PA 1: IC Lexical Analysis

due: 5pm, Friday, Sept. 23

CSCI 434T
Fall, 2011

Overview

In this programming assignment, you will implement the scanner for your IC compiler. The IC language specification document is available on the course web page. You will build the scanner using the JFlex lexical analyzer generator. Examples and documentation for this tool can be found on the JFlex home page and from the course resources web page.

Implementation Details

You will implement the lexical analyzer using JFlex. You will also build a driver program for the lexer, and a test suite. You are required to implement the following:

- `class Token`. The lexer returns an object of this class for each token. The `Token` class must contain at least the following information:
  - `id`, an identifier for the kind of token matched (where the type of `id` should be the `sym` enumeration below);
  - `value`, an arbitrary object holding the specific value of the token (e.g. the character string, or the numeric value);
  - `line`, the line number where the token occurs in the input file.

  The different kinds of tokens must be placed in a file `sym.java` containing an enumeration `sym` with the following structure:

  ```
  public enum sym {
    IDENTIFIER,
    LESS_THAN,
    INTEGER,
    ...
  }
  ```

  Note: in the next assignment, this file will be automatically generated by the parser generator `java_cup`.

- `lexer.flex` specification. Compiling this specification with JFlex must produce the `Lexer.java` file containing the lexical analyzer generator. The generated scanner will produce `Token` objects.

- `class Compiler`. This will be the main class of your compiler at the end of the semester. At this point, this class is just a testbed for your lexer. It takes a filename as an argument, opens that file, and breaks it into tokens by successively calling the `next_token` method of the generated lexer. The code should print a representation of each token read from the file to the standard output, one token per line. Your output must include the following information: the token identifier, the value of the token (if any), and the line number for that token. At the command line, your program must be invoked as follows:

  ```
  java ic.Compiler <file.ic>
  ```

  I have given you code in the `ic.Compiler` class to handle the optional command line argument "`-d`", as in:

  ```
  java ic.Compiler -d <file.ic>
  ```

  This option will turn on debugging messages generated by calls to `ic.Util.debug(...)`. (See the `ic.Util` javadoc and `Compiler.java` source code for more details.) If "`-d`" is not provided, calls to `ic.Util.debug(...)` will have no effect.

- `class LexicalError`. Your lexer should also detect and report any lexical analysis errors it may encounter. You must implement an exception class for lexical errors, which contains at least the line number where the error occurred and an error message. Whenever the program encounters a lexical error, the lexer must throw a `LexicalError` exception and the main method must catch it and terminate the execution. Your program must always report the first lexical error in the file.

I have provided a few utility methods in the `ic.Util` class. You are free to use these methods in your code.

Code Structure. All of the classes you write should be in or under the package `ic`, containing the following:

- the class `Compiler` containing the main method;
- the `ic.lex` sub-package, containing the `Lexer` and `sym` classes;
- the `ic.error` sub-package, containing the `LexicalError` class.

Testing the scanner. You must test your lexer. You should develop a thorough test suite that tests all legal tokens and as many lexical errors as you can think of. We will test your lexer against our own test cases – including programs that are lexically correct, and also programs that contain lexical errors.

SVN and Eclipse. You are required to manage your group project with `svn`. This is a useful tool for managing the concurrent code development by multiple persons. Such a tool will become more useful in the following assignments, which will be significantly larger than this first assignment. You should therefore use this assignment as a chance to set up your code production and testing process. You may also consider the automation of this process using makefiles, shell scripts or other similar tools.

I also strongly recommend using the Eclipse IDE. Eclipse has integrated `svn` support, as well as many other useful features such as code navigation, text completion, unit testing, and debugging. All of these can significantly help increase your productivity in this project. To further encourage you to use Eclipse, I will provide an Eclipse starter project for the project and we will spend some time during lab this week setting things up in your accounts.

Using Scala. If you wish to use Scala:

- Download the Scala version of the starter project.
- Everything above applies to the Scala project as well. You can run your IC compiler on the command line as follows:

  ```
  scala ic.Compiler <file.ic>
  ```

  The "-d" flag works as above.
- I believe the JFlex Java output and Scala will interoperate without problem. That is, you should be able to define `Token` as a Scala class and then create `Token` objects in your Java code as usual. I do, however, suggest you leave `sym` be a Java enumeration since that file will become an auto-generated file when we write the parser.
- If Eclipse gives you spurious error messages, run "make" from the command line and then "Clean..." and "Refresh" the Eclipse project. I've noticed that Eclipse occasionally fails to recompile Java files properly when it becomes sufficiently confused by compile errors in Scala source code.

Submission

To simplify submission and grading, you will turn in your programs by just checking the final versions of your code and supporting files into your `svn` repository by the deadline. (Please provide a descriptive message, such as "pa1 submission", when you commit the version you wish me to look at.) I will check out that version for grading.

As in any other large program, much of the value in a compiler is in how easily it can be maintained. For this reason, a high value will be placed here on both clarity and brevity – both in documentation and code. Make sure your code structure is well-explained in your write-up and in your javadoc documentation. (See the `readme.txt` file for how to generate javadoc from your code.)

The files you are required to submit for this assignment are:

- A brief, clear, and concise document (no more than 1-2 pages) describing the your code structure and testing strategy. Include a list of regular expressions for all the tokens that your lexer recognizes. Make sure you mention any known bugs and other information that may be useful when grading your assignment.
- All of your source code and test cases.

Your project directory should be organized as follows:

- `readme.txt` - a quick overview of how to build and run the project.
- `/ic` - all of your source code.
- `/test` - your test cases.
- `/tools` - the JLex utility that you will use in the project. You should not modify this.
- `/doc` - the generated javadoc documentation. You should not need to manually modify this.
- `/writeup` - your project write up.
- `Makefile` - `make` script to compile the file from the command line.