Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2000 Segment I: Lexical Analysis – Project Assignment Wednesday, September 6

DUE: Monday, September 11

Your scanner must be able to identify tokens of the Decaf and Espresso languages, two Java-like languages we will be compiling in 6.035. Note that the two languages have the same tokens, and so there are tokens in Decaf which will always be parsing errors. This is to make Espresso a strict superset of Decaf. The languages are described in Handout 6 and Handout 7. Given these language specifications, you must figure out what the tokens in Decaf/Espresso are.

Your scanner should note illegal characters, missing quotation marks and other lexical errors with reasonable and specific error messages. The scanner should find as many lexical errors as possible, and should be able to continue scanning after errors are found. The scanner should also filter out comments and whitespace.

What to Hand In

Provide a gzipped tar file named `username-scanner.tar.gz` in the directory `6.035` under your Athena home directory, with `username` replaced by your Athena user name. This file should contain a copy of the token declarations passed to JLex (named `Yylex`), all relevant source code, and a Makefile. Additionally, you should provide a Java archive, produced with the `jar` tool, named `username-scanner.jar` in the same directory.

Unpacking the tar file and running `make` should produce the same Java archive. The main entry point of your scanner should be a **static** function called **main** in a class named **Compiler**. With the `CLASSPATH` environment variable set to `username-scanner.jar:/mit/6.035/classes`, typing `java Compiler` <*filename*> `-target scan` at the command line should run your scanner.

Some sample code has been provided in the `6.035` locker to help you implement this interface. `/mit/6.035/provided/scanner/Makefile` can be copied into your 6.035 directory, and it will create the correct tar and jar files. See the comments at the top of this file for more details. A test script has also been provided to check that the tar and jar files exist, and runs the scanner in the jar file using the standard interface. This script can be run by typing `add 6.035; run_scanner` <*filename*>.

Your scanner should output a list of errors, if any, to the terminal, and create an output file containing a table with one line for each token in the input where each line contains the line number of the token, the kind of token, and the attribute string (see below). (See the general project specification for information on getting the command line to work.)

Consult your TA if you plan on submitting a scanner that does not behave in this manner.

Finally, be sure to submit hardcopy documentation for this project assignment, as described in Handout 4.

JLex

You will not be writing the scanner from scratch. Instead, you will generate the scanner using JLex, a Java scanner generator from Princeton University. This program reads in an input specification of regular expressions and creates a Java program to scan the specified language. More information on JLex (including the manual and examples) can be found in Handout 9 on the web at

```
  http://www.cs.princeton.edu/~appel/modern/java/JLex/
```

and in Section 2.5 of the Appel text. It is suggested that you follow the design approach outlined in the text to make your life easier in the next project assignment (the parser). You should definitely use the `java_cup.runtime.Symbol` class and define all your tokens in a `sym` class to ensure that your scanner is compatible with the CUP parser generator.

To get you started, we have provided you with a template for the lex file in

```
  /mit/6.035/provided/scanner/scanner.lex
```

which you should copy to your work directory and fill in with the appropriate regular expressions and actions. This template contains useful directives and macros. Read the JLex documentation for more information on what you find there. In order to run JLex on this file, execute the following commands:

```
setenv CLASSPATH /mit/6.035/classes
java JLex.Main scanner
```

Note that you will also need to have set the CLASSPATH as above to run your scanner program.

You may also look in

```
  /mit/6.035/provided/scanner/Tiger
```

for a more complete "template" for the expected handin, including a Makefile, and some supporting code. It comes from the Tiger source code on the web.

Details

JLex will output `scanner.lex.java`. Note that JLex merely generates a Java source file for a scanner class; it does not compile or even syntactically check its output. Thus, any typos or syntactic errors in the lex file will be propagated to the output. You should fix any such errors in the lex file before handing in your project.

In order to interface to the parser in the next assignment, your scanner should have a class named `lex` that includes the method:

```
   public java_cup.runtime.Symbol nextToken()
```

The `java_cup.runtime.Symbol` class provides a basic abstract representation for tokens. It contains 4 basic fields for information about the token:

```
type - the symbol type (a value of type int from the sym class)
left - is the left position in the original input file
right - is the right position in the original input file
value - the lexical value of type Object
```

These values are passed to the constructor for `Symbol` in this order. You can also look at the source code for the `Symbol` class in the scanner directory.

The type of a `Symbol` will be used by the CUP parser generator in the next project assignment. Every distinguishable terminal in your Decaf/Espresso working grammar must have a unique type (an integer) so that the parser generator can distinguish them. These integers should be consecutive integers beginning at 1. For program readability, you should equate each token kind to something more descriptive: for example, `sym.RETURNS = 7`. These constants should be defined as `public static final int`s within the `sym` class.

Each of the reserved words is distinguishable in the grammar, so they should each be assigned a unique kind. Some classes of tokens, however, take on many values but need not have a unique kind for each value. An example of this type of token is integers. The grammar does not distinguish between two different integer values. Thus all integers have the same kind. The value for each integer token is saved in the `value` field of the `Symbol` class.

The `value` field of `Symbol` contains descriptive information about the token. Note that the `value` field is of the `java.lang.Object` type, and thus can store a reference to any object of any type. For this part of the project, we will simply use it to store the attribute strings of some tokens, although it will be used to store more complex information in the future.

The `Symbol` class does not contain any information about the token's line number; instead it stores the values of the token's absolute position in the input file. There are two possible approaches your team can take to generate error messages that include line numbers. The approach outlined in the textbook by Appel is to define a global `ErrorMsg` object that stores all the positions of the line breaks in the input file, and calculates the row and column location of a given token with this information. Should you find this approach distasteful, another way is to subclass the `Symbol` class with a subclass that stores more information (specifically, the line number) about the token to use in error messages. There will be no differences in grading or credit for either approach, and more information on both can be found in the scanner directory.

Although most of the lexer will be generated for you, you still must create a `Compiler` class that reads the command line and calls the generated lexer if the `SCANNER` or `DEFAULT` target is requested, and behaves as specified in *What to Hand In*.

Test Cases

The provided test case for the scanner is a file containing an instance of all valid tokens in Decaf/Espresso. The test case is found in `/mit/6.035/provided/scanner/tests/public`. If your scanner passes this test case (generates all the correct tokens with correct line numbers and attribute strings) you get 10 points of the possible 30 points for implementation. The other 20 points will be based on how well your scanner performs on hidden test cases.