CS352s2014 Project 1: Lexing

January 17, 2014

1 Objective

The objective of this project is to become accustomed to the course MiniJava compiler environment and to write a MiniJava lexer, both using the JavaCC lexer generator and by hand.

2 Code

The project's code environment is available on the course web page. We are providing Java class files for a complete compiler, including the lexer, but are not providing accompanying Java source. Through the course of these projects, it will be your job to rewrite the Java source for each component, creating a complete compiler of your own design. You are advised to keep a copy of the Java class files so that you can use them to generate correct output and test against your own.

2.1 Testing

The compiler provides a number of tools for testing each stage. They are available in the `bin` directory. Two are of value for testing the lexer.

- `bin/mjlex`: Runs the lexical analyzer (lexer). `bin/mjlex examples/Factorial.java` will print out one lexical token per line. If a lexically invalid file is provided, an error will be produced.
- `bin/mjlex-manual`: Runs the hand-written lexical analyzer (lexer). (You must write a hand-written lexer before this command will work).

3 JavaCC

JavaCC is a parser generator ("compiler compiler") for Java. Given a definition of a language's grammar, it generates a parser for that language. In this project, you will use JavaCC as a lexer generator, for the lexical component of MiniJava.

A template of the lexer is available in `edu/purdue/cs352/minijava/parser/Lexer.jj-template`. Use it as a basis to create `edu/purdue/cs352/minijava/parser/Lexer.jj`. This template includes a main function and small examples; you must fill in the remainder.

`.jj` files are compiled into `.java` files using `javacc`. A copy of `javacc` is included in the project code directory. To compile `Lexer.jj`, use the following command:

```
javacc/bin/javacc -OUTPUT_DIRECTORY=edu/purdue/cs352/minijava/parser \
edu/purdue/cs352/minijava/parser/Lexer.jj
```

Please note that the included `Makefile` will also rebuild the lexer if `Lexer.jj` changes.

JavaCC's regular expressions have a slightly different syntax than the canonical syntax we've discussed in class. In particular, JavaCC's regular expressions require that you quote characters and strings. For instance, the regular expression `if`, in JavaCC, is written `"if"`. This simply allows you to write regular expressions more clearly, using whitespace between components.

Unions and both Kleene and positive closures have the same syntax as in canonical regular expressions: `(a|b)`, `(a)*`, `(a)+`.

Character classes require that their component characters be quoted: `["a"-"z"]`. JavaCC also allows multiple character groups in a single character class, e.g. `["a"-"z", "A"-"Z"]`. Character classes are inverted with a `~` symbol before the opening brace: `~["c"]`.

Tokens are given names using angle brackets, e.g. `< SYM_EQ : "=" >`.

The full documentation for JavaCC is available online at `https://javacc.java.net/doc/docindex.html`

4 Goals

For this assignment, you must implement two Java classes: `edu.purdue.cs352.parser.Lexer` and `edu.purdue.cs352.parser.LexerManual`. The former will be generated by JavaCC, and a template for it is provided in `edu/purdue/cs352/parser/Lexer.jj-template`. The latter you must write by hand, `edu/purdue/cs352/parser/LexerManual.java`.

Both classes have the same functionality:

```
java edu.purdue.cs352.parser.Lexer <filename>
java edu.purdue.cs352.parser.LexerManual <filename>
```

When either lexer is run, it expects a filename as the first argument. It reads that file and writes the lexeme for each token encountered in the file, one per line. If an error is occurred, an exception is thrown (you may define the precise exception yourself). Please note that since only lexemes are being printed, the token names are irrelevant for this assignment. For future assignments, only the token names for identifiers and numeric literals will be important (`IDENTIFIER` and `INT_LITERAL`, respectively).

A number of examples are provided in the `examples` directory. These examples will help in testing, but are *not* the examples that will be used to grade your submission. In particular, none of the provided examples are invalid MiniJava. You should create your own examples to test for errors or corner cases in your own lexer.

4.1 Example

Given the following file in `examples/test.java`:
```
trust me i_m a* minijava program!  yay
```

Your lexer should output the following:
```
trust
me
i_m
a
*
minijava
program
!
yay
```

5 Submission

You will submit your project code via the `turnin` command. Please include the entire project code directory, including all the class files provided by us, the `bin` and `examples` directories, etc.

To turn in a project directory: `turnin -c cs352 -p proj1 <project directory>`

This project is due by Friday, January 31, 2014 at 12:30PM Purdue time. Late submissions will not be accepted.

6 Grading

Your code will be tested on both correct and incorrect source code examples. We will test your code on machines similar to those in the Linux labs used by this course's PSOs. Your grade will be based on correctness in the following criteria for both the generated and hand-written lexers:

- Correct lexing of symbols and keywords.
- Correct skipping of both C++/Java-style and C-style comments.
- Correct lexing of identifiers.
- Correct lexing of numeric literals.
- Rejection of lexically incorrect program source.
- Correct and unlimited handling of any number of tokens (files of any length).