Graphing - Programming Project

Shashi Balla

November 26, 2018

Contents

Part I

Analysis

Chapter 1

Introduction

For my project I will attempt to make a graphing software. Graphing software is incredibly important in Linear Algebra and a lot of maths taught in schools is to do with Linear Algebra. Linear Algebra also links to many other aspects of Maths and hence is very important to understand. My stakeholders will consist of teachers, who will use my software to show graphs of functions, and to students who will use it to practise their graph sketching or to help them with their homework. There are many graphing tools out there, some of them shown below, however they have many downsides. I hope to make a piece of software that has as much functionality as possible, while retaining simplicity and reducing the number of downsides to an absolute minimum.

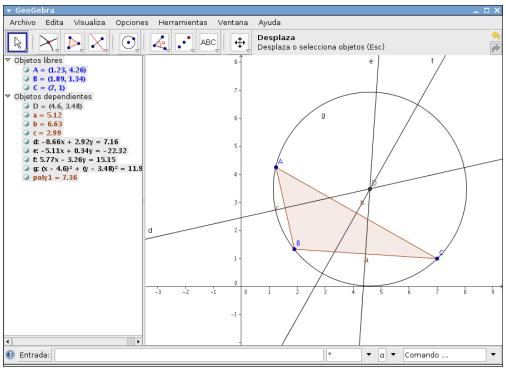


Figure 1.1: GeoGebra

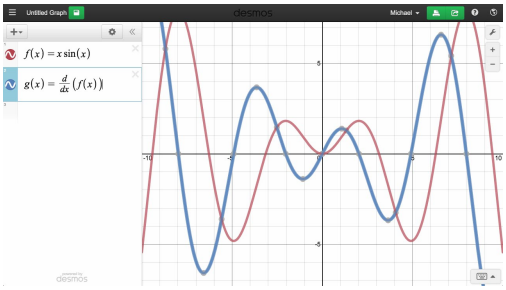


Figure 1.2: Desmos

Chapter 2

Stakeholders

My stakeholders will be comprised of teachers and students in year 10 and above. This is because graph sketching as a skill becomes very important from GCSE onwards and as such the tool will be aimed towards students in the upper years and the teachers themselves. I have interviewed several of my potential stakeholders about the software that they use and their opinions on it.

2.1 Teacher Interview

I interviewed Miss Naguthney who is a Maths teacher who teaches pupils from year 7 to 13. She uses multiple tools to aid her teaching. I asked her a couple of questions to get an idea of what she finds important in a graph drawing software.

Question	Answer	Analysis
What graph drawing software have you used?	I have used Desmos, MATLAB and GeoGebra.	These three programs are very different, ranging from a simple graph drawing software, Desmos, to a professional data presentation software, MATLAB.
Which do you think is the best and why?	I think all 3 are good. In terms of pure graph drawing I think that Desmos is the best due to its simplicity.	From this response, it is important that my program is as simple as possible so that anyone can use. I think that to make program simple it must be responsive as well, and hence I should make my solution as efficient as possible so that it can be responsive on most devices.
What do you think is the most important aspect of a graph drawing software?	The most important part of a graph drawing software are obviously the graphs themselves and the most important part of the graphs are of course their intersections with axes, turning points and any asymptotes.	I agree with all these points made and I think that my program should at least be able to identify the turning points and any intersections with the axes.

2.2 Student Interview

I find that during my sixth form studies, that graph sketching is a must have skill, and hence a graph drawing tool is important to verify that you have drawn a graph accurately. I interviewed Matthew, who is a sixth form student studying similar subjects, about graph drawing tools that he has used and his opinions about them.

Question	Answer	Analysis
What graph drawing software have you used?	GeoGebra, Desmos and Kalgebra (KDE Application)	These three programs are quite similar as they have many of the same features.
Which do you think is the best and why?	I would say GeoGebra because: • Native client so it is more responsive than a web-based app • Versatile • Easily Adjustable Axes • Very easy to focus on a part of the graph • Multiple function support However the UI looks ugly and has no dark mode to reduce strain on the eyes.	I think that my program, should have the ability to be themed, through using CSS or a text file and while I may not be able to make my program versatile, it should still be able to draw multiple functions at once and navigating the graph must be fluid.
What do you think is the most important aspect of a graph drawing software?	The two most important aspects for me are ease of use and flexible input. Specifically this would be stuff like being able to input complex functions such as sums of multiple rational functions [1] (Not complex as in $x \in \mathbb{C}$) and the ability for the software to automatically adjust the scale. An example of this is for trigonometric functions sine and cosine. They don't have high y values but they always appear very small on GeoGebra because the scale is wrong. A way to automatically set a suitable scale would be nice. Finally, I feel performance is also a must - it's frustrating to navigate the graph and have the application lag a lot.	Matthew's answer can be summarized into two points: Responsiveness and Flexible Input. I think that both of these points are valid and I should aim to make sure my application satisfies both these points.

[1] A function $f(x) = \frac{P(x)}{Q(x)}$, where $P(x)$ and $Q(x)$ are functions of x

Chapter 3

Similar Programs

During my interviews with my stakeholders, they mentioned several pieces of software that they have used. I will be analyzing 3 of those programs in particular (or similar in the case of MATLAB, since MATLAB is extremely expensive), which are:

1. Desmos [1]
2. GeoGebra [2]
3. GNU Octave[3] (MATLAB[4] Clone)

3.1 Desmos

Desmos is a closed source graphing calculator written in HTML5, which allows it to be used on many devices. It has dedicated apps on Android and on Apple devices, but has no dedicated application on a desktop environment.

The real strength of Desmos comes from its ability to create activities that students can then complete.

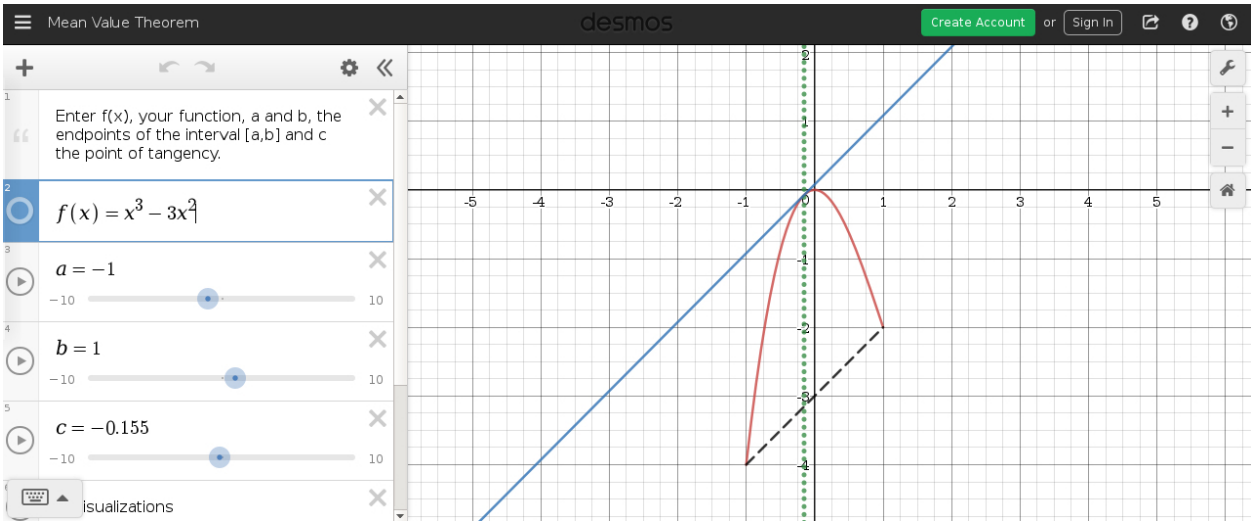


Figure 3.1: Mean Value Theorem example that students can interact with

The problem however is that it is web based and as such, can be clunky to use. The mobile app and website are especially unresponsive when zooming in or out and when panning around. Another pitfall is that Desmos tries to make itself more secure in a way by restricting input. This however makes input very difficult to do and along with the unresponsiveness, it can be impossible to input the required equation.

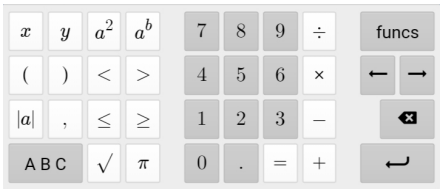


Figure 3.2: The Input

On the other hand, the actual graph drawing is excellent, with many features such as inequalities, modulus functions and the ability to draw polar functions. It also highlights turning points, intersections with axes, inflection points and in the case of trigonometric functions, it expresses them in terms of π instead of a numerical values.

Overall as a graphing calculator it does an excellent job, but has poor optimization issues and input, which I have realized are both important and need to be at the top of my list of requirements for my program.

In terms of other features, you can print, save and share your graphs to use later. I think that my program should have at least 1, if not all 3, of these features.

3.2 GeoGebra

GeoGebra is an interactive geometry, algebra, statistics and calculus application under the GNU General Public License, meaning that it is open source.

The main advantage to GeoGebra is that it can do a lot of things and is all of its features integrate well with each other. For example you can create a shape on the graph and create lines intersecting it, linking geometry with algebra. Individual line equations can also be moved around allowing for a very interactive experience. Like Desmos, you can create activities that you can interact with.

The mobile app for GeoGebra has a similar input to Desmos and as such suffers from the same problems. However it is more responsive than Desmos but only slightly. The desktop version however does not suffer from any of these problems.

As a pure graphing software, it does not automatically show the points of interest of a curve and is instead confined in a tool called the "Function Inspector". This tool is awkward to use as it only shows local maximums and minimums within a certain range (the highlighted red region), and if there are multiple roots, it does not acknowledge what the values of the roots are. Desmos does better than GeoGebra in this regard. Like Desmos it supports many functions that you can plot such as modulus and inequalities.

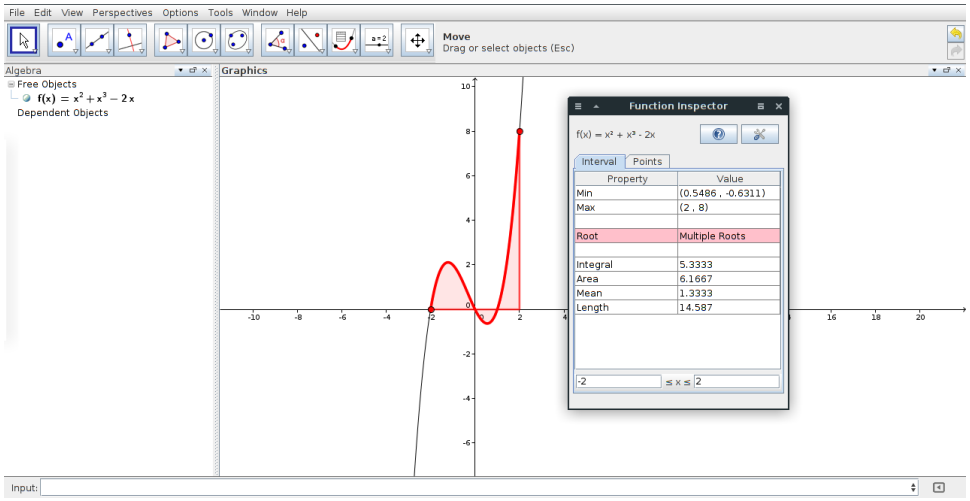


Figure 3.3: GeoGebra Function Inspector Tool

The panning and zooming in/out in GeoGebra is very fluid, with the zoom function being centered around the cursor. This means that you zoom in towards your cursor which feels more responsive than if it zooms in/out from the center of the graph. The only criticism is that you can accidentally move already drawn graphs when trying to pan, which can be frustrating.

In terms of other features, like Desmos, you can save and print your work as well as being able to export it into other formats such as JPG, PNG or even convert it into a PSTricks or TikZ environment which can be used in LaTeX documents for excellent graphs.

3.3 GNU Octave

GNU Octave is essentially an interpreter for a high level language centered around numerical calculation. GNU Octave is designed to be an open source software clone of MATLAB (MATrix LABatory).

Octave is different from GeoGebra and Desmos in the fact that it plots points that you tell it to not lines. For example if you wanted to plot $y = x^2$, you would define an array for your range in x by doing:

$$x = -10 : 0.1 : 10;$$

This creates an array that starts at -10 increments by 0.1 until 10. You would then define y by:

$$y = x. \wedge 2$$

You would then plot x and y by doing $plot(x, y)$;.

As a graphing tool Octave is not very impressive, it has limited support for implicit functions and will not highlight points of interest of a curve. Realistically this is expected since Octave is primarily made for data presentation. This means that it can produce excellent plots, including meshes and surfaces [1] in 3 dimensions. [2]

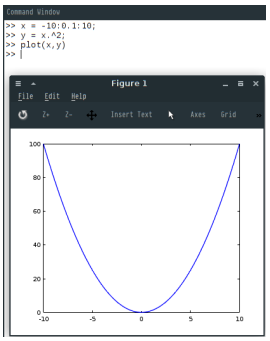


Figure 3.4: Creating a simple plot in GNU Octave

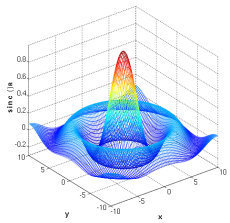


Figure 3.5: Wireframe 3D plot of the two-dimensional unnormalized sinc function

The way Octave plots (using the plot function at least) is by joining a line between consecutive points that it plots. I may use this in my program, since it seems more efficient to approximate a curve by making linear lines and joining them up than to check each pixel on the plot to check if it satisfies the given relation ship.

When inputting an algebraic expression in Octave it can be quite awkward, for example in Figure 3.4, a period is before the carat. This is unnatural for us to input and as such can cause errors. Natural input is

[1] A mesh is when a set of points in 3D space are joined up with lines to look like wire mesh. A surface is a mesh but adjacent points are used to make a solid shape instead, hence creating a surface.

[2] To clarify, a plot is where we explicitly define data that we then represent graphically, while a graph is where we define a relationship between 1 or more variables and represent this graphically.

something that I think I should focus on since it is means that the program is easier to use and requires no further reading unlike Octave.

Octave is split into two views, the main editor view and a dockable plot view (you can also have multiple plots open at once). Dockable plots, or multiple plots could be something I implement later into my project.

Octave, like the other programs, can save plots and additionally the scripts you have made. It also has a feature that saves the current state when you close it, and resumes this state when you open it again. This could also be a feature I implement later into the project.

3.4 Comparison

Overall I think that I will be using aspects of all three of the programs that I have analyzed in my program. I think I should try and make my program as simple to use as Desmos. As mentioned, the input is a problem, and hence I will try and implement an input system like GeoGebras'. All three of the programs analyzed have some form of saving their plots, so I think that this is a feature that I should implement within my own program.

In terms of UI design, both Desmos and GeoGebra have a similar layout. They have an equation input box to the left, that can be minimized and the actual graph in the center left. I think that this layout is easy to use for the user so I will implement a similar design in my own program.

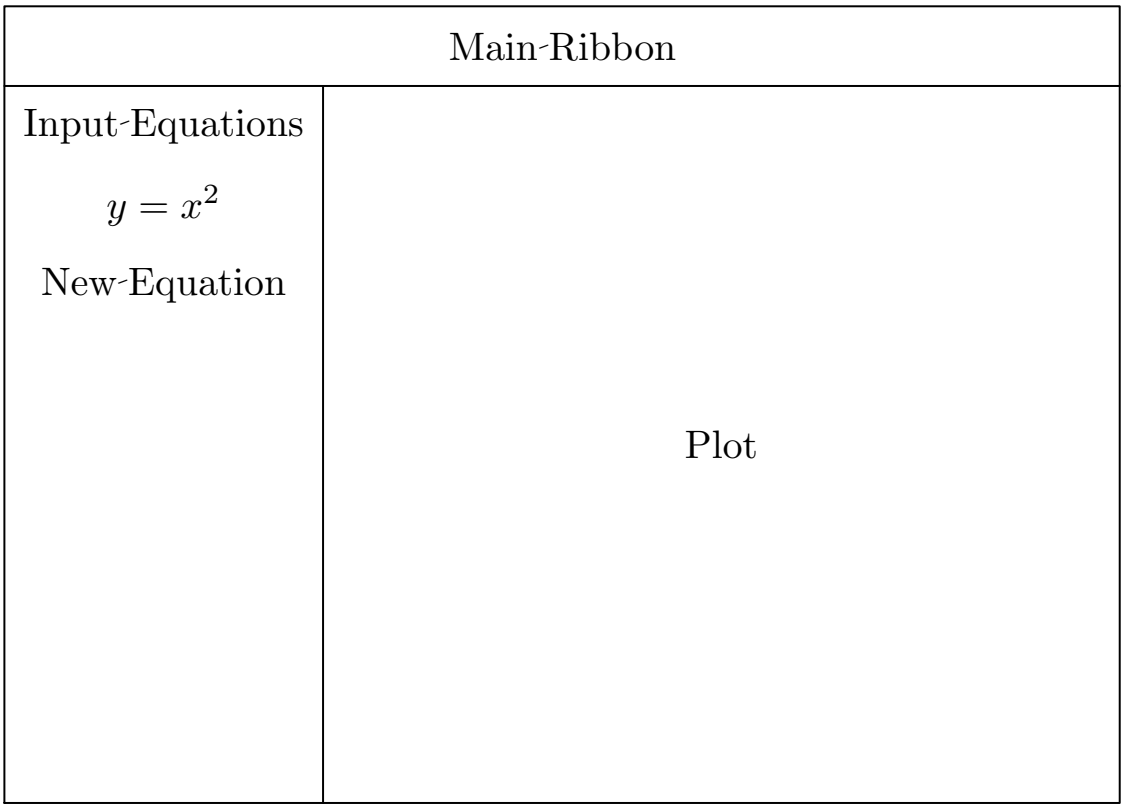


Figure 3.6: UI Design

Chapter 4

Computational Methods

4.1 Abstraction, Heuristics and Computability

Abstraction is the process of creating a model of a real-life problem by separating and highlighting important details. There are two main details that I can abstract away:

1. The input of expressions, and their conversion into a form that the computer can then manipulate.
2. The drawing of the function onto the screen.

By analyzing both parts of these details separately, we can determine if our problem is intractable.

The first part of the problem is solvable by converting the expression into an abstract syntax tree or into a post-fix stack, which is easily implementable.

The second part of the problem is solvable using heuristic methods. Most APIs, such as `OpenGL[5]` and `Vulkan`, only allow the drawing of primitives, which are points, lines and triangles. [1] This means that our curve cannot be drawn straight onto the screen and must be approximated by either using points or lines, hence there are two ways we can approximate the curve:

1. Let x be our starting value, $f(x)$ our function and dx be a small value. Now draw a line between the points $(x, f(x))$ and $(x + dx, f(x + dx))$; $(x + dx, f(x + dx))$ and $(x + 2dx, f(x + 2dx))$; ... ;$(x + n \cdot dx, f(x + n \cdot dx))$ and $(x + (n + 1) \cdot dx, f(x + (n + 1) \cdot dx))$. Here we are effectively creating line segments and joining them together to approximate the curve. As $dx \to 0$ our approximation tends towards the real curve. However as $dx \to 0$ the time taken tends to infinity and therefore we only need to make dx small enough so that the human eye can not see individual line segments. Also as a practical note, as $dx \to 0$, the pixel density required to view the curve tends to infinity, therefore dx needs to be big enough so that the dx is not less than the individual pixel width.
2. In the method above we approximated a curve by creating line segments, here we will approximate a curve by using individual points. Let x be our starting value, $f(x)$ our function and dx be a small value. Now draw a point at $(x, f(x))$; $(x + dx, f(x + dx))$; $(x + 2dx, f(x + 2dx))$; ... ;$(x + n \cdot dx, f(x + n \cdot dx))$. This can take a lot of time since calculate where to draw a pixel w number of times, where w is the pixel width. In some cases, for complicated functions, it may be intractable.

By using the first method we can model a curve to good accuracy within good time, therefore our second part of the problem is computationally possible albeit, using a heuristic method.

[1] Strictly this isn't true, but most other things we can draw using these APIs usually consist of a combination of primitives, usually triangles.

4.2 Logical Approach

Many parts of this problem require clear decision making, making sure that user inputs are valid and making my solution as efficient as possible. I could validate and standardize user input by using `RegEx[6]`. I will definitely use standard structures in my solutions, and therefore will be able to logical traverse these structures to get the required solution.

4.3 Decomposition

Decomposition is where a problem is broken down into smaller sub-problems that are manageable and are solved individually and combined to make the complete solution. As stated in the abstraction section, I can break down the problem into two main parts. These parts could then be broken down further into classes and functions such as:

1. A function class which consists of:
   (a) Converting an expression into a form the computer can manipulate.
   (b) Evaluating the expression for a value x, where $x \in \mathbb{R}$.
   (c) Calculating the roots and turning points of the function.
2. A plot class which consists of:
   (a) Drawing functions onto the screen
   (b) Heuristically adjusts the scale so the graph looks good
   (c) Drawing the axes

This is just an example of how I can decompose the problem and therefore the problem will be decomposed more.

4.4 Divide and Conquer and Concurrent Processing

Divide and Conquer is where a problem is recursively broken down into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. While this may seem similar to decomposition, Divide and Conquer relates to algorithms. My solution may be using trees, and as such I can use a Divide and Conquer algorithm to convert an expression into a tree by looking at sub-expressions within the expression. This can then be further improved by using Concurrent Processing to process each sub-expression individually. I can also use Concurrent Processing to draw multiple lines or functions on the screen at once.

Chapter 5

Requirements

From my research and analysis I have created a set of requirements that I aim to fulfill. I have split these requirements into development iterations that they will be linked with. Each iteration has a theme that most of the requirements will follow.

- Iteration I - Emphasis on Core Functionality
  - Plot a explicit function in x
  - Plot multiple functions on the same plot
- Iteration II - Emphasis on the User
  - Zoom in/out of the graph
  - Pan around the graph
  - Plot special functions such as trigonometric functions, logarithms, modulus functions, etc.
  - Identify intersection with the axes
  - Multiple plots that you can switch between
  - Save plots as pictures
  - Save workspace to resume later
  - Dark Theme
- Iteration III - Advanced Features
  - Identify intersections between functions
  - Differentiate explicit continuous functions with respect to x
  - Polar equations
  - Implicit equations
  - Parametric equations
  - LaTeX equation support
  - Identify turning points

I will use `Java` to accomplish this task. `Java` is platform independent meaning that I do not have to compile for many different systems. It also means that if I want to create a mobile version I will only have to recreate the UI. Within `Java` I will use `JavaFX` for my UI. `JavaFX` allows for customisation of UI elements through `css` files, as well as an easy interface to draw objects onto through the screen, through its `Canvas` class. I am also comfortable with the language which makes it the best language for this project.

Part II

Prototype I

Chapter 6

Design

6.1 Decomposition

My program will be decomposed into two main parts the GUI and the parsing of functions. The GUI will consist of the input of functions and the drawing of the functions. Many parts of this project can be achieved in a cleaner fashion using an OOP approach rather than a procedural approach.

6.1.1 OOP vs Procedural Approach Justification

An Object Oriented approach for my program is better than a procedural approach due to its complexity. An OOP approach allows parts of a program to be independent, and this means that these parts can be tested individually, allowing for simple debugging.

Specific to my project, Java is wholly based around the OOP approach and hence it makes more sense to use an OOP approach rather than a procedural one. Also many parts of my program can use the OOP concepts of inheritance and polymorphism. An example of this is the layer system I will be using later on to draw functions onto the screen. There are many different types of functions (again which can use inheritance), and each of these will have a coresponding layer. Each type of layer will draw, have very simlar attributes and other similar features which will be very slightly different for each type of function. By making all these different layers inherit a base layer class, it means that all of them can be accessed generally, especially when drawing. They can also all be stored in an array of the type of the superclass, and accessed (outside of the class) in a standard way making the solution cleaner and easier to test.

6.2 Function Class

Our basic `Function` class will contain two main methods:

- Parse - This method will convert the user's input into a data structure that we can use to evaluate. This data structure will be stored as a private attribute.
- Evaluate - This method will take a value of x and input it into our function and return the value $f(x)$.

Our class will contain more methods and attributes later (colour of the line, roots, turning points etc.) but these can be considered later as these are quite small parts. There are two distinct methods to parse a function:

- We can convert the input into a data structure, which we can then use to evaluate a value x.
- We can convert the input into its equivalant in a scripting language, such as `Lua[7]`, and then use the scripting language to evaluate a value x.

An example of the second is shown below:

```
1 import org.luaj.vm2.*;
2 import org.luaj.vm2.lib.jme.*;
3 public double evaluate(double x) {
4         ScriptEngineManager mgr = new ScriptEngineManager();
5         ScriptEngine e = mgr.getEngineByName("luaj");
6         e.put("x", x);
7         e.eval("y = math.sqrt(x)");
8         return e.get("y");
9 }
```

The problem with using scripts is that they are take a huge hit on performance as you are effectively creating a virtual machine during your program. Especially in `Java` where a VM is used to run your programs, creating a VM inside a VM is not very effiecient. Also we may make upto 10000 evaluate calls to render multiple lines, and therefore we will need to be as effiecient as possible. While the first method is harder to implement, it will have better performance, and it will be easier to debug as we are not using external tools. Therefore we will use the first method.
There are two main structures we can convert a mathematical input into:

- Binary Trees
- Stack Based Programming

both of which we will analyze in detail.

6.2.1 Analysis of Algebraic Expressions

Algebraic Expressions are quite important in this project and as such it is important to break them down and understand what they really mean. The most basic idea in evaluating expressions is **BIDMAS**, which stands for **Brackets**, **Indices**, **Division**, **Multiplication**, **Addition** and **Subtraction**. It signifies the order that we must do operations on an expression (It is sometimes called **BODMAS** where the **O** stands for **Orders**) with the order of significance going from the first letter to the last. For example, let us use BIDMAS with the following expression:

$$3(4^2 + 2)$$

According to BIDMAS we will first look for brackets. There are brackets! The expression inside those brackets is $4^2 + 2$. We then apply BIDMAS again. There are no brackets but we do hae indices. We apply the indice function which produces $16 + 2$. Applying BIDMAS again we see that we must do addition which produces 18. Finally we have the expression 3(18) which is multiplication and it produces 54. Here we repeatedly took the most significant operator and applied its function until we fully evaulated our expression. Later we will instead take the least significant operator as this will allow us to split our expression down in a more structured manner.

In the example above there was nuance that we ignored. This was when we identified 3(18) actually meant $3 * (18)$. This implicitly represented multiplication and while as humans it is easy for us to process it, it is impossible for a computer to know this. It is therefore important that we remove these inconsistencies before we properly convert our input into a structure. It is also important that we strip away all whitespace before we start as this will allow our input to be more consistent. Here is a list of all these inconsistencies that we need to remove:

- Any instance of ax where $a \in \mathbb{R} : a \neq 0$ is to be converted to $a * x$.[1]
- Any instance of $a($ and $)a$ where a is not an operator, is to be converted to $a * ($ and $) * a$ respectively..[2]
- Any instance of $(f(x))(g(x))$ is to be converted to $(f(x)) * (g(x))$.[3]
- Any instance of $! - f(x)$ where ! is to be any operator (e.g. $*$ or $/$) is to be converted to $!(-f(x))$.[4]
- Any instance of $-f(x)$ at the start or next to an opening bracket is to be converted to $0 - f(x)$.[5].

[1] For example $4x$ is to be converted to $4 * x$

[2] For example $4(x + 1)$ is to be converted to $4 * (x+!)$

[3] For example $(x + 4)(x - 3)$ is to be converted to $(x + 4) * (x - 3)$

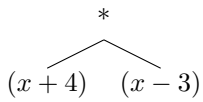
[4] If there is a negate symbol next to another operator, we need to make sure that the negate symbol is not treated as an operator (even though we treat it like an operator in certain situations in the next step)

[5] Both these expressions are equivalent but the second allows us to reduce ambiguity if there is a negate symbol at the start of an expression e.g. $-x + 4$ would be treated as $0 - x + 4$ and $(-x)$ is $0 - x$.

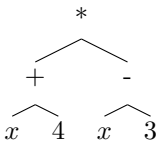
6.2.2 Binary Trees

The second stage that a compiler goes through is called Syntax Analysis and this is where the code is transformed into an Abstract Syntax Tree. What we are trying to achieve is very similar to that (albeit on a smaller scale). If we take a function $f(x) = x + 4$, this would be transformed into the binary tree:

If we take something more complicated such as $f(x) = (x + 4)(x - 3)$, we need to first remove consistencies as discussed at the start of this section. Therefore the function would become $f(x) = (x + 4) * (x - 3)$. Now when we convert this into a syntax tree, we find the least significant operator (the operation we would do last), make it our root node and split the parameters that it is operating and make those parameters the child nodes of the root node. Here $*$ is the least significant operator and $x+4$ and $x-3$ become the child nodes.



We can then repeat what we did above on the child nodes. In $x + 4$, $+$ is the least significant operator and in In $x - 3$, $-$ is the least significant operator. Therefore we now get:



We now stop as we cannot split this down any further. What we notice here is that only the bottommost nodes are actually values, the rest are operators. This is significant, because what we just did was a recursive algorithm, where we split each node down, until a node does not contain any operators.
Now if we want to get $f(1)$, we simply replace every instance of x with 1 and then perform the recursive algorithm outlined below.

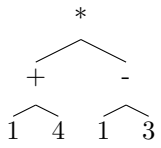
Algorithm 1: Evaluate: Binary Tree Version

```
1 function evaluate(tree):
2 |   if tree.height = 0 then
3 |   |   return tree.root.value
4 |   else
5 |   |   return evaluate(tree.leftsubtree) tree.root.value evaluate(tree.leftsubtree)
6 |   end
7 end
```

This recursive algorithm has base case "tree.height = 0", which essentially checks if are at the bottom of our tree. We know that the bottommost nodes are actually values, therefore we can return this value as we cannot go deeper into our tree. Our recursive case is where we are not at the bottom of our tree. We know that at this point, the root node is an operator [6]. Therefore we operate on the nodes below it.

[6] It is important to realize that tree.root.value is just an operator, and in theory we can simply just write it as shown above but in implementation this will require us to use if statements to check which operator it actually is as the operators will most likely be of data type string

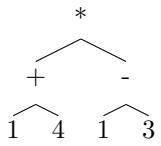
First let us replace every instance of x with 1.



Now the root node has value "+", i.e. an operator therefore we take the left subtree which is,

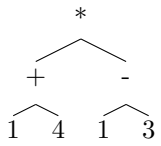
and again the root node is an operator so we take the left subtree again, which is 1. This is not an operator, so we return this value, going back to our previous call which is,

Now we take the right subtree, which is 4. Again, this is not an operator so we return this value, going back to our previous call. Now we have **return** $1 + 4$, therefore we return 5 to our previous call which is,



We now repeat the process for the right subtree,

which returns -2. We then go back to our original call to get $5 * -2$ which returns -10.
The recursive tree for this is exactly the same as the original tree that we started with.



This is significant as this means that the number of recursive calls made is the number of edges, which is $n - 1$ where n is the total number of operands and operators combined.

6.2.3 Stack Based Programming

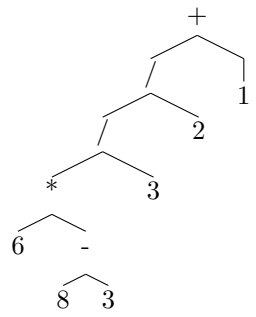
Stack Based Programming[8] is all about using a stack data structure, and manipulating the items within it to get the desired result. Firstly we have to convert our *Infix Notation* into *Reverse Polish Notation*. Infix notation is the "normal" way of writing algebra, where the operator is inbetween its operands. Reverse Polish notation or Postfix notation is where the operator is after its operands. The advantage of this notation is that no brackets are required. For example $4+3$ in infix notation would be $4 \quad 3 \quad +$ in postfix notation. To evaluate this expression we push each individual operator/value, from left to right, one by one, to a stack. If the value is an operator then it pops however many inputs it would normally take off the stack, perform the operation, on the values we just popped off, and then add that new value back to the stack. For example if we take the infix expression,

$$(((6 * (8 - 3)/3))/2) + 1$$

we can convert this to the equivalent postfix expression,

$$6 \quad 8 \quad 3 \quad - \quad * \quad 3 \quad / \quad 2 \quad / \quad 1 \quad +$$

The way we get this is by looking for the least significant operator (the one we would consider last) and putting it to the end. We then take the operands of the operator which just removed, and repeat until we are left with only values. This seems very similar to the binary tree solution. In fact, if we convert this into a binary tree,



and if we then perform post-order depth traversal. So first we go all the way down the left hand side of the tree to get 6. We then visit its sibling, $-$, which has more children. So we visit 8, 3 then $-$. So far we have

$$6 \quad 8 \quad 3 \quad -$$

We then move up to $*$ and add this to our list. We then visit its sibling 3 and add this to our list. We have now got

$$6 \quad 8 \quad 3 \quad - \quad * \quad 3$$

We then go up to / and add this to our list. We then visit its sibling 2 and add this to our list. We have now got

$$6 \quad 8 \quad 3 \quad - \quad * \quad 3 \quad / \quad 2$$

We then go up to / and add this to our list. We then visit its sibling 1 and add this to our list. We then go up to the root node, $+$, and add this to our list. Finally we have,

$$6 \quad 8 \quad 3 \quad - \quad * \quad 3 \quad / \quad 2 \quad / \quad 1 \quad +$$

This is identical to our post-fix notation. This is important as later on we can use this fact to make our solution as effiecient as possible.
Now if we start to evaluate this expression we get,

$$[6] \quad \begin{bmatrix} 8 \\ 6 \end{bmatrix} \quad \begin{bmatrix} 3 \\ 8 \\ 6 \end{bmatrix}$$

we have reached an operator, $-$, so we now pop 2 items of the stack, 3 and 8, and perform the operation, $8 - 3 = 5$. We now push this value onto the top off the stack and resume our evaluation.

$$\begin{bmatrix}5\\6\end{bmatrix}$$

Again we have reached a operator so we repeat what we did before. Take 5 and 6 off the stack, push $5 * 6 = 30$ onto the stack.

$$[30]\ \begin{bmatrix}3\\30\end{bmatrix}$$

Take 3 and 30 off the stack, push $30/3 = 10$ onto the stack.

$$[10]\quad \begin{bmatrix}2\\10\end{bmatrix}$$

Take 2 and 10 off the stack, push $10/2 = 5$ onto the stack.

$$[5]\quad \begin{bmatrix}1\\5\end{bmatrix}$$

Finally Take 1 and 5 off the stack, push $5 + 1 = 6$ onto the stack. We are left with 6 and have now got our answer. Just like with the binary trees, if we have an unknown x, we can simply replace the x with a value when we want. The algorithm to evaluate post-fix notation is shown below,

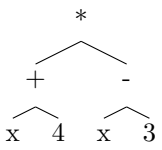
Algorithm 2: Evaluate: Stack Based Version

```
1  function evaluate(list):
2      Stack stack
3      Array temp
4      Real out
5      Real x
6      foreach i in list do
7          if i is an operator then
8              pop = i.numberOfInputs
9              temp = new Array[pop]
10             for j=0 to pop by 1 do
11                 temp[j] = stack.pop()
12             end
13             x = i.input(temp[1],temp[2],...,temp[n])
14             stack.push(x)
15         else
16             stack.push(i)
17         end
18     end
19     out = stack.pop()
20     return out
21 end
```

Here we are treating the operator to be an object for simplicity. In practice, all these values will be of data type String, therefore we will probably use some sort of selection contruct, either *if* or *switch-case* statements, to determine if i is an operator and if it is, which one is it.

6.2.4 Analysis of the two Methods

When inititially parsing the infix expression $f(x)$, the binary tree version will have time complexity $O(n)$ where n is the number of operators (n will be the number of operators, and m will be the number of operands in $f(x)$) in $f(x)$. The stack based version will convert the infix expression into a binary tree, and then perform depth-first traversal to convert the tree into post-fix. Depth-first traversal has a time complexity of $O(m+n)$ (every traversal visits every node, and there are $n+m$ nodes). This means that to initially parse the infix expression, the binary tree version has a smaller time complexity.
When evaluating $f(x)$ for a specific value of x, both versions have a time complexity of $O(m+n)$. This is because the binary tree version visits every node, $m+n$ (what we do is essentially post-order depth-first traversal but applying an operation each time), and the stack based version has $m+n$ items in the list that it goes through. Here we are assuming that each operation between operands takes equal time. This is a reasonable assumption to make because we are comparing two algorithms and they have the same input, $f(x)$, therefore we can remove the steps that it takes to complete the operations.
Now from a pure time complexity point of view, we can say that the binary tree version is better as the initial parsing is quicker. However when it comes to space complexity this is not true. When evaluating $f(x)$, the binary tree version uses a lot more memory because it creates subtrees every time it calls a recursive function. On the otherhand, with the stack based version, the maximum space that could be occupied is $(m+n)+m)$. $(m+n)$ is the size of the orignal list that we pull values from, and m is the maximum size our stack could ever get (we never add an operator to the stack). From a memory point of view a binary tree is not very effieciently stored. Most languages don't have a binary tree construct, and implementing your own with primitive arrays (each child is an array) is ineffiecient. For example the binary tree,



would be represented as (actual implementation would have pointers),

$$[*[+[x,4]],[-[x,3]]]$$

If we then stored this in contiguous memory,

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ * & + & x & 4 & - & x & 3 \end{bmatrix}$$

Now when doing post-order depth-first traversal, we jump between the memory locations,

$$\begin{bmatrix} 2 & 3 & 1 & 5 & 6 & 4 & 0 \\ x & 4 & + & x & 3 & - & * \end{bmatrix}$$

This is horribly ineffiecient because our stride is not consistent. This is unlike our stack based version, where we always have a stride of 1, no matter what. While this is not that big of a problem for one evaluate call, when 2000 evaluate calls are made, this makes a massive impact. We could, in theory, modify the order that we store our tree into one which is effiecient when doing a depth-first traversal but this is what we did when we converted our tree into post-fix notation. Therefore using stacks to process input is best for performance even though, the initial parsing takes longer. However this does not mean that we are not going to consider binary trees. This is because we need binary trees to initially parse our infix expression, therefore we will use stacks with binary trees to parse our user input. We need to inplement trees and stacks in Java before we can start parsing our infix expression.

6.2.5 Implementing Stacks and Trees

Since stacks and trees are a prerequisite to our function class, we will design them now.
Our stack does not need to become infinitely big, as the size of the `Stack`, as discussed in the last section, is m where m is the number of operands. Therefore we can initialize an array of size m and manipulate the array to make it act like a `Stack`.
Our binary tree will be quite simple, containing only the bare minimum of what our binary tree needs, with an added traversal function which returns the post-order depth-first traversal of the tree. For simplicity we will use a private static helper function, which will take a tree and a stack as parameter. This stack which is passed in by reference will be edited in every recursive call. We will also make our left and right tree attributes public. This is because our root tree will be public, and as the left and right sub-trees are simply different copies of the root tree, it isn't consistent to make the right and left private while keeping the root tree public.

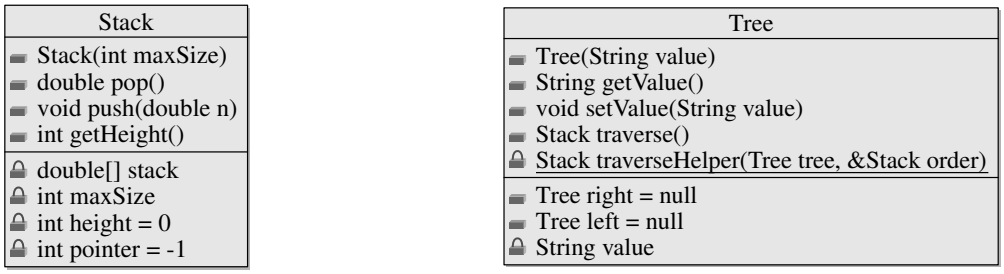


Figure 6.1: Class Diagrams

Algorithm 3: Post-Order Depth-First Traversal Helper

```
1 function traverseHelper(Tree tree,Stack &order):
2 |   if tree == null then
3 |   |   return;
4 |   else
5 |   |   traverse(tree.left,order)
6 |   |   traverse(root.right,order)
7 |   |   order.add(root.data)
8 |   end
9 end
```

Algorithm 4: Post-Order Depth-First Traversal

```
1 function traverse():
2 |   Stack order
3 |   traverseHelper(this,order)
4 |   return order
5 end
```

6.2.6 Parsing Algorithm

6.2.6.1 Least Significant Operator

An important part of the parsing stage, is the ability to find the least significant operator within an expression. If there are no operators -1 will be returned. If an operator is inbetween within a pair of brackets it is automatically discounted. We do this by respectively incrementing or decrementing a variable when we reach an opening or closing bracket to monitor when we are within a pair of brackets.

Algorithm 5: Least Significant Operator Position

```
1  function leastSigOperatorPos(String input):
2      Integer parenthesis = 0
3      Integer leastSigOperatorPos = -1 /* stores the position of the least significant
           operator so far                                                          */
4      Integer leastSigOpcode = 1000
5      Character[] operators = ["+",' "-", "/", " * ", " ^"] /* stores each operator in order of
           incresing significance                                                   */
6      Integer currentOpcode /* the current index of the operator in the array operators  */
7      Character currentChar
8      for i=0 to (input.size - 1) by 1 do
9          currentChar = input[i]
10         if currentChar in operators then
11             currentOpcode = operators.find(currentChar)
12             if (currentOpcode ≤ leastSigOpcode) and (parenthesis == 0) then
13                 leastSigOperatorPos = i
14                 leastSigOpcode = currentOpcode /* Update the least significant operator so
                       far, as it is now the current character                    */
15         else if currentChar == "(" then
16             parenthesis++
17         else if currentChar == ")" then
18             parenthesis−−
19         end
20     end
21     return leastSigOperatorPos
22 end
```

6.2.6.2 Removing Brackets

The function above assumes that there is no whitespace and that there are no brackets enclosing the entire expression (e.g. $(x - 4)$). We can deal with our whitespace issue in our constructor[7] however we need to make another function check for and remove any brackets surrounding an expression.

However algorithm 6 has some issues. For example if we have the expression $\frac{x+1}{x+2}$, this would be input as, $(x + 1)/(x + 2)$. Now if we apply algorithm 6 which removes enclosing brackets we get $x + 1)/(x + 2$. This is completely wrong, as in this case we do not want to any remove brackets at all. The significant issue here is that we only want to remove the enclosing brackets, if they are ***matching***.

To do this the algorithm 7 is more suited. This algorithm is based on 6 but checks for matching brackets. It does this by using a variable that increments everytime there is a opening bracket and decrements every time there is a closing bracket. When the variable becomes 0 the matching bracket has been found. If this

[7] In our function class we will store the original input so we can show the user, therefore we do not need to remove whitespace here

happens at the end the opening and closing brackets are removed, else nothing happens. This algorithm will also throw an exception if there are unequal number of opening and closing brackets. This is so that we can inform the user later of the error that they have made and so that we can kill the process instantly rather than letting this error have consequences later on (probably during the evaluation of a value).

Algorithm 6: Check for and remove any Brackets surrounding an input

```
1  function checkBracket(String input):
2      Boolean done = False
3      while !done do
4          done = True
5          if (input[0] == '(') and (input[input.size - 1] == ')') then
6              done = False
7              input = input.substring(1, input.size - 2)
8          end
9      end
10     return input
11 end
```

Algorithm 7: Check for and remove any Matching Brackets surrounding an input

```
1  function checkBracket(String input):
2      Boolean done = False
3      if input.count('(') != input.count(')') then
4          throw "There is an unequal number of opening and closing brackets"
5      while !done do
6          done = True
7          if input[0] == '(' and input[input.size - 1] == ')' then
8              Integer countMatching = 1
9              for i=1 to (input.size - 2) by 1 do
10                 if countMatching == 0 then
11                     return input
12                 else if input[i] == '(' then
13                     countMatching++
14                 else if input[i] == ')' then
15                     countMatching–
16                 end
17             end
18             if countMatching == 1 then
19                 done = False
20                 input = input.substring(1, input.size - 2)
21     end
22     return input
23 end
```

6.2.6.3 RegEx

Another important part of the parsing stage is to standardize the input. This is where we convert any inconsistencies discussed in section 6.2, page 16. The easiest way to do this is to use `RegEx[6]`. `RegEx` stands for regular expression and is a standardized form of pattern recognition in strings, usually used during syntax analysis during compilation of software. Many languages support `RegEx` in some form or another and Java is no exception. These were our 5 inconsistencies that we needed to fix:

1. Any instance of ax where $a \in \mathbb{R} : a \neq 0$ is to be converted to $a * x$.
2. Any instance of $a($ and $)a$ where a is not an operator, is to be converted to $a * ($ and $) * a$ respectively.
3. Any instance of $(f(x))(g(x))$ is to be converted to $(f(x)) * (g(x))$.
4. Any instance of $! - f(x)$ where ! is to be any operator (e.g. $*$ or /) is to be converted to $!(-f(x))$.
5. Any instance of $-f(x)$ at the start or next to an opening bracket is to be converted to $0 - f(x)$.

For the first and second examples, we look around every instance of x, (or) and if the adjacent characters are not operators or brackets then we replace with $*x$ or $x*$. Therefore we can combine the first and second examples to use two separate `RegEx` expressions to deal with the case where we have x after and where we have x before.

The `RegEx` expression for the first case is **"([^\(\)\+\-*\/\^])([\(a-z])"** with the replacement expression being **"$1*$2"**. If we take the `RegEx` expression, it creates two capture groups, *"$1"* and *"$2"*, which are *"([^\(\)\+\-*\/\^])"* and *"([\(a-z])"* respectively. A capture group stores a set of characters for each match that is made, so that we can perform actions on it later. The first capture group checks if the first character, in the substring that is currently being checked, is not any of the operators or brackets[8]. The not is signified by the first ^. The second capture group checks if the second character, in the substring that is currently being checked, is a constant/variable [9] , signified by the *"a-z"* or is an opening bracket *"("*. If both capture groups return true then a match is found and the match is replaced with *"$1*$2"* where *"$1"* is the first capture group, *"$2"* is the second capture group and the * asterisk between them signifying the multiply.

The `RegEx` expression for the second case is **"([\)a-z])([^\(\)\+\-*\/\^])"** with the replacement expression being **"$1*$2"** again. This expression does the same as the first but checks for the reverse order i.e. xa and instead checks for a) instead of (as we are checking the back of a substring instead of the start.

For the third inconsistency, the `RegEx` expression is **"\)\("** with the replacement expression being **")*("**. This expression returns a match if it finds a) followed by a (. If it finds a match it then replaces the entire match with *")*("*.

For the fourth inconsistency, the `RegEx` expression is **"([\+\-*\/\^])-([^\+\-*\/\(\)\^]*)"** with the replacement expression being **"$1(-$2)"**. The `RegEx` expression returns a match when there is an operator followed by a minus sign followed by any number of characters that are not operators or brackets. There are two capture groups. The first is *"([\+\-*\/\^])"* and this captures the operator. The second is *"([^\+\-*\/\(\)\^]*)"* and this captures the expression after the minus sign. The match is then replaced by the first capture group, followed by an opening bracket, a minus sign, the second capture group, then a closing bracket.

For the fifth inconsistency, the `RegEx` expression is **"(^|\()-"** with the replacement expression being **"$10-"**. The `RegEx` expression returns a match when it is either the start of a line, signified by the ^, or[10] a (followed by a minus sign. The start of the line of bracket is captured and is used in the replacement expression, when the match is replaced with a 0- preceded by either a bracket or nothing depending on if the start of a line or an opening bracket was captured.

[8] the reason there are so many backslashes is because a lot of the operators are actually key characters in `RegEx` and a blackslash is an escape character which means that it signifies to treat the next character as a pure character

[9] While we are only dealing with the variable x at the moment, this allows for constants to be used in the future and be dealt with correctly.

[10] the or keyword in `RegEx` is signified by |

6.2.6.4 Creating the Tree and the Stack

Using the functions above we can create a syntax tree from our expression. Before we dive into this, remember that we will convert this into a `Stack` later to process, since it is more efficient (section 6.2.4). To create the `Tree` we will use the Divide and Conquer methodology by recursively splitting the original expression until it becomes a single constant or variable. We can know if we should split the expression and if so where we should split the function by using our Least Significant Operator function. Our recursive case will return a `Tree` where the left and right nodes are made up of the trees of the two sub-expressions and the root node will be the operator that we split our original expression with. Our base case will return a `Tree` containing the constant or variable that is remaining. For example if we have the expression "$x^2 + 4$", we will first split this expression by the least significant operator which is the "+". We then make this our root node and our left and right trees will be what is left and right of that operator. This will be repeated for each sub-tree until only no operators remain. So our bottom nodes will be x, 2 and 4.

Algorithm 8: Create a Binary Tree for an Algebraic Expression

```
1  function createTree(String expression):
2      expression = checkBracket(expression)
3      Integer leastSigOperatorPos = leastSigOperatorPos(expression)
4      if leastSigOperatorPos == -1 then
5          return new Tree(expression)
6      else
7          String operator = expression[leastSigOperatorPos]
8          String a = expression.substring(0, leastSigOperatorPos)
9          String b = expression.substring(leastSigOperatorPos+1,expression.length - 1)
10         return new Tree (operator,createTree(a),createTree(b))
11     end
12 end
```

Within this algorithm I can use concurrent processing to process each sub-expression individually. To do this I will create 2 threads to create the 2 individual sub-trees. I will then start these threads and on the root thread wait for these threads to finish (usually done using a method called "join"), and then return the new `Tree` object made up of the 2 individual sub-tree.

Algorithm 9: Create a Binary Tree for an Algebraic Expression (Using Concurrent Processing)

```
1  function createTree(String expression):
2      expression = checkBracket(expression)
3      Integer leastSigOperatorPos = leastSigOperatorPos(expression)
4      if leastSigOperatorPos == -1 then
5          return new Tree(expression)
6      else
7          String operator = expression[leastSigOperatorPos]
8          String a = expression.substring(0, leastSigOperatorPos)
9          String b = expression.substring(leastSigOperatorPos+1,expression.length - 1)
10         Thread threadA = new Thread (createTree(a))
11         Thread threadB = new Thread (createTree(b))
12         threadA.join()
13         threadB.join()
14         return new Tree (operator,createTree(a),createTree(b))
15     end
16 end
```

6.2.7 Substitute Algorithm

To actually plot the functions we need to be able to substitute values into the actual functions to know where to draw our points. Eventually we should be able to substitute multiple variables/constants into our function, but for now we will only substitute an x value into our function. This can be an extension for my next iteration.
Now algorithms 3 and 4 on page 23 traverse a tree and output a list (we will implement this using a stack, within the constructor). However the most significant operations and values, i.e. the actions we would do first according to **BIDMAS** [11] , are at the front of the list. This means that in a stack they would be at the bottom, which means they would be popped off last which is not what we want when we evaluate a function. While we can reverse the stack, this is inefficient and also defeats the purpose of using a stack.

However this is fixed by the need of this `substitute` algorithm. We will need to traverse this stack to replace the variables/constants with actual values anyway, so while we do this we can reverse the stack, fulfilling to purposes.

To accomplish this we will create a new stack, of the same height as the original, and then pop each value of the original stack, checking if it is a variable, and if it is substituting the value required. The value will then be pushed onto the new stack. This new stack will then be returned. We also will need to create a copy of the old stack and manipulate that so that we retain the original stack for other substitution calls.

In order to make my function class more generic, I will introduce a new attribute of type `character` called `parameter`. In all our examples our parameter has been x, but we may eventually use different parameters. Within this algorithm, instead of checking for the character x, I will instead check for the character `parameter`. Within this algorithm I have assumed that a `character` is equivalent to a `String` of

Algorithm 10: Substitute into a function

```
1  function substitute(Double x):
2      Stack copy = postFixStack                                   // Copy the Post-Fix Stack
3      Stack substituteStack = new Stack (this.postFixStack.getHeight())  /* Create new Stack the
        same size to reverse it into */
4      for  i=0 to copy.getHeight() do
5          String pop = copy.pop()
6          if pop == parameter then
7              pop = x
8          substituteStack.push(pop)
9      end
10     return substituteStack
11 end
```

length 1 (on line 6). This is obviously not the case, therefore during implementation I will need to cast the character to a `String` or do another method that allows me to compare the two values.

[11] **BIDMAS** stands for **Brackets**, **Indices**, **Division**, **Multiplication**, **Addition** and **Subtraction**. It signifies the order that we must do operations on an expression. It is sometimes called **BODMAS** where the **O** stands for **Orders**.

6.2.8 Evaluate Algorithm

The `evaluate` algorithm is very similar to algorithm 2, the stack-based evaluate algorithm. However we will pass in the inputs as parameters and use the `substitute` algorithm within our `evaluate` algorithm to replace our variables with the actual values. We will also hardcode the selection and verification of which operator to use instead of assuming that an operator is an object we can manipulate. A precondition of this algorithm is that we are assuming that the original expression is valid, otherwise this algorithm may not behave as intended.

Algorithm 11: Evaluate the Expression for a Value of x

```
1  function evaluate(Double x):
2      Stack substituteStack = substitute(x)
3      Stack evaluateStack = new Stack (postFixStack.getHeight())
4      for  i=0 to substituteStack.getHeight() do
5          String pop = subStack.pop()
6          Double a,b
7          switch pop do
8              case "+" do
9                  b = evaluateStack.pop()
10                 a = evaluateStack.pop()
11                 evaluateStack.push(a + b)
12                 break
13             case "−" do
14                 b = evaluateStack.pop()
15                 a = evaluateStack.pop()
16                 evaluateStack.push(a − b)
17                 break
18             case "∗" do
19                 b = evaluateStack.pop()
20                 a = evaluateStack.pop()
21                 evaluateStack.push(a ∗ b)
22                 break
23             case "/" do
24                 b = evaluateStack.pop()
25                 a = evaluateStack.pop()
26                 evaluateStack.push(a/b)
27                 break
28             case "∧" do
29                 b = evaluateStack.pop()
30                 a = evaluateStack.pop()
31                 evaluateStack.push(a^b))
32                 break
33             default do
34                 evaluateStack.push(Double (pop))
35                 break
36             end
37         end
38     end
39     return evaluateStack.pop()
40 end
```

6.2.9 Constructor and the Class as a Whole

The constructor for the `Function` class will be quite simple. We take an expression as an input and:

1. Remove Whitespace from the expression
2. Standardize the expression (using the RegEx rules)
3. Create the Binary Tree
4. Create the Post-Fix Stack from the Binary Tree

Algorithm 12: Function Class Constructor

```
1 function Function(String expression):
2 |   expression = expression.strip()
3 |   expression = standardize(expression)
4 |   this.expression = expression this.binaryTree = createTree(this.expression)
5 |   this.postFixStack = this.binaryTree.traverse()
6 end
```

The class as a whole will look something like the diagram below.

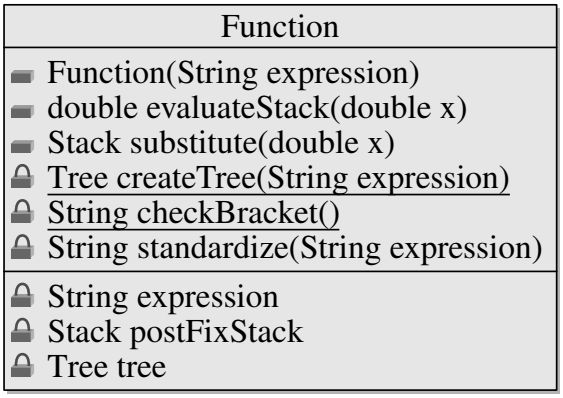


Figure 6.2: `Function` Class

6.3 Graphical User Interface

The Graphical User Interface will be achieved using one the many toolkits within `Java`. Specifically I will be using `JavaFX[9]`. `JavaFX` is unique in the fact that the GUI can be generated using two approaches:

1. Within the code itself, we can define how the GUI[10] is comprised. This is done by instantiating Objects of specific component Classes, such `Pane`, `Label`, `Button` etc. all inheriting a top class called `Node`, and using their methods to join them together. The advantage of this approach is that since these are simply classes, we can create child classes from them allowing us to create custom components that we can control.

2. The UI is defined using `.xml` files (aptly named `FXML[11]` files). A controller class is defined alongside it. This controller class is referenced within the the `FXML` file and when the `FXML` file is loaded, the controller class is initialized along with it. This controller allows us to "control" the UI nodes. While this approach is very simple to implement and the design of the UI looks clean, it can be somewhat convoluted to allow the controller classes to interact with other controller classes.

The real strength of `JavaFX` however comes from the fact that these two approaches can be intertwined as needed, making solutions unique and versatile.

The GUI can be decomposed into smaller parts as shown by the hierarchical diagram shown below. The parts in **bold** are the parts to be implemented and designed in Prototype I and the rest will be completed in Prototype II.

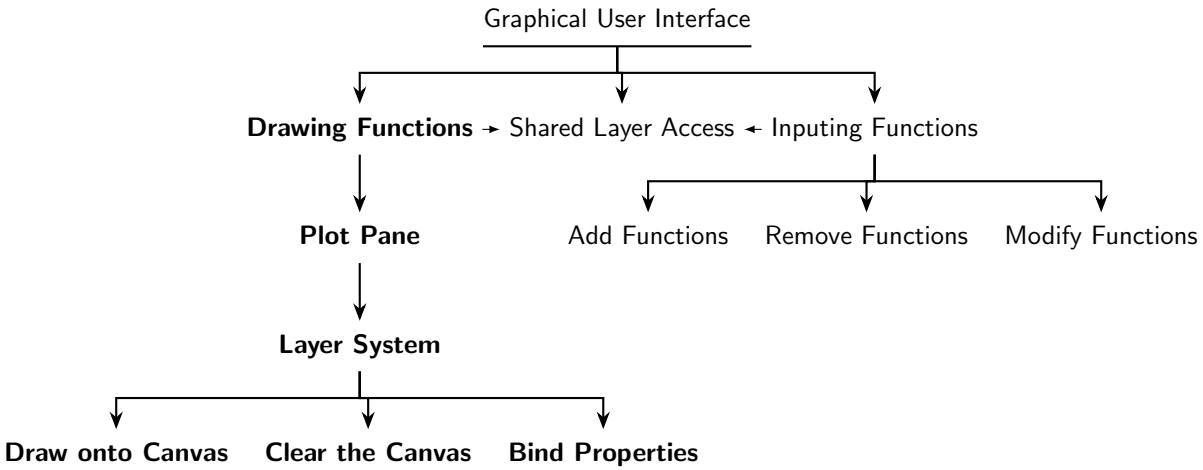


Figure 6.3: Graphical User Interface Hierarchical Diagram

Many of the features above will be discussed in greater detail later, however the Shared Layer Access and Properties will be touched upon now.

As mentioned before it can be hard to make two controller classes interact, as such one solution[12] is an object that both controller classes can reference. This class will contain attributes and methods that both classes need to interact with each other.

However, how do these classes know when something changes and hence know when to update their corresponding UI elements. The answer is through a class called `Property[13]`. A `Property` can bind another `Property`, one updating when the other one changes. Objects called `Listener` can also be added to properties. A `Listener` can be programmed to do a specific action when a `Property` is updated. This action could be a function, or updating a variable. Using the class `Property` allows for a responsive UI, updating whenever the user does an action.

6.3.1 Overview of the Graphical User Interface

The GUI will look something like:

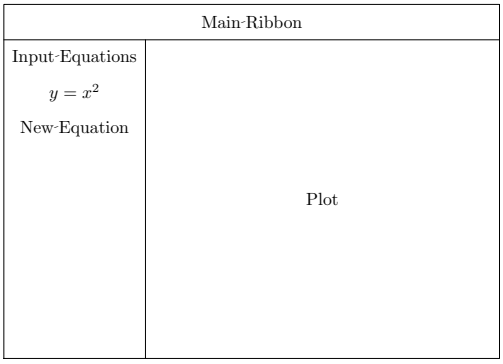


Figure 6.4: UI Design

as showcased in the analysis.

Within `JavaFX` there is a hierarchy[14] of a standard GUI components. This is shown below:

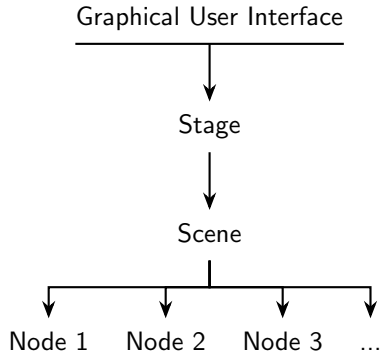


Figure 6.5: `JavaFX` Components Hierarchical Diagram

Now we can manipulate the the nodes beneath `Scene`. It is standard to make the first `Node` beneath `Scene` some kind of `Pane`. For this project a `BorderPane` is most suited since it has multiple objects of type `Region`, that can hold one `Node` each. A diagram of this is shown to the right showcases how a `BorderPane` is arranged[15] and way it behaves.

The center `Region` can consist of the plots themselves, and the left `Region` can consist of the input. The top `Region` could be used for the ribbon and other buttons to manipulate the plot.

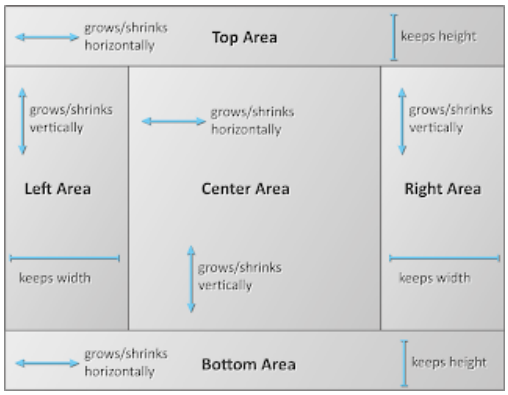


Figure 6.6: `BorderPane` Architecture

6.3.2 Layer System

Drawing onto the screen using `JavaFX` can be achieved using the `Canvas[16]` object. This is a node, which allows us to draw shapes, lines, arcs, on to the screen. As discussed in the analysis, we can approximate a curve by drawing many lines however one of our requirements was to draw multiple functions. From the article cited above, there are two ways to achieve this:

1. Draw every function onto one `Canvas` and add this to the root `Pane`.

2. Create a `Canvas` for every function and draw each function onto the corresponding canvasses and add all these canvasses onto the root `Pane`. One canvas will be paired with one function and bundled `Layer`. This only works because canvasses are by default transparent.

The advantage of the first is that it is simple to implement. However if there are multiple types of functions, i.e. an explicit function and an implicit function, then each one of these functions must have a draw function associated with it. This can become slightly convoluted, because then we must check what type of function something is, before we draw it.

Although the second is harder to implement, it is by far the superior. A `Canvas` is essentially an abstracted `OpenGL` view, and hence it is almost impossible[17] to draw multiple things on the same canvas at once. However it is possible to draw onto multiple canvasses at once, which boosts performance on multicore system. This also means that we can create a class for each type of function (explicit, implicit, parametric, etc.), making this system more modular. It is important to realize that only the topmost `Canvas` receives input (keyboard and mouse events) and hence we can create a `Layer` solely to handle input.

However before we handle the canvasses themselves, we must concern ourselves with how we can draw a line at a specific place.

6.3.2.1 Coordinate Systems

The coordinate system that we are familiar with is the Cartesian Coordinate System. Cartesian Coordinates take a fixed point as reference, usually called the origin (symbolized by O), and specify a point by taking the distance from the origin in n directions, where n is the dimension of the system. The directions are vectors, that are all perpendicular to each other, and have magnitude 1, i.e. unit vectors. This graphing software will be dealing with a 2-Dimensional Cartesian Coordinate System and in this system, up and down are defined as the y direction, with up being positive and down being negative; whereas right and left are defined as the x direction, with right being positive and left being negative. Within our graphing program we will only show a section of the x–y plane. This is because:

1. It is intractable to show the entire x–y planesince the plane goes on for an infinite distance in both the x and y directions.

2. It is not helpful to the user to see the entire plane even if they could. A graphing tool should enable the user to highlight parts of the graph that is important to them.

However a `JavaFX Canvas` doesn't exactly use a Cartesian Coordinate System. It specifies a position on itself taking the top-left corner as the origin, right as the positive x direction and down as the positive y direction. It is important to realize that negative coordinates do not exist in this system. The maximum x and y values are the width and height, respectively, in pixels.

We must therefore find a way to convert between these two coordinate systems. Since our viewport is predefined, let us label what the maximum and minimum x and y values in our viewport (with respect to our original Cartesian Coordinate System). Let these values be `minX`, `maxX`, `minY` and `maxY`.
Now let us define how much the width and height of each pixel on the canvas is `''worth''`. This *"worth"* specifies how much going across or down a pixel width or height, is in terms of our original Cartesian Coordinate System. This is best demonstrated with an example. If our canvas shows the x–y planein the x direction from 0 to 10, and our canvas is 1000 pixels wide, then our pixel worth is the amount we have

traversed in the x direction, divided by the number of pixels, in this case $\frac{(10-0)}{1000} = 0.01$. The algorithm for this is shown below:

Algorithm 13: Calculate the *"Worth"* of each Pixel

```
1 function updatePixelWorth():
2 |   pixelWorthX = (maxX - minX)/canvas.getWidth()
3 |   pixelWorthY = (maxY - minY)/canvas.getHeight()
4 end
```

Now that we have the pixel worth, we can convert between our coordinate systems. To do this we take the vector from the origin, $O = (x_{min}, y_{max})$, to the point, $P = (x, y)$ in the x or y direction. We then calculate how many pixel *"worths"* this is and this gives us the pixel on the canvas that this coordinate is at. It is important to realize that going down in a `JavaFX Canvas` is actually positive, relative to the canvas, hence our two algorithms are slightly different. This means our vector is actually

$$\vec{d} = \begin{bmatrix} (x - x_{min}) \\ (y_{max} - y) \end{bmatrix}$$

Hence our position within the canvas is:

$$\vec{p} = \begin{bmatrix} \left(\frac{x - x_{min}}{x_{worth}}\right) \\ \left(\frac{y_{max} - y}{y_{worth}}\right) \end{bmatrix}$$

The algorithms below convert between x and y coordinates [12] respectively.

Algorithm 14: Convert the x Coordinate

```
1 function convertX(Double x):
2 |   x = x - minX /* Calculate the horizontal distance between the point and origin     */
3 |   x = x / pixelWorthX /* Calculate the number of pixel ''worths'' this is           */
4 |   return x
5 end
```

Algorithm 15: Convert the y Coordinate

```
1 function convertX(Double y):
2 |   y = maxY - y /* Calculate the vertical distance between the point and origin      */
3 |   y = y / pixelWorthY /* Calculate the number of pixel ''worths'' this is          */
4 |   return y
5 end
```

[12] Here we assume the coordinate is inside the viewport. Realistically we should check if the coordinates are in the viewport, i.e. bigger than the width or height of the canvas, or in some cases negative. However it does not matter since we can make this check later. Even then a `JavaFX Canvas` will check if the coordinates are in the canvas viewport anyway, so it does not matter to us since this detail is abstracted away from us.

6.3.2.2 Drawing the Function

From the analysis (Section refsec:computability) I briefly described how I could approximate a curve by creating a finite number of line segments.

Firstly we take the minimum value of x, which in the viewport is x_{min}. If our function is defined by $f(x)$ then let the coordinate of the function at this value be $P_1 = (x_{min}, f(x_{min}))$. We then take a second coordinate a small value dx away, so let the coordinate be $P_2 = (x + dx, f(x + dx))$. We now draw a line between these two points. We then make $P_1 = P_2$ and make P_2 with $x + 2 \cdot dx$. We repeat this process until $x + n \cdot dx \geq x_{max}$. We obviously need to convert the x and y values from the Cartesian coordinates into the canvas coordinates do we can do this when drawing the line. dx will also be a finite value that is very small. We will define this by taking the number of line segments we want, and working out dx for that number of steps. This is simply done by finding out how much each step is *"worth"* within the viewport, i.e. $dx = (\texttt{maxX} - \texttt{minX})/\texttt{steps}$. The algorithm for this is shown below.

Algorithm 16: Draw a Function in the Viewport

```
1  function draw():
2  |   Double x1 = minX
3  |   Double y1 = f.evaluate(x1)
4  |   Double dx = (maxX - minX)/steps
5  |   for x2=minX + dx to maxX by dx do
6  |   |   y2 = f.evaluate(x2)
7  |   |   drawLine(convertX(x1), convertY(y1), convertX(x2), convertY(y2))
8  |   |   x1 = x2
9  |   |   y1 = y2
10 |   end
11 end
```

A precondition of this is that `f` is a valid function. We also need to make sure we do not divide by 0, but this can be sorted out in the implementation.

6.3.2.3 Constructor and Class as a Whole

The layer class we have described in the previous sections, describe a layer for an explicit function, in terms of x, in a Cartesian Coordinate System. To take full advantage of an OOP approach, as described at the start of this section, we can use layers for input, drawing axes etc. Hence we will create a superclass called `Layer` and then make our other types of layers inherit this class. This superclass will never be instatiated, in OOP terms it is abstract. The superclass will have some basic methods and attributes which we have used during this design, such as `draw()`, the attributes of the viewport (`minX`, `maxX`, etc.), the functions to convert between coordinate systems, where some of the functions such as `draw()` will be overriden. Our `Explicit Function` and `Axes` classes are the only layers we will implement (apart from the superclass layer) in prototype 1. The input layer will be part of prototype 2. While we haven't talked about the axes layer, this will simply be drawing the lines $y = 0$ and $x = 0$. This is done just for reference so that we can verify that the functions are drawn somewhat accurately.

The constructor for the Explicit Function Layer is shown below.

Algorithm 17: Explicit Function Layer Class Constructor

```
1 function ExplicitFunctionLayer(String function):
2 |   super()
3 |   this.f = new Function(function)
4 end
```

While the superclass will never be instantiated, it can still contain a constructor. Here it will be used to initialize the canvas and its `OpenGL` context, in a standardized manner, so that we do not have repeated, redundant code for no reason.

Algorithm 18: Layer Class Constructor

```
1 function Layer():
2 |   this.canvas = new Canvas()
3 |   this.gc = canvas.getGraphicsContext2D()
4 end
```

The class diagrams for the `Layer` [13] classes are shown below:

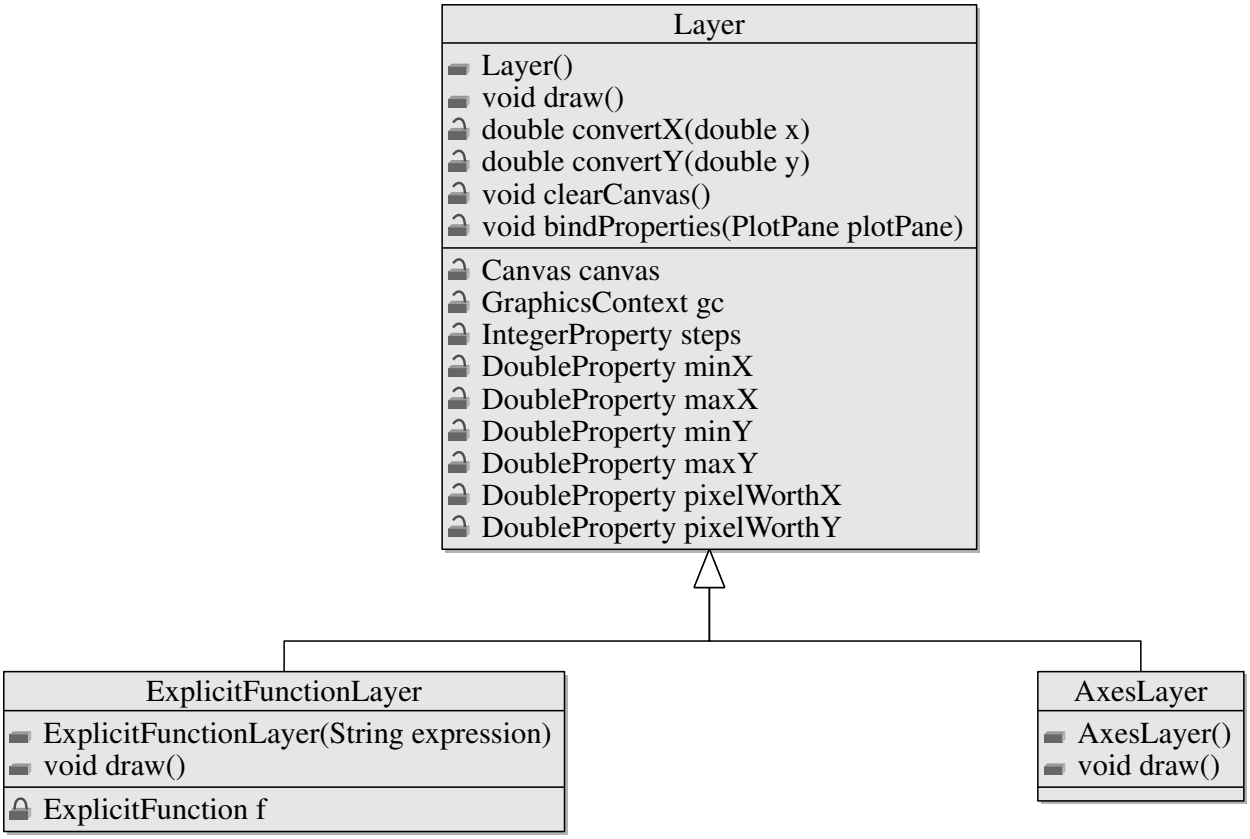


Figure 6.7: `Layer` Classes

[13] The inherited methods and attributes are protected so that they can actually be viewed and accessed by the child classes.

6.4 Plot Pane

The purpose of the `PlotPane` is to manage the layers. The `PlotPane` will tell them when to draw and notify them when something changes through the use of properties. Our `PlotPane` will inherit the `JavaFX Pane`. While we could have used `FXML`, `FXML` is more applicable when there are complicated UI components that are hard to arrange and manage. In this situation we only need to have one `Pane` hence there is no need to use `FXML`. Inheriting `Pane` also allows it to integrate well within the `JavaFX` hierarchy.

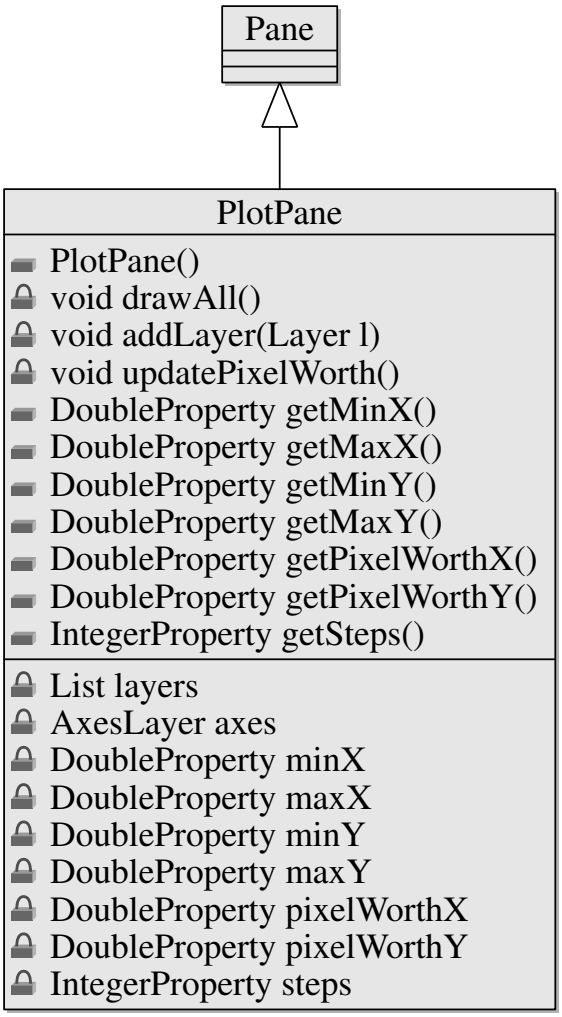


Figure 6.8: `PlotPane` Class

The `drawAll()` procedure will call the draw function for every `Layer`. We will start by doing this on a single thread, and then we will implement it using multiple threads. The `addLayer()` function will add the `Layer` to the list, and bind the layers' properties to the `PlotPane`.

Chapter 7

Implementation

When beginning to implementing my solution, I made a few decisions. I realized a couple of important ideas which I did not address in my Design:

1. My function class designed was more of a class for an expression instead of a function. This is significant because different functions will use expressions in different ways. For example, an explicit function will use one expression to define the relationship, whereas an implicit function will use two expressions to define this relationship. Therefore to make our solution expandable in the future it would make more sense to have an expression class with multiple other classes for the different types of functions.

2. The layer system only relates to the Cartesian Coordinate System. This is a problem when I expand my program later to contain the Polar Coordinate System. So what I will do to fix this problem, is to take the `Layer` class that I created, rename it to `CartesianLayer` and make this class inherit a new class called `Layer` which will be all the attributes some key methods that all layers will have. I will also make another class called `ExplicitFunctionLayer` which will inherit from `CartesianLayer`. This allows me to split up each one of my layers into the types of function that will be used.

3. `Java` allows projects to be split up into packages. To organize my implementation I will split it up into the following packages:

   (a) `application` - Contains the UI classes, controllers, `PlotPane`, etc.

   (b) `layers` - Contains the `Layer` classes.

   (c) `structures` - Contains the data structure classes, stacks, trees, expressions, functions, etc.

   (d) `exceptions` - Contains custom exceptions.

7.1 Stacks

I will be implementing Stacks using a concept called *Generic Programming[18]*. This is best explained with an example.

Imagine we have a data structure which will only ever contain one type of object but we will use this data structure in different scenarios, each with different types of object. The structure must also be independent of the objects it is storing.

Now we can use an OOP approach where we have a base abstract class called `Structure` and then we create other classes that are related to a certain type of object and make it inherit this class, e.g. `IntegerStructure` or `StringStructure`. However this can become unfeasible to do this for every single object that we use this structure for.

A more elegant approach is to use *Generics*. This is where we link a type of object with the structure. Let this type of object be of type `T`. Now we make our structure of type `T` and make all of its methods take or return an object of type `T`. This allows the structure to be versatile and be used for any object.

Here is the implementation I have created in `Java` using *Generics[19]*:

```
1  package structures;
2
3  import exceptions.StackOverflowException;
4  import exceptions.StackUnderflowException;
5
6  public class Stack<T> {
7
8          // attributes
9          // pointer variables
10         private int maxHeight;
11         private int height = 0;
12         private int pointer = -1;
13         // stack
14         private T[] stack;
15
16         // methods
17         // constructor
18         @SuppressWarnings("unchecked")
19         public Stack(int maxHeight) {
20                 this.maxHeight = maxHeight; // set this max height
21                 // create an array of this size of type "T"
22                 this.stack = (T[]) new Object[this.maxHeight];
23         }
24
25         // push an item on to the stack
26         public void push(T push) throws StackOverflowException {
27                 if (height == maxHeight) { // check that the stack isn't full
28                         throw new StackOverflowException();
29                 } else {
30                         // make the element one above the top, the new value
31                         this.stack[this.pointer + 1] = push;
32                         this.pointer++; // increment pointer variables
33                         this.height++;
34                 }
35         }
```

```
36          // pop an item off the stack
37          public T pop() throws StackUnderflowException {
38                  if (this.isEmpty()) { // check that the stack isn't full
39                          throw new StackUnderflowException();
40                  } else {
41                          // get the value of the top element
42                          // no need to make it null, it is a waste of an instruction
43                          // it also releases no memory since we are using
44                          // an array to store our stack which is a static structure
45                          T pop = this.stack[this.pointer];
46                          this.pointer--; // decrement pointer variables
47                          this.height--;
48                          return pop; // return the popped value
49                  }
50          }
51
52          // check if stack is empty, return true if so
53          public boolean isEmpty() {
54                  return this.height == 0 ? true : false;
55          }
56
57          // return the height of the stack
58          public int getHeight() {
59                  return height;
60          }
61
62          // allow for a visualization of the stack by listing it
63          // with the top element being encapsulated in square brackets
64          @Override
65          public String toString() {
66                  if (this.isEmpty()) { // if empty notify the user
67                          return "Stack is Empty";
68                  } else {
69                          // highlight the top element
70                          String out = "[" + this.stack[this.pointer] + "]";
71                          // loop through the rest of the stack backwards
72                          // concatenating each element to a string
73                          for (int i = this.pointer - 1; i >= 0; i--) {
74                                  out = out + ", " + this.stack[i].toString();
75                          }
76                          return out; // return this string
77                  }
78          }
79
80  }
```

There are two custom `Exception` classes that I have created for this class, `StackOverflowException` and `StackUnderFlowException`, in a separate package. These simply output a different message to the standard `Exception` class. The code for this is at the end of the documentation.

To just make sure that my code is working as intended I tested 4 things:

1. Popping and pushing items
2. Overflowing the stack
3. Underflowing the stack
4. Copying a stack (this is needed for the substitute algorithm)

Here is the code for test 1:

```
// add two items, pop one off, add another one
Stack<String> stack = new Stack<String>(4);
stack.push("item 1");
System.out.println("Stack -> "+stack); // expecting item1
stack.push("item 2");
System.out.println("Stack -> "+stack); // expecting items 1 and 2
String pop = stack.pop();
System.out.println("Popped item -> "+pop); // expecting items 2
stack.push("item 3");
System.out.println("Stack -> "+stack); // expecting items 1 and 3
```

Here was the output for test 1:

```
Stack -> [item 1]
Stack -> [item 2], item 1
Popped item -> item 2
Stack -> [item 3], item 1
```

Which is exactly what I expected.

To spice things up for test 2 I made a stack of integers instead of strings just to make sure the generics part is working properly. Here is the code for test 2.

```
// create a stack of max height 4
Stack<Integer> stack = new Stack<Integer>(4);
// push 4 items, the max amount
stack.push(1);
stack.push(2);
stack.push(3);
stack.push(4);
// show that the stack has 4 items
System.out.println("Stack -> "+stack);
// push another item
stack.push(5);
```

Here was the output for test 2:

```
Stack -> [4], 3, 2, 1
Exception in thread "main" exceptions.StackOverflowException:
Stack has reached its max height
        at structures.Stack.push(Stack.java:40)
        at structures.Stack.main(Stack.java:107)
```

Which is exactly what I expected.

Again for test 3 I made a stack of doubles. Here is the code for test 3.

```
// create a stack
Stack<Double> stack = new Stack<Double>(4);
// try and pop an item off
stack.pop();
```

Here was the output for test 3:

```
Exception in thread "main" exceptions.StackUnderflowException:
Stack has no items for you to pop
        at structures.Stack.pop(Stack.java:52)
        at structures.Stack.main(Stack.java:100)
```

Which is exactly what I expected.

For test 4 I made a stack of booleans. Here is the code for test 4.

```
// create a stack
Stack<Boolean> stack = new Stack<Boolean>(4);
// pop some items
stack.push(true);
stack.push(false);
// assign the value of the old stack to the copy stack
Stack<Boolean> copy = stack;
// pop an item off the copy stack to test that they are independent
copy.pop();
// output both stacks
System.out.println("Stack -> "+stack);
System.out.println("Copy -> "+copy);
```

Here was the output for test 4:

```
Stack -> [true]
Copy -> [true]
```

This is not what I expected. I expected the original stack to have an extra item. It appears as if they are mimicking each other, like they are sharing the same memory address. When I researched this issue[20] it appears that this is true. In `Java` every variable is a reference so when you assign a variable the value of an object, you actually assign the variable a reference. The exceptions to this are when you pass variables into functions, this is done by value not reference [1] , and with primitives [2]. The suggested solution was to use `Serialization[21]`. This is where an object is serialized into a `byte` array and then deserialized into an object again. This works because we are effectively converting an object into a primitive and then assigning this to a new value, which bypasses the reference issue. However this can be a hit on performance. So why don't I use the same principle in my code. Since all my attributes are primitives, if I just create a constructor which assigns the new stacks' attributes' values the original stacks' attributes, the problem should be solved.

[1] A function cannot have parameters passed by reference[22] in `Java`

[2] A language's base types of object that are inherently part of the language. In `Java` the primitives are `int`, `double`, `float`, `boolean`, `char`, `byte`, `short`, `long`.

I made a constructor that takes an existing stack as a parameter to solve the issue. In many languages this is called a *Copy Constructor*. The code for this is below:

```
// copy constructor
public Stack(Stack<T> copy) {
        // assign the new stacks' attributes' values the original stacks' attributes
        this.pointer = copy.pointer;
        this.height = copy.height;
        this.maxHeight = copy.maxHeight;
        this.stack = copy.stack;
}
```

For my new test I slightly modified it from the original by using the copy constructor. Here is the code for new test:

```
// create a stack
Stack<Boolean> stack = new Stack<Boolean>(4);
// pop some items
stack.push(true);
stack.push(false);
// assign the value of the old stack to the copy stack
Stack<Boolean> copy = new Stack<Boolean>(stack);
// pop an item off the copy stack to test that they are independent
copy.pop();
// output both stacks
System.out.println("Stack -> "+stack);
System.out.println("Copy -> "+copy);
```

Here was the output for the test:

```
Stack -> [false], true
Copy -> [true]
```

This is what I expected.

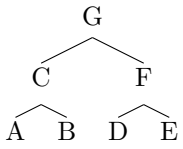
7.2 Trees

Our trees will be simpler to implement than our stacks. This is because we do not need to use generics with our trees since we will only be storing strings within them. The only hard part of this implementation was the fact that you cannot pass by reference[22] in Java. I got around this issue by simply passing this list in, and returning it each time. Here is the implementation I have created in Java:

```java
package structures;

import exceptions.StackOverflowException;

public class BinaryTree {

	// attributes
	// children
	public BinaryTree left;
	public BinaryTree right;
	// node value
	private String value;

	// methods
	// constructor - create a null tree
	public BinaryTree() {
		this.left = null;
		this.right = null;
		this.value = null;
	}

	// constructor - create leaf tree i.e. no children
	public BinaryTree(String value) {
		this.value = value;
		this.left = null;
		this.right = null;
	}

	// constructor - create a tree with the children defined
	public BinaryTree(String value, BinaryTree left, BinaryTree right) {
		this.value = value;
		this.left = left;
		this.right = right;
	}

	// return the value of the current node
	public String getValue() {
		return value;
	}

	// set the value of the current node
	public void setValue(String value) {
		this.value = value;
	}
```

```
45
46          // count the number of nodes beneath this tree + this tree
47          public int countNodes() {
48                  // set the number of nodes counted to 0
49                  int count = 0;
50
51                  // recursive case, counts the number of nodes in its children
52                  // if they are not null
53                  if (this.left != null) {
54                          count = count + this.left.countNodes();
55                  }
56                  if (this.right != null) {
57                          count = count + this.right.countNodes();
58                  }
59                  // return the count of the children
60                  // but increment it to count it self
61                  return count + 1;
62          }
63
64          // post-order depth-first traversal wrapper function
65          public Stack<String> traverse() throws StackOverflowException {
66                  // create a stack of max height
67                  // which is the number of nodes in the entire tree
68                  // this is to save memory
69                  Stack<String> order = new Stack<String>(this.countNodes());
70                  // traverse the tree
71                  order = traverseHelper(this, order);
72                  return order;
73          }
74
75          // the actual recursive traversal function
76          private static Stack<String> traverseHelper(BinaryTree tree, Stack<String> order)
          ↪  throws StackOverflowException {
77                  // base case, if the tree is null just return the unmodified stack
78                  if (tree == null) {
79                          return order;
80                  } else { // recursive case search its children if not null
81                          // since this is post-order, the order is left, right then root
82                          // traverse left
83                          order = traverseHelper(tree.left, order);
84                          // traverse right
85                          order = traverseHelper(tree.right, order);
86                          // finally push the root value into the stack
87                          order.push(tree.value);
88                          return order;
89                  }
90          }
91
92  }
```

Just to test my binary tree implementation I tried to recreate the tree below:



If this graph has post-order, depth first traversal performed on it then the output should be:

$$\begin{bmatrix} A & B & C & D & E & F & G \end{bmatrix}$$

Here is the test code:

```
BinaryTree root;
BinaryTree left = new BinaryTree("C", new BinaryTree("A"), new BinaryTree("B"));
BinaryTree right = new BinaryTree("F", new BinaryTree("D"), new BinaryTree("E"));
root = new BinaryTree("G",left,right);
// print the traversal stack
System.out.println(root.traverse());
```

Here is the output:

```
[G], F, E, D, C, B, A
```

While this appears to be different to our expected stack it isn't. If we look closely the items are actually reversed, which makes sense since a stack is FILO (First In Last Out) structure and we made the visual representation of our stack implementation reflect this property. This means that A was pushed into the stack first, which means our tree implementation is working as intended.

7.3 Expressions

My `Expression` class, the one I designed as `Function`, is quite a complex and large, therefore I will implement each method independently and test them before I use them within my other methods.

7.3.1 Standardize Expression

I created a method to standardize my expression using the `RegEx` expressions that I made. This method will be `static` since it isn't related to the object, but instead to the class as a whole. I also added support for a few constants such as e and π so that we can use them later on. Here is the code:

```
// standardize an expression to remove ambiguity
private static String standardize(String expression) {
        // remove whitespace
        expression = expression.replace(" ", "");
        // make pi one character for faster regex
        expression = expression.replace("pi", "");
        // inconsistency 1
        String regex1 = "([^\\(\\)\\+\\-\\*\\/\\^])([\\(a-z])";
        String replace1 = "$1*$2";
        // inconsistency 2
        String regex2 = "([\\)a-z])([^\\(\\)\\+\\-\\*\\/\\^])";
        String replace2 = "$1*$2";
        // inconsistency 3
        String regex3 = "\\)\\(";
        String replace3 = ")*(";
        // inconsistency 4
        String regex4 = "([\\+\\-\\*\\/\\^])-([^\\+\\-\\*\\/\\(\\)\\^]*)";
        String replace4 = "$1(-$2)";
        // inconsistency 5
        String regex5 = "(^|\\()-";
        String replace5 = "$10-";
        // perform regex
        expression = expression.replaceAll(regex1, replace1);
        expression = expression.replaceAll(regex2, replace2);
        expression = expression.replaceAll(regex3, replace3);
        expression = expression.replaceAll(regex4, replace4);
        expression = expression.replaceAll(regex5, replace5);
        // replace constants with numerical values
        expression = expression.replaceAll("e",Double.toString(Math.E));
        expression = expression.replaceAll("",Double.toString(Math.PI));
        // return the standardized expression
        return expression;
}
```

To test my module I tested each of the inconsistencies and then a complex expression just to make sure everything is working. Here is the code with comments for what I expected:

```
System.out.println(standardize("2x"));          // 2*x
System.out.println(standardize("(x)3"));          // (x)*3
System.out.println(standardize(")("));          // )*(
System.out.println(standardize("*-x"));                  // *(0-x)
System.out.println(standardize("3x(-x+1)(x+2)"));          // 3*x*(0-x+1)*(x+2)
```

Here are the results:

```
2*x
(x)*3
)*(
*(0-x)
3*x(0-x+1)*(x+2)
```

All the tests passed except the last one. It seems as though, the `RegEx` is ignoring the second instance of the multiplication for some reason. To see if it is a one time thing I ran the same `RegEx` expression again, as in I executed the `replaceAll()` functions again. For some reason this seemed to fix it as I got the expected output of $3 * x * (0 - x + 1) * (x + 2)$. To account for this, within my `standardize()` function, I will add a loop which make the `RegEx` checks are run until my expression has not changed between two consecutive `RegEx`. Here is the code[3] that I modified:

```
1  // loop until check and expression are equal
2  String check = "";
3  while(!expression.equals(check)) {
4          check = expression;
5          // perform regex
6          expression = expression.replaceAll(regex1, replace1);
7          expression = expression.replaceAll(regex2, replace2);
8          expression = expression.replaceAll(regex3, replace3);
9          expression = expression.replaceAll(regex4, replace4);
10         expression = expression.replaceAll(regex5, replace5);
11 }
```

While making this modification I originally checked if the previous and current versions of the expressions by doing `check!=expression`. This caused an infinite loop. This is because the `String` object in `Java` is quite unique in the fact that they act like primitives when assigning values (as shown in line 4 in the code above), but act like references when checking for equivalency[23]. In the example before, we were checking if they were the same reference i.e. the same object, not if they have the same contents, therefore it always evaluated as true (never same object hence true). Hence `!expression.equals(check)` is more suitable.

I tested this new function with the same arguments as before and here are the results:

```
2*x
(x)*3
)*(
*(0-x)
3*x*(0-x+1)*(x+2)
```

These are the results I expected.

[3]The full code is at the end of the section

7.3.2 Check Brackets

Like the `standardize()` function, this method is related to the class but not to a specific object hence it will be `static`. Here is the code:

```
// check for and remove enclosing matching brackets
private static String checkBracket(String input) throws UnequalBracketsException,
↪  StringIndexOutOfBoundsException {
        boolean done = false;
        // check if there are an equal number of opening and closing brackets
        if((input.length() - input.replace("(", "").length())
                        !=(input.length() - input.replace(")", "").length())) {
                throw new UnequalBracketsException(input);
        }
        while(!done) {        // repeat until there are no enclosing brackets
                done = true;// assume no enclosing matching brackets until otherwise found
                // check if there are enclosing brackets (not necessarily matching)
                if((input.charAt(0) == '(') && (input.charAt(input.length()-1) == ')')) {
                        int countMatching = 1;        // initialize the bracket count as 1 to
                        ↪  count for the opening bracket
                        // loop from the second character to one before the end
                        for(int i=1; i<input.length() - 1; i++) {
                                if (countMatching == 0) {        // if we find the matching
                                ↪  bracket before the end terminate and return the input
                                        return input;
                                } else if(input.charAt(i) == '(') {
                                        // increment if we find an opening bracket
                                        countMatching++;
                                } else if(input.charAt(i) == ')') {
                                        // decrement if we find an closing bracket
                                        countMatching--;
                                }
                        }
                        // if the last character is the matching bracket keep the outer while
                        ↪  loop going and and remove the enclosing bracket
                        if(countMatching == 1) {
                                input = input.substring(1, input.length() - 1);
                                done = false;
                        }
                }
        }
        return input;
}
```

While this algorithm is explained in the design section, the confusing part of this is the checking if there are an equal number of opening and closing brackets. There is no builtin method for this, but we can use builtin methods to achieve the same result. To do this[24] we replace all of the character that we want with empty characters and then take this away from the length of the original String.[4]

[4]Let l be the original length of the String, x the number of instances of the character and c the length of the reduced String:

$$l = x + c \implies x = l - c$$

To test this method I used the following code with comments for what I expected:

```
System.out.println(checkBracket("2x"));        // 2x - nothing changes
System.out.println(checkBracket("((x))"));        // x - nothing changes
System.out.println(checkBracket("(x-1)(x+1)"));        // (x-1)(x+1) nothing - changes
System.out.println(checkBracket("((x-1)(x+1))"));        // (x-1)(x+1)
System.out.println(checkBracket("((x)"));        // throws an exception
```

Here are the results:

```
2x
x
(x-1)(x+1)
(x-1)(x+1)
Exception in thread "main" exceptions.UnequalBracketsException: There are an unequal number of
↪  opening and closing brackets in the expression: (()
        at structures.Expression.checkBracket(Expression.java:133)
        at structures.Expression.main(Expression.java:274)
```

Which is exactly what I expected.[5]

[5]Like with the `Stack` class I created a custom exception for this.

7.3.3 Least Significant Operator

Again this method is not linked an object, hence it will be static. The code is shown below:

```
// find the least significant operator in an expression
private static int leastSigOperatorPos(String input) {
        int parenthesis = 0;
        int leastSigOperatorPos = -1;
        int leastSigOpcode = 1000;        // make the opcode super high so every operator is more significant than it
        // used a string to store the operators as it is essentially a char array but with added utility such as finding elements
        final String operators = "+-*/^";
        int currentOpcode;
        // loop through the entire input
        for(int i=0; i<input.length(); i++) {
                char currentChar = input.charAt(i);
                currentOpcode = operators.indexOf(currentChar);
                // check if it is an operator
                if(currentOpcode>=0) {
                        // check if it the same or more significant and if it is not enclosed in parenthesis
                        if((currentOpcode <= leastSigOpcode) && (parenthesis == 0)) {
                                // update the significance and the position of the operator
                                leastSigOperatorPos = i;
                                leastSigOpcode = currentOpcode;
                        }
                } else if(currentChar == '(') {
                        // increment to signify we have entered an enclosing bracket
                        parenthesis++;
                } else if(currentChar == ')') {
                        // decrement to signify we have exited an enclosing bracket
                        parenthesis--;
                }
        }
        return leastSigOperatorPos;
}
```

To test this method I used the following code with comments for what I expected:

```
System.out.println(leastSigOperatorPos("3+x^2"));        // should return 1 for the +
System.out.println(leastSigOperatorPos("(x+3)"));        // should return -1 since it is enclosed in brackets
System.out.println(leastSigOperatorPos("x*(x+3)^3"));        // should return 1 for the *
```

Here are the results:

```
1
-1
1
```

Which is exactly what I expected.

7.3.4 Create the Abstract Syntax Tree

I first implemented the normal sequential function and later I will implement the multi-threaded approach. Again like the methods before it this is a static function, since it relates to the class not the object itself.

```
// create the abstract syntax tree for the expression - single-threaded
private static BinaryTree createTree(String expression) throws UnequalBracketsException {
        // remove enclosing matching brackets
        expression = checkBracket(expression);
        // find the least significant operator, if an operator remains
        int leastSigOperatorPos = leastSigOperatorPos(expression);
        if (leastSigOperatorPos == -1) { // base case - no operators remain
                return new BinaryTree(expression);
        } else { // recursive case - recurse on the sub-expressions
                // locate and hold the operator
                String operator = String.valueOf(expression.charAt(leastSigOperatorPos));
                // split the expression into sub-expressions by the operator and recurse on
                ↪  them
                String a = expression.substring(0, leastSigOperatorPos);
                String b = expression.substring(leastSigOperatorPos + 1);
                // return the new tree containing the trees of the sub-expressions and the
                // operator as the root value
                return new BinaryTree(operator, createTree(a), createTree(b));
        }
}
```

The problem with creating a tree is that I can't actually display it in a way to check if my algorithm works. To get around this issue I can just use my traversal algorithm. I have tested my traversal algorithm and know that it works hence I can just output the stack from traversing it. I will standardize my expressions before I create the trees since this is what will happen in the constructor later.

```
//  create the standardized expressions
String a = standardize("2^x");
String b = standardize("(x+1)(x-1)");
// create the trees
BinaryTree treeA = createTree(a);
BinaryTree treeB = createTree(b);
// show the conversion between the expressions and trees
// the traverse function uses a stack so remember the output will be reversed
// [2, x, ^] but it will be reversed therefore [^, x, 2]
System.out.println(a + " -> " + treeA.traverse());
// [x, 1, +, x, 1, -, *] but it will be reversed therefore [*, -, 1, x, +, 1, x]
System.out.println(b + " -> " + treeB.traverse());
```

Here are the results:

```
2^x -> [^], x, 2
(x+1)*(x-1) -> [*], -, 1, x, +, 1, x
```

Which is exactly what I expected.

When I designed this algorithm I wrote about how I could implement it using multiple threads. While multi-threading can be very powerful, like everything in Computer Science there is a cost. This is especially true where in Java the cost of instantiating a thread[25] (on the fly at least) is incredibly expensive. In the article it talks about three main things:

1. Allocating Memory to the thread
2. Create the call stack for the thread[26, 28]
3. Initialize and link the thread to the host OS

This takes a lot of processing time, in this article[27] one user manages to spawn about 10000 threads a second, which means that it takes approximately 0.1 seconds to spawn every thread. This is to be taken with a pinch of salt however since the article is quite old (8 years in fact) and processors are better and the JVM should be more efficient nowadays than before. To test this for myself I ran their benchmark program. [6] I did change some of the parameters to make each test do the same amount of work (the same number of instructions) and made the number of threads spawned different as well. I also ran it 3 times so I could calculate a mean to able to identify anomalous data. This also alleviates some of the randomness of the schedulerI did this by compiling and executing the program 3 times, to reduce the effect of cached memory affecting the results. While the test does accommodate for this issue and allows for multiple tests, I think it is best to give the JVM no chance for optimization by just destroying and recreating the JVM. The results table and the corresponding graphs are on the next page.

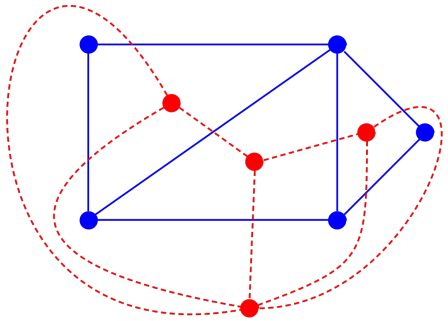
$$DO \cdot THE \cdot THREAD \cdot CREATION \cdot TEST$$

$$do \cdot the \cdot analysis$$

[6] The code for this is on the StackOverflow thread, made by a user called, at the time of writing this, "Jaan"[27]

Table 7.1: Thread Creation Test

Number of Threads	Time Taken/ms												Mean Total
	To Create				To Complete Work				To Join				
	1	2	3	Mean	1	2	3	Mean	1	2	3	Mean	
1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
5	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
20	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
100	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1000	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0



From the analysis above it seems that the effect of using a few threads seems minimal and the overhead may be outweighed by the performance gain from other parts of the function. However this may be specific to this scenario, so to test this properly I implemented the multi-threaded version which is below:

```
// create the abstract syntax tree for the expression - multi-threaded
private static BinaryTree createTreeThread(String expression)
                throws StringIndexOutOfBoundsException, UnequalBracketsException,
                ↪  InterruptedException, ExecutionException {
        // remove enclosing matching brackets
        expression = checkBracket(expression);
        // find the least significant operator, if an operator remains
        int leastSigOperatorPos = leastSigOperatorPos(expression);
        if (leastSigOperatorPos == -1) { // base case - no operators remain
                return new BinaryTree(expression);
        } else { // recursive case - recurse on the sub-expressions
                // locate and hold the operator
                String operator = String.valueOf(expression.charAt(leastSigOperatorPos));
                // split the expression into sub-expressions by the operator and recurse on
                ↪  them
                String a = expression.substring(0, leastSigOperatorPos);
                String b = expression.substring(leastSigOperatorPos + 1);
                BinaryTree tree0;
                BinaryTree tree1;
                // create the threads and execute them
                ExecutorService executor0 = Executors.newSingleThreadExecutor();
                Future<BinaryTree> future0 = executor0.submit(() -> {
                        return createTree(a);
                });
                ExecutorService executor1 = Executors.newSingleThreadExecutor();
                Future<BinaryTree> future1 = executor1.submit(() -> {
                        return createTree(b);
                });
                // return the tree values from the threads
                tree0 = future0.get();
                tree1 = future1.get();
                // shutdown the threads
                executor0.shutdown();
                executor1.shutdown();
                // return the new tree containing the trees of the sub-expressions and the
                // operator as the root value
                return new BinaryTree(operator, tree0, tree1);
        }
}
```

I then tried to test my 2 `createTree()` functions to identify if a purely single-threaded approach is more efficient than a multi-threaded approach. However the two approaches may change in effectiveness depending on the complexity of the expression we are parsing. In order to control this for a fair test we need to effectively measure the complexity hence we need to analyze the algorithm in detail.

Assuming that a single call of the `createTree()` algorithm takes the same time, t, to execute (exclude the recursive calls and the overhead that is need for them), then $\mathcal{O}(n) = (n+1)t$ where n is the number of operators. Assuming that the expression is valid, this value comes from the fact that we can split a standardized expression into by each operator. From the nature of our approach, there will be no operators on the start or the end of a valid expression, and each variable/constant is split apart by an operator[7]. Now from this property we notice that each operator has a uniquely positioned variable/constant on the right of it, apart from the first operator which has an extra variable/constant to the left of it. Now our function recurses k times where k is the number of base sub-expressions, i.e. variables/constants. However it can be hard to count these variables/constants since they can be of variable length (1.2 is a constant and has a length in terms of strings of 3, but x is also a variable but has a length of 1). On the other hand operators are incredibly easy to spot and count since there is a finite number of types of operators, and they are all of length 1. Hence it makes sense to define the complexity of our function in terms of the number of operators. This means that one of our independent variables, the one we will change, will be the number of operators. We will also change whether we use threads or not since this is the point of our experiment.

Table 7.2: Single- vs Multi-Threaded Approach Test

Number of Operators	Time Taken/ms							
	With Threads				Without Threads			
	1	2	3	Mean	1	2	3	Mean
1	nsert	nsert	nsert	nsert	nsert	nsert	nsert	nsert
5	nsert	nsert	nsert	nsert	nsert	nsert	nsert	nsert
20	nsert	nsert	nsert	nsert	nsert	nsert	nsert	nsert
100	nsert	nsert	nsert	nsert	nsert	nsert	nsert	nsert

[7]This is an inherent property within any valid expression since otherwise without this we cannot separate each variable/constant hence we cannot define explicitly where each variable/constant starts and ends. Within human expressions this is more implicit but still exists. For example $(x+1)(y-1)$ appears not to have an operator between 1 and y, but recall from the design section that this expression actual means $(x+1)*(y-1)$ hence the property is still valid. We have only applied this to operators and becomes blurred when we talk about functions within our expression, such as sin, but this does not matter since our algorithm does not deal with this.

7.3.5 Constructor

Now that the `createTree()` function is implemented, the constructor can finally be implemented. I will create two con

7.3.6 Substitute

7.3.7 Evaluate

7.4 Functions

7.5 Layers

Bibliography

[1] Desmos
https://www.desmos.com/

[2] GeoGebra
https://www.geogebra.org/

[3] GNU Octave
https://octave.org/doc/interpreter/

[4] MATLAB - MATrix LABatory
https://uk.mathworks.com/help/matlab/

[5] OpenGL
https://www.opengl.org/documentation/

[6] RegEx - Regular Expressions
https://en.wikipedia.org/wiki/Regular_expression

[7] Lua Example
http://www.luaj.org/luaj/3.0/README.html

[8] Stack Based Programming
https://en.wikipedia.org/wiki/Stack-oriented_programming

[9] JavaFX
https://en.wikipedia.org/wiki/JavaFX#JavaFX_11

[10] JavaFX example without FXML
https://en.wikipedia.org/wiki/JavaFX#JavaFX_application_example

[11] Creating a GUI using FXML
https://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm

[12] Shared Access between two or more Controllers
https://stackoverflow.com/questions/29639881/javafx-how-to-use-a-method-in-a-controller-from-another-controller

[13] Property Class from JavaFX explained
https://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm

[14] JavaFX interface Hierarchy
http://www.ntu.edu.sg/home/ehchua/programming/java/Javafx1_intro.html

[15] Border Pane Layout Explained
http://dammsebastian.blogspot.com/2012/03/javafx-20-layot-panes-borderpane.html

[16] Canvas Example
https://docs.oracle.com/javafx/2/canvas/jfxpub-canvas.htm

[17] Multithreading in OpenGL
https://stackoverflow.com/questions/11097170/multithreaded-rendering-on-opengl

[18] Generic Programming
https://en.wikipedia.org/wiki/Generic_programming

[19] Generics within Java
https://docs.oracle.com/javase/tutorial/extra/generics/index.html

[20] Duplicating objects in Java
https://stackoverflow.com/questions/12072727/duplicating-objects-in-java

[21] Serialization in Java
http://javatechniques.com/blog/faster-deep-copies-of-java-objects/

[22] Passing by Reference in Java
https://stackoverflow.com/questions/40480/is-java-pass-by-reference-or-pass-by-value

[23] Checking String Equivalency in Java
https://stackoverflow.com/questions/513832/how-do-i-compare-strings-in-java

[24] Counting instances of a character within a String in Java
https://stackoverflow.com/questions/275944/how-do-i-count-the-number-of-occurrences-of-a-char-in-a-string

[25] Costs of instantiation of threads in Java
https://stackoverflow.com/questions/5483047/why-is-creating-a-thread-said-to-be-expensive

[26] The Thread Stack in Java
https://stackoverflow.com/questions/36898701/how-does-java-jvm-allocate-stack-for-each-thread

[27] Thread Creation Rate in Java
The article - https://stackoverflow.com/questions/2117072/java-thread-creation-overhead
The benchmark I used - https://stackoverflow.com/a/4371508

[28] The Call Stack
https://en.wikipedia.org/wiki/Call_stack https://www.youtube.com/watch?v=Q2sFmqvpBe0