**Goals**

**Clear and Well-Documented**

This software is free and open source, so we encourage contributions and constructive criticism. To aid this mentality, clear and well-documented code is a must. This includes the use of meaningful function and variable names, JavaDoc compliant comments, and easy to follow code.

**Modular**

To allow this software to potentially serve more than its original purpose, modularity is a necessity. This means files should be meaningfully separated with minimal dependencies. An added benefit of this goal is that additional features are much less likely to break current features.

**Debugging and IDE Support**

Most compiler compilers don’t provide debugging support for their user’s source code making errors much harder to find and fix. This is because these compilers use customized, or even new, languages that have a small following. So, it’s a requirement to use a common language as a front-end for language developers.

**Design**

LAPS is organized into packages which can be used independently of one and other. Below are the explanations for each file and sub-package in the edu.rit.gec8773.laps package.

**Annotations**

This package contains all the annotations used during reflection of all the classes associated with a LAPS language specification.

@GrammarRule:

Types annotated with this give confirmation to the parser that it is intended to be a grammatical rule in the defined language. This should only be applied on classes in the current version.

@Token(skip=false):

This can be applied on both fields and methods, which are ***public*** and ***static***, to mark them as definitions for token types in the language specification. The names of the token types are the case-insensitive names of the fields and/or methods.

Fields are parsed as generic objects. Methods, on the other hand, are executed and the return generic objects. Once these objects are collected, the objects’ toString() methods are invoked one time before the language’s runtime and stored as the regular expression for tokens.

This annotation has a field called skip. Set this equal to true to mark the token type as text to be removed from the language’s input source.

@RunBeforeFirstInit:

This can only be applied on ***public*** and ***static*** methods in @GrammarRule types (classes). Methods found with this annotation will be invoked once **before** the **first** instance of that @GrammarRule type is instantiated. This happens when the parser accepts the defined grammatical rule from the language’s input. Be aware, if multiple methods in the same @GrammarRule type are annotated with this, there is no guarantee which order the methods will be run in.

@RunBeforeEachInit:

This can only be applied on ***public*** and ***static*** methods in @GrammarRule types (classes). Methods found with this annotation will be invoked once **before** **each** instance of that @GrammarRule type is instantiated. This happens when the parser accepts the defined grammatical rule from the language’s input. Be aware, if multiple methods in the same @GrammarRule type are annotated with this, there is no guarantee which order the methods will be run in.

@RunAfterEachInit:

This can only be applied on ***public*** **instance** methods in @GrammarRule types (classes). Methods found with this annotation will be invoked once **after** **each** instance of that @GrammarRule type is instantiated. This happens when the parser accepts the defined grammatical rule from the language’s input. Be aware, if multiple methods in the same @GrammarRule type are annotated with this, there is no guarantee which order the methods will be run in.

@Priority(value=0):

This can be applied to constructors in @GrammarRule types. The point of this is to distinguish between the ambiguity of which grammatical rule to attempt to parse first. So, all constructors take advantage of this priority system, even when not explicitly annotated. The way it works is the smaller (more negative) the value, the earlier the parser attempts to parse the corresponding constructor’s grammatical rule. If not annotated with this, default value is 0.

1. Structure
2. A walk-through of the language processor development process
3. Road Blocks
4. Results: Goals Met?
5. Future work

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);  }  } |