CompilerDesign
Project 1-A
Simple calculator language — Lexical analysis

1 Language: Simple calculator

Our first language provides only the ability to express *a sequence of arithmetic expressions on integer literals*. When compiled, each expression will be evaluated and its results printed. The expressions will use *modified prefix notation* (with parentheses, sometimes called *LISP notation*). This form of expression is less familiar to programmers, but allows for simpler parsing with lack of ambiguity about evaluation priority.

Each expression, when evaluated, yields an integer value. Examples of expressions in our language are:

```
4
(+ 2 3)
# A comment between expressions.
(- 3 97) # A comment after an expression.
(/ (* 13 100) 44)
(% (/ 88 11) (- 13 2))
```

2 Lexical analysis

The *lexeme* of this language comprises the following elements:

- **Operators:** A set of integer-arithmetic operators (`+`, `-`, `*`, `/`, `%`).
- **Enclosures:** Parentheses (`(`, `)`).
- **Integers:** A sequence of one or more decimal digits.

Lexical analysis is performed by a *lexer* or *scanner*.[1] This tool reads the sequence of characters that comprise a *source code file* and converts them to a *sequence of tokens*. Each *token* captures one element of the lexeme.

The lexer should skip over comments when producing the sequence of tokens. Any symbols or constructions that are not a valid part of the lexeme should trigger an error that (a) stops processing, and (b) shows the user clearly where the invalid characters are used.

[1] Because the Java class `Scanner` is so familiar to Java programmers, we will call our lexical analyzer a *lexer* to avoid confusion.

3 Getting started

Begin by grabbing the starting source code at:

`bit.ly/cosc-371-p1a`

Extract the archive into a new directory, and examine the code that is provided:

- `Compiler.java`: This file is a completely written top-level driver of the compiler. You will see that it opens a source code file whose name is provided by the user at the command line, and then reads the source code into a `List<Character>` object. It then passes that list of characters to the `Lexer.parse()` method, which does the work scanning the source code and returning a `List<Token>` as a result of the scan.

- `Lexer.java`: The `parse()` method is where you will do most of your work. You will see that it currently parses some tokens, but it is not complete.

- `Token.java`: A standalone class for representing each token. Note the *nested enumeration*, `Type`, that provides a unique constant to represent each member of the lexeme. You can see (in `Token` and `Lexer`) that this enumerated type can be referred to as `Token.Type`, allowing one to choose a particular value for, say, an opening parenthesis by specifying `Token.Type.OPENPAREN`.

- `Utility.java`: A class that stores some useful constants and that provides methods for communicating information to the user. In particular, note the following methods for such communication:

  - `error(String message)`: Emit a simple error message for a compilation error and halt.
  - `error(String message, int position)`: Emit an error message for a compilation error and then, based on the source code that is set by the `Compiler` after reading it from the file, mark a specific position in the source code by showing the line of code and marking a location on that line. (Providing the `position` in the source code is enough to trigger that output.) Finally, halt.
  - `abort(String message)`: Emit an error message about a failure internal to the compiler (and not a source code mistake), and then halt.
  - `warn(String message)`: Emit an error message about a possible problem and return (allowing processing to continue).
  - `debug(int level, String message)`: Emit a debugging message if the given `level` is at least the `_debuggingLevel` set at compilation.

4 Your assignment

Complete the lexer for our simple-calculator language. Complete the `Lexer.scan()` method, and add any `Token` subclasses needed to represent the sequence of lexical elements detected. (Follow the form of the examples provided!)

A good implementation of the lexer should:

- Correctly identify each token irrespective of which tokens precede or follow.
- Correctly identify any invalid character sequence that does not form a token.
- Upon identifying an error, provide a useful error message to the user, highlighting the error.

5 How to submit your work

In order to submit your assignments, we will use a shared folder in *Google Drive*. I will create a shared folder for each of you, and within that will be a subfolder for each project. For this project, copy/upload **all of the Java source code files for this project**. Even if you have not changed some of the source code provided copy them all into that shared project subfolder so that it could be build with a simple `javac *.java` command.

This assignment is due on Thursday, Sep-12, 11:59 pm.