表达式求值

中缀表达式求值

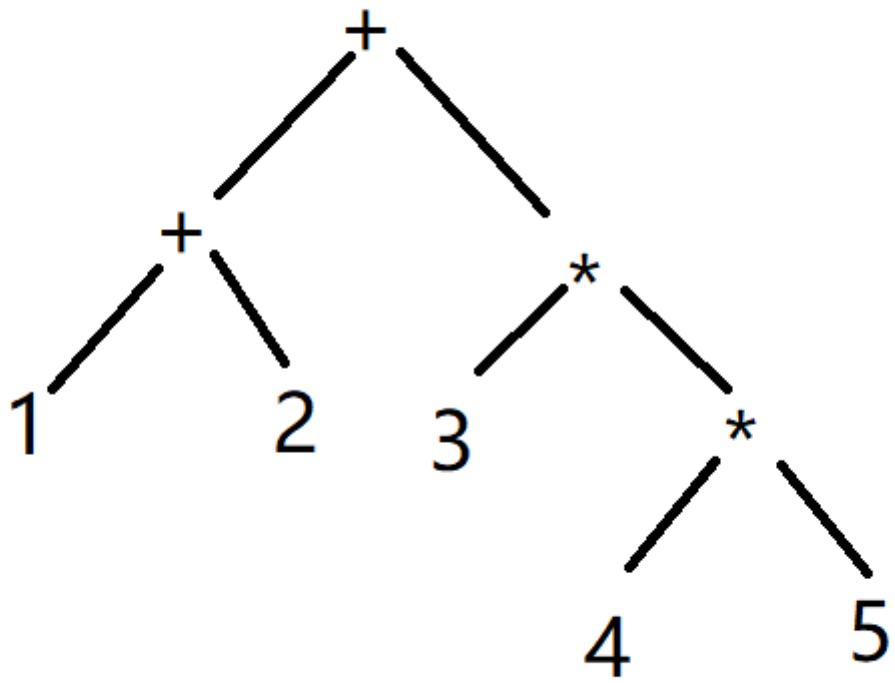
基本思想：

第一步，建树：

其实我们可以用一棵树来表示一个表达式，以表达式`1 + 2 + 3 * 4 * 5`（不带括号）为例，这棵树的构建有两个原则：

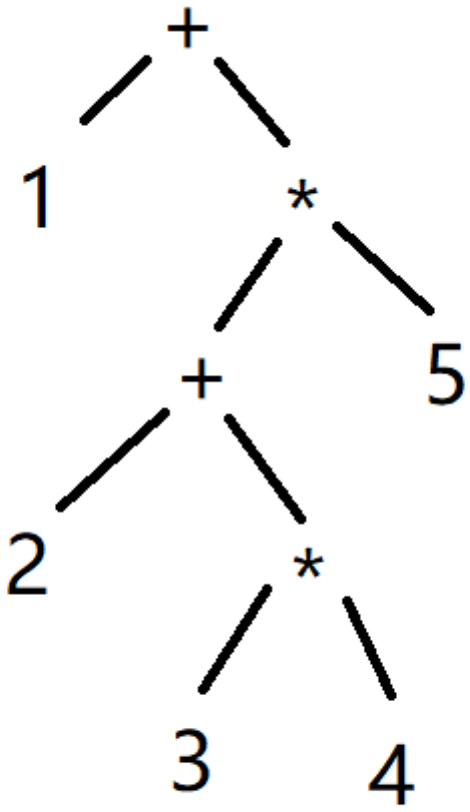
1. 叶结点都是数字，中间结点是运算符
2. 上面的运算符的优先级低于下面的运算符

于是建树如下：



第一个加号的运算级高于第二个加号，乘号的运算级高于加号，这棵树中序遍历的结果就是上面的运算式

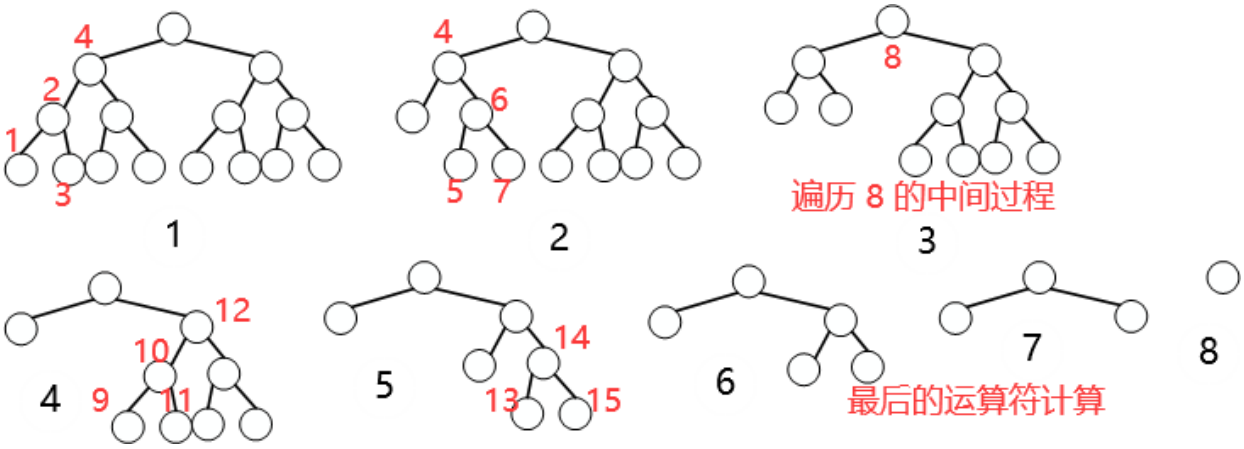
如果是带括号的式子：`1 + (2 + 3 * 4) * 5`那么建树如下：



其中由于括号的存在，括号内的运算符的优先级是大于括号外的运算符的，第一个加号的运算级最低，其次是最后一个乘号，然后是括号内的第一个加号，接着是运算优先级最高的括号内的第二个乘号，这棵树中序遍历的结果仍然是上面的那个式子

第二步，计算过程：

这棵树的计算过程与中序遍历过程是一致的，如下所示：



遍历节点 `1 2 3` 后，则 `4` 的左子树遍历完，则计算 `4` 的左子树的结果，新节点作为 `4` 的左孩子节点

继续遍历节点 `5 6 7`，则 `4` 的右子树遍历完，计算 `4` 的右子树的结果，于是 `8` 的左子树遍历完，则计算 `8` 的左子树的结果，新节点作为 `8` 的左孩子节点

理, 遍历节点 `12` 的时候计算其左子树的结果, 新节点作为 `12` 的左孩子

于是就这样将整棵树计算出来，注意不管原式子有没有带括号计算的过程都是上面这个过程

需要解决几个问题：

由于人是知道什么时候进行计算的，但是计算机不知道，所以，我们需要告诉计算机什么时候来进行计算

1. 什么时候进行计算?

当往上走的时候进行计算，比如上面的`1 3 2`结点访问完了，开始访问`4`，这时就需要计算`2`这颗子树，并将它的值作为一个叶子结点

1. 怎么知道是往上走?

注意到运算符优先级大的在下面, 运算符优先级小的在上面，所以当目前运算符的优先级比上一运算符优先级小时, 说明是往上走当目前运算符的优先级比上一运算符优先级大是, 说明是往下走

2. 怎么知道子树遍历完?

往上走的时候遍历完，比如上面访问`4`，就表示`1 2 3`这颗子树遍历完

我们用一个式子来模拟这个过程：`1 + (2 + 3 * 4) * 5`

用栈来模拟树的计算过程：

数据结构:

由于是模拟中序遍历树的过程, 所以要用栈数据结构，由于是有运算符和数字两个对象, 所以要用两个栈来存储

算法：

我们用以下四步来模拟上面树的计算过程：

1. 数字

数字并不会产生计算过程, 所以只需提取数字, 将数字压栈

2. 括号

括号分为两个运算符`(`和`)`

遇到`(`说明会往下走, 所以只需将`(`压栈

遇到`)`说明会往上走, 所以要计算括号表示的子树的结果, 所以要逆向计算运算符直至遇到`(`

3. 普通二元运算符

如果当前运算符优先级比上一运算符高, 说明是往下走, 则只需将运算符压栈

如果当前运算符优先级比上一运算符低, 说明是往上走, 则需要一直计算上一运算符直至当前运算符优先级比上一运算符高

我们用式子`1 + ( 2 + 3 * 4) * 5`举例：

1. 如果栈顶是+，即将入栈的是+，栈顶优先级高，需要先计算，再入栈；
2. 如果栈顶是+，即将入栈的是*，栈顶优先级低，直接入栈；

3. 如果栈顶是*，即将入栈的是+，栈顶优先级高，需要先计算，再入栈；
4. 如果栈顶是*，即将入栈的是*，栈顶优先级高，需要先计算，再入栈；
5. 如果是左括号，则直接入栈，不计算
6. 如果是右括号，则一直计算，直到遇到左括号

可以用一个运算符的表来记录：

优先级比较		入栈运算符	
		+	*
栈顶运算	+	>	<
	*	>	>

那么下面来模拟整个过程：

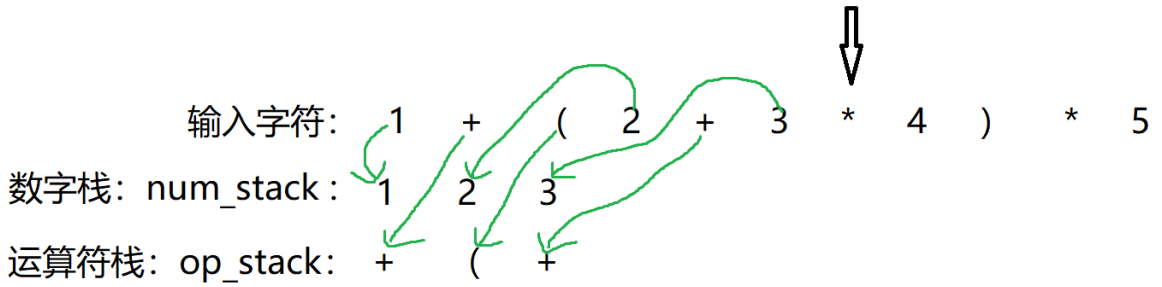
一开始，读入整个表达式，数字和字符栈都为空：

输入字符：1 + (2 + 3 * 4) * 5

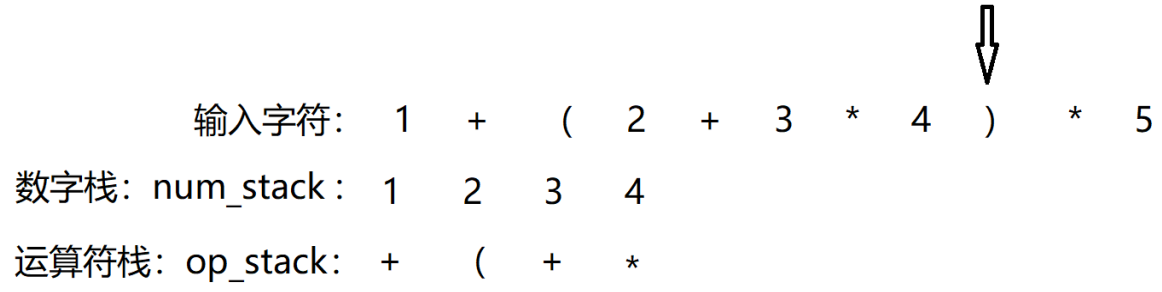
数字栈：num_stack：

运算符栈：op_stack：

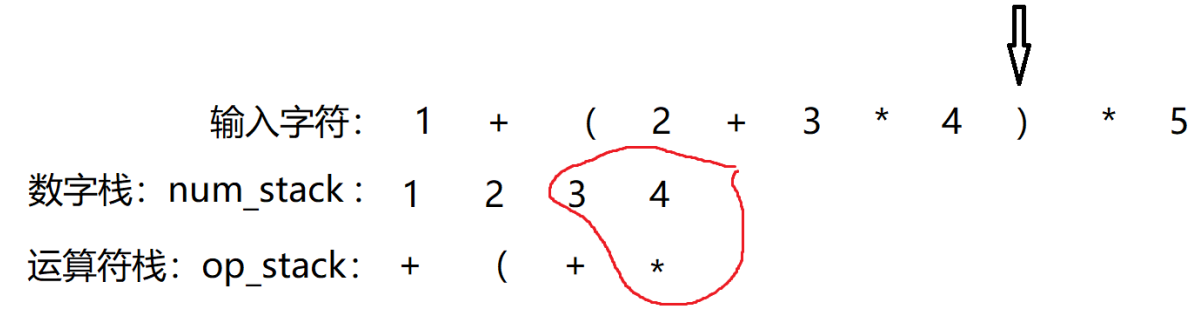
我们不断读入，由于这个过程中有一个左括号，需要特殊处理，直接入栈不需要计算，所以我们一直读入直到遇到第一个乘号：



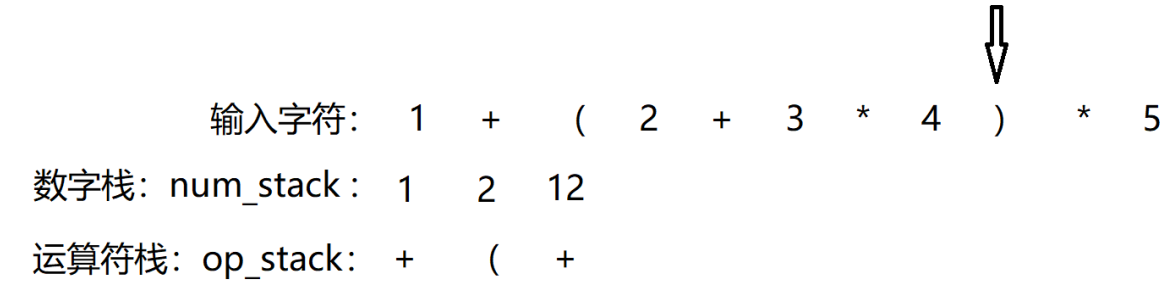
发现乘号的优先级大于栈顶的`+`号，这时不做计算，继续读入，直到遇到了右括号`)`：



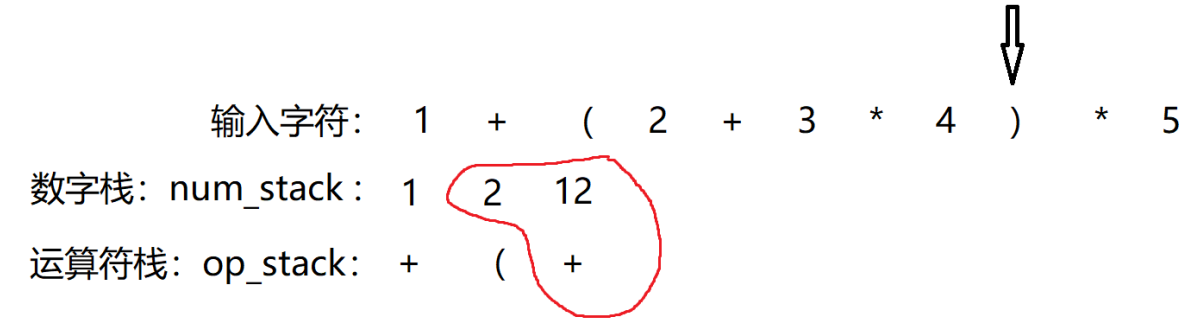
开始计算：



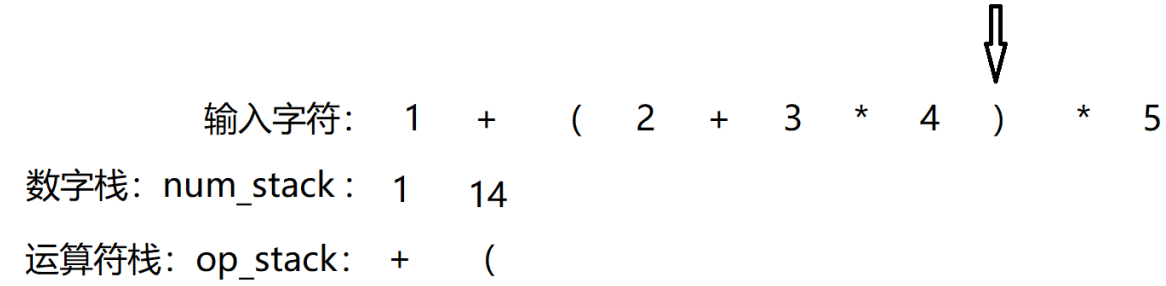
两个操作数和一个操作符弹出栈，然后将计算结果入栈：



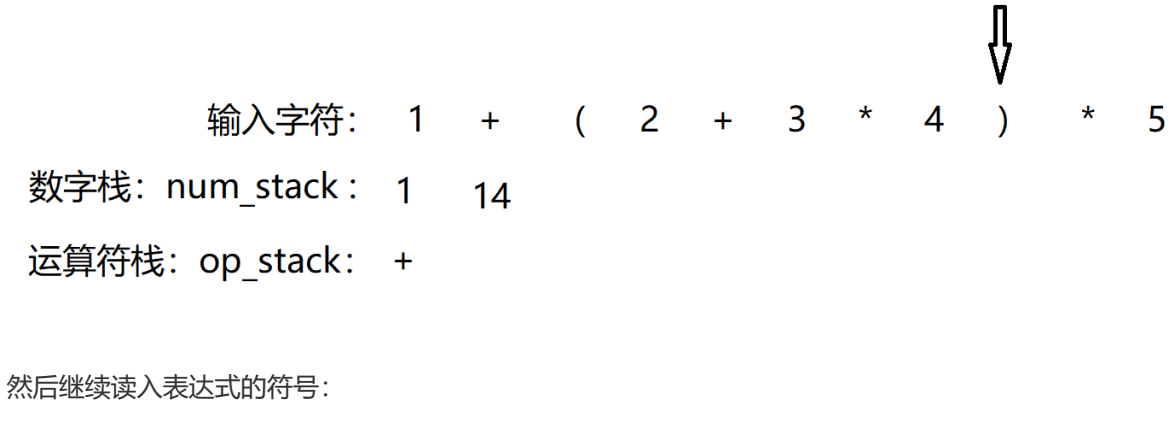
运算符栈顶不是左括号`(`继续计算：



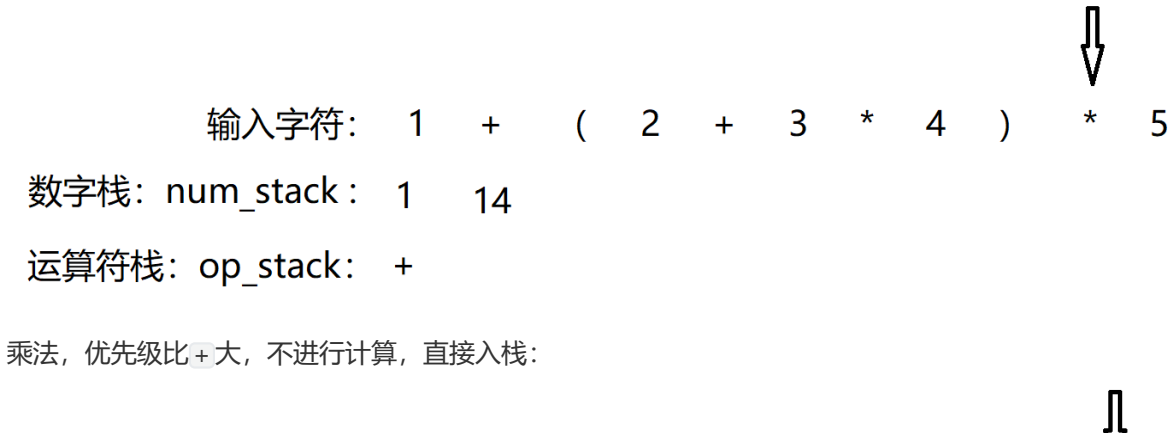
得到：



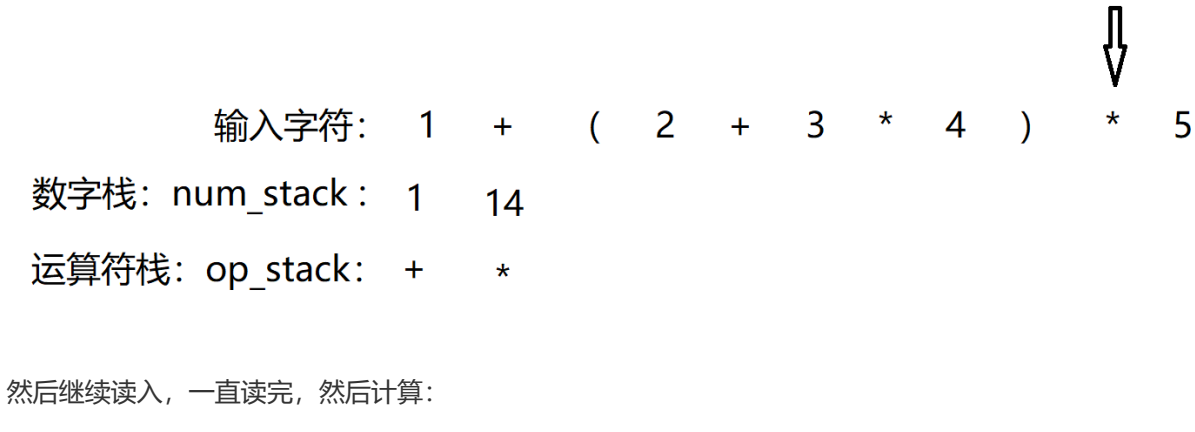
操作符栈顶是（计算结束，并将左括号弹出：



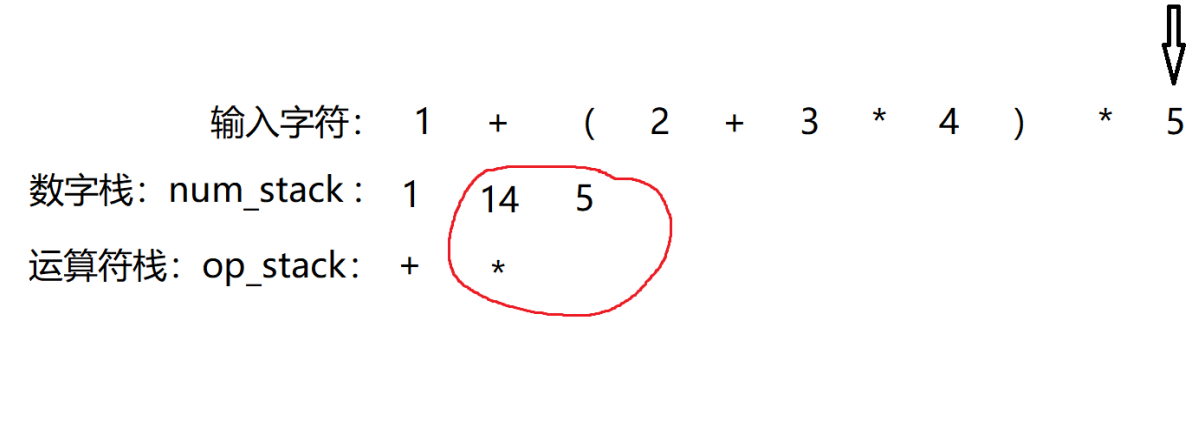
然后继续读入表达式的符号：

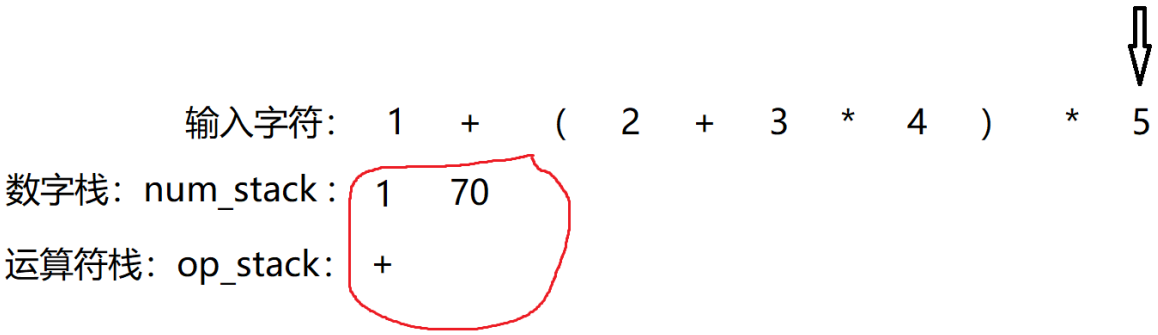


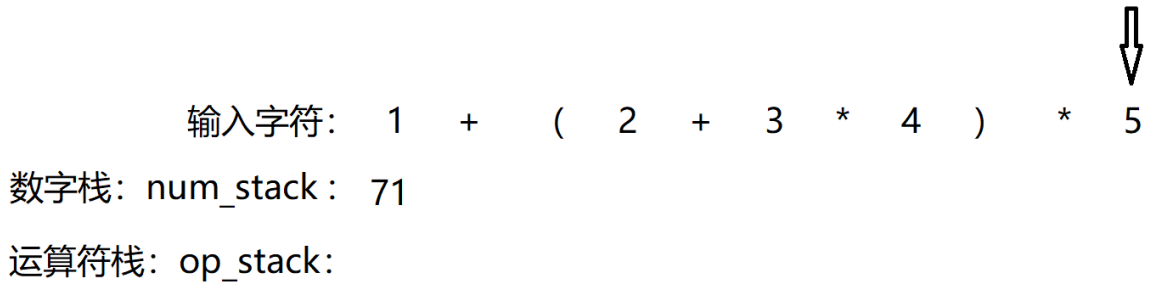
乘法，优先级比+大，不进行计算，直接入栈：



然后继续读入，一直读完，然后计算：







数字栈的栈顶就是最后的结果：`71`

其实总的来说，用栈来模拟数的计算是这样的一个过程：

1. 当前运算符的优先级小于栈顶的运算符时出栈并计算这个操作，来模拟树的向上走
2. 当前运算符的优先级大于栈顶的运算符时运算符直接入栈操作，来模拟树的向下走
3. 当表达式带括号的时候树是不一样的，但是运算符的优先级顺序仍然符合建树规则，此时算法中遇到左括号直接入栈，遇到右括号一直计算直到遇到左括号这个操作就是用来模拟树中对括号中表达式的操作
4. 数字直接入栈，以及符号栈为空时当前符号直接入栈其实用来模拟树的中序遍历

其实上述的模拟过程可以有严格的证明，但是需要一些严格的数学表达，如果对证明感兴趣可以看这里，这个证明只是对树计算正确性的证明，并没有证明我们的算法为什么是对的，我有一个不成熟的想法就是利用数学工具建立栈模拟向树计算的严格映射来证明二者的等价性，证明树计算是对的也就证明了栈模拟式对的。

代码实现：

给定一个表达式，其中运算符仅包含 `+,-,*,/` （加 减 乘 整除），可能包含括号，请你求出表达式的最终值。

注意：

- 数据保证给定的表达式合法。
- 题目保证符号 `-` 只作为减号出现，不会作为负号出现，例如， `-1+2` ，`(2+2)*(-(1+1)+2)` 之类表达式均不会出现。
- 题目保证表达式中所有数字均为正整数。
- 题目保证表达式在中间计算过程以及结果中，均不超过 $2^{31}-1$。
- 题目中的整除是指向 0 取整，也就是说对于大于 0 的结果向下取整，例如 $5/3=1$，对于小于 0 的结果向上取整，例如 $5/(1-4)=-1$。
- C++和Java中的整除默认是向零取整；Python中的整除 `//` 默认向下取整，因此Python的 `eval()` 函数中的整除也是向下取整，在本题中不能直接使用。

输入格式

共一行，为给定表达式。

输出格式

共一行，为表达式的结果。

数据范围

表达式的长度不超过 10^5。

输入样例：

```
(2+2)*(1+1)
```

输出样例：

```
8
```

```cpp
#include<iostream>
#include<cstring>
#include<algorithm>
#include<stack>
#include<unordered_map>

using namespace std;

stack<int> num;
stack<char> op;

void eval(){
    //读取第二个运算数
    auto b = num.top();
    num.pop();

    //读取第一个运算数
    auto a = num.top();
    num.pop();

    //取出运算符
    auto c = op.top();
    op.pop();

```

```
    int x;
    //进行计算
    if(c == '+') x = a + b;
    else if(c == '-') x = a - b;
    else if(c == '*') x = a * b;
    else x = a / b;

    num.push(x);
}

int main(){
    unordered_map<char,int> pr{{'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}};

    string str;
    cin >> str;//读入表达式

    for(int  i = 0; i < str.size(); i++){
        auto c = str[i];
        if(isdigit(c)){
            int x = 0, j = i;
            while(j < str.size() && isdigit(str[j])){
                x = x * 10 + str[j ++] - '0';//当前如果是数字，则将字符转化为数字，注意数字12是先读入1，再读入2，所以需要1*10 + 2

            }
            num.push(x);
            i = j - 1;//因为for循环会自动加一，所以再这里需要减掉一

        }else if(c == '('){//若是左括号，直接将括号压入栈中
            op.push(c);

        }else if(c == ')'){//若是右括号，则反复计算，一直遇到左括号
            while(op.top() != '('){
                eval();
            }
            op.pop();//再将左括号弹出

        }else{//正常的运算符
            //若运算符栈中有符号，并且当前运算符的优先级小于栈中的运算符的优先级
            //则进行计算
            //运算符优先级相等的话也要进行计算，因为运算符是从左到右的优先级
            //栈中此时最多有两个运算符，跟运算符优先级的种类有关
            //假如当前的运算符比里面的运算符的优先级都大，则必须全部计算完
            //例如：1+3*4+2，读到第二个加号的时候栈中其实没有进行任何运算
            while(op.size() && pr[op.top()] >= pr[c]){
                eval();
            }
            //运算符栈中为空，直接将当前运算符压入栈中
            //运算符栈顶的优先级大于当前运算符的优先级，表示上一个运算符的优先级小于当前运算符的优先级，压入栈中
            op.push(c);
        }
```

```
    }
    //当整个表达式入栈读完了之后，一直计算，由于优先级个数不超过2，所以栈中符号的个数也不会超过两个
    //可以用整个表达式外面套着一层大括号来理解
    while(op.size()) eval();
    cout << num.top() << endl;

    return 0;
}
```

后缀表达式求值

后缀表达式的计算很简单，就是一个单纯的栈，遇到符号就把栈中的数字计算，然后结果入栈

题目描述：

读入一个后缀表达式（字符串），只含有0-9组成的运算数及
加（+）、减（-）、乘（*）、除（/，整除）四种运算符，以及空格。

每个运算数之间用一个空格隔开，不需要判断给你的表达式是否合法。以'#'作为结束标志。

保证中间的运算过程不会爆INT

输入格式：
一行字符

输出格式：
一个整数，表示表达式的值。

输入样例：

```
16 9 4 3 +*-#
1
```

输出样例：

```
-47
1
```

数据规模与约定：

表达式长度小于255
————————————————————

核心代码：

```
#include<iostream>
#include<cstring>
using namespace std;
```

```
int stack[260];
int top=0;

int main()
{
    char a[260];
    cin.getline(a,260);
    int i=0,x=0,t1,t2;
    while(a[i]!='#')
    {
        if(a[i]>='0' && a[i]<='9')  {
        x=0;
            //用于处理数字大于10的情况
            while(a[i]>='0' && a[i]<='9')
            {
                x=x*10+a[i]-'0';
                i++;
            }
            stack[++top]=x;
        }
        if(a[i]=='+')
        {
            t1=stack[top--];
            t2=stack[top--];
            stack[++top]=t1+t2;
        }
        else if(a[i]=='*')
        {
            t1=stack[top--];
            t2=stack[top--];
            stack[++top]=t1*t2;
        }
        else if(a[i]=='-')
        {
            t2=stack[top--];
            t1=stack[top--];
            stack[++top]=t1-t2;
        }
        else if(a[i]=='/')
        {
            t2=stack[top--];
            t1=stack[top--];
            stack[++top]=t1/t2;
        }
        i++;
    }
    cout<<stack[top];
    return 0;
}

```

参考:

1. 栈模拟中缀表达式求值的过程与简单分析
2. C++ 后缀表达式求值
3. 表达式求值：多图讲解运算符优先级+详细代码注释