

原

算术表达式的合法性判断与求值（下）

2014年10月07日 11:05:47 神奕 阅读量: 4775 更多

版权声明：本文为博主原创文章，未经博主允许不得转载。 <https://blog.csdn.net/lsong694767315/article/details/39852507>

<< [算术表达式的合法性判断与求值（上）](#)

上一篇讲到，通过编译原理的方法（词法分析和语法分析）来判断字符串表示的算术表达式的合法性。这一篇，接着讲在算术表达式合法的情况下，对表达式进行求值。

问题：给定一个字符串，只包含 '+'、'-'、'\*'、'/'、数字、小数点、'('、')'。  
要求：(1) 判断该算术表达式是否合法；(2) 如果合法，计算该表达式的值。

### 三、算术表达式的求值

表达式的求值是栈应用的一个典型范例。我们一般通过**后缀表达式（逆波兰式）**进行求值，因为对后缀表达式求值比直接对中缀表达式求值简单很多。**中缀表达式**不仅依赖运算符的优先级，而且还要处理括号，而后缀表达式中已经考虑了运算符的优先级，且没有括号。

所以，这里对表达式的求值分两个步骤进行：首先，把中缀表达式转换为后缀表达式，然后，对后缀表达式求值。

#### 1) 中缀转后缀

在把中缀转后缀的过程中，需要考虑操作符的优先级。根据《数据结构与算法分析》一书中的描述，我们需要利用一个**栈**（存放操作符）和一个**输出字符串Output**，从左到右读入中缀表达式：

1. 如果字符是操作数，将它添加到 Output。
2. 如果字符是操作符，从栈中弹出操作符，到 Output 中，直到遇到左括号 或 优先级较低的操作符（并不弹出）。然后把这个操作符 push 入栈。
3. 如果字符是左括号，无理由入栈。
4. 如果字符是右括号，从栈中弹出操作符，到 Output 中，直到遇到左括号。（左括号只弹出，不放入输出字符串）
5. 中缀表达式读完以后，如果栈不为空，从栈中弹出所有操作符并添加到 Output 中。

好了，下面直接上代码：

```
1  #include <iostream>
2  #include <string>
3  #include <stack>
4  using namespace std;
5
6  // 获取运算符的优先级
7  int prior(char c)
8  {
9      switch (c)
10     {
11         case '+':
12         case '-':
13             return 1;
14         case '*':
15         case '/':
16             return 2;
17         default:
18             return 0;
19     }
20 }
21
22 // 判断是否是运算符
23 bool isOperator(char c)
24 {
25     switch (c)
26     {
27         case '+':
28         case '-':
29         case '*':
30         case '/':
31             return true;
32         default:
33             return false;
34     }
35 }
36
37 // 中缀转后缀
38 string getPostfix(const string& expr)
39 {
40     string output; // 输出
41     stack<char> s; // 操作符栈
42     for(int i=0; i<expr.size(); ++i)
43     {
44         char c = expr[i];
45         if(isOperator(c))
46         {
47             while(!s.empty() && isOperator(s.top()) && prior(s.top())>=prior(c))
48             {
49                 output.push_back(s.top());
50                 s.pop();
51             }
52             s.push(c);
53         }
54         else if(c == '(')
55         {
56             s.push(c);
57         }
58         else if(c == ')')
59         {
60             while(s.top() != '(')
61             {
62                 output.push_back(s.top());
63                 s.pop();
64             }
65             s.pop();
66         }
67         else
68         {
69             output.push_back(c);
70         }
71     }
```

```

72     while (!s.empty())
73     {
74         output.push_back(s.top());
75         s.pop();
76     }
77     return output;
78 }
79
80 int main()
81 {
82     string expr = "a+b*c+(d*e+f)*g";
83     string postfix = getPostfix(expr);
84     cout << expr << endl << postfix << endl;
85     return 0;
86 }

```

相信应该不需要我再解释什么了，请对照上面的规则看代码。

## 2) 后缀表达式求值

得到了后缀表达式以后，对后缀表达式的求值就变得非常简单了。只需要使用一个栈，从左到右读入后缀表达式：

1. 如果字符是操作数，把它压入堆栈。
2. 如果字符是操作符，从栈中弹出两个操作数，执行相应的运算，然后把结果压入堆栈。（如果不能连续弹出两个操作数，说明表达式不正确）
3. 当表达式扫描完以后，栈中存放的就是最后的计算结果。

好了，话不多说，直接上代码：

```

1  #include <iostream>
2  #include <string>
3  #include <stack>
4  using namespace std;
5
6  int prior(char c)
7  {
8      switch (c)
9      {
10         case '+':
11         case '-':
12             return 1;
13         case '*':
14         case '/':
15             return 2;
16         default:
17             return 0;
18     }
19 }
20
21 bool isOperator(char c)
22 {
23     switch (c)
24     {
25         case '+':
26         case '-':
27         case '*':
28         case '/':
29             return true;
30         default:
31             return false;
32     }
33 }
34
35 string getPostfix(const string& expr)
36 {
37     string output; // 输出
38     stack<char> s; // 操作符栈
39     for(int i=0; i<expr.size(); ++i)
40     {
41         char c = expr[i];
42         if(isOperator(c))
43         {
44             while(!s.empty() && isOperator(s.top()) && prior(s.top())>=prior(c))
45             {
46                 output.push_back(s.top());
47                 s.pop();
48             }
49             s.push(c);
50         }
51         else if(c == '(')
52         {
53             s.push(c);
54         }
55         else if(c == ')')
56         {
57             while(s.top() != '(')
58             {
59                 output.push_back(s.top());
60                 s.pop();
61             }
62             s.pop();
63         }
64         else
65         {
66             output.push_back(c);
67         }
68     }
69     while (!s.empty())
70     {
71         output.push_back(s.top());
72         s.pop();
73     }
74     return output;
75 }
76
77 // 从栈中连续弹出两个操作数
78 void popTwoNumbers(stack<int>& s, int& first, int& second)
79 {
80     first = s.top();
81     s.pop();
82     second = s.top();
83     s.pop();
84 }

```

```

83         s.pop();
84     }
85
86     // 计算后缀表达式的值
87     int expCalculate(const string& postfix)
88     {
89         int first, second;
90         stack<int> s;
91         for(int i=0; i<postfix.size(); ++i)
92         {
93             char c = postfix[i];
94             switch (c)
95             {
96                 case '+':
97                     popTwoNumbers(s, first, second);
98                     s.push(second+first);
99                     break;
100                 case '-':
101                     popTwoNumbers(s, first, second);
102                     s.push(second-first);
103                     break;
104                 case '*':
105                     popTwoNumbers(s, first, second);
106                     s.push(second*first);
107                     break;
108                 case '/':
109                     popTwoNumbers(s, first, second);
110                     s.push(second/first);
111                     break;
112                 default:
113                     s.push(c-'0');
114                     break;
115             }
116         }
117         int result = s.top();
118         s.pop();
119         return result;
120     }
121
122     int main()
123     {
124         string expr = "5+2*(6-3)-4/2";
125         string postfix = getPostfix(expr);
126         int result = expCalculate(postfix);
127         cout << "The result is: " << result << endl;
128         return 0;
129     }

```

注意，示例中的操作数都是单个的字符（0-9），但是通常的表达式不会是这种特殊情况，这就是我们需要对表达式进行词法解析的原因。

## 四、解决问题

好了，下面我们就结合上篇讲的[词法分析](#)对一个含有整数或小数的表达式进行求值。

因为操作数不再是单个字符（个位数），我们需要对表达式进行词法解析。这里经过解析后，将(单词, 种别编码)对存入到一个 `vector<pair<string, int>>` 中，所以我们的中缀转后缀、后缀表达式求值都是对这个 `vector` 结构进行遍历。

假设表达式已经判断为合法，求值的完整代码如下：

```

1  #include <iostream>
2  #include <sstream>
3  #include <string>
4  #include <vector>
5  #include <stack>
6  #include <utility>
7  using namespace std;
8
9  int word_analysis(vector<pair<string, int>>& word, const string expr)
10 {
11     for(int i=0; i<expr.length(); ++i)
12     {
13         // 如果是 + - * / ( )
14         if(expr[i] == '(' || expr[i] == ')' || expr[i] == '+'
15            || expr[i] == '-' || expr[i] == '*' || expr[i] == '/')
16         {
17             string tmp;
18             tmp.push_back(expr[i]);
19             switch (expr[i])
20             {
21                 case '+':
22                     word.push_back(make_pair(tmp, 1));
23                     break;
24                 case '-':
25                     word.push_back(make_pair(tmp, 2));
26                     break;
27                 case '*':
28                     word.push_back(make_pair(tmp, 3));
29                     break;
30                 case '/':
31                     word.push_back(make_pair(tmp, 4));
32                     break;
33                 case '(':
34                     word.push_back(make_pair(tmp, 6));
35                     break;
36                 case ')':
37                     word.push_back(make_pair(tmp, 7));
38                     break;
39             }
40         }
41         // 如果是数字开头
42         else if(expr[i]>='0' && expr[i]<='9')
43         {
44             string tmp;
45             while(expr[i]>='0' && expr[i]<='9')
46             {
47                 tmp.push_back(expr[i]);
48                 ++i;
49             }
50             if(expr[i] == '.')
51             {
52                 ++i;
53                 if(expr[i]>='0' && expr[i]<='9')

```

```

54         {
55             tmp.push_back('.');
56             while(expr[i]>='0' && expr[i]<='9')
57             {
58                 tmp.push_back(expr[i]);
59                 ++i;
60             }
61         }
62         else
63         {
64             return -1; // .后面不是数字, 词法错误
65         }
66     }
67     word.push_back(make_pair(tmp, 5));
68     --i;
69 }
70 // 如果以.开头
71 else
72 {
73     return -1; // 以.开头, 词法错误
74 }
75 }
76 return 0;
77 }
78
79 // 获取运算符的优先级
80 int prior(int sym)
81 {
82     switch (sym)
83     {
84         case 1:
85         case 2:
86             return 1;
87         case 3:
88         case 4:
89             return 2;
90         default:
91             return 0;
92     }
93 }
94
95 // 通过 种别编码 判定是否是运算符
96 bool isOperator(int sym)
97 {
98     switch (sym)
99     {
100         case 1:
101         case 2:
102         case 3:
103         case 4:
104             return true;
105         default:
106             return false;
107     }
108 }
109
110 // 中缀转后缀
111 vector<pair<string, int>> getPostfix(const vector<pair<string, int>>& expr)
112 {
113     vector<pair<string, int>> output; // 输出
114     stack<pair<string, int>> s; // 操作符栈
115     for(int i=0; i<expr.size(); ++i)
116     {
117         pair<string, int> p = expr[i];
118         if(isOperator(p.second))
119         {
120             while(!s.empty() && isOperator(s.top().second) && prior(s.top().second)>=prior(p.se
121             {
122                 output.push_back(s.top());
123                 s.pop();
124             }
125             s.push(p);
126         }
127         else if(p.second == 6)
128         {
129             s.push(p);
130         }
131         else if(p.second == 7)
132         {
133             while(s.top().second != 6)
134             {
135                 output.push_back(s.top());
136                 s.pop();
137             }
138             s.pop();
139         }
140         else
141         {
142             output.push_back(p);
143         }
144     }
145     while (!s.empty())
146     {
147         output.push_back(s.top());
148         s.pop();
149     }
150     return output;
151 }
152
153 // 从栈中连续弹出两个操作数
154 void popTwoNumbers(stack<double>& s, double& first, double& second)
155 {
156     first = s.top();
157     s.pop();
158     second = s.top();
159     s.pop();
160 }
161
162 // 把string转换为double
163 double stringToDouble(const string& str)
164 {
165     double d;
166     stringstream ss;
167     ss << str;
168     ss >> d;

```

```

169         return d;
170     }
171 }
172 // 计算后缀表达式的值
173 double expCalculate(const vector<pair<string, int>>& postfix)
174 {
175     double first, second;
176     stack<double> s;
177     for(int i=0; i<postfix.size(); ++i)
178     {
179         pair<string, int> p = postfix[i];
180         switch (p.second)
181         {
182             case 1:
183                 popTwoNumbers(s, first, second);
184                 s.push(second+first);
185                 break;
186             case 2:
187                 popTwoNumbers(s, first, second);
188                 s.push(second-first);
189                 break;
190             case 3:
191                 popTwoNumbers(s, first, second);
192                 s.push(second*first);
193                 break;
194             case 4:
195                 popTwoNumbers(s, first, second);
196                 s.push(second/first);
197                 break;
198             default:
199                 s.push(stringToDouble(p.first));
200                 break;
201         }
202     }
203     double result = s.top();
204     s.pop();
205     return result;
206 }
207
208 int main()
209 {
210     string expr = "(1.5+2.5)*2-0.5";
211     vector<pair<string, int>> word;
212     int err_num = word_analysis(word, expr);
213     if (~1 == err_num)
214         cout << "Word Error!" << endl;
215     else
216     {
217         double result = expCalculate(getPostfix(word));
218         cout << expr + " = " << result << endl;
219     }
220     return 0;
221 }

```

为了防止精度的损失，不论是整数还是小数，在这里都通过stringToDouble()函数转为 double 浮点数。

## 附：字符串转int/float/double类型

方法一：atoi、atof

在C语言的头文件 stdlib.h 里提供了两个函数，用于将字符串转换为整数或浮点数。函数原型分别为：

```

1 int atoi(const char *nptr);           // 字符串转整数
2 double atof(const char *npnr);       // 字符串转浮点数

```

方法二：stringstream

在C++里，可以利用 stringstream 方便的将 string 转换为 int、float、double：

```

1 double stringToDouble(const string& str) {
2     double d;
3     stringstream ss;
4     ss << str;    // 把字符串写入字符流
5     ss >> d;      // 输出到double
6     return d;
7 }
8
9 string doubleToString(const double& d) {
10    string str;
11    stringstream ss;
12    ss << d;
13    ss >> str;
14    return str;
15 }

```

通过 stringstream 将 string 转换为 int 或 float 与上面的方法是一样的，只需要改一下变量的类型就可以了。

(全文完)

