Students:

This content is controlled by your instructor, and is not zyBooks content. Direct questions or concerns about this content to your instructor. If you have any technical issues with the zyLab submission system, use the **Trouble with lab** button at the bottom of the lab.

20.21 Ritchie: Mini Math Programming Language

Program 4

Due: Sunday 11/10/19 , 11:59PM

Grade: 90% Zybooks unit tests, 10% I will hand grade your assignments When I grade your assignments I will be looking for:

- Author documentation
- Good variable names
- Comments: Write appropriate comments and / or java documentation (for an example of this go to blackboard -> Programming -> Programming Guidelines -> Example program)
- Correct implementation of the classes (IE no hard coding the answers).

Extra Credit due date: 11/3/19, 11:59 PM. In order to earn extra credit on this program you may turn it in "early" (i.e. any time before the extra credit due date). If you submit your program after the extra credit due date then you will no longer be eligible for extra credit, even if you previously submitted before the due date. The extra credit is 15% of your earned score towards your program grade. This means if you get a 50 / 50 on this program and you turn it in before the extra credit due date, you would get an extra 15% of 50 putting your score at: 57.5 / 50. But if your score was 30 / 50 and you turned it in for the extra credit you would get an extra 15% of 30 putting your score at: 34.5 / 50.

Program Background

In this assignment you will be creating a mini compiler that can understand a mini math programming language. You will read in a file, tokenize the contents of the file using the "space" character as a delimiter, assign each token to its respective class, compute the result of each valid mathematical expression, then write out the result to a file called: "output.txt"

But first a little background…

A **Compiler** is a computer program that translates computer code written in one programming language (the source language) into another language (the target language). In your case you will be translating a mini math programming language into Java, then executing the Java code to find the result of a mathematical expression. Often a compiler is used to translate some high level

programming language (like Java) into a low level language (like Assembly or machine code), but in our case we can use it to translate one high level language into a another.

A compiler is made up four main parts: *pre-processing*, *lexical analysis*, *parsing*, and *semantic analysis*. In this assignment we will focus on 3 of the 4 parts: *lexical analysis*, *parsing*, and *semantic analysis*.

Lexical analysis can also be referred to as *tokenization*. You already did this in the previous assignment! So really this assignment is only addressing two new parts: *parsing* and *semantic analysis*. If you need a refresher on tokenization, please reread the description in the last assignment.

Parsing (also known as syntactic analysis) is the process of analyzing strings for syntactic correctness. What does that mean? Essentially parsing is checking that word is spelled correctly. For example, in Java when some one attempts to create a variable name that begins with a number, the compiler shows an error, this is because the parser has detected invalid syntax for the variable name. A parser is also used to determine that an entire line is structurally valid, for example:

```
int x = 5; // valid
5 = x int // not valid
```

The "5 = x int" would throw an error, this is because the parsing step of Java's compiler has determined that line to be structurally / syntactically invalid. For this assignment, the only thing your parser will need to worry about is if a number is a valid number, a math operation is a valid math operation, and the structure of the math expression is correct (more on this later).

Semantic Analysis is a process that gathers necessary semantic information from the source code. What does semantic mean? And what is the difference between semantic and syntax. Remember *syntax* is referring to valid grammar (i.e. something is spelled / structured correctly). Where as *semantic* refers to giving meaning to a word or expression. For example, while the *parser* might have determined that: int x = 5; is syntactically valid java code, the *semantic analysis* step determines that the user wants to create a variable called x, that is of type int, and assign it to the value 5. In your program, your semantic analysis step will determine the meaning of the given math operation (i.e. if the operation symbol is a "minus sign" then that expression will be doing *subtraction*, or if the operation symbol is a "plus sign" then that operation will be doing *addition*). Then your program will write out the computed out put based on the semantic analysis (i.e. if your semantic analysis step determined an expression will be doing subtraction, then subtract the two numbers associated with that expression and write out the result to the output file).

Program Description

The mini math language should essentially act as a rudimentary calculator, where a mathematical operation is applied to only two numbers. It will be comprised of mathematical expression that are specifically formatted as followed:

```
number1 number2 operation
```

For example, the following is a **valid** expression in our mini math language:

```
5 2 -
```

This means: "5 - 2" and therefore, when computed, the result should be: "3". The mathematical operations our mini math programming language excepts are as follows:

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulo)

The only valid number type should be an integer (i.e. floats, doubles, and String are considered invalid number types). If an output to a mathematical expression would be and non-integer number, then convert that result to an integer before writing out the result to the output file. For example the following are **invalid**:

```
asdf 5 -
5 t +
5.5 3 *
5.45 asdf %
```

Each number and the operation should be separated by a space character. Multiple spaces are also valid. So for example the following are both **valid** in our mini math programming language:

```
5 3 %
5         3          %
```

The order in a mathematical expression is important, for example the following are **invalid** due to the operations location in the expression:

```
10 - 3
 + 10 3
```

There can only be two valid numbers in a mathematical expression. For example the following are all **invalid**:

```
10 3 2 - -
3 5 6 7 +
6 -
```

There can only be one mathematical expression per line. For example the follow is **invalid**:

```
10 3 -        4 5 *
```

Your program should write out the computed result of valid mathematical expressions into a file called output.txt. For example if the following are the contents in some input file:

```
6 6 +
10 3 -
4 5 *
7 2 /
8      3       %
```

The resulting output .txt file should contain:

```
12
 7
 20
 3
 2
```

Notice the order of the results in the output file matches the order of the expressions in the input file. For invalid expressions your program should output error messages to the output.txt file. The following are the only error message types expected:

```
ERROR: Cannot divide by zero
ERROR: Invalid number: asdf
ERROR: Invalid Operator: =
ERROR: Invalid Expression
```

The "asdf" and "=" in the above example are just examples of an invalid token and and invalid operator (respectively). The order that errors should be caught are as follows:

- Invalid expression (i.e. not enough or too many tokens on a line, there should be exactly three)
  - The associated error statement is: ERROR: Invalid Expression
  - This error has the highest precedence
- Invalid operator (i.e. a character (or characters) is in the operator location and is not a valid operator)
  - The associated error statement is: ERROR: Invalid Operator: = Where the "=" is representative of the violating character, in this case the equals symbol
- Invalid LEFT number (i.e. the left number is checked for validity before the right number)
  - The associated error statement is: ERROR: Invalid number: left Where the "left" is representative of the violating number, in this case the string "left"
- Invalid RIGHT number (i.e. the right number is checked for validity after the left number)
  - The associated error statement is: ERROR: Invalid number: right Where the "right" is representative of the violating number, in this case the string "right"
- Cannot divide by zero (i.e. You are not allowed to divide a number by zero)
  - The associated error statement is: ERROR: Cannot divide by zero
  - This error has the lowest precedence
  - Note that modulo should also handle this error

Given the following input file:

```
not 5 op
not 5 +
5 asdf -
left right +
6 0 /
5 5 5 -
5 +
```

The output.txt file should contain:

```
ERROR: Invalid Operator: op
ERROR: Invalid number: not
ERROR: Invalid number: asdf
ERROR: Invalid number: left
ERROR: Cannot divide by zero
ERROR: Invalid Expression
ERROR: Invalid Expression
```

Your program should ignore empty lines and / or lines filled with only spaces on them. For example given the following input file content:

```
6 3 /

4 6 *


8 5 %
```

The output.txt file should contain:

```
2
24
3
```

Notice that you do not need to print empty lines in the output file.

Program to turn in:

You will turn in multiple files:

- Token.java
- Constant.java
- Add.java
- Subtract.java
- Multiply.java
- Divide.java

- Modulo.java
- Builder.java
- TokenBuilder.java
- Tokenizer.java
- Parser.java

It may seem like a lot, but most of the files are small and contain similar code. Also, I will be giving you the Parser.java file. The following goes over the contents of each file. You may add any fields or methods you deem necessary, as long as each file contains the specified class, methods, constructors, and fields. The files below are organized in the order that I recommend you accomplish this program (that is, I recommend you start with Token.java and finish with Tokenizer.java)

Token.java

Description: This class provides the basis to our whole mini compiler design. It will at the root of the polymorphic behavior in this assignment.

This file should contain:

- An **abstract class** called Token
- Fields:
  - A protected field of type Token that represents the left number in a mathematical expression
  - A protected field of type Token that represents the right number in a mathematical expression
  - A protected field of type String that represents the value of a token
- Constructors:
  - A constructor that takes in one input argument of type string. The protected field of type String should be assigned to (set equal to) this input.
  - A constructor that takes in two input arguments, each of type Token. The protected field of type Token representing the left number should be assigned to the first input argument. The protected field of type Token representing the right number should be assigned to the second input argument.
- Abstract Methods
  - Declare an abstract method called evaluate, that have no input parameters and returns type string.

Constant.java

This file should contain:

- A **class** called Constant that inherits from abstract class Token
- Constructors:
  - A constructor that takes in one input parameter of type String. This input parameter should be passed into the inherited constructor from Token as an argument.
- Methods

- Implement the inherited abstract method evaluate. Evaluate should return the inherited field of type String, representing the value of a token. If the string is not an int then the appropriate error message should be returned instead.
- Override the toString method and have it return the inherited field of type String that represents the value of a token

Add.java

This file should contain:

- A **class** called Add that inherits from abstract class Token
- Constructors:
  - A constructor that takes in two input parameters each of type Token. These input parameters should be passed into the inherited constructor from Token as arguments. The first input parameter should be the first argument to the inherited Token constructor, and the second input parameter should be the second argument to the inherited Token Constructor.
- Methods
  - Implement the inherited abstract method evaluate. Evaluate should return the String representation of the result from adding together the integer representations of evaluate calls on the inherited fields of type Token that represents the left number and the right number. Evaluate can also return an appropriate error message.
  - Override the toString method and have it return the left token's toString, the right token's toString, and the "+" symbol, each separated by a single space. For example it might return: "5 6 +".

Subtract.java

This file should contain:

- A **class** called Subtract that inherits from abstract class Token
- Constructors:
  - A constructor that takes in two input parameters each of type Token. These input parameters should be passed into the inherited constructor from Token as arguments. The first input parameter should be the first argument to the inherited Token constructor, and the second input parameter should be the second argument to the inherited Token Constructor.
- Methods
  - Implement the inherited abstract method evaluate. Evaluate should return the String representation of the result from subtracting together the integer representations of evaluate calls on the inherited fields of type Token that represents the left number and the right number. Evaluate can also return an appropriate error message.
  - Override the toString method and have it return the left token's toString, the right token's toString, and the "-" symbol, each separated by a single space. For example it might return: "5 6 -".

Multiply.java

This file should contain:

- A **class** called Multiply that inherits from abstract class Token
- Constructors:
    - A constructor that takes in two input parameters each of type Token. These input parameters should be passed into the inherited constructor from Token as arguments. The first input parameter should be the first argument to the inherited Token constructor, and the second input parameter should be the second argument to the inherited Token Constructor.
- Methods
    - Implement the inherited abstract method evaluate. Evaluate should return the String representation of the result from multiplying together the integer representations of evaluate calls on the inherited fields of type Token that represents the left number and the right number. Evaluate can also return an appropriate error message.
    - Override the toString method and have it return the left token's toString, the right token's toString, and the "*" symbol, each separated by a single space. For example it might return: "5 6 *".

Divide.java

This file should contain:

- A **class** called Divide that inherits from abstract class Token
- Constructors:
    - A constructor that takes in two input parameters each of type Token. These input parameters should be passed into the inherited constructor from Token as arguments. The first input parameter should be the first argument to the inherited Token constructor, and the second input parameter should be the second argument to the inherited Token Constructor.
- Methods
    - Implement the inherited abstract method evaluate. Evaluate should return the String representation of the result from dividing together the integer representations of evaluate calls on the inherited fields of type Token that represents the left number and the right number. Evaluate can also return an appropriate error message. (Do not forget that you cannot divide by zero)
    - Override the toString method and have it return the left token's toString, the right token's toString, and the "/" symbol, each separated by a single space. For example it might return: "5 6 /".

Modulo.java

This file should contain:

- A **class** called Modulo that inherits from abstract class Token

- Constructors:
  - A constructor that takes in two input parameters each of type Token. These input parameters should be passed into the inherited constructor from Token as arguments. The first input parameter should be the first argument to the inherited Token constructor, and the second input parameter should be the second argument to the inherited Token Constructor.
- Methods
  - Implement the inherited abstract method evaluate. Evaluate should return the String representation of the result of finding the modulus of the integer representations of evaluate calls on the inherited fields of type Token that represents the left number and the right number. Evaluate can also return an appropriate error message. (Do not forget that you cannot divide by zero)
  - Override the toString method and have it return the left token's toString, the right token's toString, and the "%" symbol, each separated by a single space. For example it might return: "5 6 %".

Builder.java

Description: This interface defines what it means to be a builder and is implemented by the TokenBuilder class

This file should contain:

- An **generic interface** called Builder that uses a *generic type* called E
- Methods:
  - Specify a method called build that has an input argument of type String and returns the generic type

Parser.java

Description: This abstract class will give the TokenBuilder some nice help with the parsing step of this program.

This file is given to you. If you look through it you will notice that I handle the operator and expression errors for you using some special looking code. Don't worry about that, your job is to look at where the createToken method is called within the parse method. Keep that in mind as you implement the createToken method in the TokenBuilder class. Notice the createToken method is abstract.

TokenBuilder

Description: This class will build and return a single master token given a line of text, this master token can be used to find the result of that line of text, whether that be a number or an error.

This file should contain:

- A **class** called TokenBuilder that inherits from the abstract class Parser and implements the interface Builder

- Fields:
  - A private field of type String that will holder the delimiter used for creating tokens
- Constructors:
  - A default constructor that assigns the private field of type String to a comma
  - A constructor that has two input parameters, the first of type String and the second of type ArrayList of Strings. private field of type String should be assigned to the input parameter of type String. The inherited ArrayList of Strings called "keywords" should be assigned to the second input parameter that is also of type ArrayList of Strings.
- Methods:
  - Implement the inherited method "createToken". The input argument is an Array of Strings, starting at index zero it should hold the first number of a mathematical expression, the second number, and the operator, in that order. Depending on the operator return the appropriate Token type. For example if the operator is a plus symbol "+", then return an Add object. The input arguments to the operator Token constructor should be two objects of type Constant, each holding one of the numbers from the mathematical expression.
  - Implement the "build" method. Return the result from calling the inherited method "parse". Where the input to parse should be an array of Strings formed by delimiting the String input to the build method.
  - Implement setters for the inherited keywords variable and for the private field of type String holding the specified delimiter.

Tokenizer.java (the name of this class should really be MathCompiler.java but I am trying to keep things simple for you)

This file should be very similar to the one from your last program. It should contain:

- A **class** called: Tokenizer
- Fields:
  - A private variable that is an ArrayList of Token objects. (This should is the same as last the program)
  - A private variable that is an ArrayList of Strings. This will hold the various operators to our mini math programming language
  - A private variable of type TokenBuilder.
- Constructors:
  - A default constructor that initializes the ArrayList of Token objects. Initializes the ArrayList of Strings and adds all of the necessary math operators to that ArrayList of Strings. Initializes the TokenBuilder object and sets it's keywords field and its private String field to a space delimiter.
- Methods:
  - Tokenizer should have a public method called: tokenizeFile. This method is the EXACT SAME as your previous program, the only difference is now you should read in from the file one line at a time (not one word at a time) and pass each line to the TokenBuilder

and store the token it returns in a similar manner as last program. You might also want to check for empty lines here.
    - Tokenizer should have only a getter for the ArrayList of Token objects (not a setter) This method should be the same as last programs
        - The getter should be called: getTokens and should return an Array of Token objects, NOT an ArrayList of Token objects
    - Tokenizer should have a method called: writeTokens This method is identical to your last programs with the exception of what you are writing out to the file now is the result of calling the evaluate method on each token.
- Override the toString method inherited from the Object class, have it return the same value as calling the toString method on the ArrayList of Token objects (This should also be the same as the last program)

Suggestions and Hints

- Remember to close your files
- You might find the String methods: Trim and isEmpty useful
- You might find the Scanner methods: hasNextLine and nextLine useful
- You might find Integers method: parseInt useful (notice that it throws and exception)
- If you are feeling unsure of how to use an ArrayList or some of the File IO objects, be sure to reread zybooks: ch 7.12 and ch 10.5.
    - You can also search on the internet for java's documentation on these objects (remember that java is owned by Oracle, so the documentation website url should be something along the lines of: docs.oracle.com/…etc)
- You may find the String class's "split" method useful
- I suggest for your delimiter you use the regex: "\\s+" I leave it to you to figure out what this does
- Remember you can test your code by creating your own file, let's say for example: Application.java, creating a class called: Application, and creating a "main" method. Then you can create your own tokenizer object and test some of the methods. (Just remember not to turn in this extra file)
- Do not turn in Token.java or Tokenize.java with a "main" method in either of these files.
    - So don't have this: public static void main(String[] args) … etc

LAB ACTIVITY 20.21.1: Ritchie: Mini Math Programming Language 60 / 60

Submission Instructions

Downloadable files

`Parser.java` and `test.txt` **Download**

Compile command

```
javac Tokenizer.java Token.java
Add.java Builder.java
```

```
Constant.java Divide.java
Modulo.java Multiply.java
Parser.java Subtract.java
TokenBuilder.java -Xlint:all -
encoding utf-8
```

We will use this command to compile your code

Upload your files below by dragging and dropping into the area or choosing a file on your hard drive.

Tokenizer.java	Drag file here or **Choose on hard drive.**	**Token.java**	Drag file here or **Choose on hard drive.**
Add.java	Drag file here or **Choose on hard drive.**	**Builder.java**	Drag file here or **Choose on hard drive.**
Constant.java	Drag file here or **Choose on hard drive.**	**Divide.java**	Drag file here or **Choose on hard drive.**
Modulo.java	Drag file here or **Choose on hard drive.**	**Multiply.java**	Drag file here or **Choose on hard drive.**
Parser.java	Drag file here or **Choose on hard drive.**	**Subtract.java**	Drag file here or **Choose on hard drive.**
TokenBu...r.java	Drag file here or **Choose on hard drive.**		

Submit for grading

Latest submission - 11:34 PM on 11/10/19 — Submission passed all tests ✓ — Total score: 60 / 60

☐ Only show failing tests — **Download this submission**

1: Unit test: test.txt ^ — 1 / 1

Test Correct File Input

Your output

```
No IOException should be thrown... GOOD!!
```

2: Unit test: wrongfile.txt 1 / 1

Test InCorrect File Input

Your output

```
IOException should be thrown... GOOD! The
wrongfile.txt (No such file or directory)
```

3: Unit test: Constant.java 2 / 2

Test evaluate and toString on valid token: 5

Your output

```
Correct! evaluate returned: 5
Correct! toString returned: 5
```

4: Unit test: Constant.java Invalid 2 / 2

Test evaluate and toString on Invalid token: "notatoken"

Your output

```
Correct! evaluate returned: ERROR: Invalid
Correct! toString returned: notatoken
```

5: Unit test: Add.java 2 / 2

Test evaluate and toString on valid Constants: 5 and 6

Your output

```
Correct! evaluate returned: 11
Correct! toString returned: 5 6 +
```

6: Unit test: Add.java Invalid 2 / 2

Test evaluate and toString on Invalid Constants: 5 and abcd

Your output

```
Correct! evaluate returned: ERROR: Invalid
Correct! toString returned: 5 abcd +
```

7: Unit test: Subtract.java 2 / 2

Test evaluate and toString on valid Constants: 10 and 3

Your output

```
Correct! evaluate returned: 7
Correct! toString returned: 10 3 -
```

8: Unit test: Subtract.java Invalid 2 / 2

Test evaluate and toString on Invalid Constants: 5 and dcba

Your output

```
Correct! evaluate returned: ERROR: Invalid
Correct! toString returned: 5 dcba -
```

9: Unit test: Multiply.java 2 / 2

Test evaluate and toString on valid Constants: 10 and 3

Your output

```
Correct! evaluate returned: 30
Correct! toString returned: 10 3 *
```

10: Unit test: Multiply.java Invalid 2 / 2

Test evaluate and toString on Invalid Constants: 5 and cdba

Your output

```
Correct! evaluate returned: ERROR: Invalid
Correct! toString returned: 5 cdba *
```

11: Unit test: Divide.java 2 / 2

Test evaluate and toString on valid Constants: 10 and 2

Your output

```
Correct! evaluate returned: 5
Correct! toString returned: 10 2 /
```

12: Unit test: Divide.java Invalid 2 / 2

Test evaluate and toString on Invalid Constants: bbbb and 5

Your output

```
Correct! evaluate returned: ERROR: Invalid
Correct! toString returned: bbbb 5 /
```

13: Unit test: Modulo.java 2 / 2

Test evaluate and toString on valid Constants: 23 and 7

Your output

```
Correct! evaluate returned: 2
Correct! toString returned: 23 7 %
```

14: Unit test: Modulo.java Invalid 2 / 2

Test evaluate and toString on Invalid Constants: aaaa and 5

Your output

```
Correct! evaluate returned: ERROR: Invalid
Correct! toString returned: aaaa 5 %
```

15: Unit test: TokenBuilder 10 / 10

Test CreateToken, tests each type of token

Your output

```
Correct! Expected: class Add
Correct! evaluate call returned correctly
Correct! toString call returned correctly
Correct! Expected: class Subtract
Correct! evaluate call returned correctly
Correct! toString call returned correctly
Correct! Expected: class Multiply
Correct! evaluate call returned correctly
Correct! toString call returned correctly
Correct! Expected: class Divide
Correct! evaluate call returned correctly
Correct! toString call returned correctly
Correct! Expected: class Modulo
Correct! evaluate call returned correctly
Correct! toString call returned correctly
```

16: Unit test: Tokenizer 6 / 6

Test getTokens with test.txt

```
Correct! Expected token toString: 5 5 +
Correct! Expected token toString: 5 5 -
Correct! Expected token toString: 5 5 *
Correct! Expected token toString: 10 0 /
Correct! Expected token toString: 40 0 %
```

Your output

```
Correct! Expected token toString: asdf 4 %
Correct! Expected token toString: 5 df +
Correct! Expected token toString: 5 5 =
Correct! Expected token toString: 5 5 5 *
Correct! Expected token toString: 5 5 *
Correct! Expected token toString: 7
Correct! Expected token toString: *
```

17: Unit test: Tokenizer 9 / 9

Test writeTokens with test.txt

Your output

```
Correct!! You have correctly written all o
```

18: Unit test: Tokenizer 9 / 9

Test writeTokens with test2.txt

Your output

```
Correct!! You have correctly written all o
```

5 previous submissions

11:31 PM on 11/10/19	60 / 60	View
11:16 PM on 11/10/19	60 / 60	View
10:55 PM on 11/10/19	51 / 60	View
10:54 PM on 11/10/19	42 / 60	View
10:52 PM on 11/10/19	42 / 60	View

Trouble with lab?