Binary Tree Applications

Parse Tree

With the implementation of our tree data structure complete, we now look at an example of how a tree can be used to solve some real problems. In this section we will look at parse trees. Parse trees can be used to represent real-world constructions like sentences or mathematical expressions.

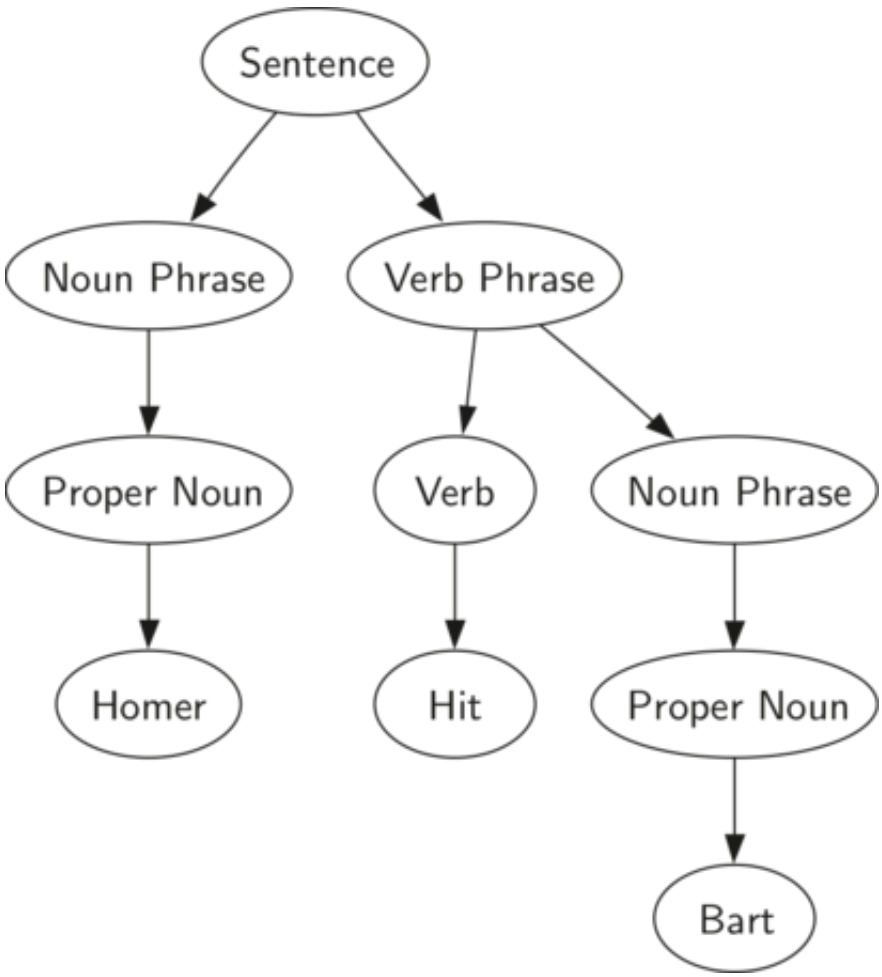


Figure 1: A Parse Tree for a Simple Sentence

Figure 1 shows the hierarchical structure of a simple sentence. Representing a sentence as a tree structure allows us to work with the individual parts of the sentence by using subtrees.

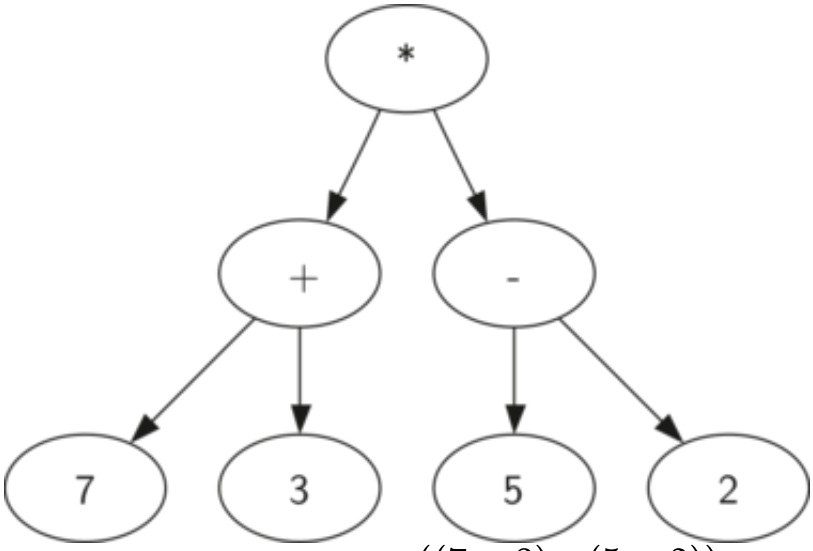


Figure 2: Parse Tree for $((7+3)*(5-2))$

We can also represent a mathematical expression such as $((7+3)*(5-2))$ as a parse tree, as shown in *Figure 2*. We have already looked at fully parenthesized expressions, so what do we know about this expression? We know that multiplication has a higher precedence than either addition or subtraction. Because of the parentheses, we know that before we can do the multiplication we must evaluate the parenthesized addition and subtraction expressions. The hierarchy of the tree helps us understand the order of evaluation for the whole expression. Before we can evaluate the top-level multiplication, we must evaluate the addition and the subtraction in the subtrees. The addition, which is the left subtree, evaluates to 10. The subtraction, which is the right subtree, evaluates to 3. Using the hierarchical structure of trees, we can simply replace an entire subtree with one node once we have evaluated the expressions in the children. Applying this replacement procedure gives us the simplified tree shown in *Figure 3*.

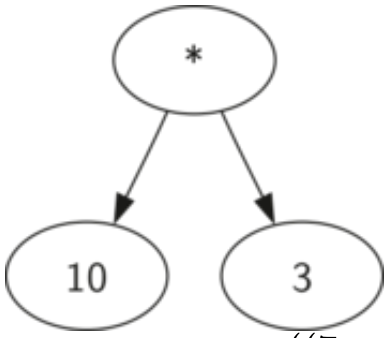


Figure 3: A Simplified Parse Tree for $((7+3)*(5-2))$

In the rest of this section we are going to examine parse trees in more detail. In particular we will look at

- How to build a parse tree from a fully parenthesized mathematical expression.
- How to evaluate the expression stored in a parse tree.
- How to recover the original mathematical expression from a parse tree.

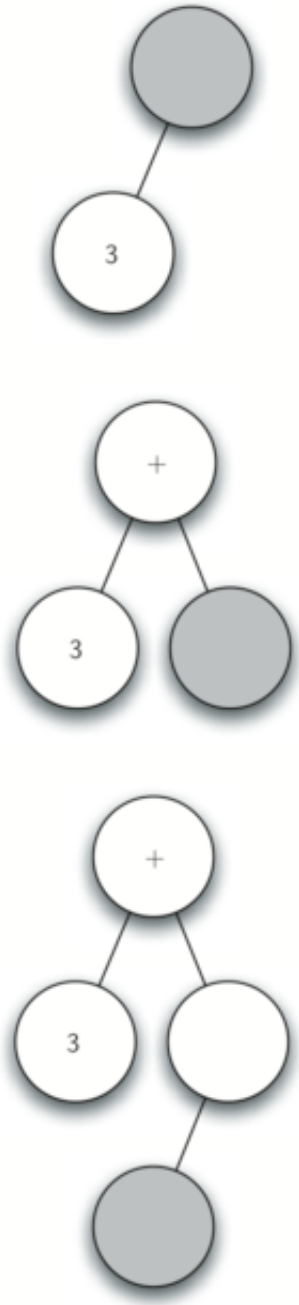
The first step in building a parse tree is to break up the expression string into a list of tokens. There are four different kinds of tokens to consider: left parentheses, right parentheses, operators, and operands. We know that whenever we read a left parenthesis we are starting a new expression, and hence we should

create a new tree to correspond to that expression. Conversely, whenever we read a right parenthesis, we have finished an expression. We also know that operands are going to be leaf nodes and children of their operators. Finally, we know that every operator is going to have both a left and a right child.

Using the information from above we can define four rules as follows:

1. If the current token is a `'('`, add a new node as the left child of the current node, and descend to the left child.
2. If the current token is in the list `['+','-','/','*']`, set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child.
3. If the current token is a number, set the root value of the current node to the number and return to the parent.
4. If the current token is a `')'`, go to the parent of the current node.

Before writing the Python code, let's look at an example of the rules outlined above in action. We will use the expression $(3 + (4 * 5))$. We will parse this expression into the following list of character tokens `['(', '3', '+', '(', '4', '*', '5' ,')',')']`. Initially we will start out with a parse tree that consists of an empty root node. *Figure 4* illustrates the structure and contents of the parse tree, as each new token is processed.


3
+
3
+
3

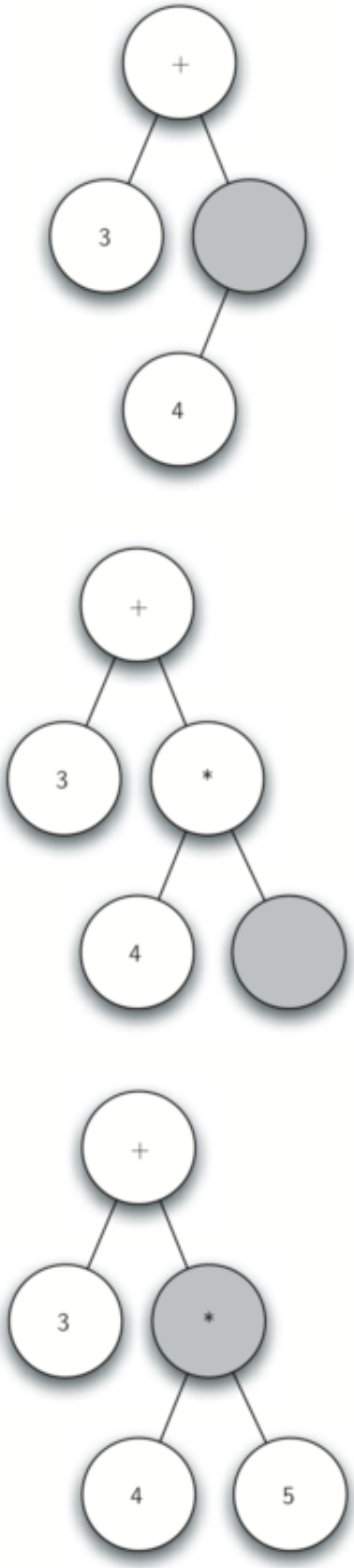


Figure 4: Tracing Parse Tree Construction

Using *Figure 4*, let's walk through the example step by step:

a. Create an empty tree.
b. Read (as the first token. By rule 1, create a new node as the left child of the root. Make the current node this new child.
c. Read 3 as the next token. By rule 3, set the root value of the current node to 3 and go back up the tree to the parent.
d. Read + as the next token. By rule 2, set the root value of the current node to + and add a new node as the right child. The new right child becomes the current node.
e. Read a (as the next token. By rule 1, create a new node as the left child of the current node. The new left child becomes the current node.
f. Read a 4 as the next token. By rule 3, set the value of the current node to 4. Make the parent of 4 the current node.
g. Read * as the next token. By rule 2, set the root value of the current node to * and create a new right child. The new right child becomes the current node.
h. Read 5 as the next token. By rule 3, set the root value of the current node to 5. Make the parent of 5 the current node.
i. Read) as the next token. By rule 4 we make the parent of * the current node.
j. Read) as the next token. By rule 4 we make the parent of + the current node. At this point there is no parent for + so we are done.

From the example above, it is clear that we need to keep track of the current node as well as the parent of the current node. The tree interface provides us with a way to get children of a node, through the `getLeftChild` and `getRightChild` methods, but how can we keep track of the parent? A simple solution to keeping track of parents as we traverse the tree is to use a stack. Whenever we want to descend to a child of the current node, we first push the current node on the stack. When we want to return to the parent of the current node, we pop the parent off the stack.

Using the rules described above, along with the `Stack` and `BinaryTree` operations, we are now ready to write a Python function to create a parse tree. The code for our parse tree builder is presented in *ActiveCode 1*.

```
 1 from pythonds.basic.stack import Stack
 2 from pythonds.trees.binaryTree import BinaryTree
 3
 4 def buildParseTree(fpexp):
 5     fplist = fpexp.split()
 6     pStack = Stack()
 7     eTree = BinaryTree('')
 8     pStack.push(eTree)
 9     currentTree = eTree
10     for i in fplist:
11         if i == '(':
12             currentTree.insertLeft('')
13             pStack.push(currentTree)
14             currentTree = currentTree.getLeftChild()
15         elif i not in ['+', '-', '*', '/', ')']:
16             currentTree.setRootVal(int(i))
17             parent = pStack.pop()
18             currentTree = parent
19         elif i in ['+', '-', '*', '/']:
20             currentTree.setRootVal(i)
21             currentTree.insertRight('')
22             pStack.push(currentTree)
23             currentTree = currentTree.getRightChild()
24         elif i == ')':
25             currentTree = pStack.pop()
26         else:
27             raise ValueError
28     return eTree
29
30 pt = buildParseTree("( ( 10 + 5 ) * 3 )")
31 pt.postorder()  #defined and explained in the next section
32
```

ActiveCode: 1 Building a Parse Tree (parsebuild)

Run

Save Load

The four rules for building a parse tree are coded as the first four clauses of the `if` statement on lines 11, 15, 19, and 24 of *ActiveCode 1*. In each case you can see that the code implements the rule, as described above, with a few calls to the `BinaryTree` or `Stack` methods. The only error checking we do in this function is in the `else` clause where we raise a `ValueError` exception if we get a token from the list that we do not recognize.

Now that we have built a parse tree, what can we do with it? As a first example, we will write a function to evaluate the parse tree, returning the numerical result. To write this function, we will make use of the hierarchical nature of the tree. Look back at *Figure 2*. Recall that we can replace the original tree with the simplified tree shown in *Figure 3*. This suggests that we can write an algorithm that evaluates a parse tree by recursively evaluating each subtree.

As we have done with past recursive algorithms, we will begin the design for the recursive evaluation function by identifying the base case. A natural base case for recursive algorithms that operate on trees is to check for a leaf node. In a parse tree, the leaf nodes will always be operands. Since numerical objects like integers and floating points require no further interpretation, the `evaluate` function can simply return the value stored in the leaf node. The recursive step that moves the function toward the base case is to call `evaluate` on both the left and the right children of the current node. The recursive call effectively moves us down the tree, toward a leaf node.

To put the results of the two recursive calls together, we can simply apply the operator stored in the parent node to the results returned from evaluating both children. In the example from *Figure 3* we see that the two children of the root evaluate to themselves, namely 10 and 3. Applying the multiplication operator gives us a final result of 30.

The code for a recursive `evaluate` function is shown in *Listing 1*. First, we obtain references to the left and the right children of the current node. If both the left and right children evaluate to `None`, then we know that the current node is really a leaf node. This check is on line 7. If the current node is not a leaf node, look up the operator in the current node and apply it to the results from recursively evaluating the left and right children.

To implement the arithmetic, we use a dictionary with the keys `'+'`, `'-'`, `'*'`, and `'/'`. The values stored in the dictionary are functions from Python's operator module. The operator module provides us with the functional versions of many commonly used operators. When we look up an operator in the dictionary, the corresponding function object is retrieved. Since the retrieved object is a function, we can call it in the usual way `function(param1,param2)`. So the lookup `opers['+'](2,2)` is equivalent to `operator.add(2,2)`.

Listing 1

```
1  def evaluate(parseTree):
       opers = {'+':operator.add, '-':operator.sub, '*':operator.mul, '/':opera
2  tor.truediv}

3      leftC = parseTree.getLeftChild()
       rightC = parseTree.getRightChild()

4      if leftC and rightC:
           fn = opers[parseTree.getRootVal()]
5          return fn(evaluate(leftC),evaluate(rightC))
       else:
6          return parseTree.getRootVal()
7
8
9
1
0
1
1
```

Finally, we will trace the `evaluate` function on the parse tree we created in *Figure 4*. When we first call `evaluate`, we pass the root of the entire tree as the parameter `parseTree`. Then we obtain references to the left and right children to make sure they exist. The recursive call takes place on line 9. We begin by looking up the operator in the root of the tree, which is `'+'`. The `'+'` operator maps to the `operator.add` function call, which takes two parameters. As usual for a Python function call, the first thing Python does is to evaluate the parameters that are passed to the function. In this case both parameters are recursive function calls to our `evaluate` function. Using left-to-right evaluation, the first recursive call goes to the left. In the first recursive call the `evaluate` function is given the left subtree. We find that the node has no left or right children, so we are in a leaf node. When we are in a leaf node we just return the value stored in the leaf node as the result of the evaluation. In this case we return the integer 3.

At this point we have one parameter evaluated for our top-level call to `operator.add`. But we are not done yet. Continuing the left-to-right evaluation of the parameters, we now make a recursive call to evaluate the right child of the root. We find that the node has both a left and a right child so we look up the operator stored in this node, `'*'`, and call this function using the left and right children as the parameters. At this point you can see that both recursive calls will be to leaf nodes, which will evaluate to the integers four and five respectively. With the two parameters evaluated, we return the result of `operator.mul(4,5)`. At this point we have evaluated the operands for the top level `'+'` operator and all that is left to do is finish the call to `operator.add(3,20)`. The result of the evaluation of the entire expression tree for $(3 + (4 * 5))$ is 23.

Tree Traversals

Now that we have examined the basic functionality of our tree data structure, it is time to look at some additional usage patterns for trees. These usage patterns can be divided into the three ways that we access the nodes of the tree. There are three commonly used patterns to visit all the nodes in a tree. The difference between these patterns is the order in which each node is visited. We call this visitation of the nodes a “traversal.” The three traversals we will look at are called **preorder**, **inorder**, and **postorder**. Let’s start out by defining these three traversals more carefully, then look at some examples where these patterns are useful.

preorder

In a preorder traversal, we visit the root node first, then recursively do a preorder traversal of the left subtree, followed by a recursive preorder traversal of the right subtree.

inorder

In an inorder traversal, we recursively do an inorder traversal on the left subtree, visit the root node, and finally do a recursive inorder traversal of the right subtree.

postorder

In a postorder traversal, we recursively do a postorder traversal of the left subtree and the right subtree followed by a visit to the root node.

Let’s look at some examples that illustrate each of these three kinds of traversals. First let’s look at the preorder traversal. As an example of a tree to traverse, we will represent this book as a tree. The book is the root of the tree, and each chapter is a child of the root. Each section within a chapter is a child of the chapter, and each subsection is a child of its section, and so on. *Figure 5* shows a limited version of a book with only two chapters. Note that the traversal algorithm works for trees with any number of children, but we will stick with binary trees for now.

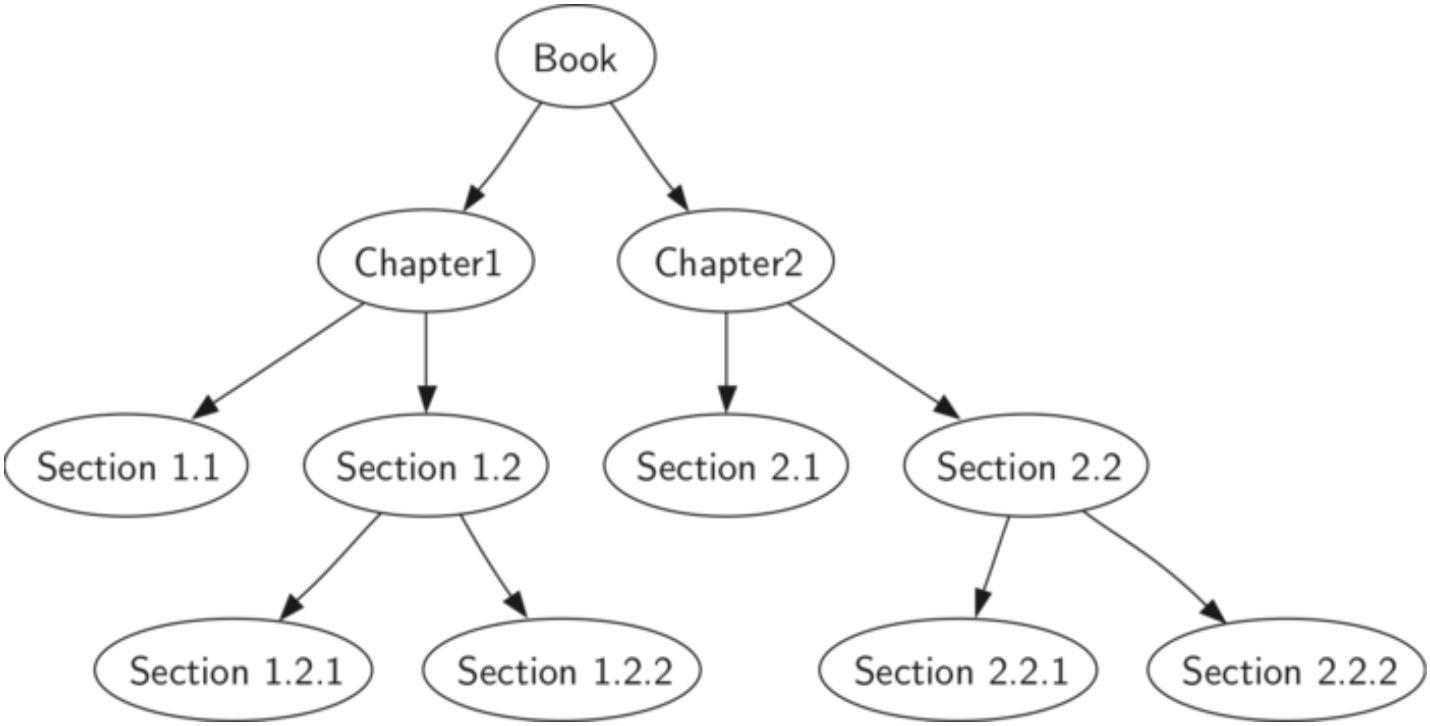


Figure 5: Representing a Book as a Tree

Suppose that you wanted to read this book from front to back. The preorder traversal gives you exactly that ordering. Starting at the root of the tree (the Book node) we will follow the preorder traversal instructions. We recursively call `preorder` on the left child, in this case Chapter1. We again recursively call `preorder` on the left child to get to Section 1.1. Since Section 1.1 has no children, we do not make any additional recursive calls. When we are finished with Section 1.1, we move up the tree to Chapter 1. At this point we still need to visit the right subtree of Chapter 1, which is Section 1.2. As before we visit the left subtree, which brings us to Section 1.2.1, then we visit the node for Section 1.2.2. With Section 1.2 finished, we return to Chapter 1. Then we return to the Book node and follow the same procedure for Chapter 2.

The code for writing tree traversals is surprisingly elegant, largely because the traversals are written recursively. *Listing 2* shows the Python code for a preorder traversal of a binary tree.

You may wonder, what is the best way to write an algorithm like preorder traversal? Should it be a function that simply uses a tree as a data structure, or should it be a method of the tree data structure itself? *Listing 2* shows a version of the preorder traversal written as an external function that takes a binary tree as a parameter. The external function is particularly elegant because our base case is simply to check if the tree exists. If the tree parameter is `None` , then the function returns without taking any action.

Listing 2

```
def preorder(tree):
    if tree:
        print(tree.getRootVal())
        preorder(tree.getLeftChild())
        preorder(tree.getRightChild())
```

We can also implement `preorder` as a method of the `BinaryTree` class. The code for implementing `preorder` as an internal method is shown in *Listing 3*. Notice what happens when we move the code from internal to external. In general, we just replace `tree` with `self`. However, we also need to modify the base case. The internal method must check for the existence of the left and the right children *before* making the recursive call to `preorder`.

Listing 3

```
def preorder(self):
    print(self.key)
    if self.leftChild:
        self.left.preorder()
    if self.rightChild:
        self.right.preorder()
```

Which of these two ways to implement `preorder` is best? The answer is that implementing `preorder` as an external function is probably better in this case. The reason is that you very rarely want to just traverse the tree. In most cases you are going to want to accomplish something else while using one of the basic traversal patterns. In fact, we will see in the next example that the `postorder` traversal pattern follows very closely with the code we wrote earlier to evaluate a parse tree. Therefore we will write the rest of the traversals as external functions.

The algorithm for the `postorder` traversal, shown in *Listing 4*, is nearly identical to `preorder` except that we move the call to print to the end of the function.

Listing 4

```
def postorder(tree):
    if tree != None:
        postorder(tree.getLeftChild())
        postorder(tree.getRightChild())
        print(tree.getRootVal())
```

We have already seen a common use for the postorder traversal, namely evaluating a parse tree. Look back at *Listing 1* again. What we are doing is evaluating the left subtree, evaluating the right subtree, and combining them in the root through the function call to an operator. Assume that our binary tree is going to store only expression tree data. Let's rewrite the evaluation function, but model it even more closely on the `postorder` code in *Listing 4* (see *Listing 5*).

Listing 5

```
1
2  def postordereval(tree):
       opers = {'+':operator.add, '-':operator.sub, '*':operator.mul, '/':opera
3  tor.truediv}
       res1 = None
4      res2 = None
       if tree:
5          res1 = postordereval(tree.getLeftChild())
           res2 = postordereval(tree.getRightChild())
6          if res1 and res2:
               return opers[tree.getRootVal()](res1,res2)
7          else:
               return tree.getRootVal()
8
9
1
0
1
1
```

Notice that the form in *Listing 4* is the same as the form in *Listing 5*, except that instead of printing the key at the end of the function, we return it. This allows us to save the values returned from the recursive calls in lines 6 and 7. We then use these saved values along with the operator on line 9.

The final traversal we will look at in this section is the inorder traversal. In the inorder traversal we visit the left subtree, followed by the root, and finally the right subtree. *Listing 6* shows our code for the inorder traversal. Notice that in all three of the traversal functions we are simply changing the position of the `print` statement with respect to the two recursive function calls.

Listing 6

```
def inorder(tree):
  if tree != None:
      inorder(tree.getLeftChild())
      print(tree.getRootVal())
      inorder(tree.getRightChild())
```

If we perform a simple inorder traversal of a parse tree we get our original expression back, without any parentheses. Let's modify the basic inorder algorithm to allow us to recover the fully parenthesized version of the expression. The only modifications we will make to the basic template are as follows: print a left parenthesis *before* the recursive call to the left subtree, and print a right parenthesis *after* the recursive call to the right subtree. The modified code is shown in *Listing 7*.

Listing 7

```
def printexp(tree):
  sVal = ""
  if tree:
      sVal = '(' + printexp(tree.getLeftChild())
      sVal = sVal + str(tree.getRootVal())
      sVal = sVal + printexp(tree.getRightChild())+')'
  return sVal
```

Notice that the `printexp` function as we have implemented it puts parentheses around each number. While not incorrect, the parentheses are clearly not needed. In the exercises at the end of this chapter you are asked to modify the `printexp` function to remove this set of parentheses.

© Copyright 2013 Brad Miller, David Ranum. Created using Sphinx (http://sphinx.pocoo.org/) 1.1.3.

78 readers online now | **Not logged in** | Back to top