Lexical Analyzer

CMS 330, Spring 2017

Due Sunday, February 4, at 11:59 PM

Description

Write a lexical analyzer for a small programming language. This project is based on the language described in Per Brinch Hansen's *On Pascal Compilers*, the *TinyBasic* language, and Bob Nystrom's *Lox* language described in the free book *Crafting Interpreters*.

This is Phase I of a two-phase project. After completing this part, you'll have a lexer that can take an input program and convert it to a stream of tokens. The second part, which we'll talk about next week, is to create a parser and interpreter that can **actually execute the program** defined by the token sequence.

The project zip file includes the example programs we reviewed in class, a set of test programs, and a Python test script. Run the script using

```
prompt$ python test.py
```

If you report the correct tokens for each test program, you'll receive full credit. There are 10 tests, including a working build, and each one is worth 10 points. As always, I reserve the right to deduct points for programs that are unclear, undocumented, or exhibit poor style.

You may complete this project in Java. Upload your finished `Lexer.java` and `Token.java` files to your project repository on GitHub.

Output

Your lexer **should not print anything to the console**. It should only return an `ArrayList` of `Token` from the `analyze` method. The test harness will print the tokens and verify that they are correct.

Token List

Here is the complete list of tokens your program must recognize, with the token's symbol in the left column and its name in the right column.

Keyword names are *not* case sensitive. Your program must recognize keywords in any mixture of upper and lowercase letters.

A few tokens have special instructions. Read the following section carefully.

`and`	`AND`
`:=`	`ASSIGN`
`:`	`COLON`
`,`	`COMMA`
`{ }`	`Comment (see below)`
`/`	`DIVIDE`
`end`	`END`
`end of file`	`EOF (see below)`
`=`	`EQUAL`
`>`	`GREATER_THAN`
`>=`	`GREATER_THAN_OR_EQUAL`
`if`	`IF`
`integer`	`INTEGER (see below)`
`(`	`LEFT_PAREN`
`<`	`LESS_THAN`
`<=`	`LESS_THAN_OR_EQUAL`
`-`	`MINUS`
`%`	`MOD`
`identifier`	`NAME (see below)`
`not`	`NOT`
`<>`	`NOT_EQUAL`
`or`	`OR`
`+`	`PLUS`
`print`	`PRINT`
`program`	`PROGRAM`
`return`	`RETURN`
`)`	`RIGHT_PAREN`
`" "`	`STRING (see below)`
`sub`	`SUB`
`*`	`TIMES`
`while`	`WHILE`

Token Details

`NAME` is used for identifiers starting with a lowercase or uppercase letter followed by any sequence of lowercase letters, uppercase letters, digits, or the underscore character. Report the name of the identifier as the value of the token.

`INTEGER` is used for literal integer values. The only numbers you must recognize are **positive integers** created from the digits 0-9. You do not need to recognize negative numbers, decimals, or scientific notation. Report the value of the number as the value of the token.

`EOF` is reported when the analyzer reaches the end of the input file. This should always be the last token recognized in programs that do not have errors.

Strings are enclosed in double quotes; for example,

```
"Hello, World!"
"This is a string."
```

The value of the `STRING` token is a Java `String` with the text enclosed in the quotes. You can assume, for simplicity, that you don't have to deal with escape characters or quotes inside a string.

Comments are enclosed in curly braces, { and }. Anything from the first instance of a left brace to the next appearance of a right brace is ignored. Note:

- Reaching the end of the file in a comment is an error. Your program should call `Driver.error` with the message `End of file in comment.` and the line number of the error.
- An unmatched right brace } is an error.
- Comments may break across lines.
- Curly braces are the only way to denote comments in this language.
- You do not need to generate a token for comments.

The lexer ignores whitespace characters. Any other character should generate an error message by calling `Driver.error`.