# Build Binary Expression Tree in Python

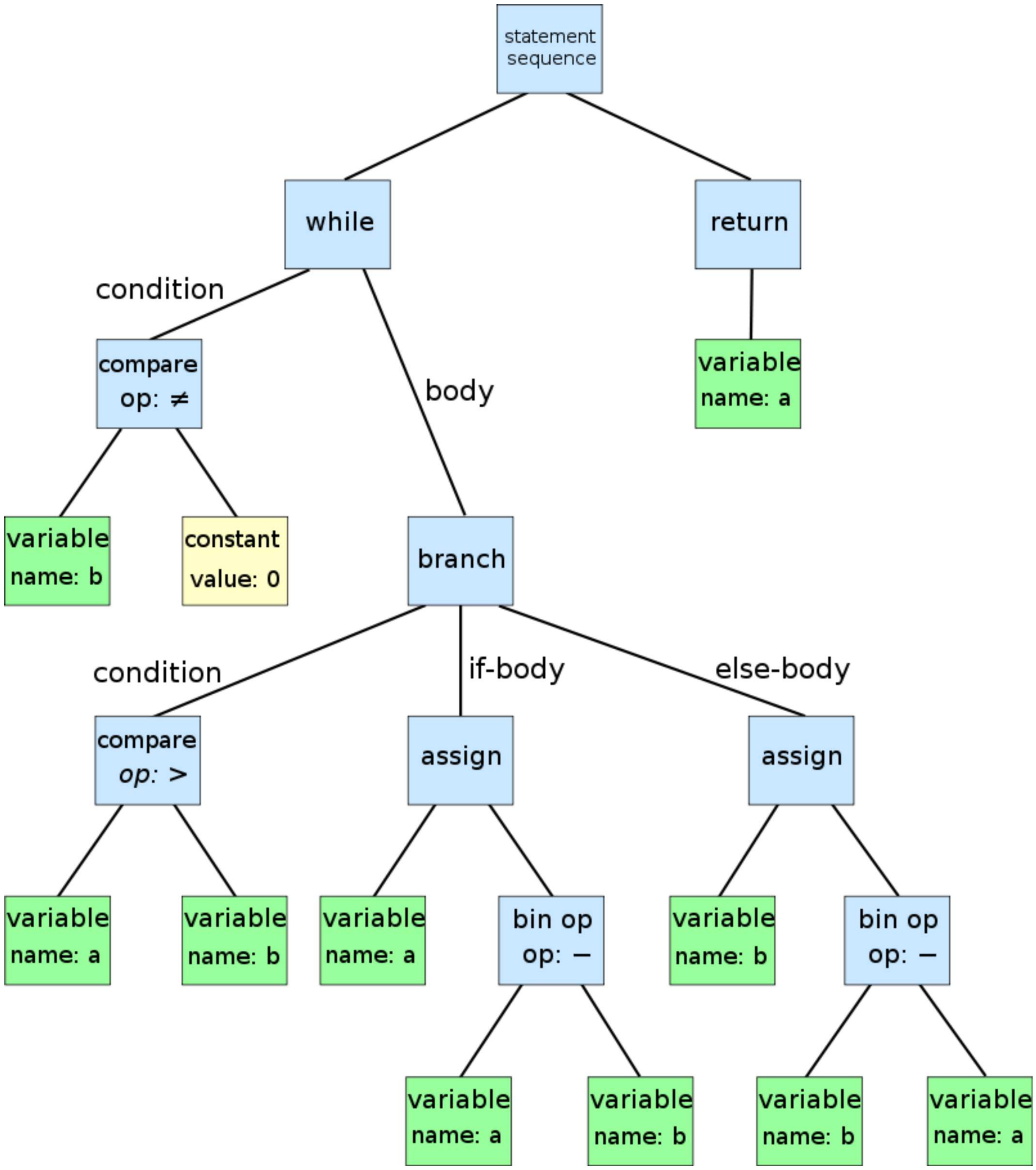
Sukhrob Golibboev Follow
Dec 15, 2019 · 5 min read ★

Photo by Safar Safarov on Unsplash

Have you ever wondered how a programming language reads expressions in source code and evaluates them to run the program? Well, if you said yes please continue reading this post.

• • •

It turns out the compiler or interpreter parses the source code and builds a data structure for representing the expressions in a way it can evaluate. This structure is called an **abstract syntax tree (AST)** and nearly all programming languages use ASTs during program compilation or execution. In this post, I'll show you a simplified type of syntax tree to explain the concepts which are fundamental to ASTs, even though they are much more complex.



Abstract Syntax Tree. Source: Wikipedia

## Trees are everywhere

In computer science, the tree structure is widely used to solve many real-world problems. Here is a list of use cases of tree data structure in various applications:

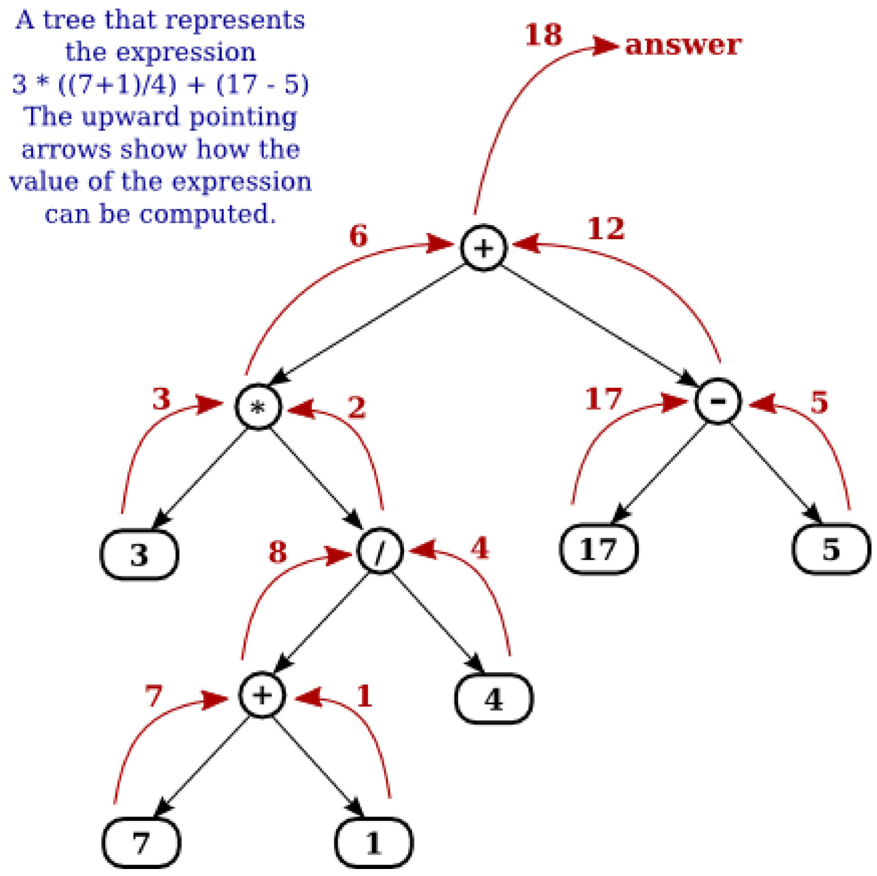
- Are you using a social network? A tree structure is used to suggest a new friend or lets you search people among 2.5 billion people less than a second.
- Databases use a tree for indexing its elements for fast accessing when a query is requested.
- Have you heard of decision tree-based learning? A machine-learning algorithm.
- Do you know the domain name server (DNS)? It also uses a tree data structure.
- All software applications in the world somehow they are using map/hashmap. Hashmap has its internal implementation is an AVL tree.

As you can see there are different kinds of trees they can be implemented to a specific application. For this post, we use a ***binary expression tree*** to tackle the problem and evaluate an expression.

. . .

## Binary Expression Tree.

Binary Expression Tree — is a specific kind of a binary tree used to represent expressions. Two common types of expressions that a binary expression tree can represent are algebraic and boolean expressions. In this article, we will talk about how we can first transform an algebraic expression into a tree, then evaluate the expression tree to determine its value.


A tree that represents the expression 3 * ((7+1)/4) + (17 - 5) The upward pointing arrows show how the value of the expression can be computed.
18
answer
6
12
3
2
17
5
8
4
7
1
+
*
-
/
+
3
17
5
4
7
1

Binary expression tree representation. Source

## How Humans Reads

First, let's take look at the below expression. We read this expression left to right and evaluate it as we go and in the end, we have the final result. While we humans are reading the expression we know which term should be evaluated first and later use its value to evaluate the next term that might use the result of the previous term.

```
12 + 6 - 5 * 2 = 8
```

## Examples

In the following first example, we just read from left to right and evaluate its expression in one go.

```
First example: 8 - 2 = 6
```

But what if we have something like this, a less simple example(second example):

```
Second example: 8 - 2 * 3 = 2
```

Now we evaluate term (2*3) before we do (8–2) because we know `multiplication` has higher precedence order than `subtraction`. So we evaluate (2*3) first then (8–2).

> *As we read the expressions, we should consider if we need to evaluate the specific term now or later, it depends on the precedence order of the operators used in the expression, the higher precedence the term has the earlier it is evaluated.*

We also need to consider the associativity rule. You can read about precedence order and associativity rule if you need a refresher.

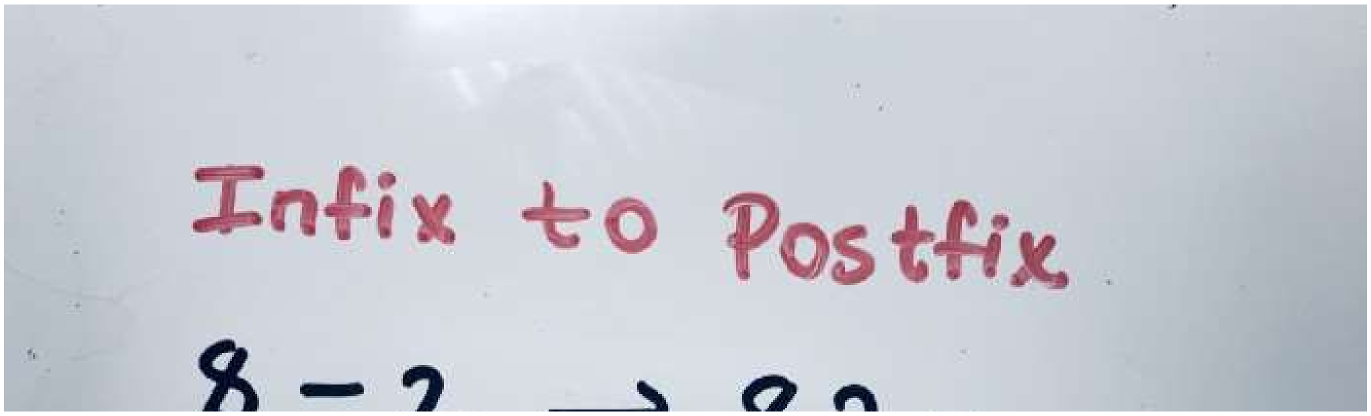
So why we are talking about these simple rules that we all learned when we were in elementary school. Because this is how our mind reads and calculates the expression once we've internalized the rules of arithmetic from primary school. Let's go back to computers, how the compiler reads the expression?

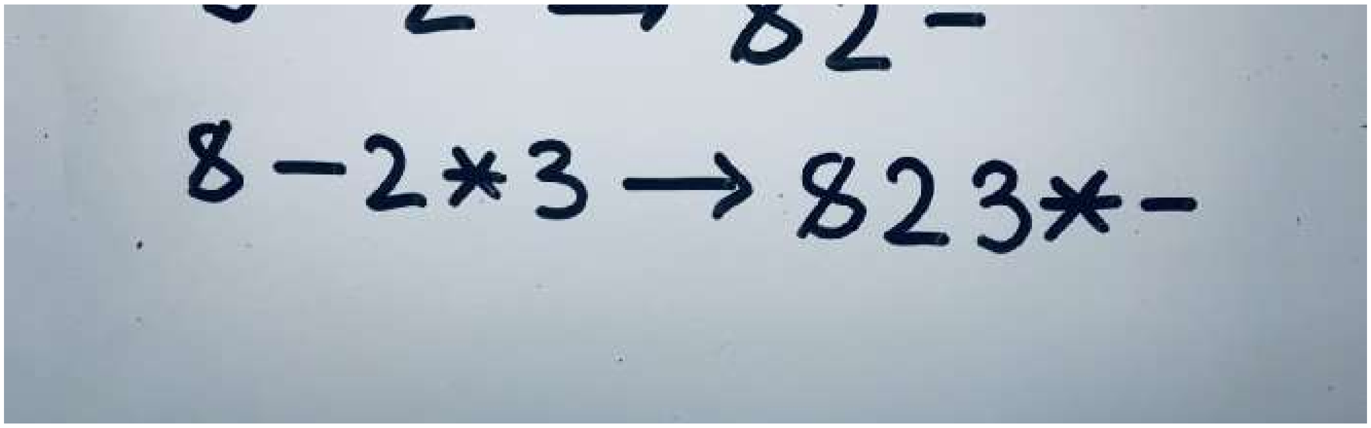
### How Compiler Reads

We just talked about how a human brain reads the expression, we read it left to right and we know which term needs to be evaluated when. and it is called ***infix expression***. But if we try to replicate the same process with computers, and write a code to read infix expression and build a tree based on it, it would take much longer than if we use **postfix expression** to build the tree to evaluate the expression.

### Infix and Postfix Expression

Infix expression is the exact expression user inputs operators are in between of operands, on the other hand, in postfix expression operators come after operands.





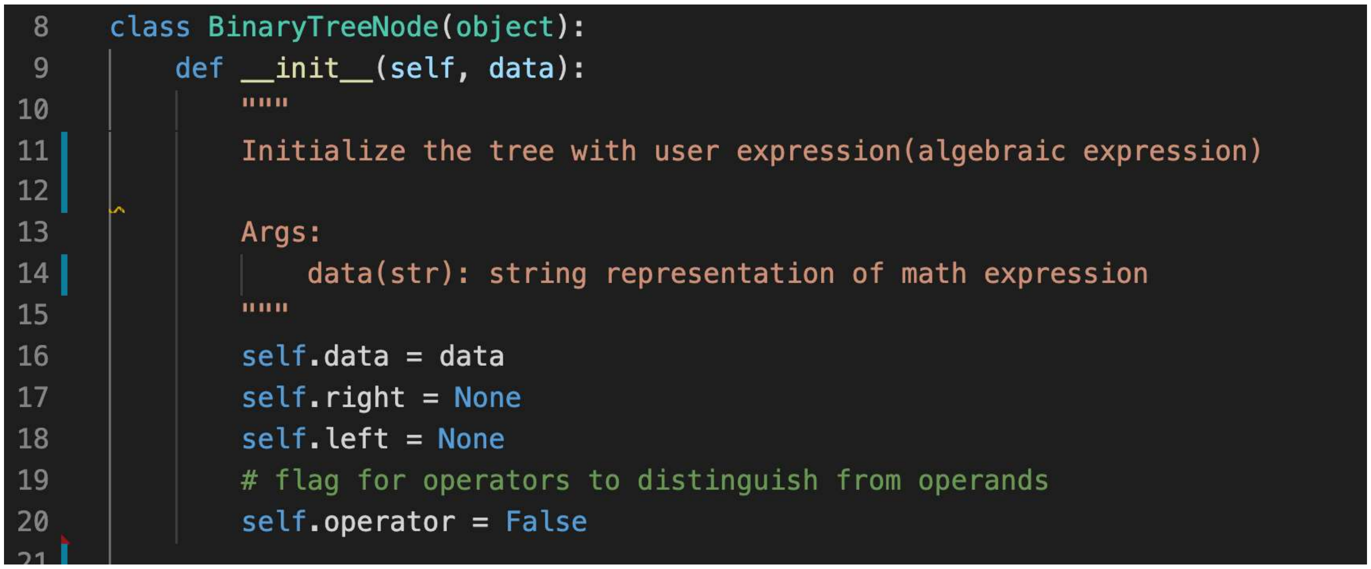
Infix to postfix conversion.

Since we are using postfix expression building the tree we need to convert our infix expression into postfix. I am not going to cover how we do that, because infix to postfix conversion alone can be another separate blog post. Instead, I will show the code I wrote for. But you can watch this amazing video on how to convert infix to postfix.

. . .

## Build the Tree

The main property of building binary expression tree — operators will be the branches(interior nodes) and the operands(constants/numbers) will be stored in leaf nodes of the tree. So, each parent node guaranteed to have two children: right and left. Each node will store data, left and right child.



```python
class BinaryTreeNode(object):
    def __init__(self, data):
        """
        Initialize the tree with user expression(algebraic expression)

        Args:
            data(str): string representation of math expression
        """
        self.data = data
        self.right = None
        self.left = None
        # flag for operators to distinguish from operands
        self.operator = False
```

Binary Expression Tree node init method

If we want to build the tree using the infix expression. First, we need to determine which operator is going to the root in the tree expression or the operator we evaluate at the last. But as we talked before it is a time-consuming operation so we convert it from infix to postfix expression. Below is the implementation.

```python
def infix_to_postfix(self, infix_input: list) -> list:
    """
    Converts infix expression to postfix.
    Args:
        infix_input(list): infix expression user entered
    """

    # precedence order and associativity helps to determine which
    # expression is needs to be calculated first
    precedence_order = {'+': 0, '-': 0, '*': 1, '/': 1, '^': 2}
    associativity = {'+': "LR", '-': "LR", '*': "LR", '/': "LR", '^': "RL"}
    # clean the infix expression
    clean_infix = self._clean_input(infix_input)

    i = 0
    postfix = []
    operators = "+-/*^"
    stack = deque()
    while i < len(clean_infix):

        char = clean_infix[i]
        # print(f"char: {char}")
        # check if char is operator
        if char in operators:
            # check if the stack is empty or the top element is '('
            if len(stack) == 0 or stack[0] == '(':
                # just push the operator into stack
                stack.appendleft(char)
                i += 1
            # otherwise compare the curr char with top of the element
            else:
                # peek the top element
                top_element = stack[0]
                # check for precedence
                # if they have equal precedence
                if precedence_order[char] == precedence_order[top_element]:
                    # check for associativity
                    if associativity[char] == "LR":
                        # pop the top of the stack and add to the postfix
                        popped_element = stack.popleft()
                        postfix.append(popped_element)
                    # if associativity of char is Right to left
                    elif associativity[char] == "RL":
                        # push the new operator to the stack
                        stack.appendleft(char)
                        i += 1
                elif precedence_order[char] > precedence_order[top_element]:
                    # push the char into stack
                    stack.appendleft(char)
                    i += 1
                elif precedence_order[char] < precedence_order[top_element]:
                    # pop the top element
                    popped_element = stack.popleft()
                    postfix.append(popped_element)
        elif char == '(':
            # add it to the stack
            stack.appendleft(char)
            i += 1
        elif char == ')':
            top_element = stack[0]
            while top_element != '(':
                popped_element = stack.popleft()
                postfix.append(popped_element)
                # update the top element
                top_element = stack[0]
            # now we pop opening parenthases and discard it
            stack.popleft()
            i += 1
        # char is operand
        else:
            postfix.append(char)
            i += 1
        #     print(postfix)
        # print(f"stack: {stack}")

    # empty the stack
    if len(stack) > 0:
        for i in range(len(stack)):
            postfix.append(stack.popleft())
    # while len(stack) > 0:
    #     postfix.append(stack.popleft())

    return postfix
```

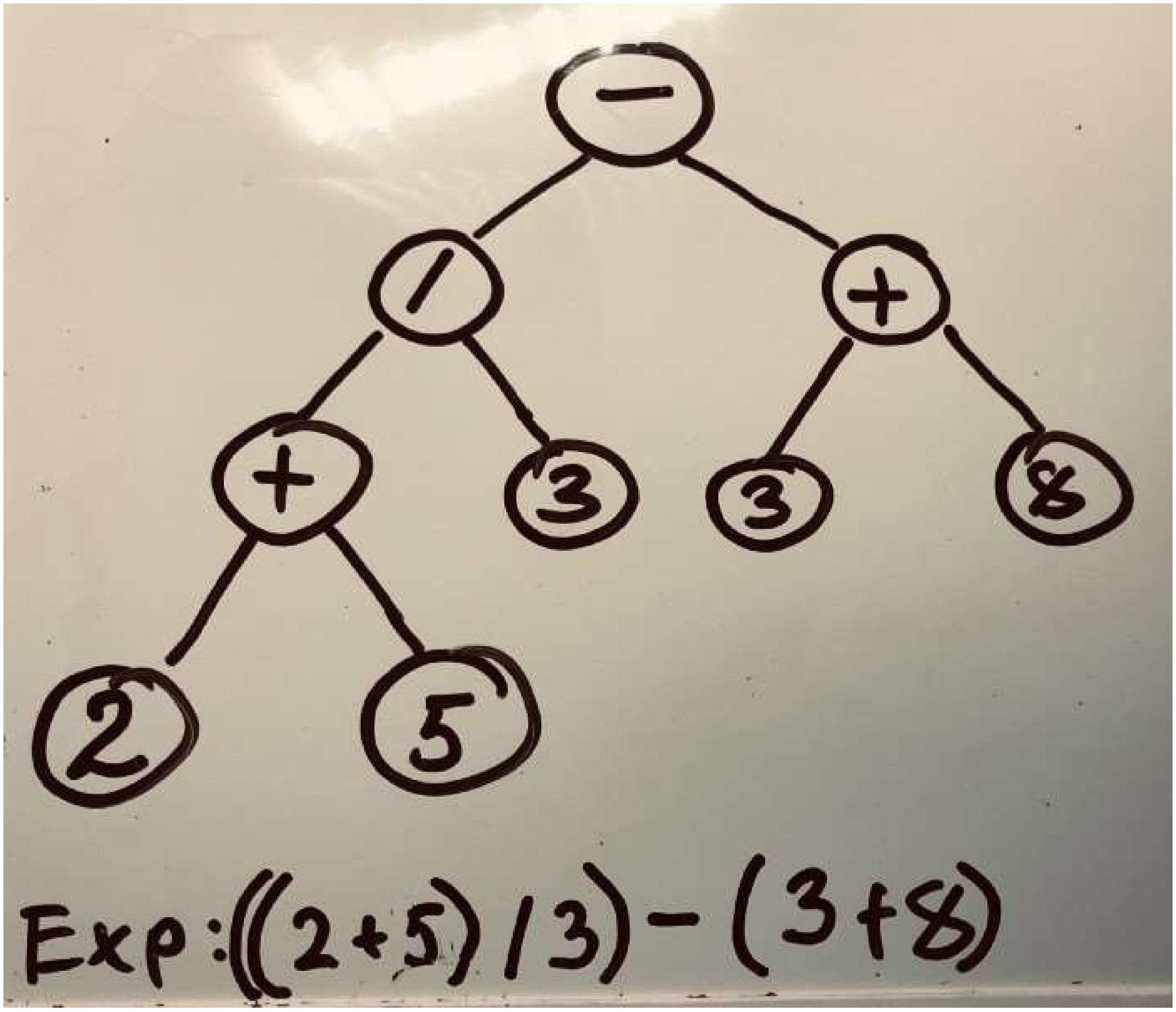
```
        return postfix
```

infix_to_postfix.py hosted with ♡ by GitHub view raw

Infix to postfix conversion code.

Once we convert the infix expression, we can build the tree, by inserting each term into the correct type of node.

So, we insert this expression `((2+5)/3)-(3+8)` into our tree it should build the below tree:



• • •

## Evaluate the Tree

Now we have a binary expression tree produced from given numerical expression. In order t evaluate this tree, we need to do recursive traversal and evaluate each parent node value using its children. Below is the recursive function to evaluate the tree.

```python
def evaluate(self, node=None) -> float:
        """
        Calculate this tree expression recursively

        Args:
            node(BinaryTreeNode): starts at the root node
        """
        # initialize

        if node is None:
            node = self.root

        # empty tree
        if node is None:
            return 0

        # check if we are at the leaf, it means it is a operand
        if node.is_leaf():
            node.value = float(node.data)
            val = float(node.data)

            return val

        left_value = self.evaluate(node.left)
        right_value = self.evaluate(node.right)

        # addition
        if node.data == "+":

            return left_value + right_value
        # subtraction
        elif node.data == "-":

            print(f"node value: {node.value}")
            return left_value - right_value
        # division
        elif node.data == "/":
            return left_value / right_value
        # multiplication
        elif node.data == "*":
            return left_value * right_value
        # power
        else:
            return left_value ** right_value
```

binary_expression_tree_evaluate.py hosted with ♡ by GitHub view raw

Evaluate method for binary expression tree

## Resources

- https://ruslanspivak.com/lsbasi-part7/
- https://en.wikipedia.org/wiki/Abstract_syntax_tree

Programming Trees Data Structures Binary Tree Python