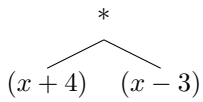
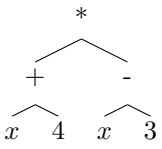
0.0.1 Binary Trees

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

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



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



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

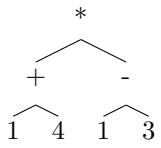
Algorithm 1: Evaluate: Binary Tree Version

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

This recursive algorithm has base case "tree.height = 0", which essentially checks if are at the bottom of our tree. We know that the bottommost nodes are actually values, therefore we can return this value as we cannot go deeper into our tree. Our recursive case is where we are not at the bottom of our tree. We know that at this point, the root node is an operator [1]. Therefore we operate on the nodes below it.

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

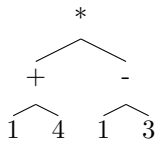
First let us replace every instance of x with 1.



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

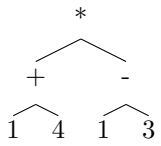
and again the root node is an operator so we take the left subtree again, which is 1. This is not an operator, so we return this value, going back to our previous call which is,

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



We now repeat the process for the right subtree,

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



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