PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
DEPARTAMENTO DE INGENIERÍA

Algoritmia

Laboratorio 4

Aplicación de los tipos de datos principales: Listas, Colas, Pilas, Árboles y Grafos

Pilas y procedimientos recursivos (tomado textualmente del libro *Estructura de Datos y Algoritmos* por Aho, Hopocroft y Ullman)

La recursión simplifica la estructura de muchos programas. Sin embargo, en algunos lenguajes las llamadas a procedimientos tienen un costo mucho mayor que el de las proposiciones de asignación, de modo que la ejecución de un programa puede resultar más rápida, en un factor constante considerable, si se eliminan las llamadas recursivas. Al decir esto no se propugna la eliminación habitual de la recursión o de otras llamadas a procedimientos; es frecuente que la sencillez estructural justifique el tiempo de ejecución. Sin embargo, podría ser deseable eliminar la recursión en las porciones de los programas que se ejecutan con mayor frecuencia, y el proposito de análisis que sigue es ilustrar cómo se puede convertir los procedimientos recursivos en procedimientos no recursivos mediante una pila definida por el programador.

Ejemplo 2.3. Considere dos soluciones, una recursiva y otra no recursiva, a una versión simplificada del clásico *problema de la mochila*, en el cual se da un *objetivo o* y una colección de pesos p_1, p_2, ..., p_n (enteros positivos). Se pide determinar si existe una selección de pesos que totalice exactamente *o*. Por ejemplo, si $o = 10$ y los pesos son 7, 5, 4, 4 y 1, se puede elegir el segundo, el tercero y el quinto, ya que $5 + 4 + 1 = 10$.

La justificación del nombre "problema de la mochila" es que se desea llevar en la espalda no más de *o* kilogramos, y se tiene para elegir un conjunto de objetos de pesos dados. Se supone, además, que la utilidad de los objetos es proporcional a su peso[1], y por ello se desea cargar la mochila con un peso lo más cercano posible al objetivo.

En la figura 2.25 se puede ver una función *mochila* que opera en una matriz

pesos: **array**[1..*n*] **of** integer.

La llamada a *mochila*(*s,i*) determina si existe una colección de los elementos entre *peso*[*i*] y *peso*[*n*] que sume exactamente *s*, e imprime sus pesos de ser tal el caso. Lo primero que *mochila* hace es determinar si se puede responder de inmediato. Específicamente, si $s = 0$, entonces el conjunto vacío de pesos es una solución. Si $s < 0$, no puede haber solución, y si $s > 0$ e $i > n$, entonces ya no hay más pesos que considerar y, por tanto, no se puede encontrar una suma de pesos igual a *s*.

Si no se tiene ninguno de estos casos, entonces sencillamente se hace la llamada a *mochila*($s - p_i$, $i + 1$) , para ver si existe una solución que incluya a p_i. Si esa solución existe, todo el problema está resuelto y la solución incluye a p_i, de modo que éste se imprime. Si no hay solución, se efectúa la llamada a *mochila*(s, $i + 1$), para ver si existe una solución que no use p_i.

1 En el "verdadero" problema de la mochila, se dan valores de utilidad y pesos, y se pide maximizar la utilidad de los objetos transportados, con una restricción del peso.

Eliminación de la recursión de cola[2]

A menudo, es posible eliminar en forma mecánica la última llamada que un procedimiento se hace a sí mismo. Si un procedimiento $P(x)$ tiene como último paso una llamada a $P(y)$, entonces es posible reemplazar la llamada a $P(y)$ por una asignación $x := y$, seguida de un salto al principio del código de P. Aquí, y puede ser una expresión, pero x debe ser una parámetro pasado pro valor, de modo que su valor se almacena en una localidad de memoria privada de esta llamada a P[3]. Por supuesto, P podría tener más de un parámetro, en cuyo caso se trataría exactamente igual que x e y.[4]

```
        function mochila ( objetivo: integer; candidato: integer ): boolean;
          begin
(1)         if objetivo = 0 then
(2)               return ( true )
(3)         else if (objetivo  < 0)  or (candidato > n) then
(4)               return(false)
            else { considera soluciones con y sin  candidato }
(5)         if mochila(objetivo–pesos[candidato], candidato + 1) then
                  begin
(6)                   writeln(pesos [candidato] );
(7)                   return (true)
                  end
            else { la única solución posible es sin candidato }
(8)               return (mochila(objetivo, candidato + 1))
          end; { mochila }
```

Fig. 2.25. Solución recursiva al problema de la mochila.

Este cambio funciona, porque volver a ejecutar P con el nuevo valor de x equivale exactamente a llamar a $P(y)$ y luego volver de esta llamada. Obsérvese que el hecho de que algunas de las variables de P tengan valores en la segunda llamada no tiene consecuencias. P no podría usar ninguno de esos valores, pues si se hubiera llamado a $P(y)$ como se intentaba en un principio, esos valores no habrían estado definidos.

En la figura 2.25 se ilustra otra variante de la recursión de cola; ahí, el último paso de la función *mochila* sólo devuelve el resultado de llamarse a sí misma con otros parámetros. En una situación tal, suponiendo que los parámetros se pasan por valor (o por referencia, si el mismo parámetro es el que se pasa a la llamada), se puede reemplazar la llamada por asignaciones a los parámetros y un salto al principio de la función. En el caso de la figura 2.25, se puede reemplzar la línea (8) por

candidato := *candidato* + 1;
goto *principio*

donde *principio* es una etiqueta que se va a asignar a la proposición (1). Obsérvese que no se necesita ningún cambio a *objetivo*, puesto que su valor pasa intacto como primer parámetro. De hecho, se puede advertir que, al no haber cambiado *objetivo*, las pruebas de las proposiciones (1) y (3) que lo incluyen están destinadas a fallar y, por tanto, es posible omitir las líneas (1) y (2), realizar sólo la prueba *candidato* > *n* en la línea (3) y luego proseguir directamente a la línea (5).

---

2 Nota del profesor. He tratado ser fiel al libro, pero en este caso creo que una mejor traducción sería “recursión final”, es decir cuando después de la llamada recursiva no se realiza operación alguna.

3 Alternativamente, x podría pasarse por referencia si y es x.

4 Nota del profesor. Algunos compiladores “inteligentes” internamente transforman un código con recursividad final en uno iterativo. Un ejemplo es el compilador de Erlang.

Eliminación completa de la recursión

El procedimiento de eliminación de la recursión de cola suprime la recursión por completo sólo cuando la llamada recursiva se encuentra al final del procedimiento y tiene la forma adecuada. Existe un enfoque más general que convierte cualquier procedimiento (o función) recursivo en no recursivo, pero que introduce una pila definida por el programador. En general, una celda de esa pila contendrá:

1. los valores actuales de los parámetros del procedimiento;

2. los valores actuales de todas las variables locales del procedimiento, y

3. una indicación de la dirección de retorno, esto es, del lugar a donde deberá devolver el control cuando la invocación actual del procedimiento termine.

En el caso de la función *mochila*, se puede hacer algo más sencillo. Primero, se observa que cada vez que se hace una llamada (que implica meter un registro a la pila), *candidato* se incrementa en 1. Así pues, se puede dejar *candidato* como una variable global, incrementando su valor a 1 cada vez que se mete un registro en la pila y disminuyéndolo en 1 cada vez que se saca un registro.

Una segunda simplificación posible consiste en mantener dentro de la pila una "dirección de retorno" modificada. En términos estrictos, la dirección de retorno para esta función es un lugar de otro procedimiento que llama a *mochila*, la llamada de la línea (5) o la llamada de la línea (8). Estas tres posibilidades se representan por una variable "de estado" que tiene uno de los tres valores siguientes:

1. *ninguno,* que indica que la llamada procede de fuera de la función *mochila.*

2. *incluido*, que indica la llamada de la línea (5), la cual incluye *pesos*[*candidato*] dentro de la solución, o

3. *excluido*, que indica la llamada de la línea (8), la cual excluye a *pesos*[*candidatos*].

Si se almacena esta variable de estado como la dirección de retorno, se puede manejar *objetivo* como una variable global. Cuando el estado cambia de *ninguno* a *incluido*, se sustrae *pesos*[candidato] de *objetivo*, y se suma de nuevo cuando el estado cambia de *incluido* a *excluido*. Como ayuda para representar el efecto de retorno de *mochila* cuando indican si se ha hallado la solución, se usa una variable global *bandera_éxito*. Una vez que *bandera_éxito* toma el valor verdadero, lo conserva y hace que se saquen de la pila los registros, imprimiendo aquellos pesos que tienen asociado un estado *incluido*. Con estas modificaciones, se puede declarar la pila como una pila de estados, mediante:

```
type
    estados = (ninguno, incluido, excluido);
    PILA = {declaración adecuada para una pila de estados}
```

La figura 2.26 muestra el procedimiento resultante no recursivo *mochila*, que opera en un arreglo de *pesos* como antes. Aunque este procedimiento puede ser más rápido que la función *mochila* original, es claramente más largo y más difícil de entender. Por ello sólo se debe eliminar la recursión cuando la velocidad sea muy importante.

```
procedure mochila (objetivo: integer );
    var
        candidato: integer;
        bandera_éxito: boolean;
        P: PILA;
    begin
        candidato := 1;
        bandera_éxito := false;
        ANULA(P);
        METE(ninguno, P); { asigna valor inicial a la pila considerando pesos[1] }
        repeat
            if bandera_éxito then begin
                { saca de la pila e imprime los pesos incluidos en la solución }
                if TOPE(P) = incluido then
                    writeln (pesos[candidato]);
                candidato := candidato-1;
                SACA(P)
            end
            else if objetivo = 0 then begin { se encontró la solución }
                bandera_éxito := true;
                candidato := candidato -1;
                SACA(P)
            end
            else if (((objetivo < 0) and (TOPE(P) = ninguno))
                    or (candidato > n)) then begin
                { no hay solución posible con las elecciones hechas }
                candidato := candidato -1;
                SACA(P)
            end
            else { aún no hay decisión, se considera el estado del candidato actual }
            if TOPE(P) = ninguno then begin { se intenta primero incluyendo al
                candidato }
                objetivo := objetivo-pesos[candidato];
                candidato := candidato +1;
                SACA(P); METE(incluido, P); METE (ninguno, P)
            end
            else if TOPE(P) = incluido then begin { se intenta excluyendo al
                        candidato }
                objetivo := objetivo + pesos[candidato];
                candidato := candidato +1;
                SACA(P); METE(excluido, P); METE(ninguno, P)
            end
            else begin { TOPE(P) = excluido; la elección actual no da resultado }
                SACA(P);
                candidato := candidato -1
            end
        until VACIA (P)
    end; { mochila }
```

Fig. 2.26. Procedimiento no recursivo para el problema de la mochila.

El problema de Josefo

Flavio Josefo fue un famoso historiador del primer siglo. Aparentemente, su talento matemático le permitido vivir lo suficiente para alcanzar la fama. En efecto, durante la guerra judeo-romana, Josefo formó parte de un grupo de 41 rebeldes judíos que fueron cercados por los romanos. Prefiriendo morir a rendirse, decidieron formar un circulo y procediendo en el sentido de las agujas de un reloj, matar a uno de cada tres rebeldes hasta que sólo quedaran dos que, en teoría, se suicidarían entonces. Sin embargo, a Josefo y a un amigo suyo esta idea no les parecía demasiado buena y decidieron situarse adecuadamente para ser los dos últimos supervivientes. ¿Cómo podrían Josefo y su amigo haber decidido las posiciones adecuadas de haber dispuesto de un ordenador (portátil) y de una implementación en C de listas circulares enlazadas?

El siguiente texto ha sido tomado de:

http://interactivepython.org/runestone/static/pythonds/Trees/bintreeapps.html

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.

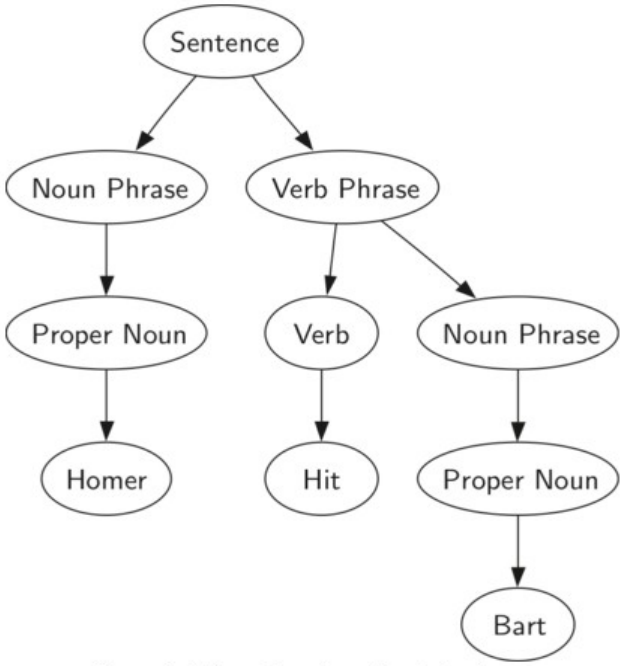


Figure 1: A Parse Tree for a Simple Sentence

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.

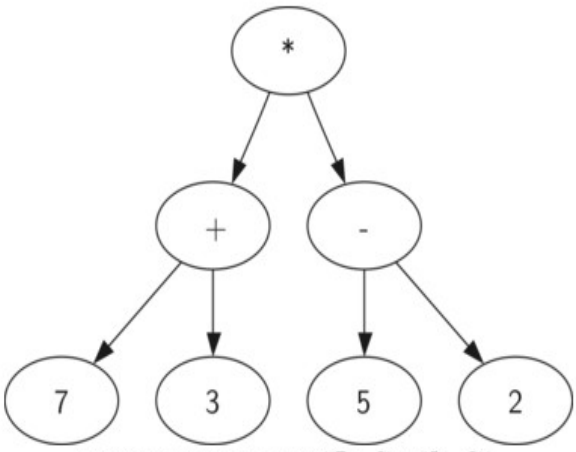


Figure 2: Parse Tree for ((7 + 3) ∗ (5 − 2))

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***.

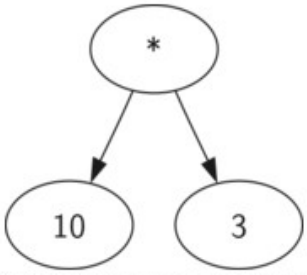


Figure 3: A Simplified Parse Tree for ((7 + 3) ∗ (5 − 2))

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.

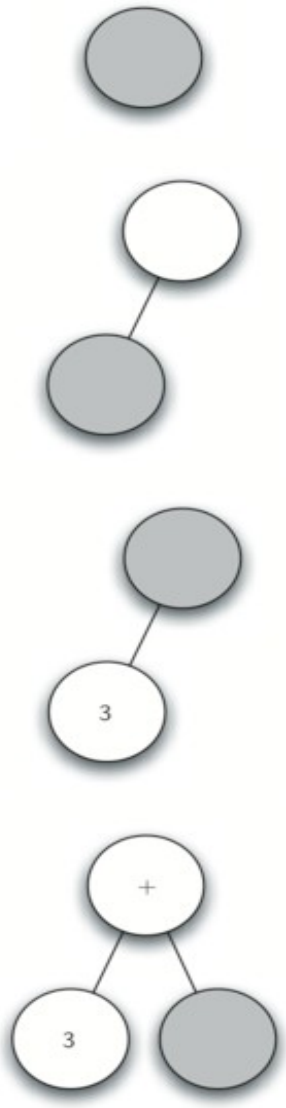
The first step in building a parse tree is to break up the expression string into a list of tokens. There are four different kinds of tokens to consider: left parentheses, right parentheses, operators, and operands. We know that whenever we read a left parenthesis we are starting a new expression, and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



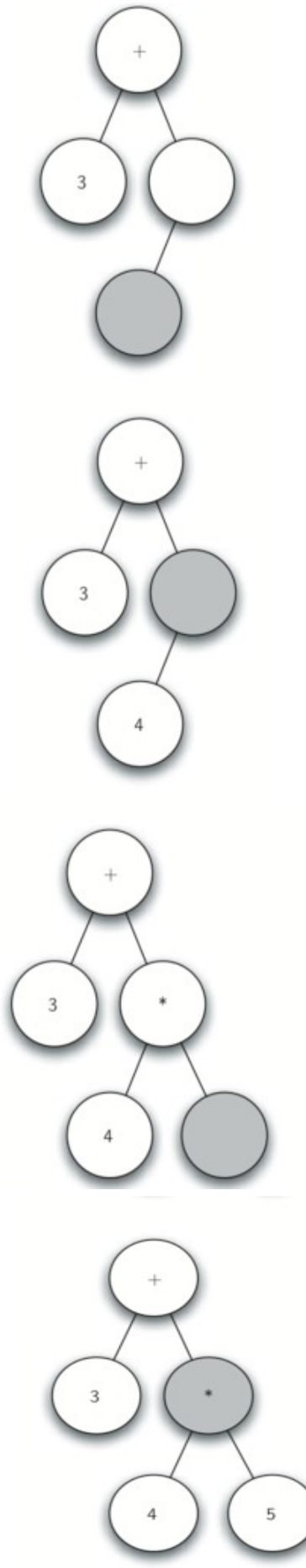


Figure 4: Tracing Parse Tree Construction

Using ***Figure 3***, let's walk through the example step by step:

1. Create an empty tree.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. Read) as the next token. By rule 4 we make the parent of * the current node.
10. 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 ***Active Code 1***

```
from pythonds.basic.stack import Stack
from pythonds.trees.binaryTree import BinaryTree

def buildParseTree(fpexp):
    fplist = fpexp.split()
    pStack = Stack()
    eTree = BinaryTree('')
    pStack.push(eTree)
    currentTree = eTree
    for i in fplist:
```

```
            if i == '(':
                currentTree.insertLeft('')
                pStack.push(currentTree)
                currentTree = currentTree.getLeftChild()
            elif i not in ['+', '-', '*', '/', ')']:
                currentTree.setRootVal(int(i))
                parent = pStack.pop()
                currentTree = parent
            elif i in ['+', '-', '*', '/']:
                currentTree.setRootVal(i)
                currentTree.insertRight('')
                pStack.push(currentTree)
                currentTree = currentTree.getRightChild()
            elif i == ')':
                currentTree = pStack.pop()
            else:
                raise ValueError
        return eTree

pt = buildParseTree("( ( 10 + 5 ) * 3 )")
pt.postorder()  #defined and explained in the next section.
```

Active Code: 1 Building a Parse Tree (parsebuild)

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 ***Active Code 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
2
3   def evaluate(parseTree):
        opers = {'+':operator.add, '-':operator.sub, '*':operator.mul, '/':operator.trued
4   iv}
5
        leftC = parseTree.getLeftChild()
        rightC = parseTree.getRightChild()
6
        if leftC and rightC:
7           fn = opers[parseTree.getRootVal()]
            return fn(evaluate(leftC),evaluate(rightC))
8       else:
            return parseTree.getRootVal()
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.