Graphing - Programming Project

Shashi Balla

September 20, 2018

Contents

Part I

Analysis

Part II

Prototype 1

Chapter 1

Design

1.1 Decomposition

We can break this project into 2 major portions which can be handled separately until the very end:

- The Input - This includes parsing the user's inputted functions and storing it in a format which we can then use for the second portion.
- The Output - This is simply drawing the inputted functions onto the screen using the OpenGL API and adding some other bits such as intersections axes etc.

There is also the User Interface, but this is quite small compared to these two portions.

1.1.1 The Input

The Input consists of the Functions that the user inputs. We can create a standardized class called "function" which will allow us to deal with it independently. Our output will only require us to input a value, x, and return a value, y, and we can include a function within this class which allows to do this. There are two ways we can parse our input:

- 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, and then use the scripting language to evaluate a value x.

```
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)"); //here we are square rooting x and returning it,
8  //but for the general case, we would look for all special cases and
9  //convert it into something lua can interpret
10 //e.g. ``x^2'' becomes ``math.pow(x,2)''
11   return e.get("y");
12 }
```

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. We will be making upto 2000 evaluate calls during one render, 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.

1.1.1.1 Function Class

Our 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 $(x+4)*(x-3)$

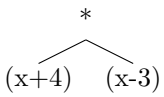
[4] If there is a negate symbol next to another operator, we need to make sure that the negate symbol is not treated as an operator (even though we treat it like an operator in certain situations in the next step)

[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 $0-x$.

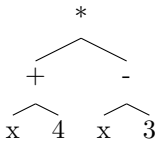
1.1.1.1.1 Binary Trees

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.

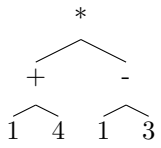
Algorithm 1: Evaluate: Binary Tree Version

```
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

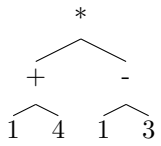
First let us replace every instance of x with 1.



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

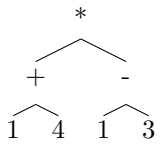
and again the root node is an operator so we take the left subtree again, which is 1. This is not an operator, so we return this value, going back to our previous call which is,

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.

1.1.1.1.2 Stack Based Programming

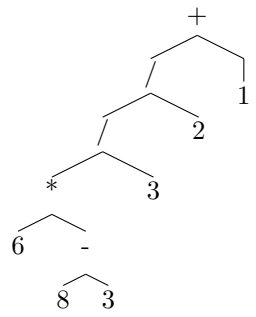
Stack Based Programming is all about using a stack data structure, and manipulating the items within it to get the desired result. Firstly we have to convert our *Infix Notation* into *Reverse Polish Notation*. Infix notation is the "normal" way of writing algebra, where the operator is inbetween its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

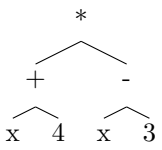
Algorithm 2: Evaluate: Stack Based Version

```
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.

1.1.1.1.3 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.

1.1.1.1.4 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.

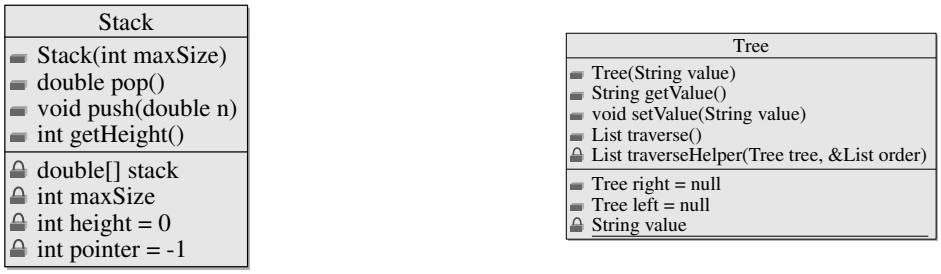


Figure 1.1: Class Diagrams

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
```

1.1.1.1.5 Parsing Algorithm

An important part of the parsing stage, is the ability to find the least significant operator within an expression.

Algorithm 5: Least Significant Operator Position

```
1  function leastSigOperatorPos(String input):
2      int parenthesis = 0
3      int leastSigOperatorPos = -1 /* stores the position of the least significant operator
           so far                                                                          */
4      int leastSigOpcode = 1000
5      char[] operators = ["+",' "-", "/", " * ", " ^"] /* stores each operator in order of
           incresing significance                                                          */
6      int currentOpcode /* the current index of the operator in the array operators     */
7      char 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
```

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.

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
```

[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

However the above algorithm 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 our current algorithm 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 following algorithm is more suited. 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 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              int 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 point countMatching would be 1 for a
                   standard expression                                                  */
18                 done = False
19                 input = input.subString(1, input.size - 2)
20             (/* otherwise it is not a standard expression and there is something wrong
                   */) throw "There is an unequal number of opening and closing brackets"
21         else
22         end
23     end
24     return input
25 end
```

Another important part of the parsing stage is to standardize the input. This is where we convert any inconsistencies discussed in section 1.1.1.1, page 5. 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 is signified by |

Bibliography

[1] insert something