Problem Statement:

Create a language which accepts addition(+), subtraction(-), multiplication(\*), division(/) of integers. The order of these operations should not be considered ambiguous. Multiplication and division have equal priority. Addition and subtraction have equal priority. Multiplication and division take priority over addition and subtraction. The only number literals that should be accepted are non-negative integers. Note: This means you don’t have to worry about double negative numbers, such as –(–3).

Step 1: Creating the top-level grammar rule

First, let’s annotate our Calculator class with “@GrammarRule”. This confirms to LAPS that you, as the programmer, intend to make the class define a grammar rule. After the class is set as a grammar rule, let’s make a constructor for the grammar rule which accepts a MathExpression (another grammar rule we are yet to make). We can save that constructor argument in local field, so we can evaluate it after parsing.

Once parsing of the Calculator is done, LAPS looks for an instance method with no parameters and an annotation of “@RunAfterEachInit”. If such a method is found it is run with “this” being the Calculator instance created during parsing. In our case, the method we want run is called “calculate”. This will evaluate our MathExpression.

Side Note: Typically, it’s a good idea to skip whitespace in a language, so that is implemented using lines 7-8.

Calculator.java

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | | 8 | | 9 | | 10 | | 11 | | 12 | | 13 | | 14 | | 15 | | 16 | | 17 | | 18 | | 19 | | import edu.rit.gec8773.laps.annotation.GrammarRule;  import edu.rit.gec8773.laps.annotation.RunAfterEachInit;  import edu.rit.gec8773.laps.annotation.Token;  @GrammarRule  public class Calculator {  @Token(skip=true)  public static String WHITESPACE = "\\s+";  private MathExpression exp;  public Calculator(MathExpression exp) {  this.exp = exp;  }  @RunAfterEachInit  public void calculate() {  System.out.println(this.exp.evaluate());  }  } |

Step 2: Defining our MathExpression Grammar Rule

In our language, we basically have a list of numbers separated by binary operators (e.g. addition, multiplication, etc.). Since there isn’t currently a list interface in LAPS, we need to define a recursive Grammar Rule.

Let’s start out with the recursive case. This will consist of a number then a binary operator then another math expression. Our base case is pretty simple, as well. It only consists of a number.

The tricky part here is what order do you want to check these rules. That’s we should add “@Priority” annotations to indicate which rule gets considered first. The ordering is similar to Unix process niceness value, so the smaller the value, the further forward the rule is in the ordering (e.g. -2 then -1 then 5).

MathExpression.java:1

|  |
| --- |
| import edu.rit.gec8773.laps.annotation.GrammarRule;  import edu.rit.gec8773.laps.annotation.Priority;  import edu.rit.gec8773.laps.annotation.RunAfterEachInit;  import edu.rit.gec8773.laps.annotation.Token;  @GrammarRule  public class MathExpression implements Comparable<MathExpression> {  @Token public static String NUMBER = "\\d+";  private Integer value = null;  private BinaryOperator operator;  private MathExpression operandA;  private MathExpression operandB;  @Priority(-1)  public MathExpression(String number, BinaryOperator op, MathExpression exp) {  this.operandA = new MathExpression(number);  this.operandB = exp;  this.operator = op;  }  @Priority(1)  public MathExpression(String number) {  this.value = number == null ? null : Integer.parseInt(number);  this.operator = new BinaryOperator(null);  } |

Step 3: Finishing up the Grammar

In our MathExpression class, we accepted a BinaryOperator as one of our grammar rule parameters. Now, we must define what that is. First, let’s add a token type for the operators we support (+,-,/,\*). Once we define our token, we can accept it in a grammar rule. In our case, we define our grammar rule to only accept an operator.

BinaryOperator.java:1

|  |
| --- |
| import edu.rit.gec8773.laps.annotation.GrammarRule;  import edu.rit.gec8773.laps.annotation.Token;  import java.util.Arrays;  import java.util.Map;  import java.util.function.BiFunction;  import java.util.function.Function;  import java.util.stream.Collectors;  @GrammarRule  public class BinaryOperator implements Comparable<BinaryOperator> {  @Token public static final String OPERATOR =  "[\\+\\-\\/\\\*]";  private final BinaryOperation operation;  public BinaryOperator(String operator) {  this.operation = BinaryOperation.operations.get(operator);  } |

Step 4: Adding some Semantics

Let’s start out by defining our operations in an enum. The first parameter is the operator’s symbol; the second is the priority of the operation (smaller number means higher priority); the last is a function which takes in two integers and returns their result in whichever operation they are preforming.

To quickly and easily get each operation, we can use a map of operator symbols to their corresponding operations. Now that we have our corresponding operations, we can compare the operations priority and calculate our operation on two integers.

BinaryOperator.java:2

|  |
| --- |
| private enum BinaryOperation {  ADDITION("+", 2, Integer::sum),  SUBTRACT("-", 2, (a,b) -> a-b),  MULTIPLY("\*", 1, (a,b) -> a\*b),  DIVIDE("/", 1, (a,b) -> a/b),  NUMBER(null, 0, null);  public static final Map<String, BinaryOperation> operations =  Arrays.stream(BinaryOperation.values())  .collect(Collectors.toMap(  binaryOperation -> binaryOperation.symbol,  Function.identity()  ));  public final String symbol;  public final int priority;  public final BiFunction<Integer, Integer, Integer> computer;  BinaryOperation(String symbol, int priority, BiFunction<Integer, Integer, Integer> computer) {  this.symbol = symbol;  this.priority = priority;  this.computer = computer;  }  }  @Override  public String toString() {  return operation.symbol;  }  @Override  public int compareTo(BinaryOperator other) {  return operation.priority - other.operation.priority;  }  public int calculate(int operandA, int operandB) {  return this.operation.computer.apply(operandA, operandB);  }  } |

Step 5: Restructuring our Parse Tree into a Syntax Tree

You may have noticed by now that the recursive grammar rule definition in our MathExpresion class helps us easily create a linked list style parse tree. We want to make an abstract syntax tree, meaning we should order the tree in by the priority of each operator. To do this, we need to do some reordering of our nodes in the tree.

Here is the algorithm that we will use. If there is a value (meaning the MathExpression is a number), then we do nothing. Otherwise, if this MathExpression’s BinaryOperator has a higher priority than its operand B, then operand B becomes the new root node and replaces this MathExpression, but we still need this MathExpression so we store it in a temporary variable. To keep all the numbers in the same order, we need to copy the operand A from the old operand B to the operand B of the old this. To finish up, the old this replaces the new root’s operand A. Below is an example going through the algorithm.

**Key**

**Circles**: MathExpresions which are already ordered.

**Squares**: MathExpressions which are not ordered.

|  |  |
| --- | --- |
| Tree | Temporary Hold |
| this  B | Copy the root of the tree to a temporary variable.  this  B |
| B | Replace the current root of the tree with operand B.  this  B |
| B | Replace the old root’s operand B with operand A\*.  this |
| this  B | Replace B’s first operand with the temporary variable and we are done.  this |

MathExpression.java:2

|  |
| --- |
| private static final MathExpression temp =  new MathExpression(null);  @RunAfterEachInit  public void orderTree() {  if (value != null)  return;  if (compareTo(operandB) < 0) {  this.copyTo(temp);  operandB.copyTo(this);  this.operandA.copyTo(temp.operandB);  temp.copyTo(this.operandA);  }  }  @Override  public int compareTo(MathExpression other) {  return operator.compareTo(other.operator);  }  private void copyTo(MathExpression to) {  to.operator = this.operator;  to.operandA = this.operandA;  to.operandB = this.operandB;  to.value = this.value;  }  public int evaluate() {  if (value != null)  return value;  return operator.calculate(operandA.evaluate(), operandB.evaluate());  }  } |

Step 6: Evaluating our MathExpressions

As you may have noticed in previous steps, there are these floating evaluate and calculate methods. These methods focus on interpreting our Abstract Syntax Tree. In the Calculator class, the calculate method, run by LAPS after parsing is complete, starts the evaluation of the root MathExpression node then prints the result. Within the MathExpresion class, the evaluate method evaluates its operands then uses the operator to calculate its value. The operator calculates its result using the computer BiFunction typed field within the BinaryOperation enum we made in step 3.