1. Motivation

I took PLC in Spring 2019 at the time we were using a program called PLCC to create programming languages. PLCC acted as a black box for most students and seemed to take away from the course’s content. Using software that was hard to understand made it harder for students to understand the point of the software. For the students who wanted to dive into the source code to try to get a better grasp of the software, they were quickly turned off by the lack of comments, obfuscated variable names, and poor code structure. Initially, I was interested in fixing PLCC by refactoring and commenting the code, but after carefully considering the design of PLCC, I figured it would be better to just make a new system entirely called LAPS.

1. Goals
   1. Student Readable and Well-Documented

One of the shortcomings of PLCC was that the source code didn’t provide any useful information for how the software ran. This inspired LAPS to be well-organized into multiple files, have useful variable and function names, and written in Java (a very common programming language amongst undergraduate students).

* 1. Modular

If someone wants to modify or enhance PLCC, it is very hard due to the source code is all in one file.

* 1. Debugging Support/Full IDE Support

1. Design
   1. Structure
   2. A walk-through of the language processor development process
2. Integer Calculator Example

The Goal:

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

|  |
| --- |
| 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);  }  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());  }  } |

BinaryOperator.java

|  |
| --- |
| 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 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;  }  }  private final BinaryOperation operation;  public BinaryOperator(String operator) {  this.operation = BinaryOperation.operations.get(operator);  }  @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);  }  } |

1. Road Blocks
2. Results: Goals Met?
3. Future work