empty sequence of digits with a leading non-zero digit (*e.g.* `1`, `123`);

  - an *octal* literal is a sequence with a leading digit 0 followed by a non-empty sequence of octal digits 0-7 (*e.g.* `00`, `0123`);
  - a *hexadecimal* literal is a sequence with a two-character prefix, 0X or 0x, followed by a non-empty sequence of hexadecimal digits, *i.e.* digits plus letters a through f (both upper and lower cases are allowed) (*e.g.* `0X0`, `0x123`).

  An integer literal's value must be in the range of 0 to 2,147,483,647 ($2^{31} - 1$). (Note that the value is always non-negative. A negative integer constant, such as `-3`, is constructed from an unary minus operator and an integer literal.)

- A *double* literal contains a non-empty sequence of digits, and a decimal point. The decimal point may appear anywhere in the sequence (*e.g.* `.0123`, `012.3`, `0123.`). This literal corresponds to Java's 64-bit double literal, except that miniJava does not support the exponent form.
- A *string* literal contains a sequence of ASCII characters, except double quotes (`"`), carriage returns (`\r`), and newlines (`\n`), delimited between a pair of double quotes. A string literal can be of arbitrary length, including zero. Note that the beginning and ending double quotes are not part of the string literal, and should not be included in the lexeme.
- *Comments* can be in two forms: a single-line comment starts with `//` and ends with the first occurrence of either an newline character (`\n`) or end-of-file character (`EOF`). A block comment starts with `/*` and ends with the first occurrence of `*/`. In both cases, any ASCII characters can be included in the comments.
- The following are miniJava's *operators* and remaining *delimiters*:

```
operator  = "+"|"-"|"*"|"/"|"&&"|"||"|"!"|"=="|"!="|"<"|"<="|">"|">="
delimiter = "="|";"|","|"."|"("|")"|"["|"]"|"{"|"}"
```

Task 1. `Lexer1.java` (7 points)

Your first task is to implement a lexer manually to recognize all the tokens of miniJava. A starter version of this lexer is in `Lexer1.java`. It contains token code and token representation definitions. You should not change these definitions.

```
enum TokenCode {
  // Tokens with multiple lexemes
  ID, INTLIT, DBLLIT, STRLIT,

  // Keywords
  CLASS, EXTENDS, STATIC, PUBLIC, MAIN, VOID, BOOLEAN, INT, DOUBLE, STRING,
  TRUE, FALSE, NEW, THIS, IF, ELSE, WHILE, RETURN, SYSTEM, OUT, PRINTLN,

  // Operators and delimiters
  ADD, SUB, MUL, DIV, AND, OR, NOT, EQ, NE, LT, LE, GT, GE,  ASSGN,
  SEMI, COMMA, DOT, LPAREN, RPAREN, LBRAC, RBRAC, LCURLY, RCURLY;
}

static class Token {
  TokenCode code;
  String lexeme;
  int line;           // line # of token's first char
  int column;         // column # of token's first char
  ...
}
```

Your task is to complete the two routines, `nextChar()` and `nextToken()` in this program.

- The `nextChar()` routine extends the version in Lab 2 with a tracking of both line and column numbers.
- The `nextToken()` routine is the main lexer routine. In normal cases, it should return a `Token` object for the next valid token from input. When the end of input is reached, it should return a `null`. This routine should detect all forms of lexical errors, and throw an exception with a descriptive message for each case. (See the *Requirements* Section below.)

  Note that the `line` and `column` fields of a `Token` object should be set to the line and column numbers of the token's *first* character. The `Lexer1.java` programs in Lab 2 does not track the line number, and the column number recorded in those programs does not pointing to the first character of a token.

Task 2. `Lexer2.jj` (3 points)

Your second task is to implement a lexer using JavaCC. Write a JavaCC specification for miniJava's tokens. Call your program `Lexer2.jj`. You can use any of the `Lexer2.jj` program from Lab 2 as a starter version.

The requirement on this lexer is similar, *i.e.*, it should recognize all valid tokens, and detect and report all lexical errors.

Compiling and Testing

The two driver programs `RunLexer1.java` and `RunLexer2.java` are similar to those in Lab 2. They repeatedly call `nextToken()` to get all tokens and print them in a nice form. You can use the provided `Makefile` to compile your lexers individually:

```
linux> make lexer1
linux> make lexer2
```

or just type `"make"` to compile both.

A set of test inputs, `testXX.in` are provided in the `tst` directory. Two sets of corresponding reference output are in `testXX.out1.ref` and `testXX.out2.ref` files.

The first ten inputs, `test01.in` – `test10.in`, consist of correct tokens; and they cover a broad spectrum of token cases. The second group of tests, `test11.in` – `test17.in`, consist of lexical error cases.

You can use the provided scripts to test your lexers with the test inputs as a whole:

```
linux> ./run1 tst/test*.in
linux> ./run2 tst/test*.in
```

Or, you cand test your lexers with a single input, *e.g.*:

```
linux> ./run1 tst/test01.in
```

For each input, the script will run your lexer, save its output in a file, `XX.out1` or `XX.out2`, and compare the output with the reference version. If you see a message "`matched ref`" after a test, it means your lexer's output matches exactly with the ref version. Otherwise, it means there are some differences, and you can view the diff result in the corresponding `XX.diff` file.

Requirements

For the correct-token test inputs, both of your lexers should generate output that match those in the `.ref` files.

For the lexical-error test inputs, both of your lexers should display a message indicating the error type, and the line and column numbers of the offending lexeme.

Your lexer1's error message should match the reference version with some minor differences.

For lexer2, the priority is to catch all lexical errors, and report the error location correctly. You should try to match its error message with the reference version, but in cases where you have no direct control (*e.g.* if you use lexical state for handling block comments and you want to catch "unclosed comments" error), whatever message you can get is fine.

Note that the set of provided tests do not cover all possible lexical cases. You are encouraged to find more cases to test your lexers.

Submission

Submit a single zip file containing your two lexer programs, `Lexer1.java` and `Lexer2.jj`. through the "Dropbox" on the D2L class website. (There is no need to submit `Lexer2.java`.)