CompilerDesign
Project 1-B
Simple calculator language — Syntactic analysis

1 Syntactic analysis

As we worked out in class, the grammar for our *simple calculator* language is as follows:

```
<program>         ::= <expression list>
<expression list> ::= <expression> <expression list> | <EMPTY>
<expression>      ::= <integer> | <operation>
<operation>       ::= '(' <operator> <expression> <expression> ')'
<operator>        ::= '+'  | '-'  | '*'  | '/'  | '%'
```

For this assignment, we want to parse this grammar by implementing a *recursive descent parser*. This parser matches a sequence of `Token`s to these rules. When a rule is fully matched, the parser generates an object that represents the elements of the rule. These elements are linked together by higher-level rules to form an *Abstract Syntax Tree (AST)* that represents the program. That is, at the end of parsing a valid program, there should be a single `Program` object that contains a `List<Expression>`, where each element of that list is a set of linked objects that represent the `IntegerLiteral` and `Operation` objects (both are subclasses of `Expression`) that represent each calculation to be evaluated in the program.

2 Getting started

Begin by grabbing some starting source code at:

`bit.ly/cosc-371-p1b`

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

- `Compiler.java`: This file continues to be the top-level driver of the compiler. Since the last project, it now will take result of lexing (a sequence of tokens) and pass it to the parser. The parser, in turn, should return a `Program`—the aforementioned AST.

- `Expression.java`: An *abstract class* that defines what is stored for each *expression*. You should subclass this class to represent the *integer literal* and *operation* types of expressions.

- `Operator.java`: Another *abstract class* that defines what is common for each *operator*. Again, create subclasses to represent the specific operators.

- `Parser.java`: The incomplete collection of methods that make up our *recursive descent parser*. You will see the highest-level production rules implemented as a methods at the top of the code, and then a useful helper method or two at the bottom. Specifically note the presence of `consumeWhitespace()`, which can be called wherever needed to move through arbitrary amounts of whitespace between other elements.

- `Program.java`: The object created and returned by the top-level `Parser.parse()` method, representing the top of the AST.

- `Utility.java`: Unchanged from the previous assignment.

Notice, too, what is **not included** in the provided source code:

- `Lexer.java` and `Token.java`: Copy these from your Project 1-A solution.

- Various subclasses, mentioned above, that you will need to write.

3 Your assignment

Complete the parser for our simple-calculator language. You will notice that, in the middle of `Parser.java`, there is a comment that marks where you should begin implementing methods for each of the production rules.

A good implementation of the parser should:

- Correctly accept each well-formed expression in a program.

- Correctly rejects each malformed expression.

- Builds and returns an AST that represents a syntactically correct program.

- Helpfully signals errors to the users, displaying what when wrong and where in the code the error was detected.

4 How to submit your work

Copy/upload **all of the Java source code files for this project** in the `project-1-B` folder in your shared *Google Drive* folder for this course.

This assignment is due on Thursday, Sep-19, 11:59 pm.