PA 1: IC Lexical Analysis

due: 10pm, Friday, Feb. 15

CSCI 434T
Spring, 2019

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. This project and the follow-on pieces should be written in Scala.

I will provide git repositories on our new GitLab server for each group. The repo will with an IntelliJ project that you will need to finish setting up according to the directions in the lab tutorial.

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 Java CUP parser generator.

- `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:

  ```
  scala -classpath bin ic.Compiler <file.ic>
  ```

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

  ```
  scala -classpath bin ic.Compiler -d <file.ic>
  ```

This option will turn on debugging messages generated by calls to `ic.Util.debug(...)`. (See the `ic.Util` and `Compiler` source code for more details.) If "`-d`" is not provided, calls to `ic.Util.debug(...)` will have no effect. You should use this printing mechanism to aid in testing and debugging your code, so that you can selectively turn on and off whatever logging you would like.

- `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.

Output Format. As mentioned above, you are free to print out the relevant information about tokens in any reasonable way provided that you print them out one per line with no blank lines between them. The exact content of error messages is left to you. In addition, the *last* line printed by your code should be either

```
Success.
```

or

```
Failed.
```

depending on whether or not any lexical errors were found. Please match those lines exactly to ensure my test scripts can properly validate your code. For similar reasons, the output should not contain any other text beyond what is specified here.

Utility Functions. I have provided a few utility methods in the `ic.Util` class. You are free to (and should!) use these methods in your code. In particular, make use of `Util.debug` to print diagnostic messages for debugging. Also, use Scala's `assert` function to assert that specific conditions (e.g.: preconditions, postconditions, invariants) are always true at run time.

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. There are two types of tests that you should consider writing:

- Specification Tests: These test the behavior of the whole system. In this case, that means running your compiler on different input files to ensure that it meets the requirements layed out in this handout. You should consider automating your testing as much as possible to make it easy to re-run them. I can help out with that if you'd like.
- Unit Tests: These tests exercise the behavior of individual components of your system in a modular way. I strongly suggest writing unit tests as you develop your code. Most IDEs come with unit testing support. The lab tutorial demonstrates how to run unit tests for an IntelliJ project.

We will test your lexer against our own test cases – including programs that are lexically correct, and also programs that contain lexical errors.

Scala. Scala and Java are designed to be interoperable and used together. Most of the code you write will be in Scala, but some of the auto-generated scanner and parser code will be in Java. This should not cause any problems. In particular, 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.

Submission

You will turn in your programs by simply pushing your final version of your code and supporting files to the GitLab server 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. Include all test files, etc. in your repository.

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-documented.

Also include a brief summary of your project in `README.md`. At this, simply describe your basic code structure and testing strategy. Also mention any known bugs and other information that may be useful when grading your assignment.

Your project directory should be organized as follows:

- `CONFIG.md` - a quick overview of how to build and run the project.
- `README.md` - your project write up.
- `/src` - 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.
- `Makefile` - `make` script to compile the file from the command line.

Once you have push what you believe to be your final version, please verify all of your code has been added and committed properly. You can do this by simply inspecting the files through the servers web interface, but the most reliable way is to clone a new copy of your project into a temporary directory, build it, and run it in a few short examples.