我们可以通过递归地产生一个带括号的左表达式，然后打印出在根处的运算符，最后再递归地产生一个带括号的右表达式而得到一个(对两个括号整体进行运算的)中缀表达式。这种一般的方法(左，节点，右)称为**中序遍历**(in-order traversal)。由于其产生的表达式类型，这种遍历很容易记忆。

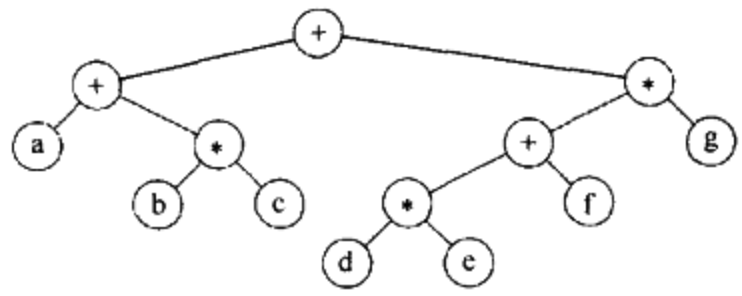



图 4-14 (a+b*c)+((d*e+f)*g)的表达式树

另一种遍历策略是递归地打印出左子树、右子树，然后打印运算符。如果我们将这种策略应用于上面的树，则将输出 a b c * + d e * f + g * +，显而易见，它就是 3.6.3 节中的后缀表示法。这种遍历策略一般称为**后序遍历**。我们稍早已在 4.1 节见过这种遍历方法。

第三种遍历策略是先打印出运算符，然后递归地打印出右子树和左子树。此时得到的表达式 ++a*bc*+*defg 是不太常用的前缀(prefix)记法，这种遍历策略为**先序遍历**，稍早我们也在 4.1 节见过。以后，我们还要在本章讨论这些遍历方法。

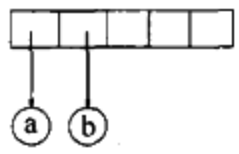
**构造表达式树**

我们现在给出一种算法来把后缀表达式转变成表达式树。由于我们已经有了将中缀表达式转变成后缀表达式的算法，因此我们能够从这两种常用类型的输入生成表达式树。这里所描述的方法酷似 3.6.3 节的后缀求值算法。我们一次一个符号地读入表达式。如果符号是操作数，那么就建立一个单节点树并将它推入栈中。如果符号是操作符，那么就从栈中弹出两棵树 $T_1$ 和 $T_2$($T_1$ 先弹出)并形成一棵新的树，该树的根就是操作符，它的左、右儿子分别是 $T_2$ 和 $T_1$。然后将这棵新树压入栈中。

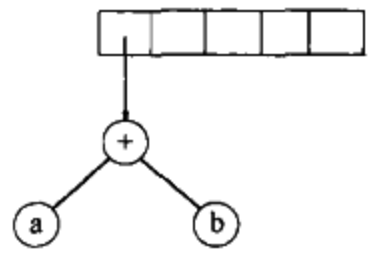
来看一个例子。设输入为

a b + c d e + * *

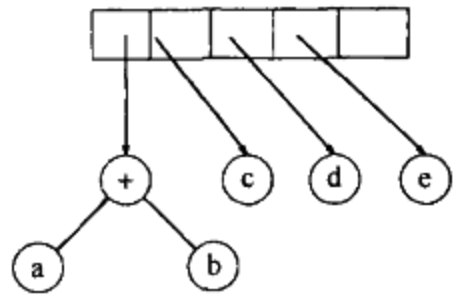
前两个符号是操作数，因此创建两棵单节点树并将它们压入栈中[⊖]。




接着，'+'被读入，因此两棵树被弹出，一棵新的树形成，并被压入栈中。




然后，c、d 和 e 被读入，在每个单节点树创建后，对应的树被压入栈中。




⊖ 为了方便起见，我们将让图中的栈从左到右增长。