Compiler Project 1: Scanner

COMPSCI 322 — Computer Languages and Compilers — Spring 2015

February 9, 2015

Important Information

- **Deadline:** Your project archive, including all code and the project report, must be uploaded to your group's Project 1 dropbox by **Monday, February 23 at 11:59 p.m.** (Whitewater time).
  - Multiple submissions are allowed until the deadline, so you can submit last-minute bug fixes if you have any. I grade only the most recent submission (prior to the deadline) for credit.
  - Late work is accepted with a 20% penalty from Tuesday, Feb. 24 until Monday, March 2 (at 11:59 p.m.), and with a 40% penalty from March 3 until March 9 (at 11:59 p.m.). You may submit a project only once after the deadline. No credit will be given for this project after March 9 for any reason unless your group has made arrangements with me before March 9.
  - Extensions are given only for circumstances that affect more than half of a group (or the entire class). For example, one ill member does not usually merit an extension, but two or three members who are ill or injured (with documentation) at the same time might earn the group an extension.
- **Fix & Resubmit grades**: Your project code must compile with no compile-time errors when you submit it. If your code does not compile for any reason (even a simple syntax error), your project will not be graded. Instead, you will receive a "Fix & Resubmit" grade, which allows you the chance to make one more submission within one week for a 30% penalty. Projects that are not fixed and resubmitted within one week of grading will receive a grade of 0 for the code.
- **Collaboration:** See the collaboration and external help policy, posted separately on D2L.
- **Grading:** This project is worth 10% of your final grade for the course. About 75% of the project grade is based on correctness and completeness of the program. The remainder is based on the project report.

Requirements

Your scanner must be able to identify tokens of the Decaf language, the simple imperative language we will be compiling in this course. The language is described in the Decaf language specification document, posted on D2L under "Compiler Projects".

Your scanner should note illegal characters, missing quotation marks, and other lexical errors with reasonable error messages, including line numbers where the errors occur. 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 not in string and character literals; do not preserve these as tokens.

When compiled, your scanner must accept a command-line argument as follows:

```
# C/C++
./a.out source-filename                    # or whatever executable name you use

# Java
java -jar jarfilename.jar source-filename  # or whatever JAR file name you use
```

Output Requirements

1. Your scanner must construct a list of 4-tuples ("quadruples", if you prefer) of the form ⟨ *line-num, column-num, token-type, lexeme* ⟩, in the order they are identified. Your scanner must define a good object type to represent these 4-tuples. `Token` is a fine name for this type. (Probably it will be a class, but in C/C++ it could be a `struct`.) Provide a way to store this data within the program so it can be used by other compiler components later (e.g., a field in the main class or a global variable).

2. For easier string processing later, your scanner must construct a "string table" that assigns an unique integer ID to every unique string literal that is scanned. See the sidebar on page 66 of *Engineering a Compiler* (*EaC)* for more information. A hashtable would be an excellent choice for this; if you don't remember how they work, see Appendix B.4 in *EaC*. (You will be glad you did this when you reach the code generation stage!)

3. Your scanner must include a facility to display each token's data on a separate line, for easier debugging and grading. Each line *must* contain the following information, from left to right:

   (a) The line number where the token appears

   (b) The token-type, if not self-explanatory. This *must* be one of the following strings: `CHARLITERAL`, `INTLITERAL`, `BOOLEANLITERAL`, `STRINGLITERAL`, `IDENTIFIER`. (These do not have to match your internal token-type names.)

   (c) The lexeme (text of the token). For `STRINGLITERAL`s and `CHARLITERAL`s, this must be the original text of the token, including quotes and escaped characters. (*Suggestion*: Use the string table to look this up.)

   Example for: `print(‘‘Hello, World!’’);`

   ```
   1 IDENTIFIER print
   1 (
   1 STRINGLITERAL "Hello, World!"
   1 )
   1 ;
   ```

4. Each error message must be printed on its own line, *before* the error's location. Error messages must include the file name, line number, and column number where the error occurs.

Testing

A set of test cases and their corresponding outputs will be posted on D2L. It's best if your scanner passes all of these test cases when you turn it in. I will test your scanner against these test cases, plus others. Points are given based on the number of passed tests.

Implementation Hints

1. Before writing any code, use the "Lexical Considerations" and "Reference Grammar" sections of the Decaf Language Specification document to identify what tokens are allowed. Make a list of the possible token types. You may want to create an enumerated type `TokenType` with values for each possible token type.

2. Consider using buffered input, as suggested in *EaC* pp. 69-71. Most languages provide facilities for buffered input. See the documentation for the standard input and output APIs in your language for more details.