CS 301 Spring 2016
Project Phase 1
February 9 – February 18

IC Lexical Analysis

Plan

For the first phase of the compiler implementation project, you will implement the scanner (a.k.a. lexer) 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 website (`http://jflex.de/manual.html`) and linked from the course website. This project uses some Java to interface with JFlex, but most of the rest of the project will be written in Scala.

Implementation Details

You will implement the lexical analyzer using JFlex, 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:

  ```
  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` 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.

Nested ML-style Comments Recognizing nested ML-style comments `(* like (* this *) one *)` correctly requires more than regular expressions. One theoretically pure option is to make them part of our grammar later, but doing so would introduce a painful amount of pollution into the grammar. Instead we will recognize these comments in the lexer by taking advantage of the actions associated with patterns to keep and use state for nested comment recognition. This is a "hack" in the sense that it requires strictly more power than regular expressions offer, so our lexer will not compile to a pure DFA. We will discuss.

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

Programming Tools

Please use Mercurial and Eclipse as outlined in the previous Tools handout and on the course webpage. Version control will pay off to support collaboration well and revert to old versions if things go wrong. It is especially useful as the project grows.This project phase is much smaller than the following phases. You

should therefore use this phase 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 provide small pieces of starter code as an Eclipse project in the `cs301-labs` Mercurial repository on Bitbucket, so using Mercurial and Eclipse makes it easy to get and integrate this code. If all of your team is comfortable and proficient with other tools (*e.g.*, the IntelliJ Scala IDE or git), feel free to use them, but please let me know. When sharing a repository with teammates on Bitbucket or elsewhere, the repository **must be private**, shared with only your team and me. (I am `bpw` on Bitbucket, Github, and GitLab.)

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, but a few notes to ensure there are no issues:

- 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` as 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 clean" 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 committing and pushing the final versions of your code and supporting files into your shared repository by the deadline. (Please provide a descriptive message, such as "Phase 1 submission", or a tag when you commit the version you wish me to grade.)

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.

Your project directory should be organized as follows:

- `readme.txt` - a quick overview of how to build and run the project.
- `/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 committed and pushed what you believe to be your final version, please double-check that all of your code has been added, committed, and pushed properly. One foolproof way to do this is to clone a fresh copy of your project from Bitbucket to a separate directory, build it, and run it in a few short examples.