Graphing - Programming Project

Shashi Balla

October 14, 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.

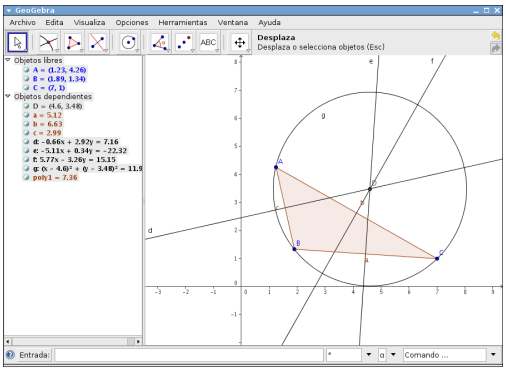


Figure 1.1: GeoGebra

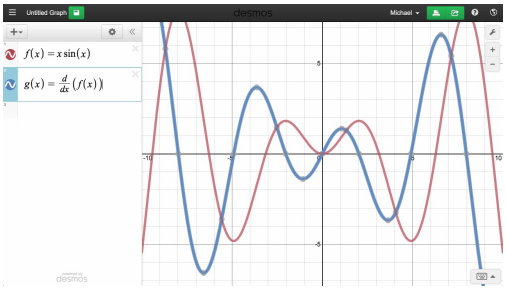


Figure 1.2: Desmos

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
2. GeoGebra
3. GNU Octave (MATLAB 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.

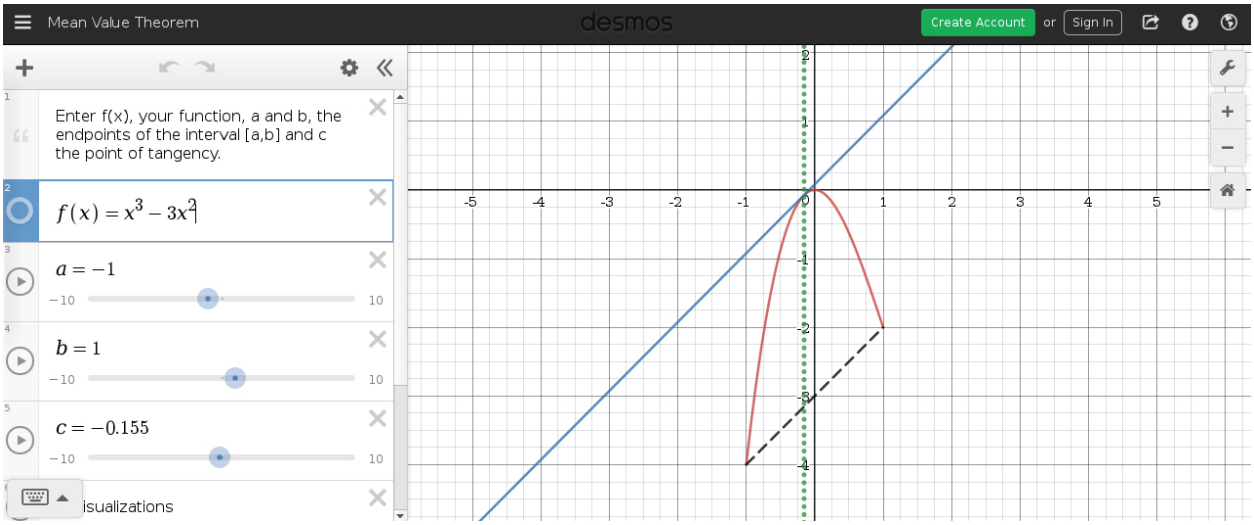


Figure 3.1: Mean Value Theorem example that students can interact with

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.

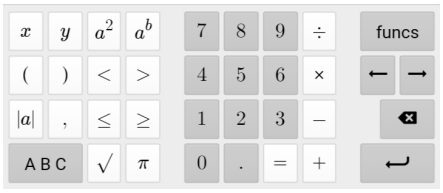


Figure 3.2: The Input

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.

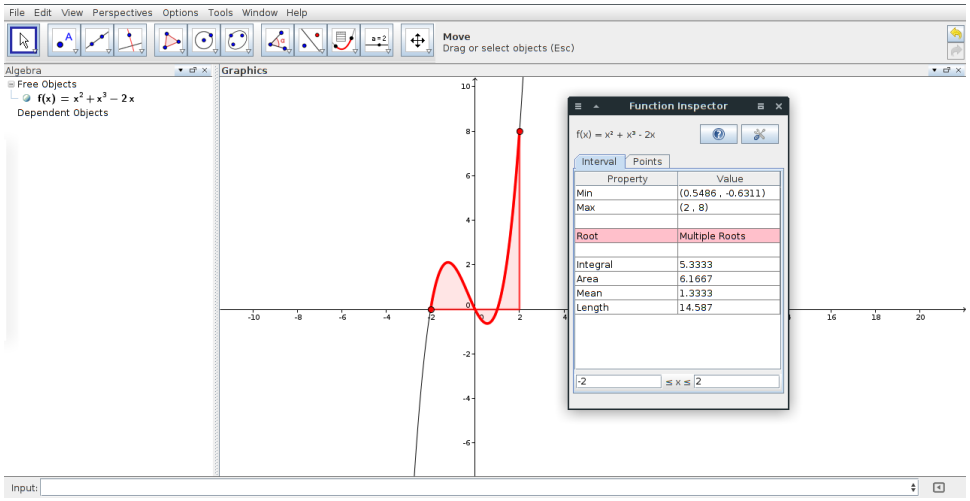


Figure 3.3: GeoGebra Function Inspector Tool

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]

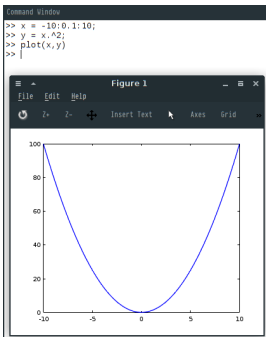


Figure 3.4: Creating a simple plot in GNU Octave

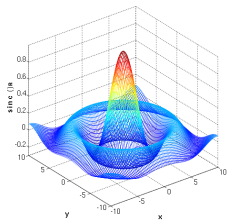


Figure 3.5: Wireframe 3D plot of the two-dimensional unnormalized sinc function

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.

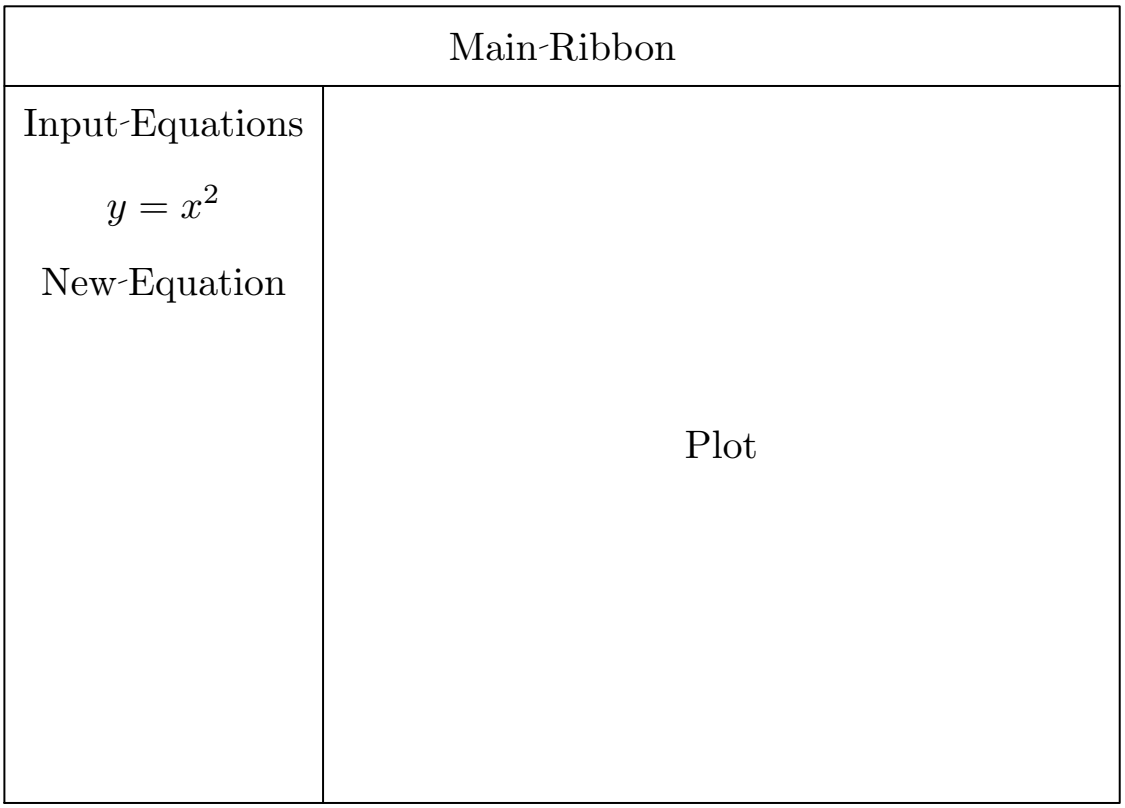


Figure 3.6: UI Design

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.

I can decompose these 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 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. 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 1 - Emphasis on Core Functionality
  - Plot a explicit function in x
  - Plot multiple functions
  - Zoom in/out of the graph
  - Pan around the graph
- Iteration 2 - Emphasis on the User
  - Identify roots and turning points
  - Plot special functions such as trigonometric functions, logarithmics, modulus functions, etc.
  - Multiple plots that you can switch between
  - Save plots as pictures
  - Save workspace to resume later
  - LaTeX equation support
  - Dark Theme
- Iteration 3 - Advanced Features
  - Identify intersections between functions
  - Differentiate explicit continuous functions with respect to x
  - Polar equations
  - Implicit equations
  - Parametric equations

I will use Java to accomplish this task. Java 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, 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 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 ways to parse an mathematical input and convert it into a structure that we can then manipulate and use. These are:

- Binary Trees
- Stack Based Programming

As a side note, there are some nuances when we write functions that make it difficult for a computer to process and we have to remember when we implement this. For example, if we have $3x$, we actually mean $3 * x$. Similarly if we have $(x-2)(2-x)$ we actually mean $(x-2)*(2-x)$. It is 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 $(x+4)*(x-3)$

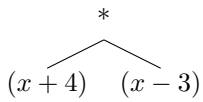
[4] If there is a negate symbol next to another operator, we need to make sure that the negate symbol is not treated as an operator (even though we treat it like an operator in certain situations in the next step)

[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 $0-x$.

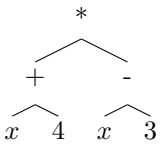
6.1.1 Binary Trees

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.

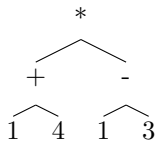
Algorithm 1: Evaluate: Binary Tree Version

```
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

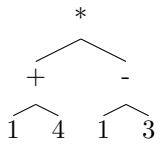
First let us replace every instance of x with 1.



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

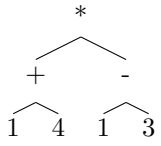
and again the root node is an operator so we take the left subtree again, which is 1. This is not an operator, so we return this value, going back to our previous call which is,

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.

6.1.2 Stack Based Programming

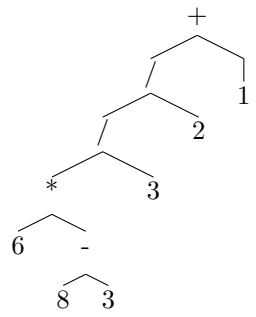
Stack Based Programming is all about using a stack data structure, and manipulating the items within it to get the desired result. Firstly we have to convert our *Infix Notation* into *Reverse Polish Notation*. Infix notation is the "normal" way of writing algebra, where the operator is inbetween its 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,

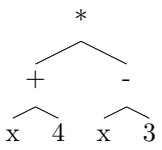
Algorithm 2: Evaluate: Stack Based Version

```
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.

6.1.3 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.

6.1.4 Implementing Stacks and Trees

Since stacks and queues are a prerequistite to our function class, we will design, implement and test 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 list as parameter. This list which is passed will be edited and returned in every recursive call. Also we will 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.

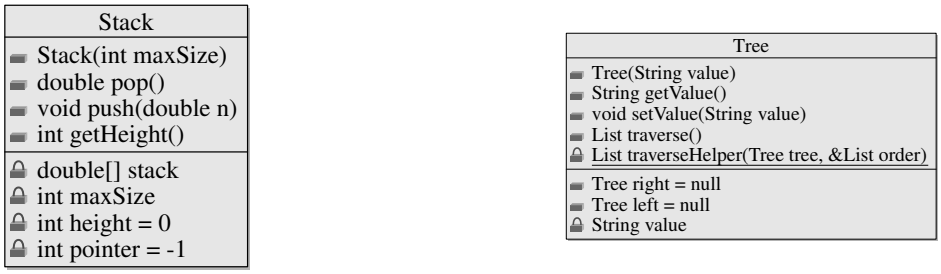


Figure 6.1: Class Diagrams

Algorithm 3: Post-Order Depth-First Traversal Helper

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

Algorithm 4: Post-Order Depth-First Traversal

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

6.1.5 Parsing Algorithm

6.1.5.1 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.

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
```

6.1.5.2 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 happens at the end the opening and closing brackets are removed, else nothing happens. This algorithm will

[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

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      while !done do
4          done = True
5          if input[0] == '(' and input[input.size - 1] == ')' then
6              Integer countMatching = 1
7              for i=1 to (input.size - 2) by 1 do
8                  if countMatching == 0 then /* if countMatching is 0, then the matching
                       closing bracket has been found before the end therefore we return the
                       input without modification                                        */
9                      return input
10                 else if input[i] == '(' then
11                     countMatching++/* If there is an opening bracket then we increment
                           */
12                 else if input[i] == ')' then
13                     countMatching–
14                     If there is an closing bracket then we decrement
15                 end
16             end
17             if countMatching == 1 then /* we haven't looped til the last character which
                   is ')' , and therefore at this poInteger countMatching would be 1 for a
                   standard expression                                                  */
18                 done = False
19                 input = input.subString(1, input.size - 2)
20             else
                   /* if not it is not an accepted expression, unequal brackets     */
21                 throw "There is an unequal number of opening and closing brackets"
22             end
23     end
24     return input
25 end
```

6.1.5.3 RegEx

Another important part of the parsing stage is to standardize the input. This is where we convert any inconsistencies discussed in section 6.1, page 14. The easiest way to do this is to use RegEx. 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 **"([^\+\-*\/\(\)\^])([\(x])"** with the replacement expression being **"$1*$2"**. If we take the ReGex expression, it creates two capture groups, *"$1"* and *"$2"*, which are *"([^\+\-*\/\(\)\^])"* and *"([\(x])"* 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 $\wedge$. The second capture group checks if the second character, in the substring that is currently being checked, is an x or a (. 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 **"([\)x])([^\+\-*\/\(\)\^])"** 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 The expression returns a match if it finds a) folllowed 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 $\wedge$, or[9] 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- preceeded 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] the or keyword in RegEx is signified by |

6.1.5.4 Creating the Tree

Using the functions above we can create a syntax tree from our expression. We will do this using 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.

Now the left side has the operator $\wedge$ so we can create another tree for this sub-expression.

Now the left and right nodes of this tree have no operators so we create a tree of consisting of these variables and constants as the root node with no child nodes.

Algorithm 8: Create a Binary Tree for an Algebraic Expression (Without Multiple Threads)

```
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
```

Bibliography

[1] insert something