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 阅读数：4775 更多

<< 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：

```cpp
#include <iostream>
#include <string>
#include <stack>
using namespace std;

// 获取运算符的优先级
int prior(char c)
{
	switch (c)
	{
		case '+':
		case '-':
			return 1;
		case '*':
		case '/':
			return 2;
		default:
			return 0;
	}
}

// 判断是否是运算符
bool isOperator(char c)
{
	switch (c)
	{
		case '+':
		case '-':
		case '*':
		case '/':
			return true;
		default:
			return false;
	}
}

// 中缀转后缀
string getPostfix(const string& expr)
{
	string output;  // 输出
	stack<char> s;  // 操作符栈
	for(int i=0; i<expr.size(); ++i)
	{
		char c = expr[i];
		if(isOperator(c))
		{
			while(!s.empty() && isOperator(s.top()) && prior(s.top())>=prior(c))
			{
				output.push_back(s.top());
				s.pop();
			}
			s.push(c);
		}
		else if(c == '(')
		{
			s.push(c);
		}
		else if(c == ')')
		{
			while(s.top() != '(')
			{
				output.push_back(s.top());
				s.pop();
			}
			s.pop();
		}
		else
		{
			output.push_back(c);
		}
	}
```

```
        while (!s.empty())
        {
                output.push_back(s.top());
                s.pop();
        }
        return output;
}

int main()
{
        string expr = "a+b*c+(d*e+f)*g";
        string postfix = getPostfix(expr);
        cout << expr << endl << postfix << endl;
        return 0;
}
```

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：

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;

int prior(char c)
{
        switch (c)
        {
                case '+':
                case '-':
                        return 1;
                case '*':
                case '/':
                        return 2;
                default:
                        return 0;
        }
}

bool isOperator(char c)
{
        switch (c)
        {
                case '+':
                case '-':
                case '*':
                case '/':
                        return true;
                default:
                        return false;
        }
}

string getPostfix(const string& expr)
{
        string output;  // 输出
        stack<char> s;  // 操作符栈
        for(int i=0; i<expr.size(); ++i)
        {
                char c = expr[i];
                if(isOperator(c))
                {
                        while(!s.empty() && isOperator(s.top()) && prior(s.top())>=prior(c))
                        {
                                output.push_back(s.top());
                                s.pop();
                        }
                        s.push(c);
                }
                else if(c == '(')
                {
                        s.push(c);
                }
                else if(c == ')')
                {
                        while(s.top() != '(')
                        {
                                output.push_back(s.top());
                                s.pop();
                        }
                        s.pop();
                }
                else
                {
                        output.push_back(c);
                }
        }
        while (!s.empty())
        {
                output.push_back(s.top());
                s.pop();
        }
        return output;
}

// 从栈中连续弹出两个操作数
void popTwoNumbers(stack<int>& s, int& first, int& second)
{
        first = s.top();
        s.pop();
        second = s.top();
```

```
        s.pop();
}

// 计算后缀表达式的值
int expCalculate(const string& postfix)
{
        int first,second;
        stack<int> s;
        for(int i=0; i<postfix.size(); ++i)
        {
                char c = postfix[i];
                switch (c)
                {
                case '+':
                        popTwoNumbers(s, first, second);
                        s.push(second+first);
                        break;
                case '-':
                        popTwoNumbers(s, first, second);
                        s.push(second-first);
                        break;
                case '*':
                        popTwoNumbers(s, first, second);
                        s.push(second*first);
                        break;
                case '/':
                        popTwoNumbers(s, first, second);
                        s.push(second/first);
                        break;
                default:
                        s.push(c-'0');
                        break;
                }
        }
        int result = s.top();
        s.pop();
        return result;
}

int main()
{
        string expr = "5+2*(6-3)-4/2";
        string postfix = getPostfix(expr);
        int result = expCalculate(postfix);
        cout << "The result is: " << result << endl;
        return 0;
}
```

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：

```
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <stack>
#include <utility>
using namespace std;

int word_analysis(vector<pair<string, int>>& word, const string expr)
{
        for(int i=0; i<expr.length(); ++i)
        {
                // 如果是 + - * / ( )
                if(expr[i] == '(' || expr[i] == ')' || expr[i] == '+'
                        || expr[i] == '-' || expr[i] == '*' || expr[i] == '/')
                {
                        string tmp;
                        tmp.push_back(expr[i]);
                        switch (expr[i])
                        {
                        case '+':
                                word.push_back(make_pair(tmp, 1));
                                break;
                        case '-':
                                word.push_back(make_pair(tmp, 2));
                                break;
                        case '*':
                                word.push_back(make_pair(tmp, 3));
                                break;
                        case '/':
                                word.push_back(make_pair(tmp, 4));
                                break;
                        case '(':
                                word.push_back(make_pair(tmp, 6));
                                break;
                        case ')':
                                word.push_back(make_pair(tmp, 7));
                                break;
                        }
                }
                // 如果是数字开头
                else if(expr[i]>='0' && expr[i]<='9')
                {
                        string tmp;
                        while(expr[i]>='0' && expr[i]<='9')
                        {
                                tmp.push_back(expr[i]);
                                ++i;
                        }
                        if(expr[i] == '.')
                        {
                                ++i;
                                if(expr[i]>='0' && expr[i]<='9')
```

```
                                {
                                        tmp.push_back('.');
                                        while(expr[i]>='0' && expr[i]<='9')
                                        {
                                                tmp.push_back(expr[i]);
                                                ++i;
                                        }
                                }
                                else
                                {
                                        return -1;  // .后面不是数字，词法错误
                                }
                        }
                        word.push_back(make_pair(tmp, 5));
                        --i;
                }
                // 如果以.开头
                else
                {
                        return -1;  // 以.开头，词法错误
                }
        }
        return 0;
}

// 获取运算符的优先级
int prior(int sym)
{
        switch (sym)
        {
                case 1:
                case 2:
                        return 1;
                case 3:
                case 4:
                        return 2;
                default:
                        return 0;
        }
}

// 通过 种别编码 判定是否是运算符
bool isOperator(int sym)
{
        switch (sym)
        {
                case 1:
                case 2:
                case 3:
                case 4:
                        return true;
                default:
                        return false;
        }
}

// 中缀转后缀
vector<pair<string, int>> getPostfix(const vector<pair<string, int>>& expr)
{
        vector<pair<string, int>> output;  // 输出
        stack<pair<string, int>> s;  // 操作符栈
        for(int i=0; i<expr.size(); ++i)
        {
                pair<string, int> p = expr[i];
                if(isOperator(p.second))
                {
                        while(!s.empty() && isOperator(s.top().second) && prior(s.top().second)>=prior(p.se
                        {
                                output.push_back(s.top());
                                s.pop();
                        }
                        s.push(p);
                }
                else if(p.second == 6)
                {
                        s.push(p);
                }
                else if(p.second == 7)
                {
                        while(s.top().second != 6)
                        {
                                output.push_back(s.top());
                                s.pop();
                        }
                        s.pop();
                }
                else
                {
                        output.push_back(p);
                }
        }
        while (!s.empty())
        {
                output.push_back(s.top());
                s.pop();
        }
        return output;
}

// 从栈中连续弹出两个操作数
void popTwoNumbers(stack<double>& s, double& first, double& second)
{
        first = s.top();
        s.pop();
        second = s.top();
        s.pop();
}

// 把string转换为double
double stringToDouble(const string& str)
{
        double d;
        stringstream ss;
        ss << str;
        ss >> d;
```

```
        return d;
}

// 计算后缀表达式的值
double expCalculate(const vector<pair<string, int>>& postfix)
{
        double first,second;
        stack<double> s;
        for(int i=0; i<postfix.size(); ++i)
        {
                pair<string,int> p = postfix[i];
                switch (p.second)
                {
                case 1:
                        popTwoNumbers(s, first, second);
                        s.push(second+first);
                        break;
                case 2:
                        popTwoNumbers(s, first, second);
                        s.push(second-first);
                        break;
                case 3:
                        popTwoNumbers(s, first, second);
                        s.push(second*first);
                        break;
                case 4:
                        popTwoNumbers(s, first, second);
                        s.push(second/first);
                        break;
                default:
                        s.push(stringToDouble(p.first));
                        break;
                }
        }
        double result = s.top();
        s.pop();
        return result;
}

int main()
{
        string expr = "(1.5+2.5)*2-0.5";
        vector<pair<string, int>> word;
        int err_num = word_analysis(word, expr);
        if (-1 == err_num)
                cout << "Word Error!" << endl;
        else
        {
                double result = expCalculate(getPostfix(word));
                cout << expr + " = " << result << endl;
        }
        return 0;
}
```

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：

```
int atoi(const char *nptr);          // 字符串转整数
double atof(const char *nptr);       // 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：

```
double stringToDouble(const string& str) {
        double d;
        stringstream ss;
        ss << str;    // 把字符串写入字符流
        ss >> d;      // 输出到double
        return d;
}

string doubleToString(const double& d) {
        string str;
        stringstream ss;
        ss << d;
        ss >> str;
        return str;
}
```

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）