Spring 2020

LAPS Development Report

Creating a language processor

Author: Geoffrey Casper

Advisor: Dr. James Heliotis

Rochester Institute of Technology DEPARTMENT of Computer Science

# Introduction

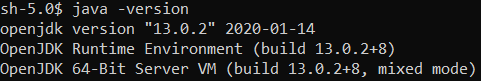
LAPS stands for LAnguage Processor & Synthesizer. It can be referred to as the LAPS system, the LAPS library, or, simply LAPS. The purpose of this software is to be an accessible, student-friendly alternative to a standard compiler compiler, such as PLCC or YACC. To avoid redundancy, this document assumes the reader knows what regular expression are, as well as, context-free grammars.

## Getting Started

LAPS takes advantage of annotations, reflection, and class loading in Java to enable developers to write their own Java classes which can be interpreted as a language specification. To define a language, there needs to be a set of tokens (lexemes) defined using regular expressions for lexical analysis, a context-free grammar in terms of those tokens for syntactic analysis, and an executable portion to process the abstract syntax tree (AST) during semantic analysis.

### Installing Java

1. Download and run the latest Java JDK 13 installer at the following [link](https://www.oracle.com/java/technologies/javase-jdk13-downloads.html) (as of 3/26/2020)
2. Open your system’s console application (Terminal, Command Prompt, etc.) and run the following and confirm your version is “13.x.x”:



1. If you don’t get that output, look up how to add Java to your PATH environment variable in your specific operating system

### Installing LAPS

1. Make a directory called “LAPS” (your LAPS directory) where you install other programs
   1. For Windows, that is “C:\Program Files”
   2. For Mac OS X, that is “/Applications”
   3. For Linux, that is “/opt”
2. Download and extract LAPS.zip (TODO insert link)
3. Just like you did for the Java JDK, add this directory to your PATH environment variable
4. For Mac OS X and other Unix-like systems run “chmod a+x laps”

### Typical First Steps

The first thing that needs to take place is annotating a Java class with @GrammarRule to tell the LAPS system that the class is indeed intended to contain a definition for the context-free grammar. Once that is done, public constructors will now represent acceptable sequences of tokens and other grammar rules. In addition, anything annotated with @Token LAPS will attempt to interpret as an uncompiled Java regular expression pattern. These token types can be accessed from your constructor grammar rule definitions by accepting a String type with the name of the field or method which defined them.

See examples/src/java/Calculator.java for a simple example.

See the Design.Annotations section for more information on how tokens types are interpreted by LAPS.

# Specification

## 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. For classes annotated with this, constructors are considered grammatical rules. In said constructors, parameters are expected to be Strings with parameter names of token types (see @Token) or other types annotated with this. Notice that when a @GrammarRule type is a constructor parameter of a @GrammarRule class, the type is initialized before the class’s constructor is executed. This should only be applied on classes in the current version.

@Token(skip=false):

This can be applied on both fields and methods in any @GrammarRule type, 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.

See the JavaDoc for Java Pattern class for information on Java regular expressions.

@RunBeforeFirstInit:

This can only be applied on *public* and *static* methods with no parameters and a void return type 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.

Example use cases:

* Initialize the global environment for a language.
* Print out an introduction for a language.
* Initialize any variable a language might need.

@RunBeforeEachInit:

This can only be applied on *public* and *static* methods with no parameters and a void return type 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.

Example use cases:

* Update a variable, i.e. number of instances created.
* Check if a variable should be reset or cleared like a List.

@RunAfterEachInit:

This can only be applied on *public* instance methods with no parameters and a void return type 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.

Example use cases:

* Reorganize the current state of the abstract syntax tree.
* Interpret a language.
* Evaluate an expression.
* Confirm the abstract syntax tree is what you expect it to be.

@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.

### Resources

This package contains all the data to compile a LAPS language specification into a single file.

Resources (class):

This is the public interface for accessing tokens and constructed parsers. It also can save this data to a file effectively saving a compiled language specification.

# Experience

## Motivation

I took the Programming Language Concepts course (PLC) in 2019 at RIT. At the time, we used a piece of software called PLCC written by Dr. Timothy Fossum. This software was great regarding the teaching resources it provided, but it lacked some features for students and contributors. My initial intention was to modify PLCC to make it less of a black box and more user-friendly, but I quickly learned that this modification would have a similar amount of work as making something new. This new project ended up being called LAnguage Processor & Synthesizer, otherwise known as LAPS.

Making LAPS gave me complete control of the architecture, feature set, and packaging. This allowed me to adhere to my goals of software modularity, comprehensible, and support for debugging and development in IDE’s. LAPS isn’t written as a typical piece of software; it’s written as a library that works together with a main program. So, anyone may come along and use a different scanner or parsing implementation. They can even remove the main and directly use LAPS integrated into their code. To enable this, the LAPS source code is heavily documented in the JavaDoc style.

## Starting Development

Before starting development, I knew LAPS needed to be written in a language many undergraduate students would understand. PLCC was originally written in Python, but since Python can enable and even encourage bad coding practices, I figured Java was the next best thing.

## Implementing Initial Ideas

Once the language was settled, I began thinking about the design structure of what LAPS eventually turned into. I started by writing a custom Java class loader which would use annotations in a package-info.java file to identify which classes would be used by LAPS. That is when I ran into a big problem, which I was unsure how to solve. Packages weren’t considered “loaded” until a class from that package has been loaded. That meant that a user of LAPS needed to pass in the path of their package, making the class loading mechanism obsolete. I was able to use the current working directory as part of the classpath and have the user pass in the fully qualified class name (e.g. path.to.classes.ClassName) of their top-level grammar rule to alleviate the issue. This took away a lot of overhead in file I/O and potential of insecure algorithms.



Once I stopped making progress on the class loader, I started coding up some of my ideas for my parser and storage for an abstract syntax tree. At this point, I didn’t really know exactly what I was doing. So, I just wanted to get all my ideas into code. I quickly realized that the storage for the abstract syntax wasn’t necessary since that would be handled by each LAPS language specification.

## Building the First Example Language

In the meantime, I created what I thought would be an example language for LAPS to process. In this draft language specification, I used my limited knowledge of annotations to annotate classes in what seemed like a reasonable manor for the use case. That is, the annotations’ arguments defined grammar rules and tokens. Since I hadn’t compiled LAPS at this stage, I didn’t realize that annotations can only receive Strings and primitive types. That made the only way to pass in valuable arguments to the annotations is to pass them by Strings, which have no compilation checks, other than having quotes surround them, making that not very IDE friendly and hard to debug. So, in lieu of using annotations to define grammar rules, I ended up moving to a different idea of using class constructors instead. Although the way annotations are used had to change, the token annotation worked flawlessly under these restrictions and ended up in the current version.

## Starting the First Testable Version

While the example language was in the works, I started creation of a recursive descent parser implementation. I started out thinking of what operations I wanted to implement based on concepts of regular expressions and context-free grammars. I started out with the most basic type of parser I could think of, a token parser. This accepts a single token and returns its String value. The next parser that made the most sense was a parser which accepts a sequence of parsable types, i.e. tokens. The next most simple parser I could think of was a parser which would accept any of one of a set of parsable types, what I referred to as an OR grammar rule. This one really confused and frustrated me, because I knew it could lead to ambiguity of which item in the set should be parsed first. I didn’t know it yet, but this would be the basic idea for the list of constructor grammar rules defined in @GrammarRule types.

As a biproduct of the architecture I have chosen, I need to create a class to help manage storage of tokens and parsers. This seems to be a common architecture for compiling, since tokens need to be accessible by both the parsers and the tokenizer. I called my implementation Resources for no reason other than I couldn’t come up with anything better at the time and it stuck. Since everything was stored so concisely, I ended up adding a feature to be able to save and load languages from a file, making it easier to pass around compiled language specifications.

## Finishing the First Testable Version

At this point, I was still a bit frustrated with the struggle I was having with OR grammar rule, so I decided to take a step back and work on another portion of the project, the scanner, or tokenizer. This portion was much easier for me to develop since I’ve written a tokenizer before in PLC and I’ve analyzed the Scan class, which has a similar function, in the PLCC source code. All that had to be done was convert a stream of characters to a stream of tokens. I did this by retaining a list of token types and then looping over those tokens to hopefully find the longest match.

Meanwhile, I went back to working on the parsers, which were still missing a crucial feature, the ability to convert a @GrammarRule type into a parser. To achieve this, I looked at the documentation for Reflection in Java. To my surprise, it was much simpler to understand. Java Class objects provide interfaces for getting their fields, methods, and constructors. All I had to do regarding the parser was collect all the constructors and convert their parameters into grammar rules using the previously made sequential and token parsers, as well as this parser recursively.

## Adding Features and Fixing Bugs

Most of the project was done by this point. Things that still needed to be done were adding error messages, making a command-line utility to use LAPS, fixing bugs within the modules, adding some features to catch erroneous things which could be in a LAPS language specification, and refactoring of files to increase modularity.

One of the major bugs I ran into was that token strings weren’t given back to the scanner by the parser when parsing failed on a grammar rule. I ended up needing to add features of buffering the input to the scanner and being able to push unparsed characters back to the front of that buffer. In addition, I had to store all of the parsed token values in a stack for each parser.

Making the command-line utility was relatively straight forward. The only involved part was making the command-line argument parser. I ended up using a HashMap of characters to lambda functions for each parameter. The arguments are used to construct the environment for the LAPS modules. Then, it opens the STDIN as the language input, makes a new scanner and parser, and executes the language.



## Future Work

There are still a few things I’d like to work on regarding LAPS. There exist better parser designs that I wish to implement. Some of the Java reflection code is intertwined with the parser code which goes against the modularity aspect of my goals. The interfaces I made for the parser and the scanner seem to be useful in the modularity aspect. Since LAPS is eventually meant to be used as a teaching tool, more example language specifications and assignments should be created. Overall, I’m very happy with what I created. LAPS is very easy to modify and grasp how it functions.

# Your First Language Made With LAPS

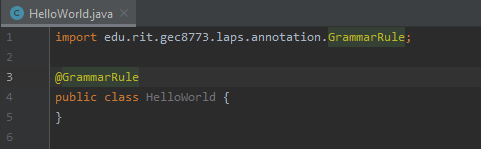
## Things to Consider

* LAPS uses Java Annotations to under what your code means, but if LAPS can’t see your code it can’t determine what your code means. TL; DR: any code you expect LAPS to see make it public.
* LAPS does NOT construct an Abstract Syntax Tree (AST) for you. You must save all the data you want to save in your constructors in your code.
* All the classes you write are yours. You can implement any interface and extend any class. Just code like you normally would in Java. Use your favorite IDE.
* The LAPS script expects the current working directory when called to be in the default package of your Java .class files (not your Java .java source code)
* All your compiled classes should be compiled with the compile flag “-parameter” to save the names of parameters in your methods and constructors

## Creating a New Project

1. Make a new empty Java project in your IDE of choice
2. Add the LAPS.jar from your LAPS directory as a library in your project
3. Add the “-parameters” compilation flag to your build configuration
4. Add a run configuration which to run the LAPS script which should be in your PATH
   1. Arguments for the script should be “-c <fully-qualified-class-name>”

## Getting Down to Business

1. Let’s start out with root node in your AST
   1. Create a new public class and call it “HelloWorld”
   2. Annotate the class with @GrammarRule to indicate this class is a grammar rule when parsing
   3. At this point you should have the following:
2. To run execute your language you can run “laps -c HelloWorld”
   1. On Windows, the LAPS command is a batch file
   2. On Unix-like systems, the LAPS command is a bash script
   3. At this point, LAPS will do nothing with exit code 0

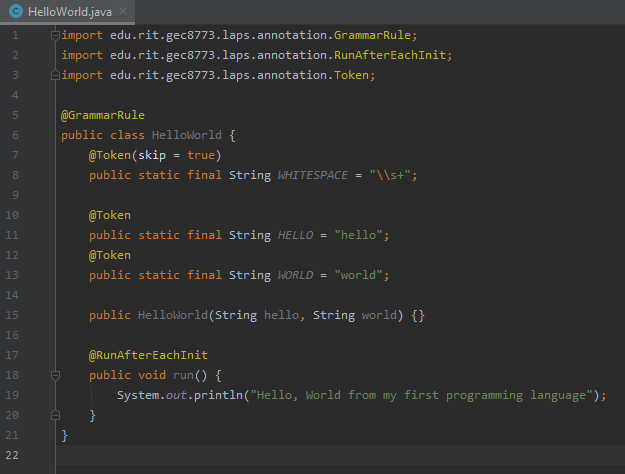
## Adding Tokens

1. Let’s add 2 tokens to start
2. In your class (HelloWorld), add 2 “public static final String” fields
   1. These fields will hold Java regular expression String values
   2. The first field can be HELLO = “hello”
   3. And the second can be WORLD = “world”
3. Once you have your fields, annotated them with @Token

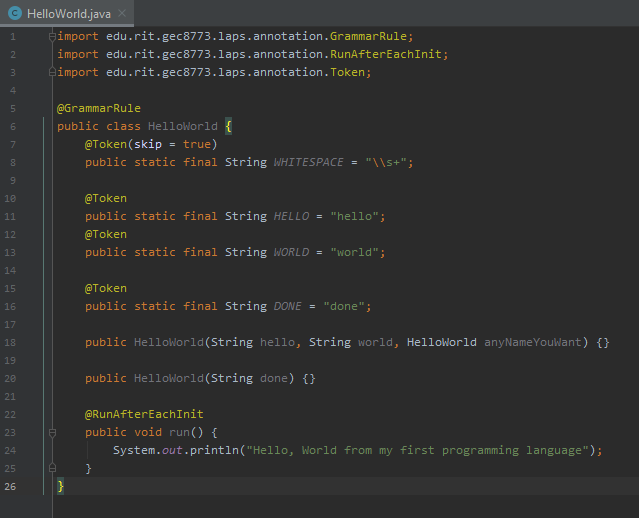
## Defining a Grammar Rule

1. In LAPS, public constructors give definitions for grammar rules
2. So, let’s make a new constructor to accept the tokens HELLO then WORLD
   1. To do this, the parameters of the constructor should be of type String which will store the value of the accepted token from the input
   2. It is very important that the names of these parameters align with the defined fields (case-insensitive); this is how LAPS determines which token to accept in a grammar rule
   3. Code update:
3. At this point, you have a language which accepts the input string “helloworld” and that’s it
   1. I know kind of useless, right?

## Making a More Useful Language

1. Something that always comes in handy in language development is the ability to skip whitespace
   1. So, let’s add a new token to accomplish that
   2. To define a skip tokens, instead of annotating your fields with @Token, annotate your fields with @Token(skip=true)
   3. Now to skip whitespace, annotate the field with @Token(skip=true), name it WHITESPACE (but LAPS doesn’t actually care about the name), and set its value to “\\s+” (the Java regular expression for 1 or more characters of whitespace)
   4. Great your language now skips whitespace
2. Let’s define some semantics for your language to make it finally do something
   1. LAPS has 3 annotations for methods to help define semantics
      1. @RunBeforeEachInit selects static methods to run before every constructor call
      2. @RunBeforeFirstInit selects static methods to run before the first constructor call
      3. @RunAfterEachInit selects instance methods to run after every constructor call
      4. Note: there is no guarantee for the order in which the selected methods run by the same annotation
   2. Define a new instance method in your class and annotate it with @RunAfterEachInit and print out “Hello, World from my first programming language” in the method
3. Code Check:

## Making a List Style Grammar Rule

1. Let’s start out with a naïve approach by adding the HelloWorld type as a parameter to our constructor and nothing else
   1. After a bit of testing, you will notice that the language does accept “hello world” however many times you add it with however many white-space characters in between (including none) as input to your language, but it never terminates unless you put in an unknown or unexpected token resulting in a failed parse
2. To fix this, let’s try to add a new empty grammar rule in the same class
   1. To do this, we must add a constructor without parameters
   2. Now give this language a test
3. You will notice the language still doesn’t terminate unless you put in an unknown or unexpected token resulting in a successful parse. What? Why? What changed?
   1. Note: This is just an edge case for the root node in the AST. In most other cases, this works
4. Before we explain this successful parse, let’s try to understand what LAPS is doing
   1. When parsing, LAPS goes through each constructor of a class containing grammar rules and tries to parse that rule based on the selected constructor
   2. If there is a constructor with no parameters, it is always parsed last
   3. If the parsing fails on a particular constructor, it goes to the next constructor
   4. If parsing fails on all the constructors in a class, it returns a failed state to the grammar rule which wanted a grammar rule from the class
5. The explanation
   1. LAPS accepts “hello world” an indefinite number of times
   2. Then when it doesn’t see a HELLO token it tries the accept nothing which always succeeds, and leaves the rest of the input unread
6. The best way to handle a list as a root node in your AST
   1. Add another token DONE = “done”
   2. And Instead of having a parameterless constructor have the constructor accept a DONE token
7. Final Product:

# Creating a More Complex Language

## 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.

See the next page for companion code from BinaryOperator.java.

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.

See the resulting language in examples/src/java/intCalculator.