Finding The Differential Of Mathematical Expressions

Joseph Rogers

Contents

1. Analysis

Identification of problem

The problem being identified is differentiating mathematical expressions. When a user wishes to differentiate mathematical expressions there should be a simple, fast, and convenient method for a user to receive an answer for the given expression. Even when expressions become complicated the program should be able to quickly return a mathematically correct differential. A program should be written which cannot only find the derivative of expressions containing numerical values and variables but also functions. It should be a portable program, which could be implemented on mobile devices as well as personal computers.

Identification of Users

A program like this would be very useful to students. For students there are calculators with the ability to differentiate, however can't differentiate functions such as 'sin', 'cos', or 'tan'. One of my clients is a school friend of mine, Tal who studies mathematics at A-level. Being a student myself I had access to many students who study mathematics to receive opinions on what the program should entail.

Existing solutions

There are a few existing solutions already. For example modern scientific calculators are able to differentiate mathematical expressions, however the calculators allowed at schools are unable to differentiate functions. This can be a problem because as expressions become more complicated they often contain functions. Although a program like this will not be allowed in class or examinations, it can still be very beneficial to students doing homework or revision. So for this reason the project should also have the ability to differentiate functions.

Other solutions such as wolfram alpha are available online. These are very clear and easy to use they can also differentiate functions, however they need an Internet connection. By writing this project using a high level language such as

python it should be available on mobiles as well as computers and should be able to run without the use of an Internet connection.

Contact with clients

As a student I had access to many students who study mathematics at A-level. This meant I was able to ask many different people for opinions on a program that differentiates mathematical expressions. One of my main clients is Tal who studies maths.

- He asked for a fast and easy process for differentiating an expression.
- It should have a straightforward user interface.
- It should be able to differentiate functions.
- It would be convenient to have a program, which could be put on a mobile device so that it would be convenient for students to use.
- It should be able to store valid inputs.
- It should return a legible output of the simplified differential.

Requirements specification

FR 1.1 Users can type 'variables' which can conform to regular expression [a-z, A-Z]*.
FR 1.2 Users can type any of the following operators '+', '-', '/', '*', '^' (plus, minus, divide, multiply, and power).
FR 1.2.1 operator '-' can be unary or binary whereas operators '/', '^', '+' and '*' must be binary.
FR 1.3 Users can type any of the following functions 'sin', 'cos', 'tan'.
FR 1.3.1 Each function listed in 1.3 are unary functions so that the argument should be contained within brackets (e.g. sin(x)).
FR 1.4 Students can type any of the following integers '0', '1', '2' ,'3' ,'4', '5', '6', '7', '8', '9' and can type decimal values to any number of decimal places.
FR 1.5 Users can type any of the following brackets '(', ')'
FR 1.6 Users can type any sequence of characters whether or not it forms a syntactically correct expression.
FR 1.7 The mathematical expression is analysed using a recursive routine, which contains recursive syntax rules.
NFR 1.8 The program should run fast and efficiently to return the answer to the given input quickly.
NFR 1.9 The program must be very simple for a typical A-level mathematics student to understand.
FR 2.0 The program must have a method of storing/saving recent inputs.
NFR 2.1 The program must have a simple user interface.

Identification of acceptable limits

The program must be very intuitive to use, it must allow the user to input their expression. It should be able to accept any input with correct syntax. It must be able to differentiate expressions. Once the expression is input the program should continue to run independently until an output is reached. The program must be able to convert the input string into a form that can be manipulated by the program. It will have to use a recursive data structure so that the program is able to deal with the infinite number of potential combinations, which could be input. The program will then have to be able to display this structure in a form that is legible by the user for its final output. If the syntax of the input is incorrect then the program should not fall over, instead it should retry the last process until the input is valid

Documented Design

Overall design

Menu

A separate file should be introduced which contains a basic user interface. In this file it should contain multiple menus, which take choices from the user. By keeping the user interface in a separate file to the calculations and datastrucure, any user interface can be implemented without affecting the functionality of the program.

Syntax

The Backus Noir form for the expected syntax looks like this:

Variable = { ID Head } { ID Tail }*
Operator = ' + ' | ' - ' | ' * ' | ' \ ' | ' ^ '
Integer = { Digit }+
Float = { Digit }+ ' . '{ Digit }+

Delimiter = ' (' | ') '
TrigFunction = 'sin' | 'cos' | 'tan'

Expression ::= <Expression> '+' <Term>
| <Expression> '-' <Term>
| <Term>

Term ::= <Term> '*' <Factor>
| <Term> '/' <Factor>
| <Factor>

Factor ::= <Factor> '^' <Primary>

| <Primary>

Primary ::= '-' <Primary>
| <Element>

Element ::= Variable
| Integer
| Float
| TrigFunction '(' <Expression> ')'

Infix

Firstly in order for the program to be able manipulate the input string it must be able to convert the input into a sequence of tokens. This means converting the string into an array and dividing up each element of the expression given. This function will also need to detect implied multiplication and where to insert multiplication symbols. It should also differentiate between unary and binary minuses and replace one with a different character so they can be dealt with differently. It will also have to place in a token for functions as a signal or to encapsulate the functions argument.

Postfix

Secondly the program will have to convert the infix form into a postfix form, which is also known as reverse polish notation. This is so that the program, without the use of brackets, can understand the order of operation of the expression. This could be done using a stack and pop method as it goes through each token in the array. In order to correctly convert from infix to postfix the program will have to able to determine which operators take precedence and know which order to put operators in the postfix when multiple operations occur. This can be determined using an algorithm such as Dijkstra's shunting yard algorithm to determine which order to append the operators to the postfix.

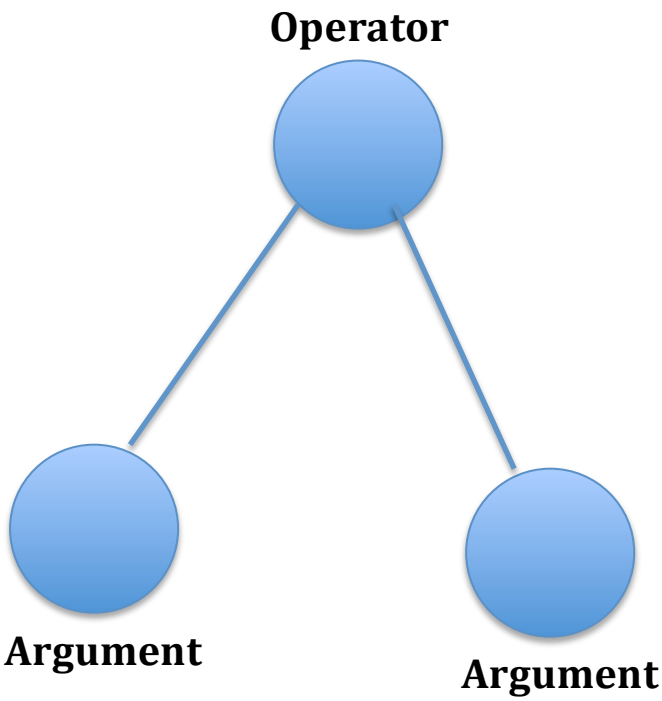
Converting '4(2*x^2 -3)/x' to postfix should look like the following example:

Current Symbol	Stack	Postfix(RPN)
'4'		'4'
'('	'('	'4'
'2'	'('	'4','2'
'*'	'(','*'	'4','2'
'x'	'(','*'	'4','2','x'
'^'	'(','*','^'	'4','2','x'
'2'	'(','*','^'	'4','2','x','2'
'-'	'(','-'	'4','2','x','2','^','*'

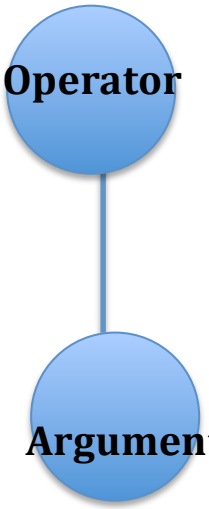
‘3’	‘(‘,’-‘	‘4’,’2’,’x’,’2’,’^’,’*’,’3’
‘)’		‘4’,’2’,’x’,’2’,’^’,’*’,’3’,’-‘
‘/’	‘/’	‘4’,’2’,’x’,’2’,’^’,’*’,’3’,’-‘
‘x’	‘/’	‘4’,’2’,’x’,’2’,’^’,’*’,’3’,’-‘,’x’
		‘4’,’2’,’x’,’2’,’^’,’*’,’3’,’-‘,’x’,’/’

Converting Postfix to a tree

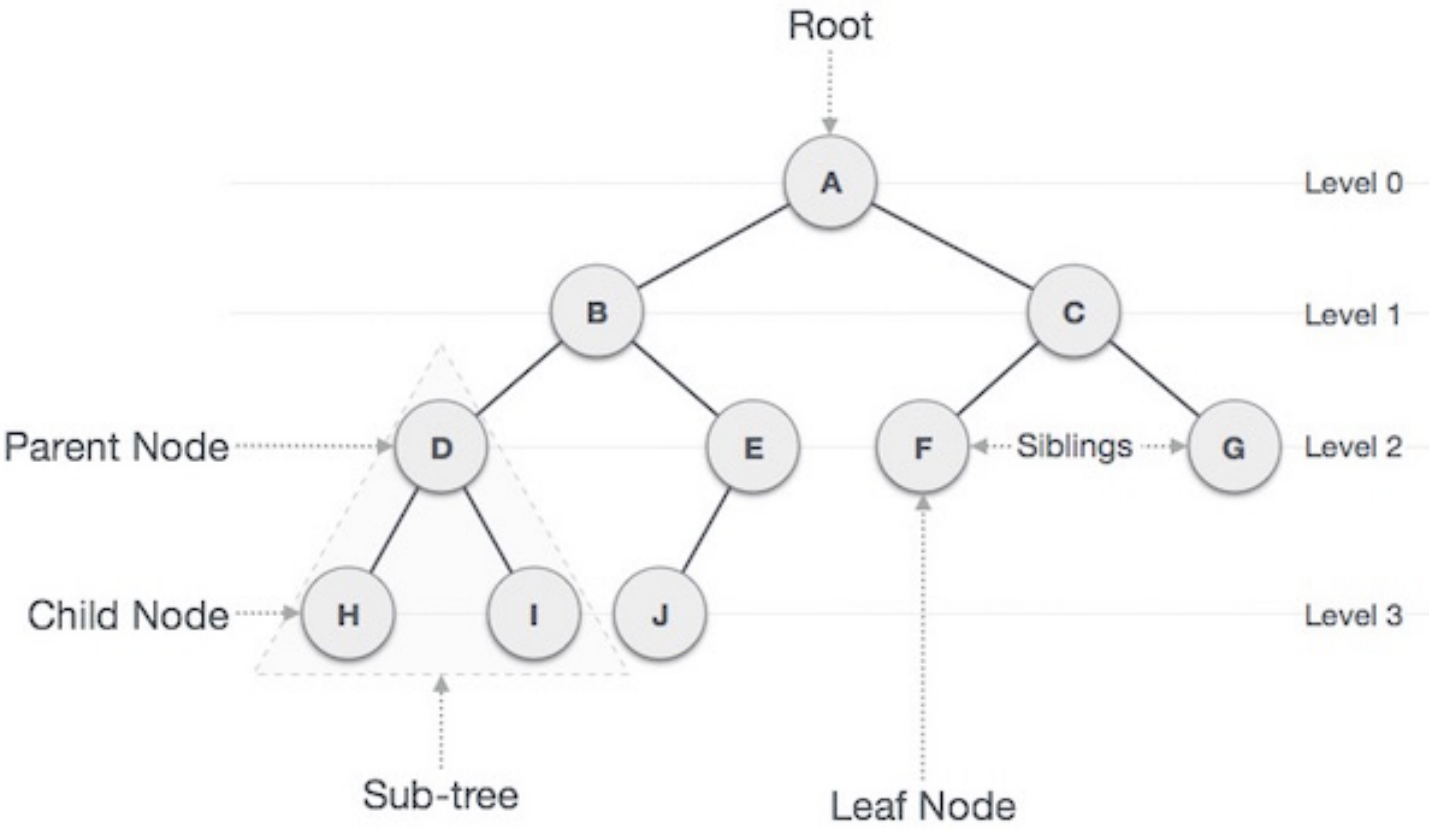
Thirdly the program will then have to be able to convert the postfix into a tree data structure. Creating nodes with children, meaning that the children of a node are the arguments for the node, can do this. A tree structure can be represented like so:



Certain operations are unary, such as the unary minus or function which only take one argument so they will only contain one child like so:

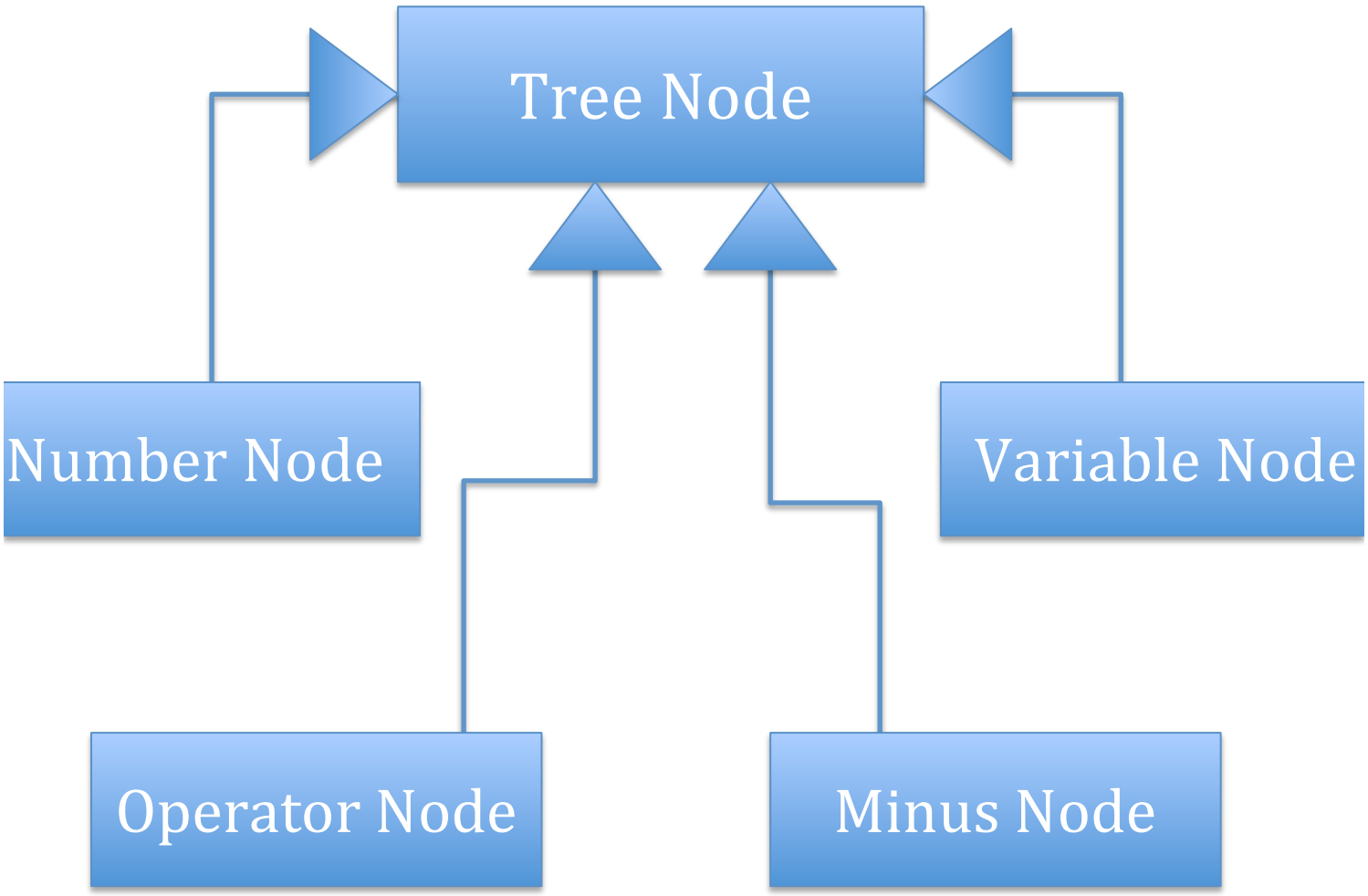


Because of the recursive nature of this structure, the size of the tree should increase with bigger expressions. More complicated inputs will create larger structures like this example I found online at https://www.tutorialspoint.com/data_structures_algorithms/images/binary_tree.jpg, which shows a clear representation of the relationship between eachof thenodes:



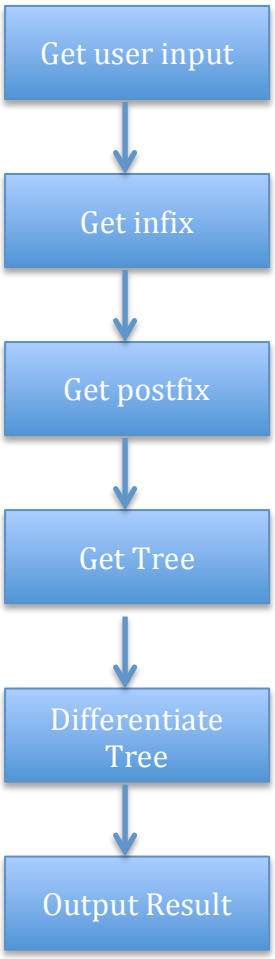
One of the solutions to creating this structure is by creating an object orientated program containing an abstract superclass of a tree node, of which all node types can then descend from. This is useful because all the node types will share similar methods such as adding a child, for methods, which will vary slightly such as differentiation, the routine can be overridden.

A class diagram to represent this structure would look like this:



The program would then have to be able to differentiate each node type correctly. If the node is a number node it should always differentiate to 0. If the node is a variable node, it should always differentiate to 1. For function nodes it should differentiate differently depending on the function. For operators they are differentiated when their children are differentiated, each operator should differentiate differently. For addition and subtraction the differential is the operator applied to its differentiated children. For multiplication if the children are not numerical then it should use the product rule; for division the quotient rule should be used; and for the power operator the chain rule should be used. The program should then have a function either in the maths expression or within each of the node types, which allow the output answer to be simplified so that it is easier to read. This means there needs to be a function which can determine which nodes are numerical, and therefore which nodes can be simplified. If a node is numerical then the operators can operate on their children to return a single numerical node.

The overall process for differentiating should roughly resemble this flow chart:



Feasibility Study and justification

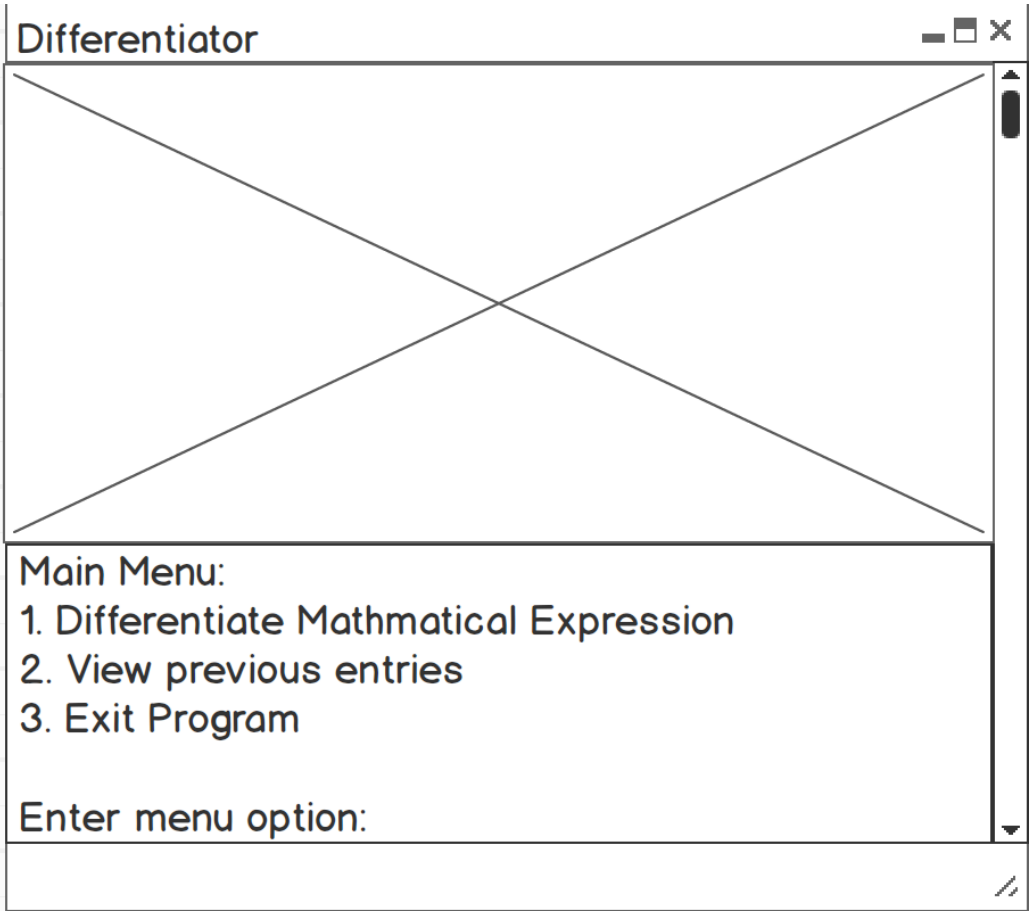
The project is most certainly feasible. Its purpose should be to differentiate mathematical expressions that are input. It should be aimed toward students or others who study mathematics that wish to differentiate expressions. This project will mainly focus on its functionality. However by separating the program into multiple files one for managing the calculations, and another just for calling those functions in the required order. It will be possible to add a third file for the user interface, which calls the functions over from the middle class, which will act as a controller between the user interface, and the file performing the calculations. The project should only take a single input from the user then run automatically for simplicity of use.

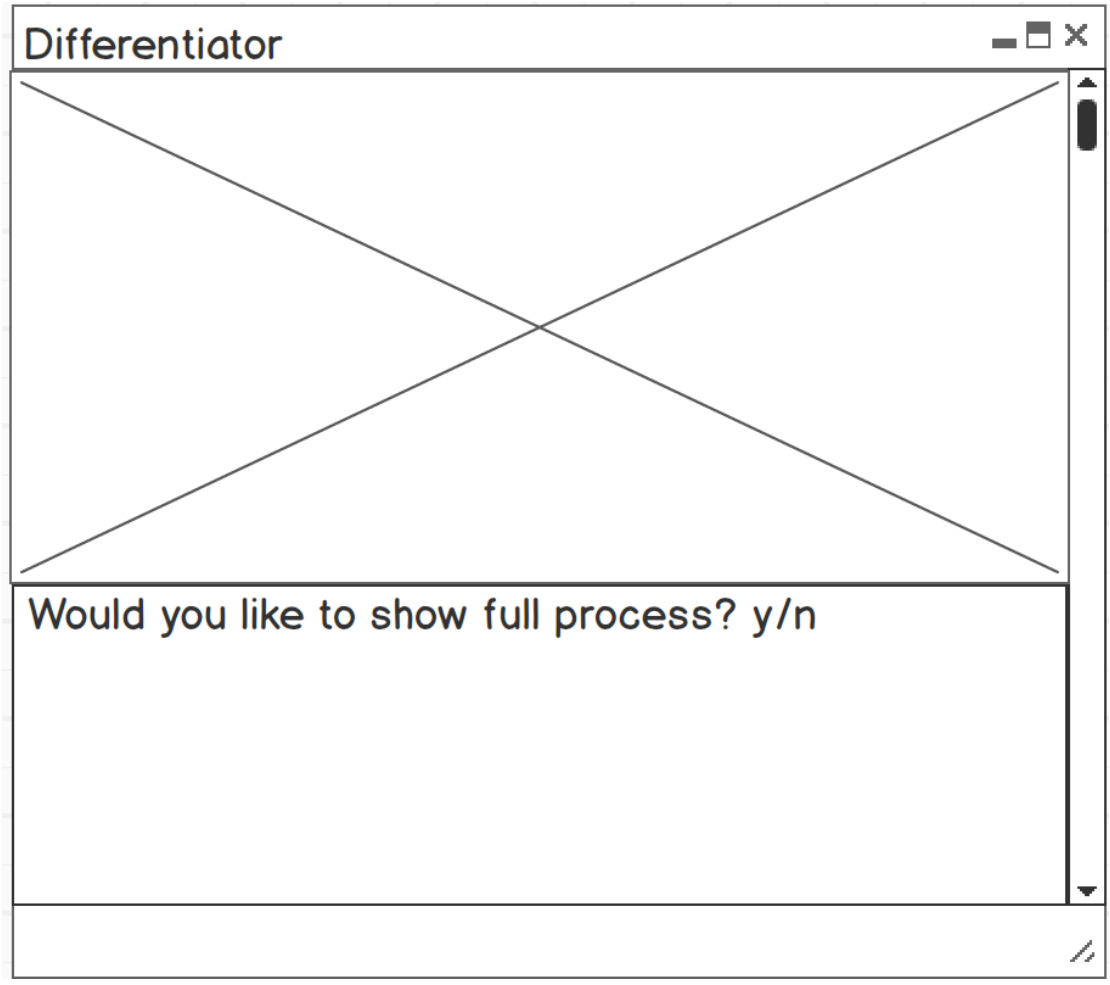
It terms of hardware requirements, it should not require expensive/modern hardware. It should not be a big task for a computer to run and by using this kind of tree data structure the program should run quickly. It should be easy to implement and by writing it in a high level language such as python, the program should be fairly portable. This means it could also potentially run on mobile devices as well as computers for ease of use.

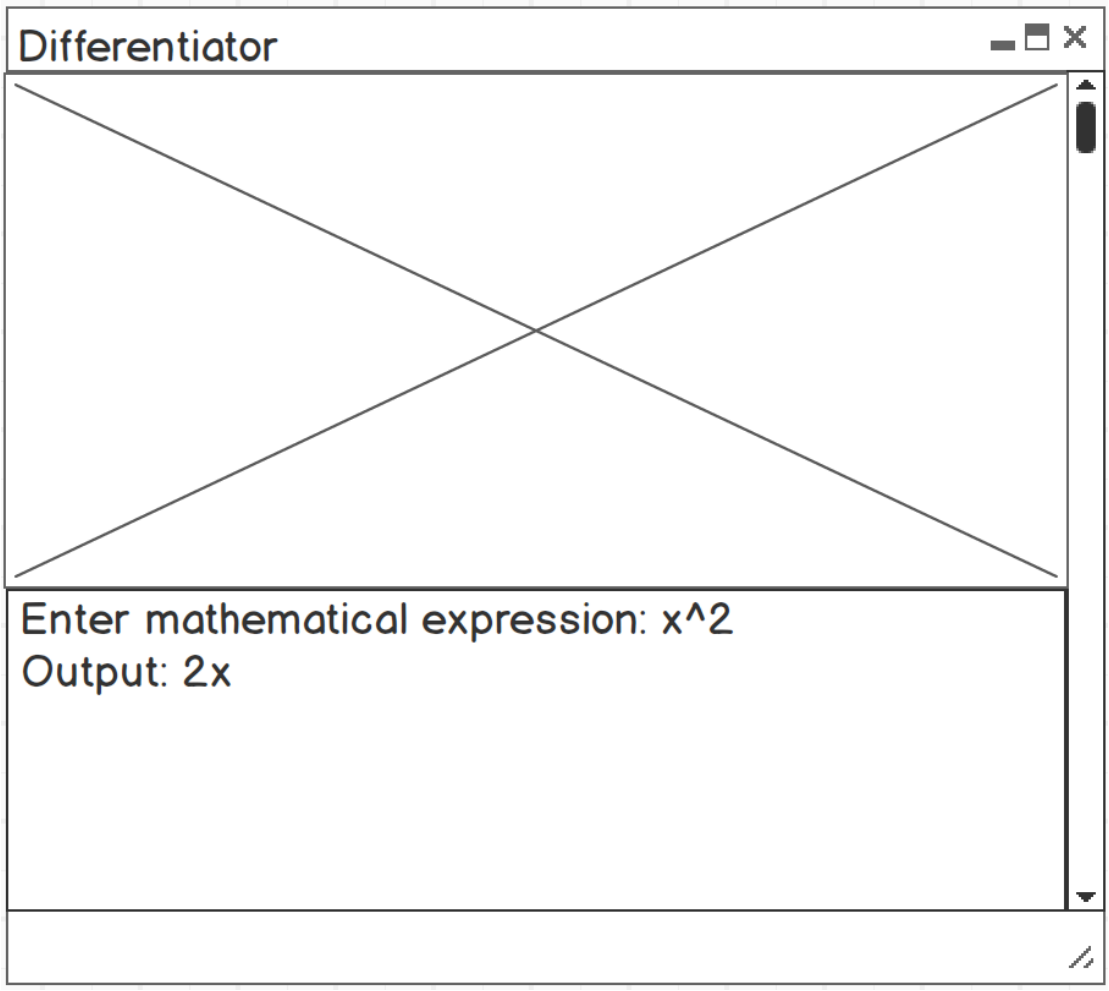
Mock ups

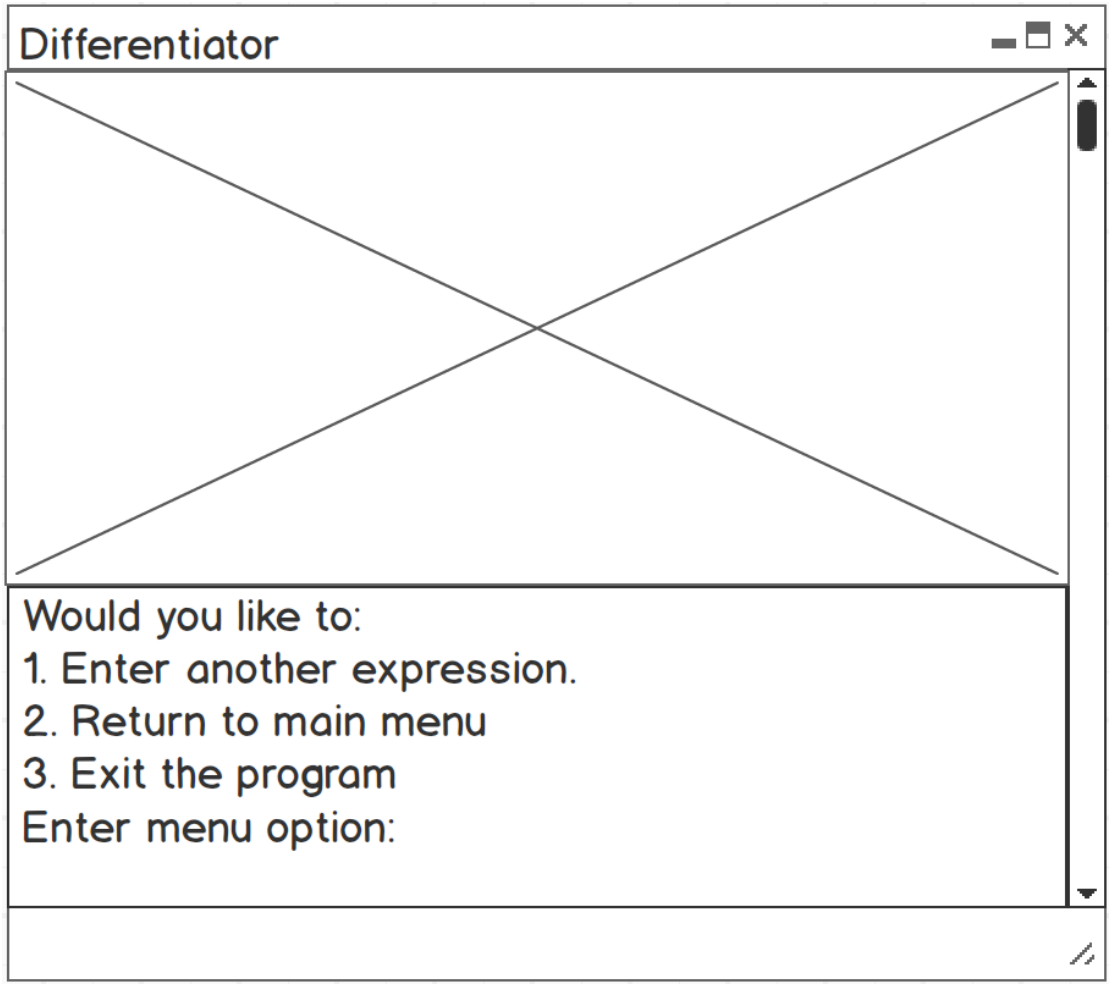
It should have a very simple design just using the console, which asks the user for the input. It should then proceed to show the user each stage and return the differentiated form of the expression underneath the input.

This project is more about functionality over the user interface, however it should be written in a way for a user interface to be easily implemented if wanted in the future.




Differentiator
Would you like to show full process? y/n


Differentiator
Enter mathematical expression: x^2
Output: 2x



Technical Solution

User Interface

```
import mathsExpression, treenodes, re, sys

class userInterface():

    def __init__(self):
        self.dictionary = {}
        self.menu()

    def differentialMenu(self):
        response = raw_input('\nwould you like to show full process? y/n: ')

        if response == 'y':

            inputString = mathsExpression.mathsExpression().getUserInputString()

            checkedString = mathsExpression.mathsExpression().impliedMultiplication(inputString)

            inputArray = mathsExpression.mathsExpression().getUserInputArray(checkedString)

            infix = mathsExpression.mathsExpression().getInfixAlgorithm(inputArray)

            print('Infix = ' + str(infix))
```

```
        postfix = mathsExpression.mathsExpression().postfixAlgorithm(infix)
        print('Postfix = ' + str(postfix))
        self.expressionTree = mathsExpression.mathsExpression().getTree(postfix)
        print('simplify the following tree : ' + self.expressionTree.treeToText() + ' to: ')

        self.simplifiedExpressionTree = self.expressionTree.simplifyTree()
        print('simplified input is : ' + self.simplifiedExpressionTree.treeToText())
        isNumerical = self.expressionTree.isNumericalNode()
        print('Is numerical is ' + str(isNumerical))
        print('Input value = ' + self.expressionTree.treeToText())
        self.derivativeTree = self.simplifiedExpressionTree.differentiate()
        print('Output value = ' + self.derivativeTree.treeToText())
        self.simplifiedTree = self.derivativeTree.simplifyTree()
        self.checkedDerivativeTree = self.simplifiedTree.binaryToNaryTree()
        self.finalDerivativeTree = self.checkedDerivativeTree.simplifyTree()
        print('Simplified output: ' + self.finalDerivativeTree.treeToText())
        outputString = self.finalDerivativeTree.treeToText()
        self.storeDictionary(inputString, outputString)
        self.finalMenu()
    elif response == 'n':
        inputString = mathsExpression.mathsExpression().getUserInputString()
        checkedString = mathsExpression.mathsExpression().impliedMultiplication(inputString)
        inputArray = mathsExpression.mathsExpression().getUserInputArray(checkedString)
        infix = mathsExpression.mathsExpression().getInfixAlgorithm(inputArray)
        postfix = mathsExpression.mathsExpression().postfixAlgorithm(infix)
        self.expressionTree = mathsExpression.mathsExpression().getTree(postfix)
        self.simplifiedExpressionTree = self.expressionTree.simplifyTree()
        isNumerical = self.expressionTree.isNumericalNode()
        self.derivativeTree = self.simplifiedExpressionTree.differentiate()
        self.simplifiedTree = self.derivativeTree.simplifyTree()
        self.checkedDerivativeTree = self.simplifiedTree.binaryToNaryTree()
        self.finalDerivativeTree = self.checkedDerivativeTree.simplifyTree()
        print('derivative : ' + self.finalDerivativeTree.treeToText())
        outputString = self.finalDerivativeTree.treeToText()
```

```
            self.storeDictionary(inputString, outputString)

            self.finalMenu()

        else:
            print('This is not a valid input')
            self.differentialMenu()

    def menu(self):

        print('\nMenu options')
        print('1. Differentiate expression')
        print('2. View previous entries')
        print('3. Exit program')

        option = raw_input('Enter menu option: ')

        if option == '1':
            self.differentialMenu()
        elif option == '2':
            self.printDictionary()
        elif option == '3':
            sys.exit()
        else:
            print('This is not a valid option')
            self.menu()

    def finalMenu(self):
        print('\nWould you like to: ')
        print('1. Enter another expression. ')
        print('2. Return to main menu ')
        print('3. Exit the program')

        option = raw_input('Enter menu option: ')

        if option == '1':
            self.differentialMenu()
        elif option == '2':
            self.menu()
        elif option == '3':
            sys.exit()

    def storeDictionary(self, inputString, outputString):
        self.dictionary.update({inputString:outputString})
        print('Expression Saved.')

    def printDictionary(self):
        print('\nRecent History: ')
        for expression in self.dictionary:
            print(expression + ' Differentiated to: ' + self.dictionary[expression])
        self.menu()


runProgram = userInterface()
```

The menu just lets the user differentiate an expression, view previous entries, or exit the program.

When differentiating an expression is chosen the user is given an option of whether or not they wish to see the whole process. If the user wishes to see the whole process then the program prints out the returned values at centain points

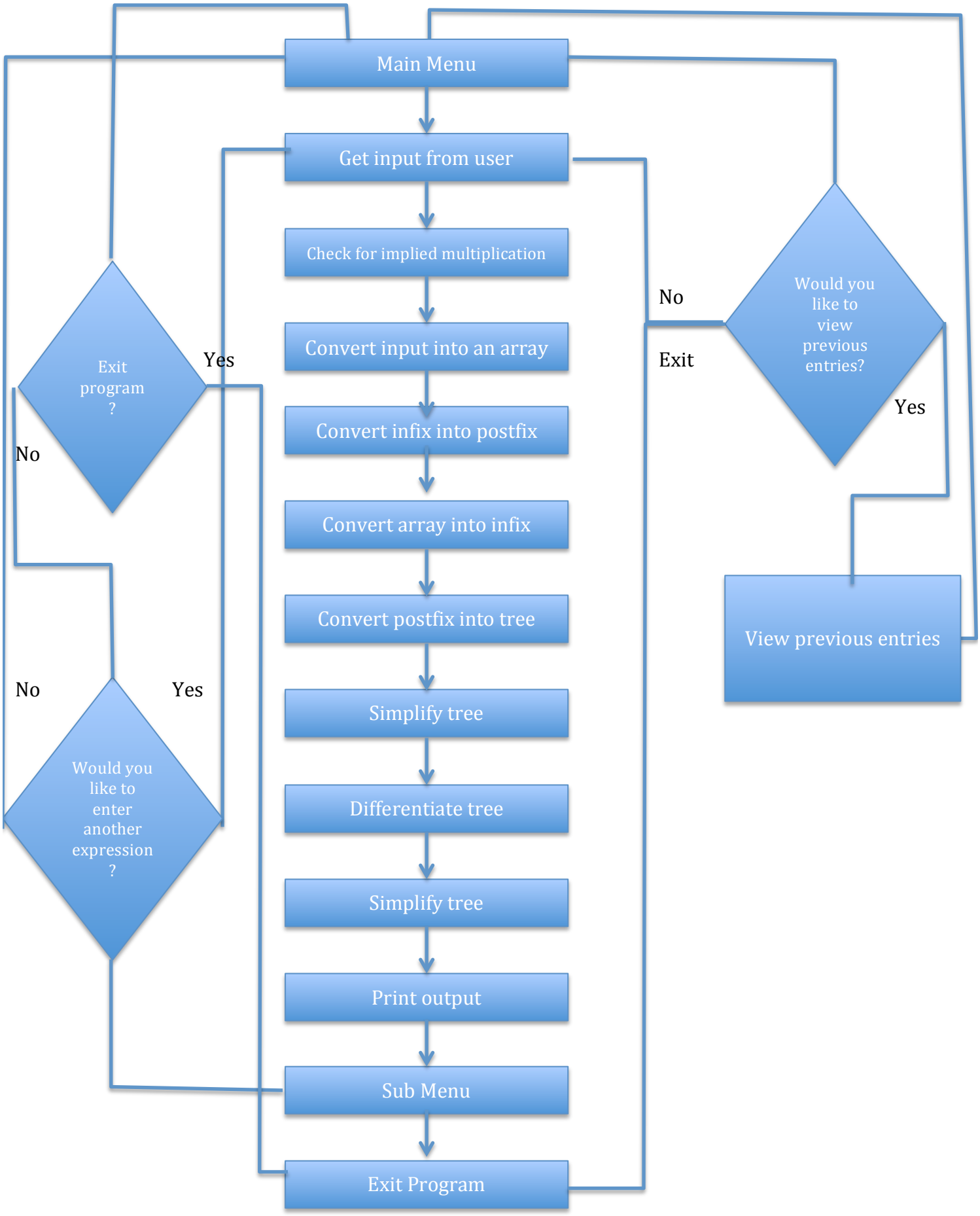
in the process. Otherwise it runs the same process without printing any of the values returned between the input and final output.

One the user has differentiated a function they then have to option of differentisting another expression, returning to the first menu, or exiting the program.

In the main menu there is an option there is an option to view previous entries. At the end of each differentiation process storeDictionary is call where the input and output are stored as keys to eachother in a dictionary called 'self.dictionary'. This way when a user wishes to view previous entries printDictionary is called which prints out all of the expressions and their keys (which are their differentials).

This file does not influence MathsExpression or TreeNodes, all it does it pass and retrieve information. All of the data manipulation is done outside of this file. By having a structure like this the program can be applied to another user interface without it having any effect of its function.

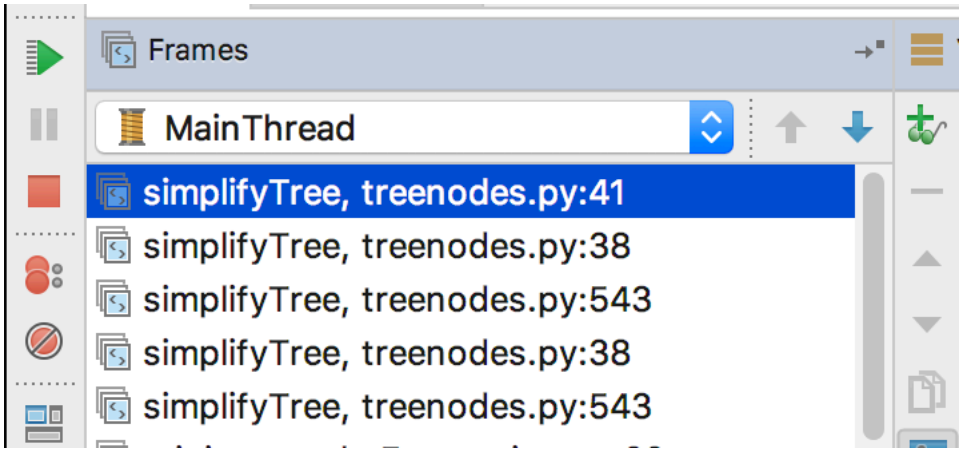
This is a flow chart representing the process of an input:



Treenodes

In order to help with differentiating expressions I created a python file called treenodes, which contains an object-orientated program for my mathsExpression file to import. I created an abstract object called a treeNode for all of my other node types to descend from, which contains common methods between all/most nodes types.

This diagram shows how the expression is represented as a tree and can clearly show how the program works up the tree. This is because when a recursive routine is called a stacking queue is created for example when simplifying a tree the root node is simplifies, which in turn simplifies its children, which then simplifies its children and so on. Each time it does this the previous node is added to a stack (first in last out). This means that the routine works its way to the bottom of the tree and all of the parent nodes above have been in a stack in the correct order of lineage.



This screenshot shows the stack that the program uses when simplifying a tree once the bottom of the tree is reached, these simplify routines are popped off and completed.

TreeNode class

The way this tree structure has been coded is recursive so that the program can handle inputs of any size. This also means because it is recursive, if the functions work on simple inputs individually it will work on complicated inputs. For example if it can run the inputs: '1+x', 'x-3', '6\5', '7*8', and 'x^2' individually then in will also be able to run more complicated inputs such as: ((1+x)*(x-3))/x^2.

```
class treeNode(object):
```

The class treeNodes is the name of the super class that all other node types are going to inherit from.

```
def __init__(self):
    self.children = [] # list of child nodes
```

The constructor contains an empty list, which will then hold the children of that node. The tree structure being created is a list of lists, which in turn contains lists and so on. By creating this list in the constructer, every node type created which inherits from treeNode will then contain a list of its children.

```
def addChild(self,Node):
    self.children.append(Node)
```

The procedure addChild is used to add a child to the node calling it. Essentially this is just adding another node to the list. New children added will then be placed last on the list. For example adding a child of numberNode(4) to an array containing numberNodes of [1,2,3] will result in a list of [1,2,3,4].

```
def insertChild(self,node, index):
    self.children.insert(index, node)
```

InsertChild is a procedure similar to that of add child except the new child canbe placed anywhere within the list. Using addChild the child will always be thelast element in the array, whereas using insertChild the position can be chosen to be the first element in the array or in the middle of the array. For example a child of numberNode(3) can be inserted into a list of numberNodes [1,2,4,5] at index 2 to result in a list of [1,2,3,4,5].

```
def isNumericalNode(self):

    if type(self) == numberNode:
        return True

    elif type(self) == variableNode:
        return False

    elif type(self) in [operatorNode, minusNode]:
        temp = True  # assume True

        for child in self.children:

            temp = temp and child.isNumericalNode()
        return temp

    else:
        return False
```

This Boolean function is helpful to determine whether or not a node/tree can be simplified. If a node is numerical i.e doesn't contain variables (such as x) or functions. This function makes nodes check and identify their own node types.

- If the type of itself is a numbernode then return as true.
- Else if itself is a variable node return false because we don't know the numerical value of 'x'.
- Else if itself is an operator or a unary minus assume it is numerical. The function then calls itself recursively on all the children of the operatorNode. By creating temp as true then temp = temp each time it loops round and returning temp. This means that if even one of its children is not numerical, the entire node is not numerical.

- Else return false. This is to catch functions because in this case functions are not being classes as numerical.

```
def simplifyTree(self):
    root = self.copy()

    for child in self.children:
        # recursively simplifies all children in the tree
        root.addChild(child.simplifyTree())

    #print('we are going to simplify: ' + self.treeToText() + ' to ' + root.treeToText())
    return root
```

The function simplifyTree on treeNode is recursive. The purpose of simplifyTree in the treeNode class is to ensureall of its children are simplified. This function is overridden in other objects that require different simplification methods, such as operatorNodes.

```
def binaryToNaryTree(self):
    root = self.copy()

    for child in self.children:
        # recursively checks children
        root.addChild(child.binaryToNaryTree())

    return root
```

This function just makes a node call binaryToNarayTree() on its children, which will be called on their children and so on. This function is overridden in the operatorNode object for creating an nary tree for commutative operators. This is to make simplification routines easier later on.

```
def equalTree(self):
    # assume trees are equal
    isEqual = True
    index = 0

    while isEqual and index in range(len(self.children)):
        self.children[index].equalTree()

    return isEqual
```

This function was introduced to check whether or not two trees are the same as each other. This was intended to be used to help with certain simplification routines within the tree.

```
def copy(self):
    raise Exception('This is an abstract method and should not have been called')
```

All other node types contain their own copy method so if this is called there has been an error somewhere as this method should be abstract and therefore shouldn't be called anywhere.

VariableNode class

```
class variableNode(treeNode):
```

This class is used for variableNodes i.e ‘x’ so all of the functions are specific to variables. It inherits from treeNode.

```
def __init__(self, variable ):
    self.variable = variable
    super(variableNode, self).__init__()
```

In its constructor it identifies self.variable as just variable. This is just to make it nicer when read by people. The constructor then calls the constructor of the super class of variableNode (which is treeNode) passing itself over as the object.

```
def differentiate(self):

    if self.variable == 'x':
        return numberNode(1.0)
```

The differentiate function just returns the differential of x, which is just 1. This function returns the number node 1 as float.

```
def treeToText(self):
    return self.variable
```

treeToText() is called when printing out a tree in a console. By returning self.variable it will be printed out as ’x’.

```
def evaluate(self):
    # this lets me know there is a problem in my code as variables should not be evaluated
    raise Exception('Cannot evaluate a variable node - replace the variables with their values first in code somewhere')
```

This is just here to help in identifying errors, as variables should never need to be evaluated. Therefore if the exception is raised then somewhere in the code evaluate has been called on a variable.

```
def copy(self):
    return variableNode('x')
```

This function is called when duplicating a tree. When a variable node is being copied, return a variableNode of ‘x’.

FunctionNode class

```
class functionNode(treeNode):
```

This class descends from treeNode and the methods are specific to functions.

```
def __init__(self, function):
    self.function = function
    super(functionNode, self).__init__()
```

The constructor of functionNode calls the constructor of it super class (which is treeNode) and passes itself in as the object.

```
def differentiate(self):
    child = self.children[0]
    diffChild = child.differentiate()
```

When differentiating a function, its differential is the differential of the functions argument multiplied by the differential of the function applied to the original argument of the function. For example the differential of sin(x) is 1*cos(x). Functions are unary operators so they will only ever take one argument. Child = self.children[0] is used to make the code easier to read. diffChild = child.differentiate() calls the differentiate function on the child

```
if self.function == 'sin':
    diffSin = functionNode('cos')
    root = operatorNode('*')

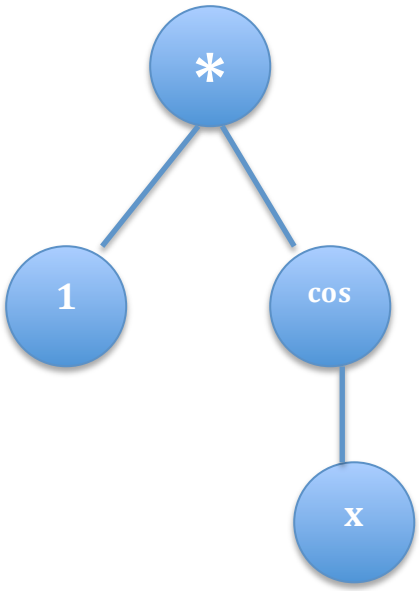
    root.addChild(diffSin)
    diffSin.addChild(child)

    root.addChild(diffChild)
```

If the function is 'sin' then 'cos' is the differential of the function and a root is created which is the operatornode of '*'. The differential of the child is added to the root (as a child), the original child is added as a child to the differentiated function. The differentiated function is then added as a child to the root. This can be visually represented for sin(x) as so:

Original Tree

Differentiated Tree




```
elif self.function == 'cos':

    sin = functionNode('sin')
    root = operatorNode('*')
    minus = operatorNode('-')

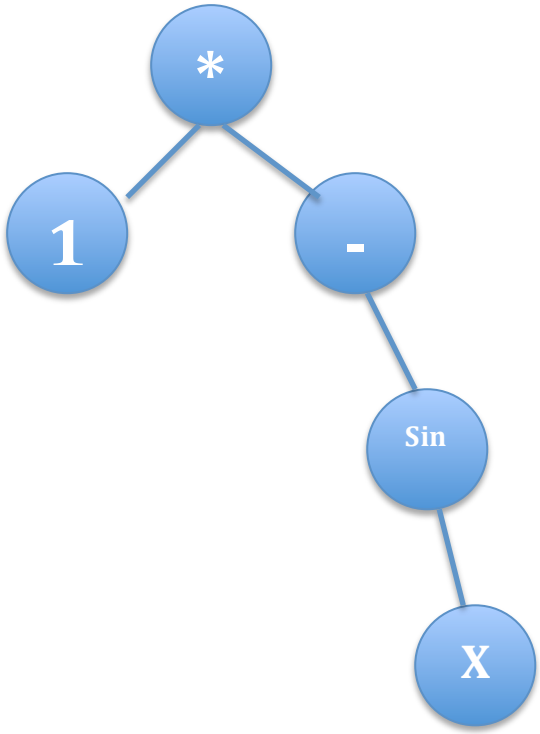
    sin.addChild(child)
    minus.addChild(sin)

    root.addChild(minus)
    root.addChild(diffChild)
```

Else if the function is ‘cos’. Then create a functioNode of ‘sin’ and create a root of an operatorNode ‘*’ because ‘cos’ differentiates to ‘-sin’ the function also creates a minus node .Then append the original child to the ‘sin’ functionNode and add the sin node as a child to the minus node.
Finally add the minus node and the differential of the child to the root. For cos(x) this can be visually represented as:

Original Tree

Differentiated Tree




```
elif self.function == 'tan':
        sec = functionNode('sec')
        subRoot = operatorNode('^')

        sec.addChild(child)

        subRoot.addChild(sec)
        subRoot.addChild(numberNode(2.0))

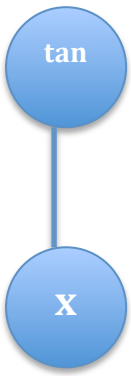
        root = operatorNode('*')

        root.addChild(subRoot)
        root.addChild(diffChild)
```

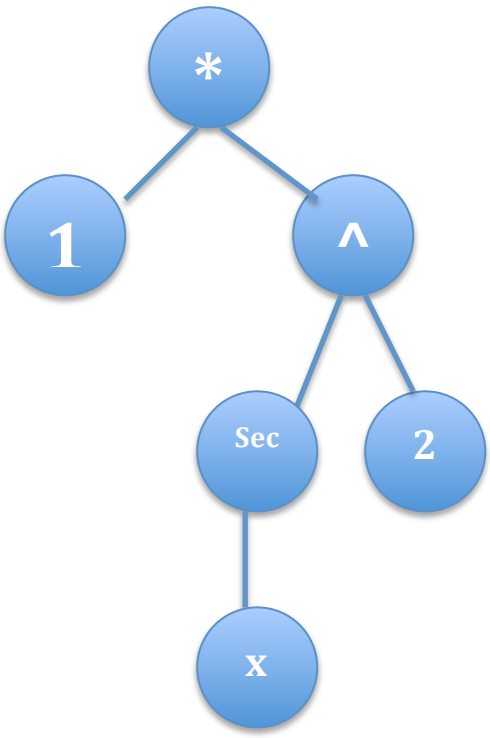
Else if the function is ‘tan’ create a functioNode of ‘sec’ and an operator node of ‘^’ because the differential of ‘tan(x)’ is ‘sec^2(x)’. Then add the original child node to the sec function node. Then add sec as a child to the subroot ‘^’ with a

number node of 2 as another child. A root of operatorNode '*' is created and the differential of the original child is added as child to the root and so is the subroot '^'. This can be visually represented for tan(x) like so:

Original Tree



Differentiated Tree




```
    else:
        raise Exception('Have not differentiated %s' %self.function)
    return root
```

Else a differentiation routine isn't available for that function. If a new function is to be introduced all that needs to be added is another 'elif' statement for that new function.

```
def copy(self):
    return functionNode(self.function)
```

When duplicating a function node just return a function node of itself.

```
def evaluate(self):
if self.function == 'sin':
    return math.sin(self.children[0].evauate())
elif self.function == 'cos':
    return math.cos(self.children[0].evaluate())
elif self.function == 'tan':
    return math.tan(self.children[0].evaluate())
```

The evaluation routine for a function returns its function and the evaluation of the functions child.

```
def treeToText(self):
    return self.function + '(' + self.children[0].treeToText() + ')'
```

For converting a function node into text, return itself and print brackets around its child converted into text.

NumberNode class

```
class numberNode(treeNode):
```

This class descends from treeNode and is specifically for number nodes, which will always be leaf nodes of the tree. This means they wont contain children.

```
def    init    (self, value):
    if not type(value) in [int,float]:
        raise Exception('number nodes must be integers or floating point numbers')

    self.value = value
    super(numberNode,self).__init__()
```

Number nodes should always be an integer or a float otherwise an error has occurred somewhere and something which shouldn't be is being assigned as a number node. The constructor then calls the constructor of its superclass (which is treeNode).

```
def evaluate(self):
    return self.value
```

When evaluating a number node, return the value of itself.

```
def differentiate(self):
    # differentiate a constant and you get 0 so return a tree with zero in it
    return numberNode(0)
```

The differential of a constant is always 0 so whether it's 1 or 1000 return a number node of 0.

```
def treeToText(self):
    return str(self.value)
```

When converting a number node to text return its own value as a string so it can be printed.

```
def copy(self):
    return numberNode(self.value)
```

To duplicate a number node return a number node of itself.

MinusNode class

```
class minusNode(treeNode):
```

This class inherits from treeNodes and is used for unary minuses because they are different from binary minuses. In the mathsExpression file it is able to differentiate between unary and binary minuses and replaces unary minuses with the character '%' which is then treated differently to '-'.

```
def __init__(self, unaryMinus):
    self.unaryMinus = unaryMinus
    super(minusNode, self).__init__()
```

The constructor just passes itself to the constructor of its super class which is treeNode.

```
def differentiate(self):

    root = minusNode('%')
    child = self.children[0]
    diffChild = child.differentiate()
    root.addChild(diffChild)

    return root
```

The differentiate function for a unary minus creates a root of minusNode '%'. It then add the differential of the child to the root and returns the root.

```
def copy(self):
    return minusNode(self.unaryMinus)
```

To copy a minus node return a minusNode of itself.

```
def treeToText(self):
    text = "-"
    child = self.children[0]
    if type(child) == operatorNode:
        text =  text + '(' + child.treeToText() + ')'
    else:
        text = text + child.treeToText()

    return text
```

When converting a unary minus into text it is printed as '-' instead of '%'. If its chid is an operator text is '-' with brackets around the child converted to text. Otherwise text is '-' and the child converted to text.

- If the child is an operator text is text with the child converted to text within brackets
- Else text is text with the child converted to text

Text is then returned.

```
def evaluate(self):
    return -self.children[0].evaluate()
```

When evaluating a unary minus return the negative of its evaluated child.

```
def simplifyTree(self):

    return self
```

When simplifying a unary minus return itself.

OperatorNode class

This is the largest class as it contains the most variation because of all of the different operator types, which are all treated differently.

```
class operatorNode(treeNode):
    def __init__(self, operatorValue):
        self.operatorValue = operatorValue

        if self.operatorValue == '+':
    self.precedence = 5
elif self.operatorValue == '-':
    self.precedence = 6
elif self.operatorValue == '*':
    self.precedence = 7
elif self.operatorValue == '/':
    self.precedence = 8
elif self.operatorValue == '^':
    self.precedence = 9
elif self.precedence == '%':
    self.precedence = 10
elif self.operatorValue in self.functions:
    self.precedence = 11
else:
    self.precedence = 1
super(operatorNode,self).__init__()
```

The constructor of this class allocates precedence to each of the operator node types. This is helpful for BIDMAS (order of operation) and also converting tree to text so the function knows where to put brackets. It then passes itself to the constructor of its super class, which is treeNode.

```
    def treeToText(self):
     text = ""
```

This function is for converting operator nodes into text. It starts with an empty string called text to which strings are added to in different ways depending on what operator type it is called on.

```
        if self.operatorValue == '+':

            if len(self.children)>0:
                text = self.children[0].treeToText()

            for i in range(1, len(self.children)):
                child = self.children[i]
                if type(child) == operatorNode and child.precedence > self.precedence :
                    text = text + '+' + '(' + self.children[i].treeToText() + ')'

                else:
                    text = text + '+' + child.treeToText()

            return text

elif self.operatorValue == '*':

            if len(self.children) > 0:
                text = self.children[0].treeToText()

                for i in range(1, len(self.children)):
                    child = self.children[i]
                    if type(child) == operatorNode and child.precedence < self.precedence :
```

```
            text = text + '*' + '(' + self.children[i].treeToText() + ')'

        else:
            text = text + '*' + child.treeToText()

    return text
```

Addition and multiplication are commutative operators so can therefore have more than 2 arguments so can therefore be an n-ary tree instead of binary. Because of this they both have very similar treeToText routines.

- If the operator is a '+' or '*' and if it has children then text is the first child converted to text. A for loop is used to go through all of its children.
- If the child is an operator node with a precedence greater than itself (+ has a precedence of 5 and '*' has a precedence of '7') then text equals text with '+' or '*' and brackets around the child converted to text.
- Else text is text with a '+' or '*' and the child converted to text. This loops though all the children of the '+' or '*' operator

Text is then returned

```
elif self.operatorValue == '-':


    if type(self.children[0]) == functionNode:
        return self.operatorValue + '(' + self.children[0].treeToText() + ')'

    else:

        left = self.children[0]
        right = self.children[1]

        leftText = self.children[0].treeToText()
        rightText = self.children[1].treeToText()

        # decide if the numerator needs brackets
        if type(left) == operatorNode and left.operatorValue in [ '-', '/']:
            text = text + '(' + leftText + ')-'
        else:
            text = text + leftText + '-'

        # decide if the denominator needs brackets
        if type(right) == operatorNode and right.operatorValue in ['+', '-', '/']:
            text = text + '(' + rightText + ')'
        else:
            text = text + rightText

        return text
```

- Else if the operator is a '-'
    - if the first child is a function return its operator value and its child converted to text within brackets.
    - Else
        - if the first child is divide or minus operator then text is text with the child converted to text within brackets.
        - If the second child is an addition, subtraction, or division operand text is text with the child converted to text within brackets.

- Else text is text with the child converted to text.

Return text

```
elif self.operatorValue == '/':

    den = self.children[1]
    num = self.children[0]

    denText = self.children[1].treeToText()
    numText = self.children[0].treeToText()

    # decide if the numerator needs brackets
    if type(num) == operatorNode and num.operatorValue in ['+', '-', '/', '*']:
       text = text + '(' + numText + ')/'
    else:
       text = text + numText + '/'

    # decide if the denominator needs brackets
    if type(den) == operatorNode and den.operatorValue in ['+', '-', '/','*']:
       text = text + '(' + denText + ')'
    else:

       text = text + denText

    return text
```

- Else if the operatoris a '\'
  - If the type of the numerator is an operator node which is a '+','-','/' or '*' then text is text with the child converted to text within brackets.
  - Else then text is text with the child converted to text.
  - If the type of the denominator is an operator node of +','-','/' or '*' then text is text with the child converted to text.
  - Else then text is text with the child converted to text.

  Return text

```
elif self.operatorValue == '^':

    left = self.children[0]
    right = self.children[1]

    leftText = self.children[0].treeToText()
    rightText = self.children[1].treeToText()

    # decide if the subject needs brackets
    if type(left) == operatorNode and left.operatorValue in [ '^','+','-', '/']:
       text = text + '(' + leftText + ')^'
    else:
       text = text + leftText + '^'

    # decide if the exponent needs brackets
    if type(right) == operatorNode and right.operatorValue in ['^','+', '-', '/']:
       text = text + '(' + rightText + ')'
    else:
       text = text + rightText
```

```
        return text
```

- Else if the operatoris a '^'
    - If the type of the first child is an operator node which is a '+','-','/' or '^' then text is text with the child converted to text within brackets.
    - Else then text is text with the child converted to text.
    - If the type of the exponent is an operator node of +','-','/' or '^' then text is text with the child converted to text.
    - Else then text is text with the child converted to text.

  Return text

```
def copy(self):
    if type(self) == operatorNode:
        return operatorNode(self.operatorValue)
```

When duplicating an operator node an operator node of itself is returned.

```
def differentiate(self):
```

The differentiation routine depends on which operator is being differentiated. This function is recursive so it is always called on the children of the operator, which in turn will differentiate its children and so on.

```
    if self.operatorValue in ['+']:
        # create a root note as a + since the D(sum)=sum(derivatives)
        root = operatorNode('+')
        for i in range(len(self.children)):
            child = self.children[i]
            childDiff = child.differentiate()
            root.addChild(childDiff)

        return root
```

- If the operator is a '+' create a root of operator node '+'
- A for loop is then used to go through all of its children; the differential of each child is then added to the root.

Return root

```
    elif self.operatorValue in ['-']:
        # create a root note as a + since the D(sum)=sum(derivatives)
        root = operatorNode('-')

        left = self.children[0]
        right = self.children[1]

        root.addChild(left.differentiate())
        root.addChild(right.differentiate())

        return root
```

- Else if the operator is a '-' create a root of an operator node '-' and append the differential of both children to the root. For minus the order of the children matter so the first child is left and the second child is right.

```
elif self.operatorValue =='*':
    # We differentiate prod(T1,T2,T3....Tn) as T1*Diff(prod(T2....Tn))+prod(T2..Tn)*Diff(T1)
    root = operatorNode('+')
    u = self.children[0]
    if len(self.children) > 2:

        v = operatorNode('*')
        for i in range(1, len(self.children)):
            v.addChild(self.children[i])

    else:
        v= self.children[1]

    du = u.differentiate()
    dv = v.differentiate()

    left = operatorNode('*')
    left.addChild(u)
    left.addChild(dv)

    right = operatorNode('*')
    right.addChild(v)
    right.addChild(du)

    root.addChild(left)
    root.addChild(right)

    return root
```

- Else if the type of the operator is a '*'. Create a root node of '+' and the first child equals u.
    - if there are more than 2 arguments then v becomes a tree with root node '*' with all of the other arguments as children.
    - Else the second child equals v.

  The function then proceeds to use the product rule for differentiation

  Du is the differential of u and dv is the differential of v
  Left is an operator node '*' v and du is then added as a child to left.

  Right is also an operator node '*' then dv and v are added as children to the right.

  Left and right are added to the root as children and root is then returned.

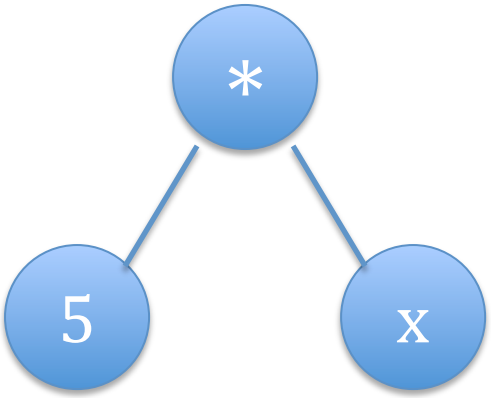
  For inputs such as 5x a function could have been implemented using the constant factor rule but another function would then have to be made to determine which nodes are suitable for it. Because the product rule still works for any multiplication input between composite functions, it was decided to always use the product rule.

$$F(x) = f(x).g(x)$$

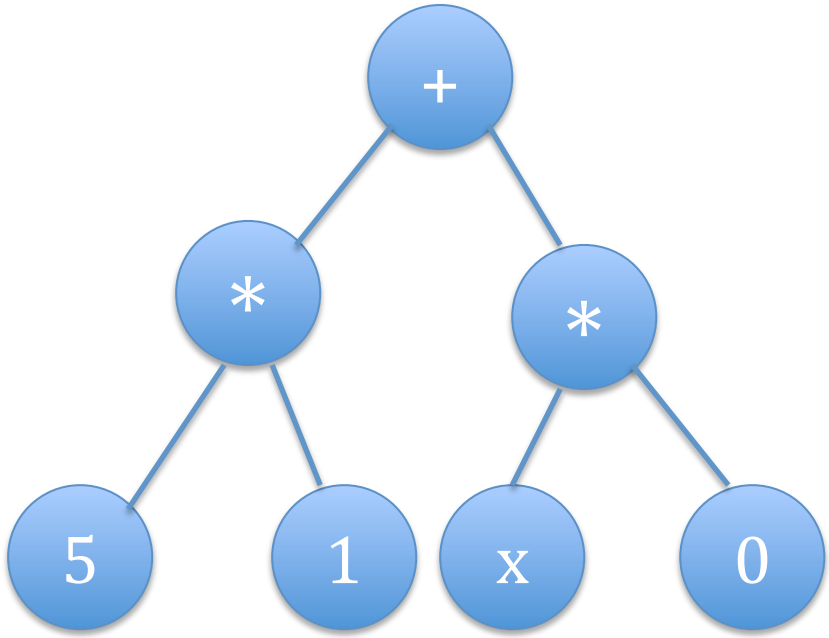
$$F'(x) = f'(x).g(x) + g'(x).f(x)$$

For example the visual representation of 5x:

Original Tree



Differentiated Tree




```
elif self.operatorValue in ['/']:
    # use quotient rule: [(v*du)-(u*dv)] /(v^2)

    root = operatorNode('/')
    numerator = operatorNode('-')
    u = self.children[0]
    v = self.children[1]
    du = u.differentiate()
    dv = v.differentiate()

    left = operatorNode('*')
    left.addChild(v)
    left.addChild(du)
```

```
right = operatorNode('*')
right.addChild(u)
right.addChild(dv)

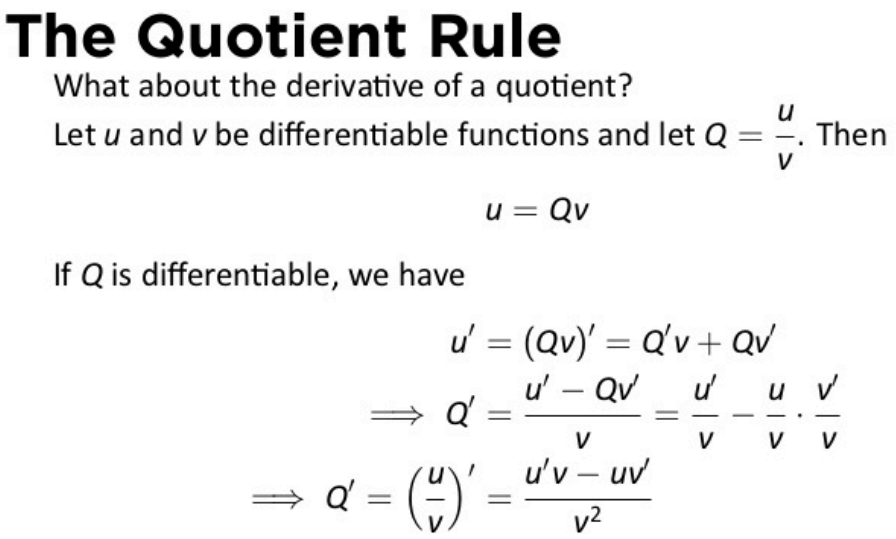
numerator.addChild(left)
numerator.addChild(right)

denominator = operatorNode('^')
denominator.addChild(v)
denominator.addChild(numberNode(2.0))

root.addChild(numerator)
root.addChild(denominator)

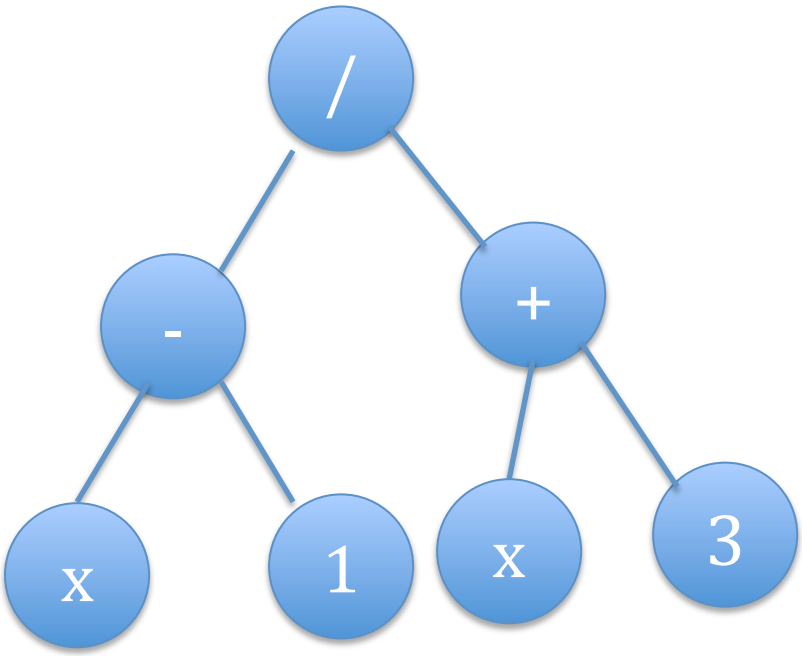
return root
```

- Else if the operator node is a '/' the quotient rule is then implemented.
  A root node is created of operator '/' and also a numerator of operator node '-'. The first child is then identified as u and the second child as v.
  The u and v are differentiated and called du and dv.
  Left is an operator node of '*' to which child v and du are added.
  Right is also an operator node '*' to which the children u and dv are added.
  Left and right are then added to the numerator node.
  A denominator node is introduced as the operator node '^'
  V is added as a child to the denominator and the number node '2' is also added as a child to the denominator to be the exponent.
  The numerator and denominator are then added as a child to the root.
  Here is a the quotient rule I found on
  http://image.slidesharecdn.com/lesson09-theproductandquotientrules011slides-110226121024-phpapp01/95/lesson-9-the-product-and-quotient-rules-slides-58-728.jpg?cb=1298722293 :

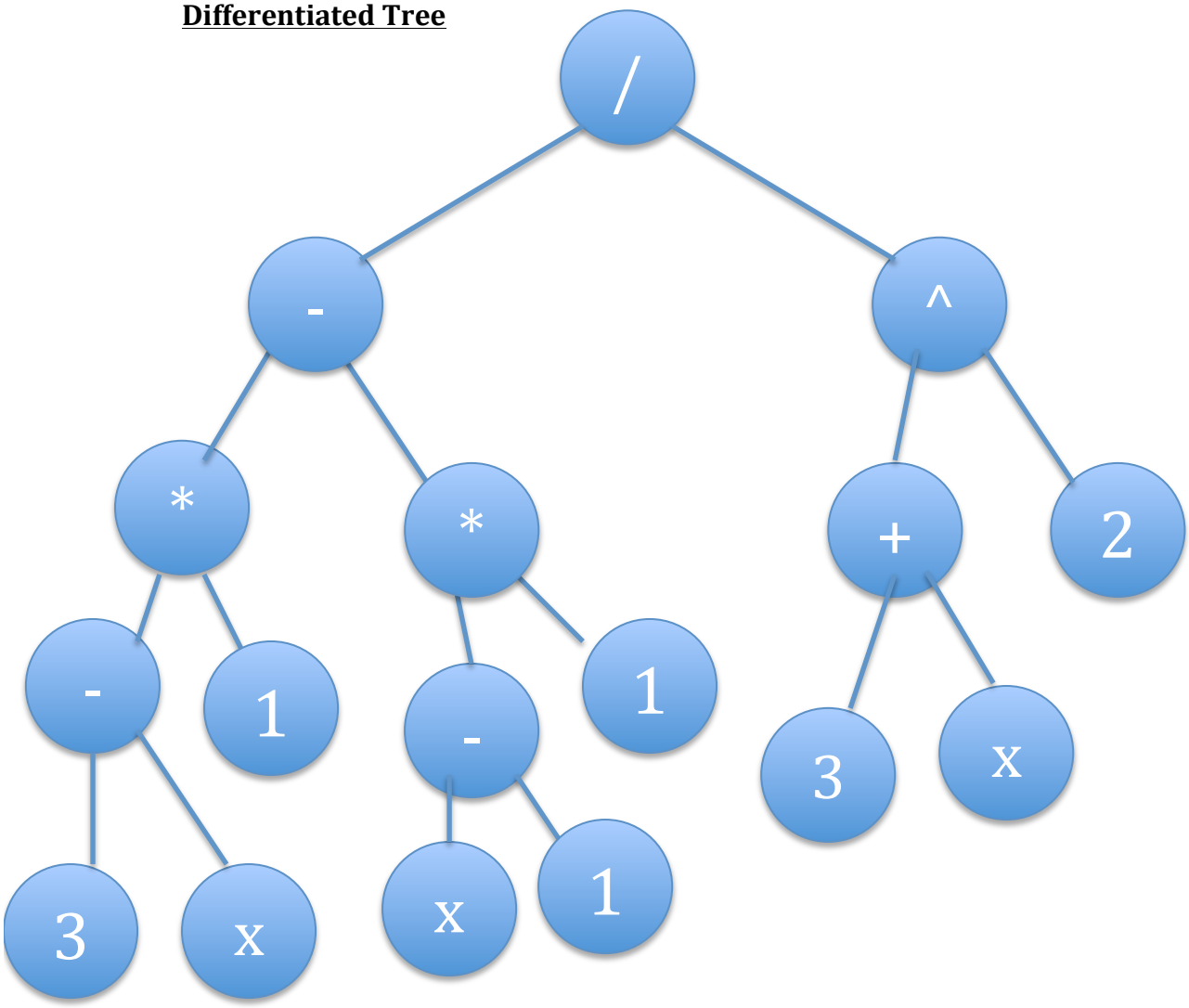


This can be visually represented with (x-1)/(x+3):

Original Tree



Differentiated Tree



$$\frac{x-1}{x-3}$$

$$\frac{d}{dx}\left(\frac{x-1}{x-3}\right) = \frac{(x-3)\frac{d}{dx}(x-1)-(x-1)\frac{d}{dx}(x-3)}{(x-3)^2}$$

$$= \frac{-2}{(x-3)^2}$$

```
elif self.operatorValue in ['^']:

        u = self.children[0]
        du = u.differentiate()
        n = self.children[1]

        root = operatorNode('*')
        powerNode = operatorNode('^')
        powerNode.addChild(u)

        nMinus1 = operatorNode('-')
        nMinus1.addChild(n)
        nMinus1.addChild(numberNode(1.0))

        powerNode.addChild(nMinus1)

        nTimesdu = operatorNode('*')
        nTimesdu.addChild(n)
        nTimesdu.addChild(du)

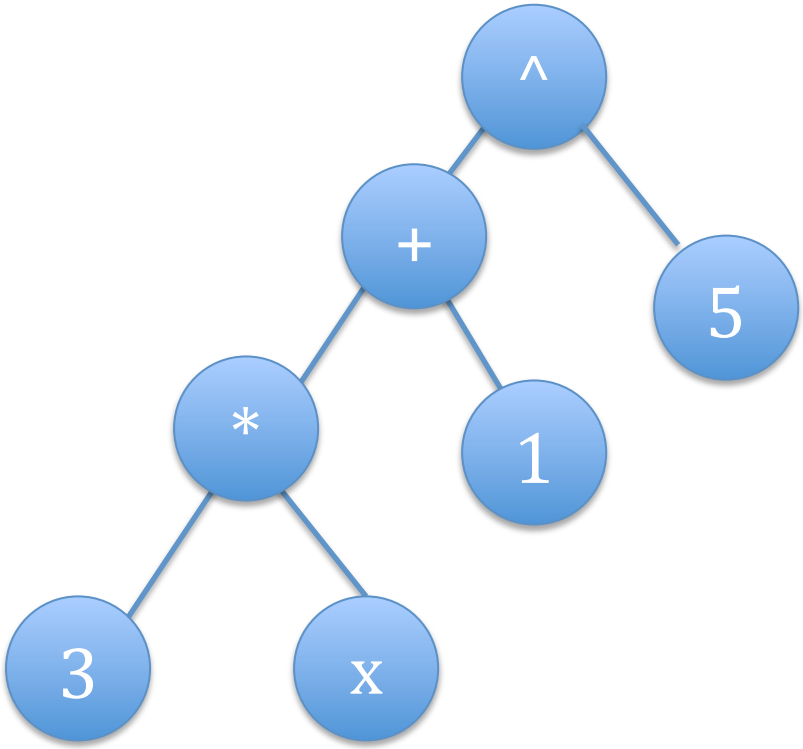
        root.addChild(nTimesdu)
        root.addChild(powerNode)

        return root
```

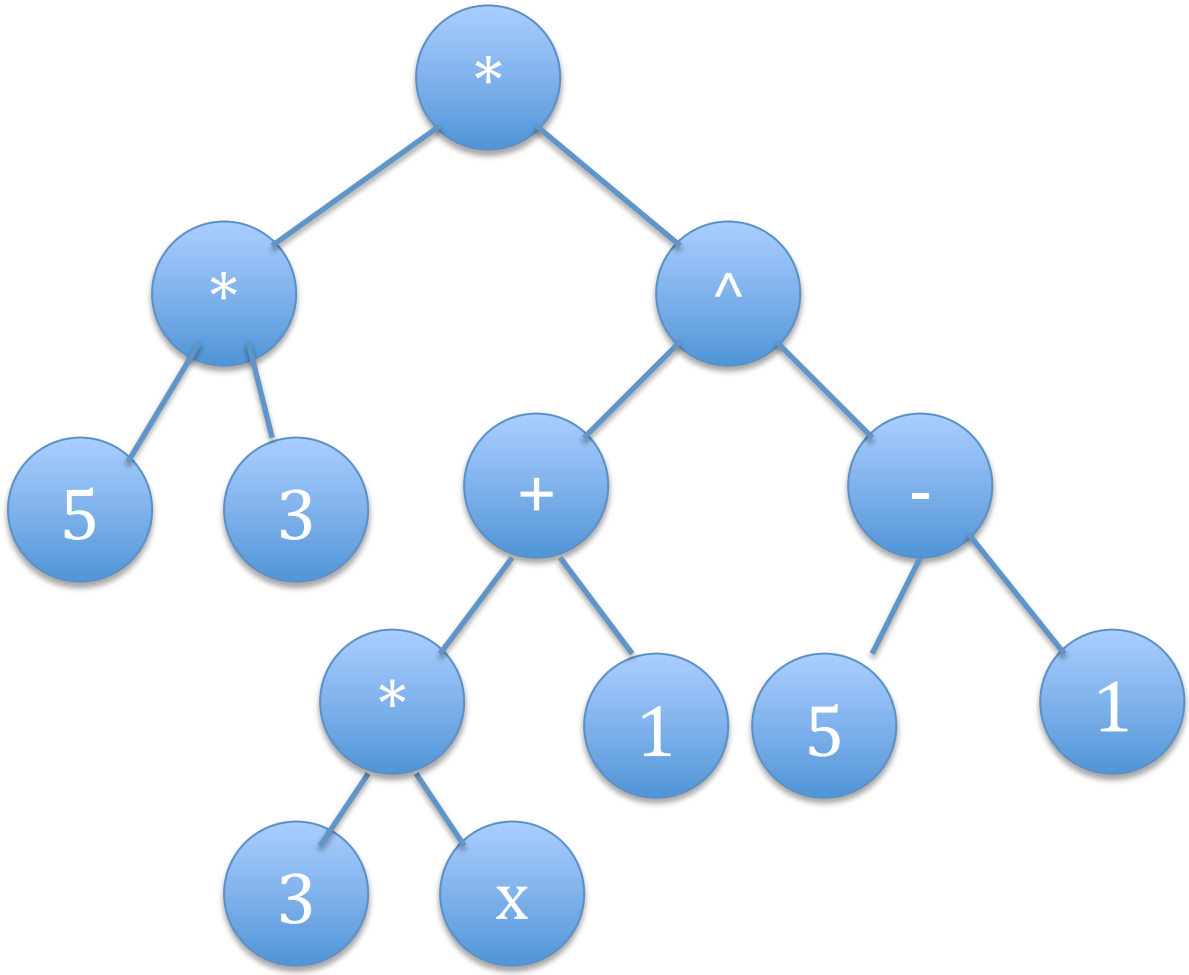
- Else if the operator node is a '^'
  U is the first child of the node. Then du is the first child differentiated and n is the second child of the node (which is the exponent).
  A root is created of operator node '^' to which u is added as a child.
  A node called nMinus1 is created as an operator node '-' to which child n is added to along with a number node of '1'.
  Then nMinus1 is added as a child to the powerNode.
  nTimesdu is introduced as an operator node '*' to which the n and du nodes are added to as children.
  nTimesdu and powerNode are then added as children to the root.

This can be visually represented for (3x+1)^5 like so

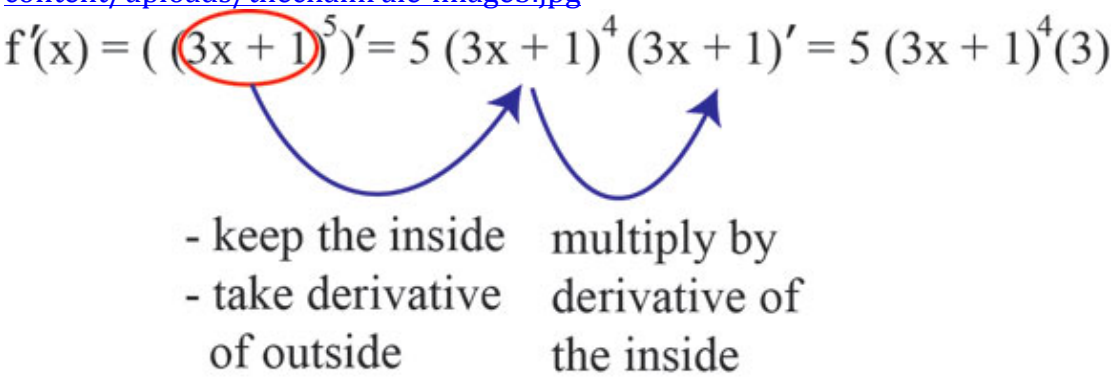
Original Tree



Differentiated Tree



This example shown was found at http://www.mathbootcamps.com/wp-content/uploads/thechainrule-image3.jpg




```
def evaluate(self):
    operators = ['+', '-', '*', '/', '^']
    functions = ['sin', 'tan', 'cos']
```

The function evaluate is used to get outputs from applying the operators to their children in a tree.

```
    if self.operatorValue in operators :


        if self.operatorValue == '+':
            temp = 0.0
            for child in self.children:
                temp = temp + child.evaluate()

            return temp
```

- If the operator is a '+' then the function goes through all of the children and evaluates them and adds them to temp each time. After going through all of the children temp is returned. If a '+' operator has 3 number nodes as children of [3,4,5] then the function will return 12. This is useful for the mathsExpression file because that operator node can then be replaced with a number node of 12.

```
        elif self.operatorValue == '*':
            temp = 1.0
            for child in self.children:
                if child == 0:
                    return 0
                else:
                    temp = temp * child.evaluate()

            return temp
```

Else if the operator is a '*' temp is 1 (because anything multiplied by 0 is 0). It then goes through all of the children and multiplies the temp value with each child's evaluation. If one of the children is equal to zero then the function returns 0 because anything multiplied by zero is zero.

```
        elif self.operatorValue == '-':
            return self.children[0].evaluate() - self.children[1].evaluate()

        elif self.operatorValue == '/':
            return self.children[0].evaluate() / float(self.children[1].evaluate())
```

```
        elif self.operatorValue == '^':
            # if self.children[1] == 1:
            #     return numberNode(1)
            return self.children[0].evaluate() ** self.children[1].evaluate()
```

For the other non-commutative binary operators, the function returns the result of the operator being applied to the evaluated form of the first and second child.

```
        else:
            raise Exception('I have not implemented %s evaluate yet',self.operatorValue)
```

This else statement is just here to catch and highlight any evaluations, which have yet to be implemented.

```
    else:
        return self.value
```

Else the node is not an operator so just return its value.

```
def evaluateVariables(self):

    if self.operatorValue == '*':
        for child in self.children:
            if child == numberNode(0):
                return 0

    else:
        return self
```

Evaluate variables is a function which just gets rid of variables that are being multiplied by 0.

```
def copy(self):
    return operatorNode(self.operatorValue)
```

The copy function just returns an operator node of itself.

```
def binaryToNaryTree(self):
    #check if operator is commutative
    if self.operatorValue in ['*', '+']:
        # create a temporary list to hold grandchildren
        naryChildList = []

        for child in self.children:

            if type(child) == operatorNode and child.operatorValue == self.operatorValue:

                for grandchild in child.children:
                    naryChildList.append(grandchild)
                self.children.remove(child)

        # add each element in list of grandchildren to self
        for element in naryChildList:
            self.addChild(element)

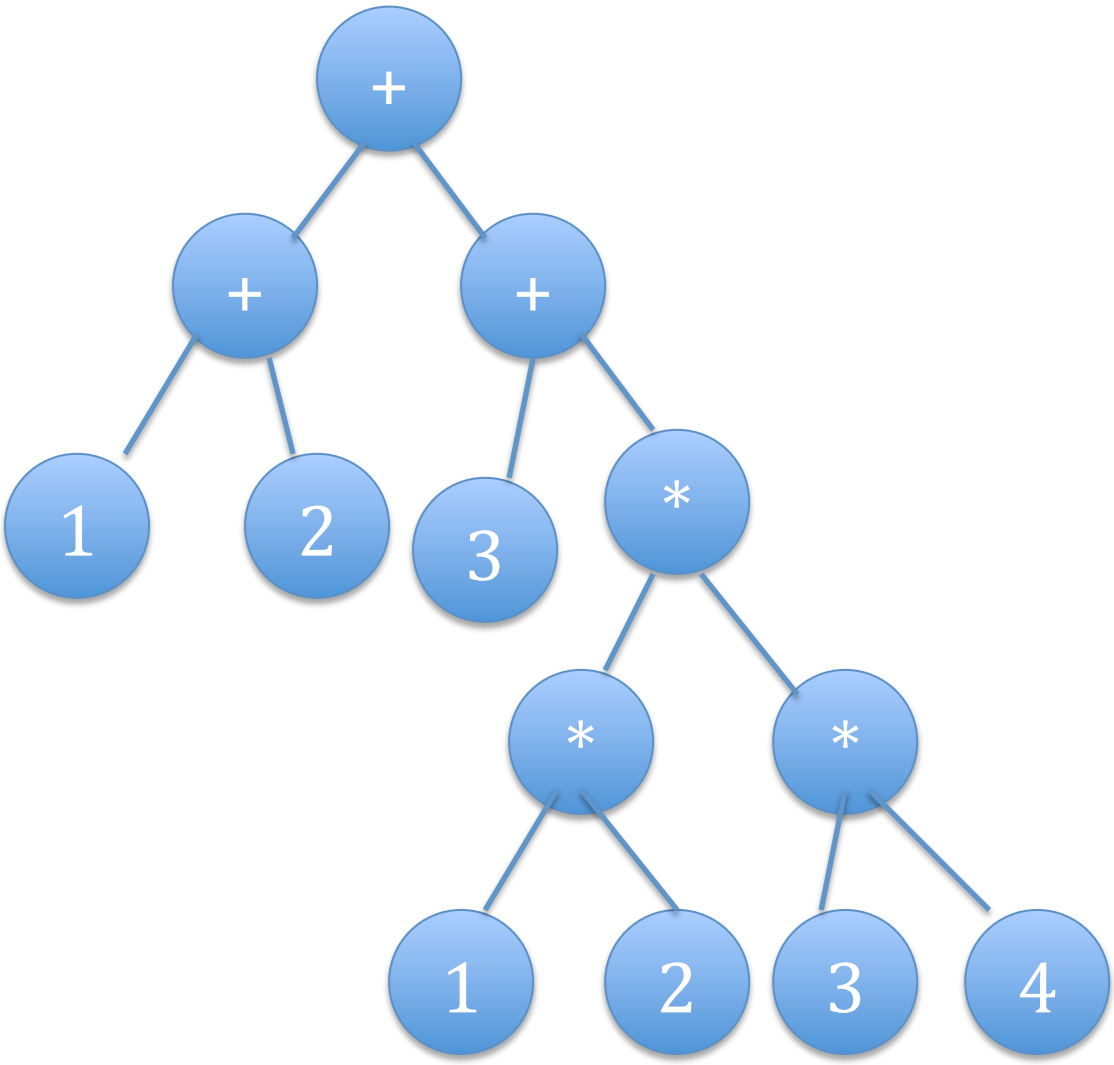
    return self
```

The binaryToNaryTree function is for the commutative operators (+,*). This uses a nested for loop which checks all of the children and if they are the same as the parent node (i.e both the same commutative operator),

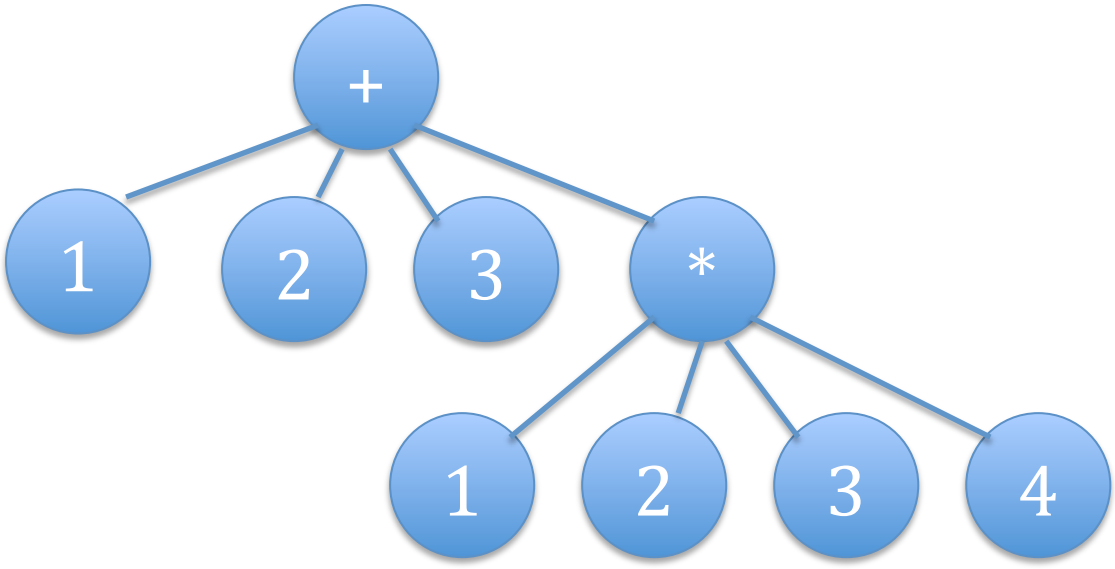
then the children of the child are added to the parent node and the child operator node is removed.

This can be represented as:

Original Tree



Binary to nary Tree




```
def simplifyTree(self):
    #first simplify children
    presimplifytext = self.treeToText()

    #print('we are going to simplify: ' + self.treeToText())
    self = super(operatorNode, self).simplifyTree()
```

This function is called to simplify trees. Self is equal to the simplifyTree() function of the super class with operator node passed as a parameter. In the super class the simplify function makes sure all of its children are simplified.

```
    if self.isNumericalNode():
        return numberNode(self.evaluate())
```

If the simplified node is a numerical node then return the evaluated node.

```
    elif self.operatorValue in ['+']:
        # go through all the children and replace the numerical nodes with a single node
        root = operatorNode('+')
        # temp holds onto a sum of all the nodes that are numerical
        temp = 0
        for child in self.children:

            if child.isNumericalNode():
                temp = temp + child.evaluate()
            # if child is not numerical add straight to root

            else:
                #if not numerical add to tree
                root.addChild(child)

        if temp != 0:
            root.addChild(numberNode(temp))

        if len(root.children) == 1:
            # +x makes no sense so just return child
            #print(' to ' + root.children[0].treeToText())
            return root.children[0]
```

```
        #print(' to ' + root.treeToText())
        return root
```

If the operator is a '+' then very similarly to the evaluate routine, it evaluates all of the children and replaces all of the numerical nodes with a singular numerical node. The node is then only added back to the tree if it isn't a 0.
If there is only one child then there is no need for the operator so replace the parent node with its child. This is done by returning the child without its parent node does this.

```
    if self.operatorValue in ['*']:
        root = operatorNode('*')
        temp = 1.0


        for child in self.children:

            if child.isNumericalNode():
                childValue = child.evaluate()

                if childValue == 0:
                    return numberNode(0)

                else:
                    temp = temp * childValue

            else:
                root.addChild(child)
```

If the operator is a '*', then similarly to the addition simplification routine it goes through all of its children and multiplies the temporary value by each child. If any of the children are equal to zero the function returns zero.
Else if the child is not a numerical node then the child is added to the root.

```
        # e.g instead of x*2 we write 2*x
        if temp != 1:
            root.insertChild(numberNode(temp), 0)

        if len(root.children)==1:
            #print(' to ' + root.children[0].treeToText())
            return root.children[0]
        #print(' to ' + root.treeToText())
        return root
```

If the temp value is not equal to one (as it will be numerical) we insert the child into the first element in the child list. This is just so that when the node is converted into text it is more legible because if the node is the product of 3 and x by inserting the child it will be printed out as 3*x as opposed to adding the child and getting x*3. The reason this is only done if the temporary value doesn't equal 1 is because for this program there is no point in returning '1*x' when it can just return 'x' for example. If there is only one child because multiplication requires at least two arguments then just return the child without the root.
Otherwise return the root.

```
if self.operatorValue in ['/']:

        # there should never be more than two children for a quotient
        if self.isNumericalNode():
            return numberNode(self.children[0]/self.children[1])
        else:
            return self

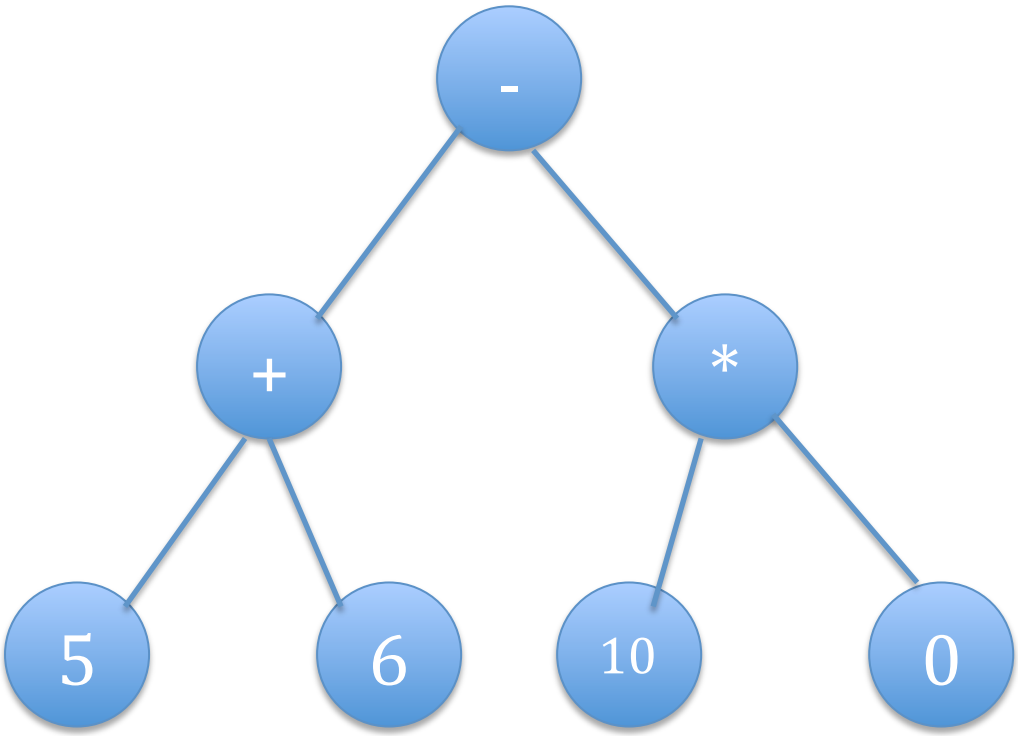
    if self.operatorValue in ['-']:

        # there should never be more than two children for a subtraction
        if self.isNumericalNode():
            return numberNode(self.children[0]-self.children[1])
        else:
            return self
```

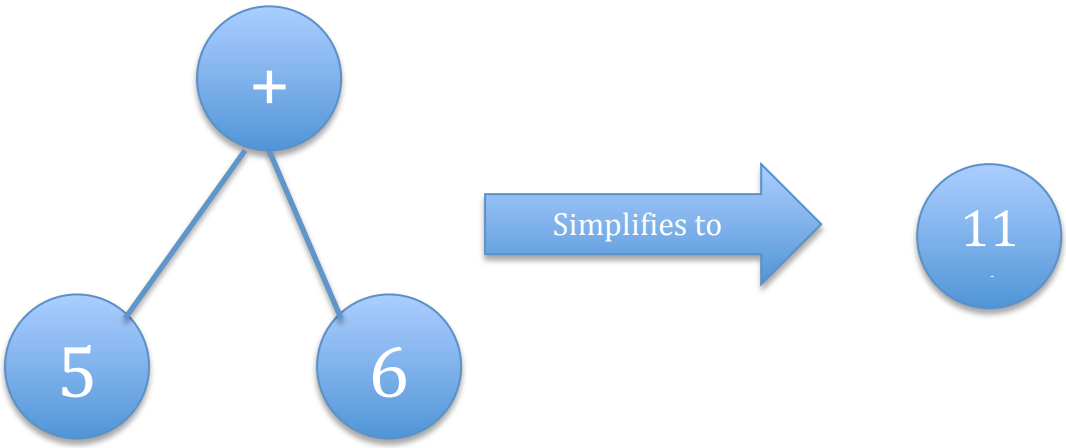
For the '/' or '-' operator if the children are numerical then the function returns the operator applied to the evaluated forms of both the children. Else return self.

An example for simplifying subtraction multiplication and addition is (5+6)-(100*0) which can be visually represented like so:

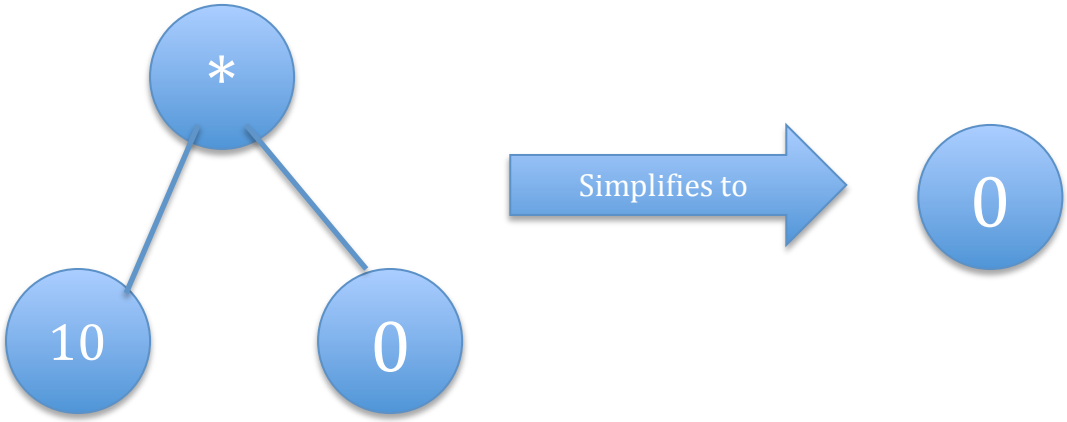
Original Tree



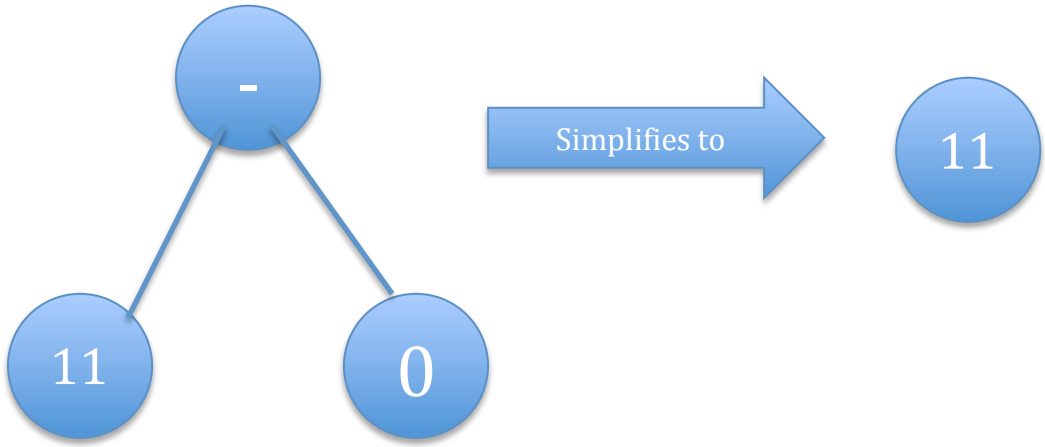
Simplifying 1st child Tree



Simplifying 2nd child Tree



Simplifying the parent Tree



Therefore the tree will be simplified overall to a single number node of 11.

```
if self.operatorValue in ['^']:

    # there should never be more than two children for a power
    base = self.children[0]
    power = self.children[1]

    if self.isNumericalNode():

        return numberNode(math.pow(base.evaluate(), power.evaluate()))
    else:

        if power.isNumericalNode():
            powerVal = power.evaluate()

            #base to the power of 1 is the base
            if powerVal == 1.0:
                return base

            # anything to the power of 0 is 1
            elif powerVal == 0.0:
                return numberNode(1)

        else:
            if base.isNumericalNode():

                baseVal= base.evaluate()
                # 0 to the power of anything is 0
                if baseVal == 0.0:
                    return numberNode(0)
                # 1 to the power of anything is 1
                elif baseVal == 1.0:
                    return numberNode(1)
```

```
        return self
```

If the operator is a '^' then the base is the first child and power is the second child.
If the node is numerical then return the base to the power.

Else

- If the base is numerical then evaluate the base
  - If the evaluated base is equal to 0 then return 1 because anything to the power of 0 is 1
  - Else if the evaluated base is equal to 1 then return the number node 1 because 1 to the power of anything is always 1.

If a value isn't returned from any of the 'if' and 'else if' statements then return self

MathsExpression

This file imports the treeNodes file for all of the functions involving the nodes. This file contains all of the routines, which are outside of the tree structure.

Getting the input from the user

```
def getUserInputString(self):
    # This function takes in an input string and converts it into an array of atomic elements

    userStringInput = raw_input(str("Please Enter expression: "))

    if userStringInput == '':
        userStringInput = '(-2+3)'

    print(userStringInput)
    return userStringInput
```

The first function I call is called getUserInput() it is a very simple function that takes the raw input from the user and returns it as a string. It also prints out the inputted string so the user can see if the inputted expression is correct. I also added an 'if' statement for testing purposes to input a set expression if left blank. This was done for testing so I didn't have to repeatedly input the same expression.

Checking for implied Multiplication

```
def impliedMultiplication(self, inputString):

    result = inputString;

    result = re.sub('([a-zA-Z]+)([0-9]*\.?[0-9]+)', '\g<1>*\g<2>', result)  # e.g. x6->x*6

    result = re.sub('([0-9]*\.?[0-9]+)([a-zA-Z]+)', '\g<1>*\g<2>', result)  # e.g. 6x->6*x
```

```
#result = re.sub('([0-9]*\.?[0-9]+)([a-zA-Z]+)', '\g<1>\u2062\g<2>', result)
result = re.sub('([0-9]*\.?[0-9]+)(\()', '\g<1>*\g<2>', result)  # e.g. // 6( -> 6*(
result = re.sub('(\))([0-9]*\.?[0-9]+)', '\g<1>*\g<2>', result)  # e.g. )6( -> )*6
return result;
```

This function called impliedMultiplication() takes the string returned from the getUserInput() function and uses regular expressions to check if the user intended multiplication. Firstly it checks if a variable (i.e 'x') is followed by a number, if so a '*' character is inserted between the two characters. This means that if the user input x5 ImpliedMultiplicaiton() returns x*5. This also checks a number followed by a variable, a number followed by a bracket and vice versa, therefore the following strings would be returned as:

x5 -> x*5

5x -> 5*x

5(…) -> 5*(…)

(…)5 -> (…)*5

Converting string into an array

```
def getUserInputArray(self, inputString):
    inputArray = []
    inputLength = len(inputString)
    index = 0
    # This for loop adds the characters into our inputList, which gives me an array which will allow me
    # to change the order of the character for RPN
    for index in range(int(inputLength)):
        inputArray.append(inputString[index])
        index = index + 1

    return inputArray
```

Once a string is returned from impliedMultiplication() it is sent to getUserInputArray(). This is a very simple function which uses the string returned from impliedMultiplication() and uses a for loop which repeats from zero to the length of the inputString, and appends each character individually to an array called 'inputArray'. Once the for loop has finished the inputArray is returned. The reason for converting the string into an array is that an array is much easier to manipulate and isolate each element atomically.

Converting array into infix

```
def getInfixAlgorithm(self, inputArray):
    # token string is used so that I can add strings as an element of the array this means that I can use
```

```
# numbers larger than a single digit and decimal numbers
userInput = inputArray
inputLength = len(userInput)
bracketCount = 0
counter = 0
token = ""
infix = []

while counter < inputLength:
    # this loop collects token a token string and adds the string to the array as an element
    # so larger numbers and decimals can be used

    character = userInput[counter]

    # This if statement skips the code past any spaces in the user input
    if character == ' ':
        counter = counter + 1

    # checks that parenthesise are matching else input is invalid
    elif character == '(':
        bracketCount = bracketCount + 1

        if len(token) > 0:
            infix.append(token)
            token = ""

        infix.append(character)
        counter = counter + 1

    elif character == ')':
        bracketCount = bracketCount - 1

        if len(token)>0:
            infix.append(token)
            token = ""

        infix.append(character)
        counter = counter + 1

    elif character in self.operators:
        if character in ['-']:

            nextCharacter = userInput[counter + 1]
            lastCharacter = userInput[counter - 1]

            if counter == 0 or nextCharacter in ['('] or lastCharacter in ['('] or lastCharacter in self.operators \
                or nextCharacter in self.functions:

                character = '%'

        if len(token) > 0:
            infix.append(token)
            token = ""

        infix.append(character)
        counter = counter + 1
        token = ""


    else:
        token = token + character
        counter = counter + 1


if len(token)>0:
    infix.append(token)
```

```
if bracketCount != 0:
    print('Error, mismatching parenthesise')
    sys.exit()

else:
    print('Infix = ' + str(infix))
    return infix
```

Once getUserInputArray() returns an inputArray, getInfixAlgorithm is used to convert the array into infix notation. This function uses a variable called 'token' which is used so that I can add individual elements from the inputArray to create a string as a token, the token is then appended to the infix array so the array can contain strings. This way the array will contain elements that require more than one character such as functions or large numbers. In order to this the function uses a while loop until the length of the input. A while loop is used here are the length of the input array is unknown. The function then uses 'if' and 'elif' statements so it has the correct response to each element in the inputArray. The statements check the character (userInput[counter]) and respond as follows:

- If the character is a ' ' (a space) then increment the counter. This means that the character is then updated to the next element in the array and the space is ignored, as we do not need it.
- Else if the character is an opening parenthesise the bracketCounter (which is initially 0) is incremented by one. If there is anything in our token it is then appended to the infix, then the parenthesise is added to the token.
- When the closing parenthesise becomes the character, bracketCounter is decreased by one. The token is then appended to the infix and the closing parenthesise is added to the token.
- Otherwise if the character is an operator ('+', '-', '/', '^') if the token string isn't empty, add the token to the infix array then add the operator. If the operator is a '-' and is identified as a unary minus (an operator with only one argument), then the character becomes a '%' instead of a '-'. This allows later functions to differentiate between a unary and binary minuses.
- Else add the character to the current token, this way the function will catch integers, functions, and variables.
- Outside the while loop if there is anything left in the token, it is appended to the infix. Next the bracketCounter is checked, if it is not equal to zero then there has been a mismatch of parenthesise, therefore an error occurs.
- Lastly the function returns the infix

Converting infix to postfix

```
def postfixAlgorithm(self, infix):
    # conversion from infix to reverse polish notation

    leftAssociativeOperators = ['*', '-', '+', '/']
    rightAssociativeOperators = ['^']
    unaryMinus = ['%']
    functions = ['sin', 'tan', 'cos']
    leftParentheses = '('
    rightParentheses = ')'
    brackets = [leftParentheses, rightParentheses]
    inputLength = len(infix)
    tokenStack = []
    postfixArray = []
    bracketCount = 0
    counter = 0


    while counter < inputLength:

        currentSymbol = infix[counter]

        if self.isNumber(currentSymbol) :
            postfixArray.append(currentSymbol)
            counter = counter + 1

        elif currentSymbol in ['%']:
            tokenStack.append(currentSymbol)
            counter = counter + 1

        elif currentSymbol in functions:
            tokenStack.append(currentSymbol)
            counter = counter + 1

        elif currentSymbol in self.operators:
            counter = counter + 1

            # using bidmas to check precedence of operators to know order of operations
            if currentSymbol in leftAssociativeOperators:
                while len(tokenStack)>0 and tokenStack[-1] in self.operators and \
                self.getTokenPrecedence(tokenStack[-1]) >= self.getTokenPrecedence(currentSymbol):
                    postfixArray.append(tokenStack.pop())
                tokenStack.append(currentSymbol)

            else:
                while len(tokenStack) > 0 and tokenStack[-1] in self.operators and \
                            self.getTokenPrecedence(tokenStack[-1]) > self.getTokenPrecedence(
                        currentSymbol):
                    postfixArray.append(tokenStack.pop())
                tokenStack.append(currentSymbol)


        elif currentSymbol == leftParentheses:
            counter = counter + 1
            if len(tokenStack) > 0 and tokenStack[-1] in self.functions:
                postfixArray.append(self.terminalToken)
            tokenStack.append(currentSymbol)

        elif currentSymbol == rightParentheses:
            counter = counter + 1

            while len(tokenStack) > 0 and tokenStack[-1] != leftParentheses:
                postfixArray.append(tokenStack.pop())

            if len(tokenStack) > 0 and tokenStack[-1] != leftParentheses:
                print ('Error occurred, mismatch of parentheses')

            else:
```

```
            tokenStack.pop()

        if len(tokenStack) > 0 and tokenStack[-1] in self.functions:
            postfixArray.append(tokenStack.pop())

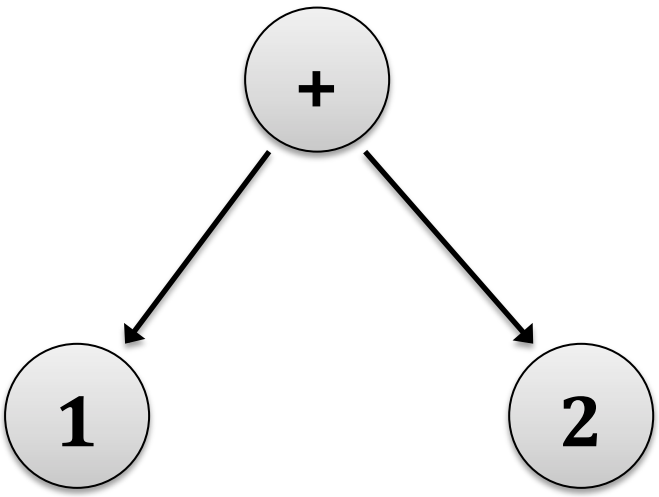
    else:
        # Its none of the others therefore it must be a variable
        postfixArray.append(currentSymbol)
        counter = counter + 1


while len(tokenStack) > 0:
    operator = tokenStack.pop()
    postfixArray.append(operator)

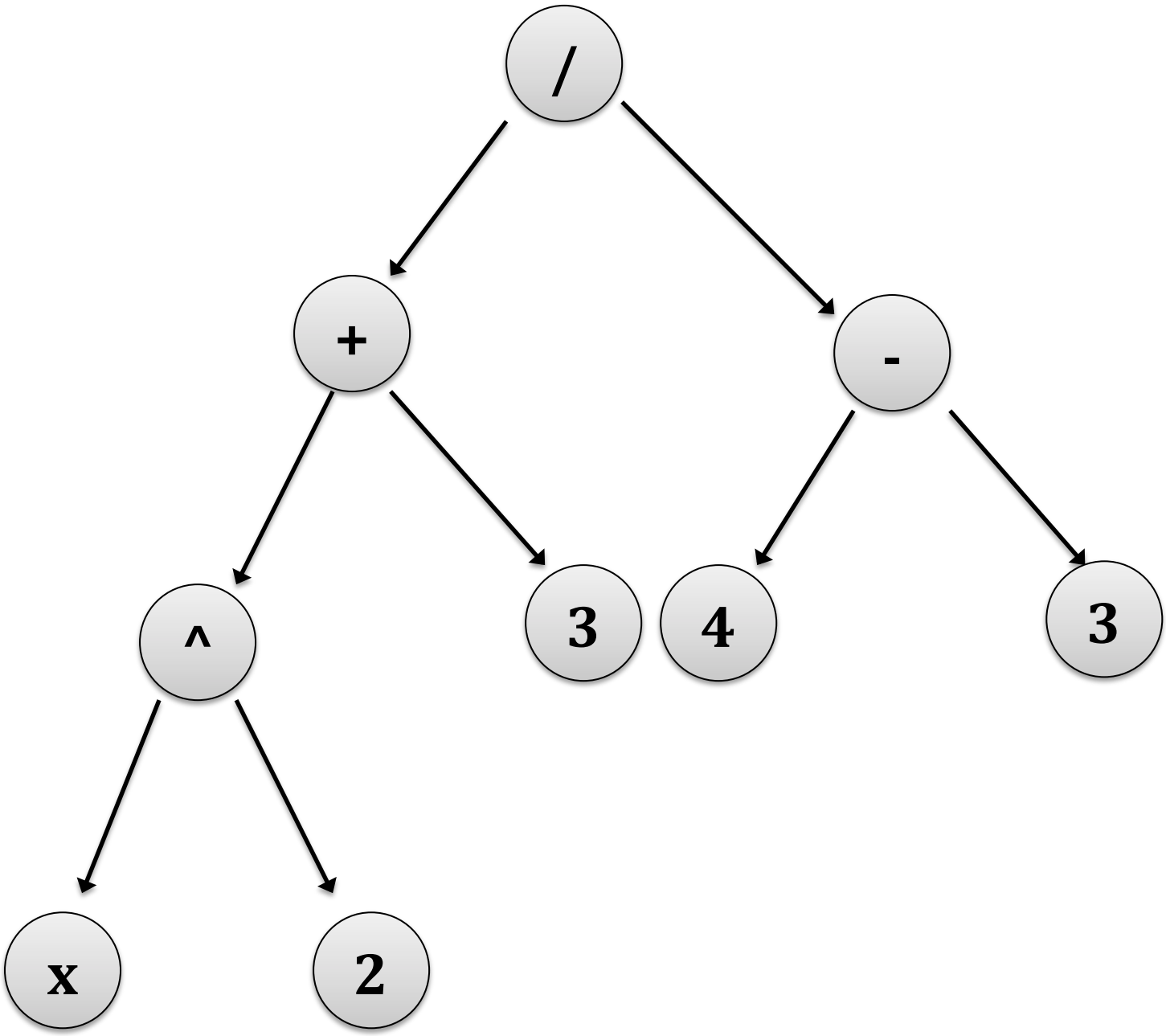
print ('Postfix = ' + str(postfixArray) )
return postfixArray
```

This next function called postfixAlgorithm() is used to convert the infix into postfix notation. The reason for this is that postfix notation does not require brackets. It does this by separating the operands and operators into a form so that an algorithms can more easily determine the operands to operate on.

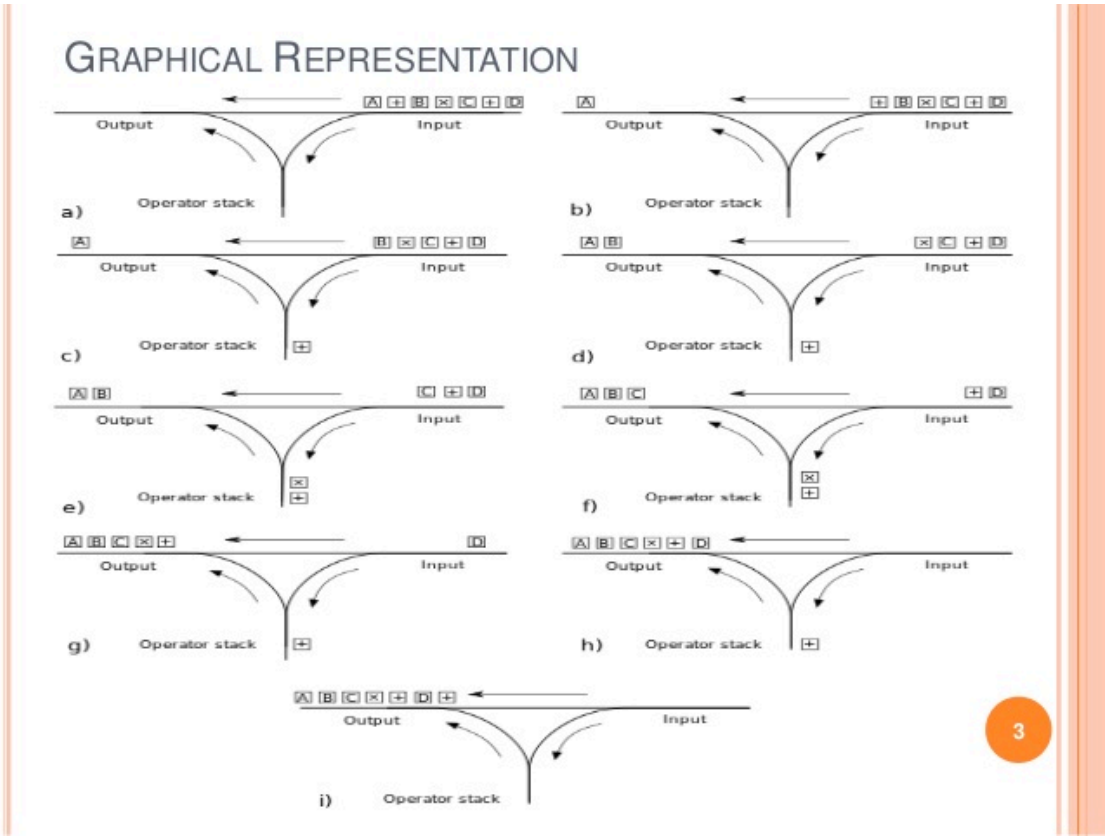
For example ‘(‘, ‘1’, ‘+’, ‘2’, ‘)’ returns ‘1’, ‘2’, ‘+’
Which can be visually represented using the tree as follows:



More complicated inputs such as ‘(‘, ‘x’, ‘^’, ‘2’, ‘+’, ‘3’, ‘)’, ‘/’, ‘(‘, ‘4’, ‘-‘, ‘3’, ‘)’ would be retuned as ‘x’, ‘2’, ‘^’, ‘3’, ‘+’, ‘4’, ‘3’, ‘-‘, ‘/’
Which can be visually represented as:



The function uses a while loop to check every element in the infix array then follows a series of ‘if’ and ‘else’ statements to act accordingly. This function is based off of Dijkstra’s shunting yard algorithm, where if there is an operator as the currentSymbol and there are operators within the tokenStack then it checks the precedence of each operator to decide which operator to append to the postfix array.



was found at: https://image.slidesharecdn.com/shuntingyardalgo-130419115510-phpapp01/95/shunting-yard-algo-3-638.jpg?cb=1366372591

A while loop is used to go through every element in the infix individually. The current symbol is the infix[counter] each time.

```
while counter < inputLength:

    currentSymbol = infix[counter]

    if self.isNumber(currentSymbol) :
        postfixArray.append(currentSymbol)
        counter = counter + 1

    elif currentSymbol in ['%']:
        tokenStack.append(currentSymbol)
        counter = counter + 1

    elif currentSymbol in functions:
        tokenStack.append(currentSymbol)
        counter = counter + 1

    elif currentSymbol in self.operators:
        counter = counter + 1

        # using bidmas to check precedence of operators to know order of operations
        if currentSymbol in leftAssociativeOperators:
            while len(tokenStack)>0 and tokenStack[-1] in self.operators and \
            self.getTokenPrecedence(tokenStack[-1]) >= self.getTokenPrecedence(currentSymbol):
                postfixArray.append(tokenStack.pop())
            tokenStack.append(currentSymbol)

        else:
            while len(tokenStack) > 0 and tokenStack[-1] in self.operators and \
```

```
                    self.getTokenPrecedence(tokenStack[-1]) > self.getTokenPrecedence(
                currentSymbol):
            postfixArray.append(tokenStack.pop())
        tokenStack.append(currentSymbol)


    elif currentSymbol == leftParentheses:
        counter = counter + 1
        if len(tokenStack) > 0 and tokenStack[-1] in self.functions:
            postfixArray.append(self.terminalToken)
        tokenStack.append(currentSymbol)

    elif currentSymbol == rightParentheses:
        counter = counter + 1

        while len(tokenStack) > 0 and tokenStack[-1] != leftParentheses:
            postfixArray.append(tokenStack.pop())

        if len(tokenStack) > 0 and tokenStack[-1] != leftParentheses:
            print ('Error occurred, mismatch of parentheses')

        else:
            tokenStack.pop()

        if len(tokenStack) > 0 and tokenStack[-1] in self.functions:
            postfixArray.append(tokenStack.pop())

    else:
        # Its none of the others therefore it must be a variable
        postfixArray.append(currentSymbol)
        counter = counter + 1


while len(tokenStack) > 0:
    operator = tokenStack.pop()
    postfixArray.append(operator)

print ('Postfix = ' + str(postfixArray) )
return postfixArray
```

- ```
  def isNumber(self,currentSymbol):

      number = (re.search('^\d*.{0,1}\d+$', currentSymbol))
      return number != None
  ```

- isNumber() is a boolean function which uses regular expressions to determine whether or not the input is numerical. It checks if all of the characters are digits and if there is a '.' All the characters after it are digits also. The function then returns 'True' or 'False' as to whether or not the input follows those conditions. There are many methods for this, in one of my earlier draughts the function looked like this:

  ```
  def isNumber(self,currentSymbol):

      try:
          float(currentSymbol)
          return True
      except ValueError:
          pass

      try:
          import unicodedata
          unicodedata.numeric(currentSymbol)
  ```
  ```

```
            return True
        except (TypeError, ValueError):
            pass

        return False
```

- This method used ‘try’ and ‘except’ statements to change the data type of the input. If it was able to change the data type successfully, the input is then numerical otherwise it is not a number.

- Else if currentSymbol is a unary minus (‘%’) append the currentSymbol to TokenStack and increment the counter.
- If the symbol is within the list of premade functions (i.e sin, cos, tan) then add the symbol to the tokenStack and update the counter.
- Else if the current symbol is an operator (‘+’, ‘-‘, ‘*’, ‘/’, ‘^’, ‘%’) increase the counter.
    - If the currentSymbol is a left associative operator
        - While the tokenStack isn’t empty and the top of the stack is an operator and the precedence of the operator at the top of the stack is equal or greater to the operator as the currentSymbol, append the top of the stack to the postfix array then appnd the currentSymbol to the tokenStack.

    - Else while the tokenStack is not empty and the top of the stack is an operator and the precedence of the operator at the top of the stack is greater than that of the currentSymbol, then append the top of the stack to the postfix and add the currentSymbol to the tokenStack.

- Else if the currentSymbol is a left parenthesise (‘(‘) increment the counter
    - If the tokenStack is not empty and the top of the stack is a function, append a terminalToken (‘&’) and add the currentSymbol to the stack. The reason a terminal token is used for a function is so that itcan be used to contain the operand within the function. This allows my later getTree function to determine what to add as a child to my function node.
- If the currentSymbol is a closing parenthesise then increment the counter.
    - Then append the top of the stack to the postfix whilst the stack is not empty and the top of the stack is not an opening parenthesise.
    - If the stack isn’t empty and the top of the stack is not a closing parenthesise then an error has occurred due to a mismatch of parenthesise.
    - Else pop the element at the top of the stack.
    - If the stack is not empty and the top of the stack is a function. Then append the function to the postfix.

- Else append the element to the postfix. If the token doesn't fall within the above 'if' statements then the token must be a variable.
- While the stack is not empty, append the top of the stack to the postfix.
- Print and return the postfixArray.

This can be visually represented using a table like so:

elements	Action performed	Stack status	Postfix expression
A	Push to postfix	(	A
-	Push	(-	A
(	Push	(-(	A
B	Put to postfix	(-(	Ab
+	Push	(-(+	Ab
C	Put to postfix	(-(+	Abc
)	Pop =, put to postfix, pop (	(-	abc+
*	Push	(-*	abc+
D	Put to postfix	(-*	abc+d
/	Pop *, put to postfix, push /	(-/	abc+d*f
F	Put to postfix	(-/	abc+d*f
)	Pop / and -	empty	abc+d*f/-

Found at https://image.slidesharecdn.com/lec-9applicationofstack-140316023053-phpapp02/95/application-of-stacks-8-638.jpg?cb=1394937131

Converting postfix into tree

```
def getTree(self, postfix):

    length = len(postfix)
    index = 0
    stack = []

    for token in postfix:
```

```
        if token == 'x':
            stack.append(variableNode(token))

        elif token == self.terminalToken:
            stack.append(token)

        elif self.isNumber(token):
            stack.append(numberNode(float(token)))


        elif token in self.functions:
            root = functionNode(token)
            while len(stack) > 0 and stack[-1] != self.terminalToken:
                root.addChild(stack.pop())
            #to remove the terminal node
            stack.pop()

            stack.append(root)

        elif token == '%':

            root = minusNode(token)
            root.addChild(stack.pop())

            stack.append(root)

        elif token in self.operators:

            root = operatorNode(token)

            right = stack.pop()
            left = stack.pop()

            if token in ['*','+']: # check if operator is commutative

                if type(left) == operatorNode and left.operatorValue == token:
                    for child in left.children:
                        root.addChild(child)

                else:
                    root.addChild(left)

                if type(right) == operatorNode and right.operatorValue == token:
                    for child in right.children:
                        root.addChild(child)

                else:
                    root.addChild(right)

            else:
                root.addChild(left)
                root.addChild(right)


            stack.append(root)

        else:
            print('sorry "'+ token + '" is not a valid input, please try again. ')
            sys.exit()


    return stack.pop()
```

This function is called to convert the postfix into a tree structure so each element can be differentiated and dealt with atomically. Otherwise the

program would have to be able to deal with every possible input, which would be much too large for this project. Because of this structure is allows the program to use recursive algorithms to differentiate, simplify, and evaluate nodes.

A for loop is used to go through each token in the postfix.

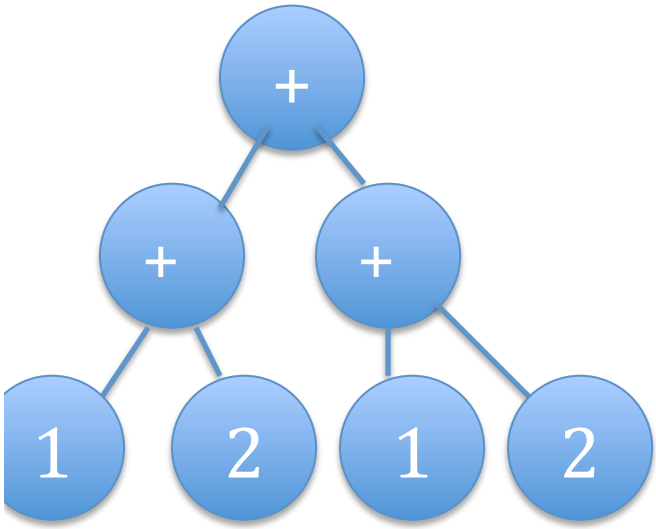
- If the token is a variable (i.e. ‘x’) it is appended to the stack.
- Else if the token is a terminal token (i.e. ‘!’) then append it to the stack.
- Else if the token is a number then it is also added to the stack.

Because these tokens will always either be leaf nodes of the tree (won’t contain any children) or just a reference point in the stack (which is what the terminal token is used for) they are added to the stack to be popped off later and potentially added as children.

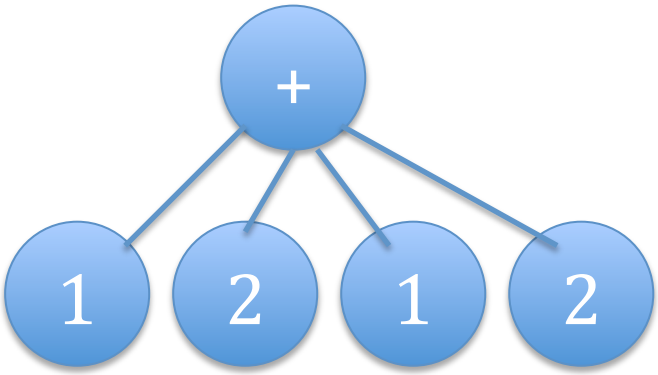
- Else if the token is a function (sin,cos,tan)
    - Create a root node of a functionNode of the token.
    - A while loop is then used until the stack is either empty or the top of the stack is a terminal token, which adds the top of the stack as a child to the root node.
    - The stack is then popped to remove the terminal node.
    - The root is then appended to the stack.
- Else if the token is a ‘%’ (unary minus)
    - A root node is crated from the minusNode of the token.
    - The top of the stack is added as a child to the minusNode
    - The minus node is then placed back onto the stack.
- Else if the token is an operator.
    - A root node is created from an operatorNode of the token.
    - Right is equal to the first item popped of the stack and left is equal to the second item popped off of the stack. This is because stacks use a first in last out structure.
    - If the token is a commutative operator (i.e. ‘+’ or ‘*’):
        - If the left is the same as the operator node then add the children of left to the root.
        - Else add left to the root
        - If the right is the same as the operator then add the children of right to the root.
        - Else add right to the root.

This is to create a nary tree for the commutative operators. The routine looks something like this:

Binary Tree



Nary Tree



- Else add left and right as children to the root because the children are not the same as the root.

- Else (if it is any other operator) add left and right as children.
- Else the operator is not recognised so an error message is printed and the program stops running.
- At the end of each loop the root is appended to the stack.

The root in the stack is returned.

Converting Tree To Text

The mathsExpression then calls the treeToText() function from the treeNodes and prints the tree out as a string to be read.

Simplifying the input tree

The input tree is then simplified to see if the tree can be shortened before differentiating.

Differentiating the input tree

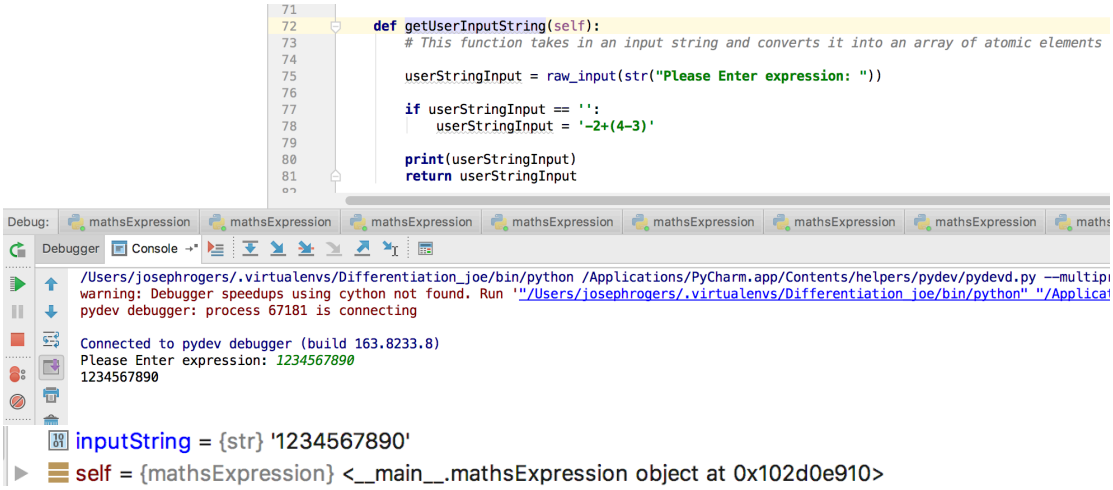
The differentiation routine from the treeNode file is then called on the simplified tree. The derivative of an expression is equal to the derivative of all of its components.

Testing

The way the program has been written is recursive and acts on each part of an expression atomically, because of this if the program can run all of the simple inputs individually with only one operand then it can run complicated inputs with all the operands combined. In the program there is an option to view the whole process of differentiating, for most of these tests I had the program show the whole process.

Input

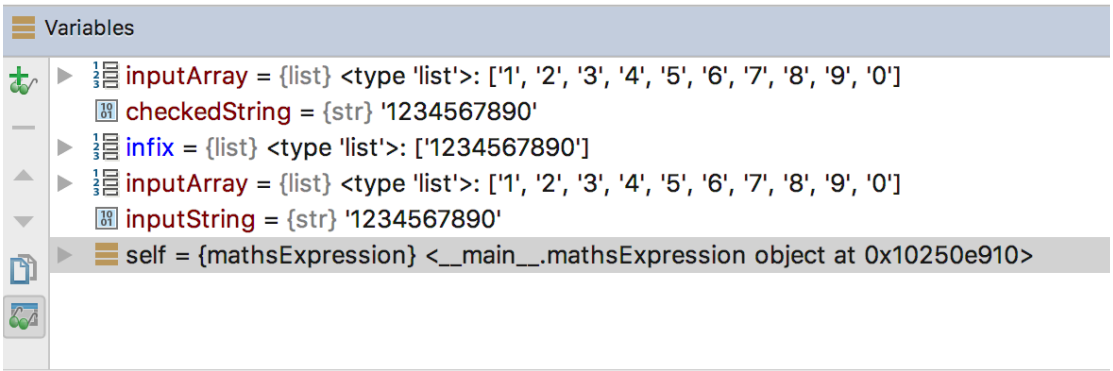
The first thing that was tested was that the program takes the correct input string.



These screenshots show an input of '1234567890' and the inputString being correct.

Array

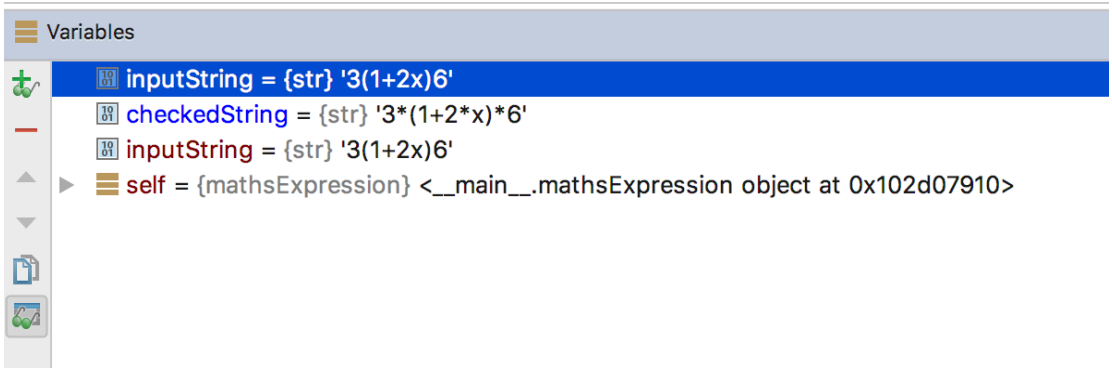
Next the program was tested has to be able to convert the string into an array by separating all of the characters individually.



As shown by the inputArray the program had the correct array.

Implied Multiplication

To text the implied multiplication an input of ‘3(1+2x)6’ was input to test if the function would return ‘*’ between the ‘3’ and ‘(‘, the ‘2’ and ‘x’, and the ‘)’ and ‘6’. So the checked string should return ‘3*(1+2*x)*6’ as shown:

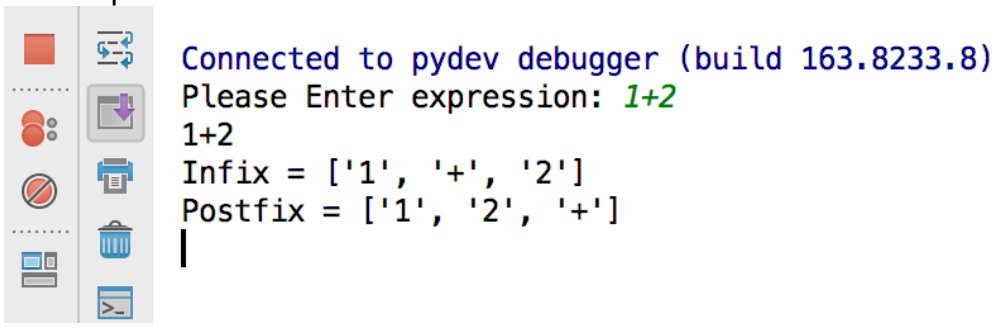


Postfix

The program then had to be tested to see if it could convert the array into infix whilst being able to separate all of the different input types i.e. operators, functions, brackets, variables, functions, along with numbers with one or more digits.

```
Connected to pydev debugger (build 163.8233.8)
Please Enter expression: (1+3x)*sin(153/7x^2)
(1+3x)*sin(153/7x^2)
Infix = ['(', '1', '+', '3', '*', 'x', ')', '*', 'sin', '(', '153', '/', '7', '*', 'x', '^', '2', ')']
```

When testing for the postfix it was firstly tested to see if it would return a correct postfix form.

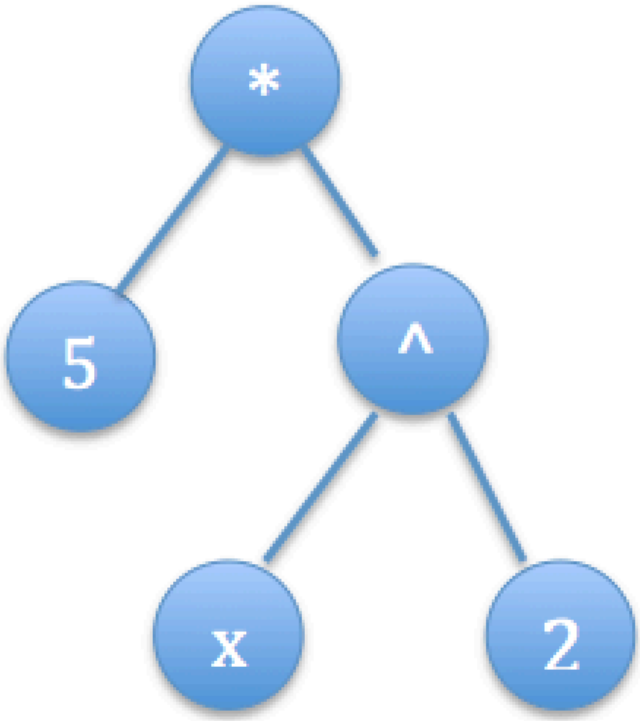


It was then tested using inputs with multiple operators in. It had to determine which operators took precedence to return the correct postfix. More complicated inputs such as ‘(1+2)^4/(3*4x^2)’

```
inputString = {str} '(1+2)^4/(3*4x^2)'
▶ postfix = {list} <type 'list'>: ['1', '2', '+', '4', '^', '3', '4', '*', 'x', '2', '^', '*', '/']
checkedString = {str} '(1+2)^4/(3*4*x^2)'
▶ infix = {list} <type 'list'>: ['(', '1', '+', '2', ')', '^', '4', '/', '(', '3', '*', '4', '*', 'x', '^', '2', ')']
▶ inputArray = {list} <type 'list'>: ['(', '1', '+', '2', ')', '^', '4', '/', '(', '3', '*', '4', '*', 'x', '^', '2', ')']
inputString = {str} '(1+2)^4/(3*4x^2)'
▶ postfix = {list} <type 'list'>: ['1', '2', '+', '4', '^', '3', '4', '*', 'x', '2', '^', '*', '/']
▶ self = {mathsExpression} <__main__.mathsExpression object at 0x102d07910>
```

Get Tree

When testing if the get tree function is correct. It the lists are checked to see if they contain the correct children.'5X^2' should look like this:



```
expressionTree = {operatorNode} <treenodes.operatorNode object at 0x10312ff10>
  children = {list} <type 'list'>: [<treenodes.numberNode object at 0x10312fe10>, <treenodes.operatorNode object at 0x10312fed0>]
    __len__ = {int} 2
    0 = {numberNode} <treenodes.numberNode object at 0x10312fe10>
      children = {list} <type 'list'>: []
        __len__ = {int} 0
      value = {float} 5.0
    1 = {operatorNode} <treenodes.operatorNode object at 0x10312fed0>
      children = {list} <type 'list'>: [<treenodes.variableNode object at 0x10312fe50>, <treenodes.numberNode object at 0x10312fe90>]
        __len__ = {int} 2
        0 = {variableNode} <treenodes.variableNode object at 0x10312fe50>
          children = {list} <type 'list'>: []
            __len__ = {int} 0
          variable = {str} 'x'
        1 = {numberNode} <treenodes.numberNode object at 0x10312fe90>
          children = {list} <type 'list'>: []
            __len__ = {int} 0
          value = {float} 2.0
      operatorValue = {str} '^'
      precedence = {int} 9
  operatorValue = {str} '*'
  precedence = {int} 7
```

As can be seen here the operator '*' has two children '5' and '^'. The 5 has no children and the '^' has two children which are an 'x' and a '2'.

Tree To Text

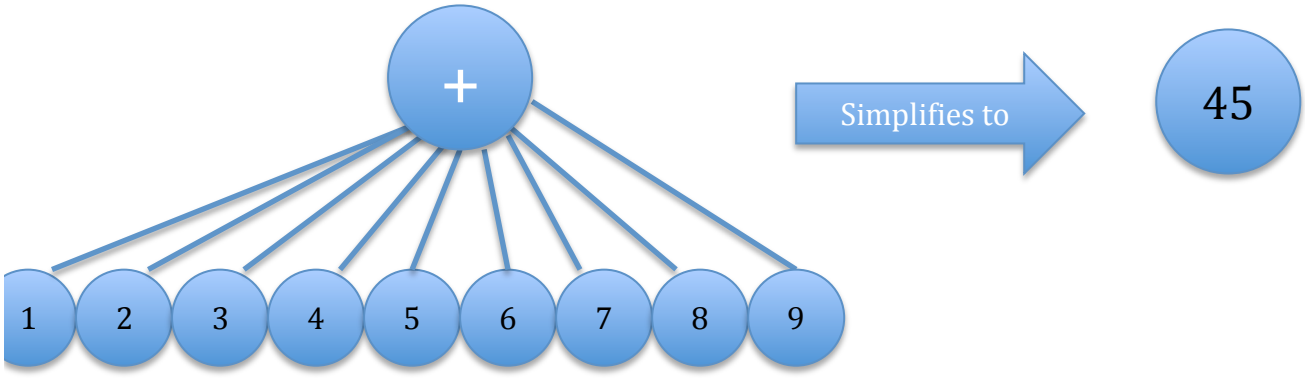
The next test is to test the treeToText() function so it returns the tree in a form legible to people.

```
Connected to pydev debugger (build 163.8233.8)
Please Enter expression: 5x^2
5x^2
Infix = ['5', '*', 'x', '^', '2']
Postfix = ['5', 'x', '2', '^', '*']
simplify the following tree : 5.0*x^2.0 to:
```

As shown the tree to text returns the tree above as '5.0*x^2.0'.

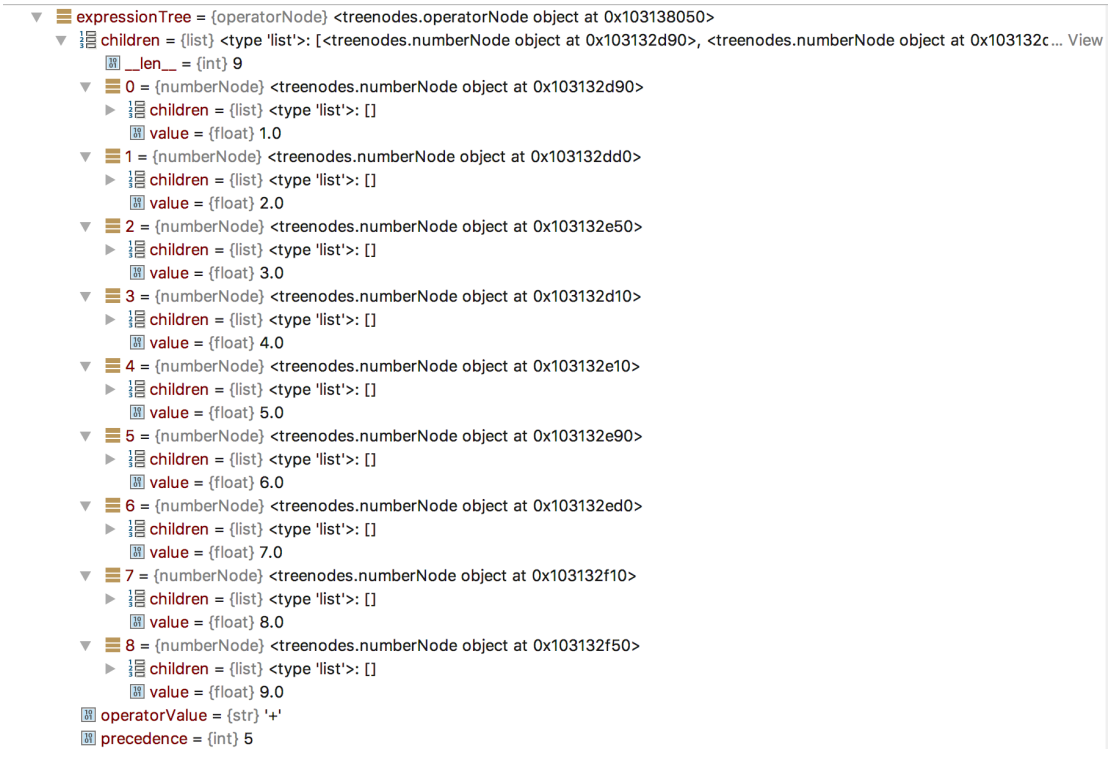
The next test is for seeing if the program is able to simplify trees.

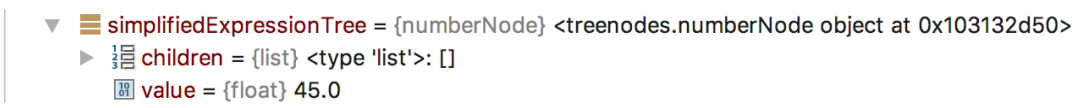
For simplifying an nary tree an input of '1+2+3+4+5+6+7+8+9'



```
xpression
sers/josephrogers/.virtualenvs/Differentiation_joe/bin/python /Users/josephrogers/PycharmProjects/
ease Enter expression: 1+2+3+4+5+6+7+8+9
2+3+4+5+6+7+8+9
fix = ['1', '+', '2', '+', '3', '+', '4', '+', '5', '+', '6', '+', '7', '+', '8', '+', '9']
stfix = ['1', '2', '+', '3', '+', '4', '+', '5', '+', '6', '+', '7', '+', '8', '+', '9', '+']
mplify the following tree : 1.0+2.0+3.0+4.0+5.0+6.0+7.0+8.0+9.0 to:
mplified input is : 45.0
```

This test shows the original tree and simplified tree converted to text. The following screen shots show that the tree was an nary tree by showing that the single operator node has inherited the children from the other operator nodes. The simplified tree shows that the operator has been applied to the input arguments and the tree has been replaced with a single number node of '45'.





Differentiation

To test the differentiation routine, each type input was tested individually. The first thing tested was differentiating numerical values (which should always differentiate to zero).

```
would you like to show full process? y/n: n
Please Enter expression: 2x^2
derivative : 4.0*x
Expression Saved.

Would you like to:
1. Enter another expression.
2. Return to main menu
3. Exit the program
Enter menu option:
```

```
Please Enter expression: 1
1
Infix = ['1']
Postfix = ['1']
simplify the following tree : 1.0 to:
simplified input is : 1.0
True
Input value = 1.0
Output value = 0
```

```
Please Enter expression: 3000
3000
Infix = ['3000']
Postfix = ['3000']
simplify the following tree : 3000.0 to:
simplified input is : 3000.0
True
Input value = 3000.0
Output value = 0
```

It was then tested to differentiate variables (x should always differentiate to 1).

```
Please Enter expression: x
x
Infix = ['x']
Postfix = ['x']
simplify the following tree : x to:
simplified input is : x
False
Input value = x
Output value = 1.0
```

Functions were then tested.

```
Please Enter expression: sin(x)
sin(x)
Infix = ['sin', '(', 'x', ')']
Postfix = ['&', 'x', 'sin']
simplify the following tree : sin(x) to:
simplified input is : sin(x)
False
Input value = sin(x)
Output value = cos(x)*1.0
Simplified output: cos(x)

Please Enter expression: tan(x)
tan(x)
Infix = ['tan', '(', 'x', ')']
Postfix = ['&', 'x', 'tan']
simplify the following tree : tan(x) to:
simplified input is : tan(x)
False
Input value = tan(x)
Output value = sec(x)^2.0*1.0
Simplified output: sec(x)^2.0
True
```

```
Please Enter expression: cos(x)
cos(x)
Infix = ['cos', '(', 'x', ')']
Postfix = ['&', 'x', 'cos']
simplify the following tree : cos(x) to:
simplified input is : cos(x)
False
Input value = cos(x)
Output value = -(sin(x))*1.0
Simplified output: -(sin(x))
True
```

The differentiation routine for the chain rule, product rule and quotient rule was then tested.

```
Please Enter expression: 3(x+1)^4
3(x+1)^4
Infix = ['3', '*', '(', 'x', '+', '1', ')', '^', '4']
Postfix = ['3', 'x', '1', '+', '4', '^', '*']
simplify the following tree : 3.0*(x+1.0)^4.0 to:
simplified input is : 3.0*(x+1.0)^4.0
False
Input value = 3.0*(x+1.0)^4.0
Output value = 3.0*4.0*(1.0+0)*(x+1.0)^(4.0-1.0)+((x+1.0)^4.0*0)
Simplified output: 12.0*(x+1.0)^3.0
True
```

```
Please Enter expression: x*34
x*34
Infix = ['x', '*', '34']
Postfix = ['x', '34', '*']
simplify the following tree : x*34.0 to:
simplified input is : 34.0*x
False
Input value = x*34.0
Output value = x*0+(34.0*1.0)
Simplified output: 34.0
True
```

```
Please Enter expression: (x-2)/(3+x)
(x-2)/(3+x)
Infix = ['(', 'x', '-', '2', ')', '/', '(', '3', '+', 'x', ')']
Postfix = ['x', '2', '-', '3', 'x', '+', '/']
simplify the following tree : (x-2.0)/(3.0+x) to:
simplified input is : (x-2.0)/(x+3.0)
False
Input value = (x-2.0)/(3.0+x)
Output value = (3.0+x*(1.0-0)-x-2.0*(0+1.0))/(3.0+x)^2.0
Simplified output: (x+3.0-(x-2.0))/(x+3.0)^2.0
True
```

Menu and Dictionary

```
would you like to show full process? y/n: n
Please Enter expression: 3(x^2-3)^4
derivative : 3.0*4.0*(2.0*x-0)*(x^2.0-3.0)^3.0
Expression Saved.

Would you like to:
1. Enter another expression.
2. Return to main menu
3. Exit the program
Enter menu option: 1

would you like to show full process? y/n: n
Please Enter expression: 3x
derivative : 3.0
Expression Saved.

Would you like to:
1. Enter another expression.
2. Return to main menu
3. Exit the program
Enter menu option: 1

would you like to show full process? y/n: n
Please Enter expression: 3x/2
derivative : 1.5
Expression Saved.

Would you like to:
1. Enter another expression.
2. Return to main menu
3. Exit the program
Enter menu option: 2

Menu options
1. Differentiate expression
2. View previous entries
3. Exit program
Enter menu option: 2

Recent History:
3x/2 Differentiated to: 1.5
3x Differentiated to: 3.0
3(x^2-3)^4 Differentiated to: 3.0*4.0*(2.0*x-0)*(x^2.0-3.0)^3.0
```

Complicated and erroneous inputs

Then finally more complicated inputs combining multiple differentiation routines.

```
Please Enter expression: (3x^2-12)/(2x-1)
(3x^2-12)/(2x-1)
Infix = ['(', '3', '*', 'x', '^', '2', '-', '12', ')', '/', '(', '2', '*', 'x', '-', '1', ')']
Postfix = ['3', 'x', '2', '^', '*', '12', '-', '2', 'x', '*', '1', '-', '/']
simplify the following tree : (3.0*x^2.0-12.0)/(2.0*x-1.0) to:
simplified input is : (3.0*x^2.0-12.0)/(2.0*x-1.0)
False
Input value = (3.0*x^2.0-12.0)/(2.0*x-1.0)
Output value = (2.0*x-1.0*(3.0*2.0*1.0*x^(2.0-1.0)+(x^2.0*0)-0)-3.0*x^2.0-12.0*(2.0*1.0+(x*0)-0))/(2.0*x-1.0)^2.0
Simplified output: (2.0*x-1.0*(3.0*2.0*x-0)-2.0*(3.0*x^2.0-12.0))/(2.0*x-1.0)^2.0
True
```

Then finally invalid inputs were tested. And also inputs intended to make the program fall over.

```
Please Enter expression: hello
hello
Infix = ['hello']
Postfix = ['hello']
sorry "hello" is not a valid input, please try again.

Process finished with exit code 0
```

```
Please Enter expression: 2x^2+3x+e
2x^2+3x+e
Infix = ['2', '*', 'x', '^', '2', '+', '3', '*', 'x', '+', 'e']
Postfix = ['2', 'x', '2', '^', '*', '3', 'x', '*', '+', 'e', '+']
sorry "e" is not a valid input, please try again.
```

```
Please Enter expression: sin(X
sin(X
Error, mismatching parenthesise
```

```
ease Enter expression: 2(x^2+4x)/3x^2
x^2+4x)/3x^2
ıfix = ['2', '*', '(', 'x', '^', '2', '+', '4', '*', 'x', ')', '/', '3', '*', 'x', '^', '2']
ıstfix = ['2', 'x', '2', '^', '4', '*', 'x', '+', '*', '3', '/', '*', 'x', '2', '^', '*']
mplify the following tree : 2.0*(x^2.0+4.0*x)/3.0*x^2.0 to:
mplified input is : 2.0*(4.0*x^2.0*x)/3.0*x^2.0
lse
nput value = 2.0*(x^2.0+4.0*x)/3.0*x^2.0
ıtput value = 2.0*((x^2.0+4.0*x)/3.0*2.0*1.0*x^(2.0-1.0)+(x^2.0*(3.0*(x^2.0*(4.0*1.0+(x*0))+(4.0*x*2.0*1.0*x^(2.0-1.0)))-x^2.0*4.0*x*0)/3.0^2.0))+((x^2.0*4.0*x)/3.0*x^2.0*0)
mplified output: 2.0*((4.0*x^2.0*x)/3.0*2.0*x+(x^2.0*(3.0*(4.0*x^2.0+(4.0*x*2.0*x))-0)/9.0))
ue

ocess finished with exit code 0
```

Evaluation

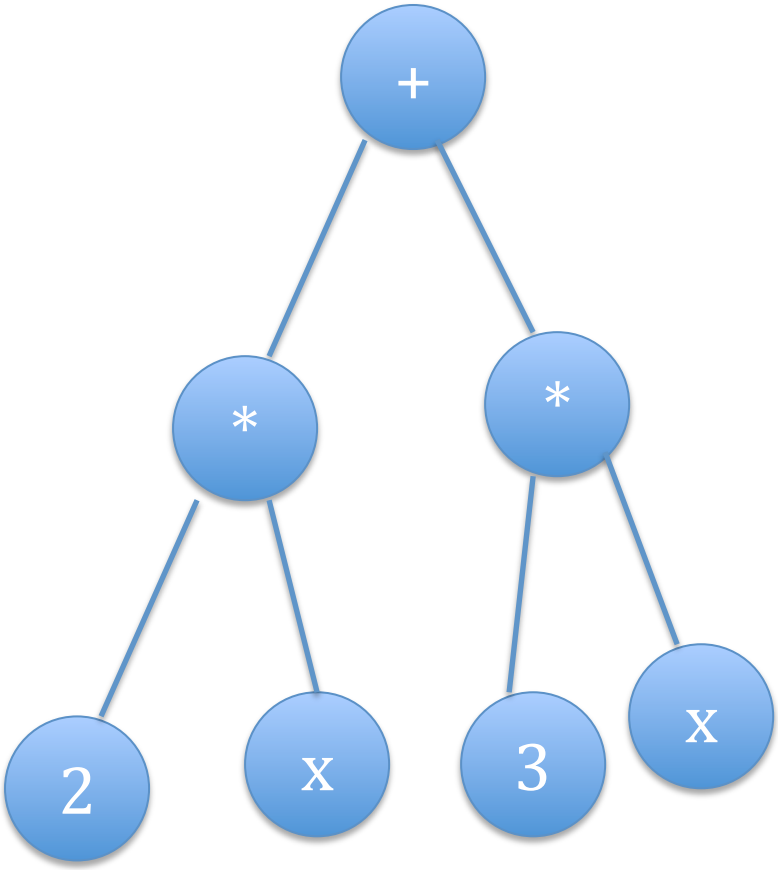
Requirements met

Functional Requirements	Was this achieved?
FR 1.1 Users can type 'variables' which can conform to regular expression [a-z, A-Z]*.	This requirement has not been completely met. At the moment the only variable which can be input is 'x'. On the one hand it would not be too difficult to write a function, which just replaces the variable with the character 'x' similar to the unary minus '%' replacing '-'. The problem lies when the user wishes to use multiple variables in a singular expression, for example differentiating y in terms of x is dy/dx which would make the trees much more complicated and a lot more coding would have to be added. In the time frame I had it was decided not to put this feature into the project, instead the program is able to differentiate a single variable.
FR 1.2 Users can type any of the following operators +, -, /, *, ^ (plus, minus, divide, multiply, and power).	Yes
FR 1.2.1 operator '-' can be unary or binary whereas operators /, ^, + and * must be binary.	This requirement has been met, however in certain cases '+' and '*' are not binary due to their commutative nature and are treated as nary however this is an improvement at it increases efficiency and makes simplification routines for them more simple.
FR 1.3 Users can type any of the following functions 'sin', 'cos', 'tan'.	This requirement has been met. Because of the nature of the object orientated programming and the trees recursive structure if any new functions were to be added, a new statement would need to be added in the functioNode class to deal with the new function and nothing else would have to be changed. This makes it very easy to add new functions such as 'arcsin' or 'log'.
FR 1.3.1 Each function listed in 1.3 are unary functions so that the argument should be contained within brackets (e.g. sin(x)).	Yes

FR 1.5 Users can type any of the following brackets (,)	Yes
FR 1.6 Users can type any sequence of characters whether or not it forms a syntactically correct expression.	The program will either loop or exit if there is an invalid input.
FR 1.7 The mathematical expression is analysed using a recursive routine, which contains recursive syntax rules.	Yes
NFR 1.8 The program should run fast and efficiently to return the answer to the given input quickly.	Yes
NFR 1.9 The program must be very simple for the user to understand.	Yes
FR 2.0 The program must have a method of storing/saving recent inputs.	The program uses a dictionary which can be used as a short term storage.
NFR 2.1 The program must have a simple user interface.	Yes

What could be improved?

Firstly the simplification routines could be much better, however the more input are to be simplified the more complicated and bigger the coding gets. One solution to further simplify answers is to add a simplify Boolean to each node and to recursively simplify the tree until all nodes return true. However this will still not fully simplify a tree. One example of this would be when a tree looks like this:



In order to simplify this tree a routine would have to be introduced which would be able to recognise that both of these can in fact be added to each other. But in order to do this it wouldn't just have to check the children of the '+' node, it would then have to check its children's children and then determine whether or not these children can be added together and them add them together. This is one of many examples where simplification routines would have to become increasingly more complicated in order to continue simplifying further.

A second improvement I would implement is a better user interface. Although this project is mainly focused on its function, if I were to have more time I would introduce a simple user interface.

A third improvement I could make is to introduce a database to remember outputs of previous inputs so the remembered output can be retrieved from the database from previous times theprogram was run.

Potential Extensions

This program could be extended to have other functions, which can be useful to students who study A-level mathematics. For example I was thinking about

creating an extension where the student could input an answer and the program would then return whether or not their answer is correct. What I started looked like this:

```
def compareAnswers(self, trueAnswer, userAnswer, domainStart, domainEnd, numPoints, failureRate):

    validCount = 0
    invalidCount = 0
    step = (domainEnd-domainStart)/numPoints
    x = domainStart

    for x in range(0, numPoints):

        if self.evaluateExpression(trueAnswer, x) == self.evaluateExpression(userAnswer, x):
            validCount = validCount + 1
            x = x + step

        else:
            invalidCount = invalidCount + 1
            x = x + step

    if invalidCount > 5:
        print('Sorry, this answer is incorrect. Would you like to see the solution?')

    else:
        print('This answer is correct! :)')

    return not invalidCount > failureRate
```

What this function does is replace variables with values ranging from 1 to 50 and if the two evaluated expression are equal to eachother then valid count increases. Otherwise the invalid count increases, as long as the invalid cound is less than five then assume both inputs are mathematically identical.

The reason the invalid counted is allowed to go up to 5 is because certain expressions might be the same but the replaced variable for one might change its mathematical value. For example x^2-6x+9 is equal to (x-3)*(x-3), although both of these are mathematically the same if x were to equal '3' then the factorised form will return 0 and the original form will return 36. So as long as invalidCount is less than 5 it is assumed they are both equal to eachother. The more values of x checked the more reliable this function would be, but checking more values would compremise efficiency of the program. In texting I checked with 50 values of x.

Another potential expression is to have a student input questions, which can then be asked later and check if the students' answers are correct fior the questions.

Code