表达式求值

波兰式、逆波兰式，中缀式表达式 表达式求值

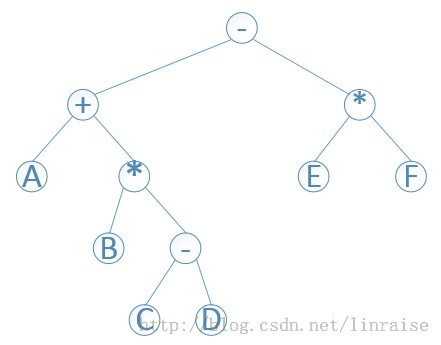
逆波兰式，波兰式，中缀式表达式的求值问题

1. 中缀表达式

人类最熟悉的一种表达式1+2，(1+2)\*3，3+4\*2+4等等都是中缀表示法。对于人们来说，也是最直观的一种求值方式，先算括号里的，然后算乘除，最后算加减，但是，计算机处理中缀表达式却并不方便，因为没有一种简单的数据结构可以方便从一个表达式中间抽出

一部分算完结果，再放进去，然后继续后面的计算（链表也许可以，但是，代价也是不菲）。

在介绍前缀，后缀表达式之前，我想先通过我们最熟悉的中缀表达式画出一棵语法树来直观认识前后缀表达式。以A+B\*(C-D)-E\*F为例：



则中缀表达式得名于它是由相应的语法树的中序遍历的结果得到的。

2. 前缀缀表达式

　　前缀表达式又叫做波兰式。同样的道理，表达式的前缀表达式是由相应的语法树的前序遍历的结果得到的。

如上图的前缀表达式为-

+ A \* B - C D \* E F

由前缀表达式求出结果有下面两种思路：

　　1.从左至右扫描表达式，如果一个操作符后面跟着两个操作数时，则计算，然后将结果作为操作数替换(这个操作符和两个操作数)，

重复此步骤，直至所有操作符处理完毕。如-+A\*B-CD\*EF，扫描到-CD时，会计算C-D=C',表达式变成：-+A\*BC'\*EF

继续扫描到\*BC',计算B\*C'=B',表达式变成:-+AB'\*EF,继续+AB'，依此类推。

　　2.由1.知，要多遍扫描表达式，并且需要将3个字符替换成1个，比较繁锁，我们可以用一个栈S2来实现计算，扫描从右往左进行，

如果扫描到操作数，则压进S2，如果扫描到操作符，则从S2弹出两个操作数进行相应的操作，并将结果压进S2(S2的个数出2个进1个),

当扫描结束后，S2的栈顶就是表达式结果。

3. 后缀表达式

　　后缀表达式又叫做逆波兰式。它是由相应的语法树的后序遍历的结果得到的。如上图的后缀表达式为：

A B C D - \* + E F \* -

由前缀表达式求出结果十分方便，只需要用一个栈实现：

我们可以用一个栈S2来实现计算，扫描从左往右进行，如果扫描到操作数，则压进S2，如果扫描到操作符，则从S2弹出两个操作数

进行相应的操作，并将结果压进S2(S2的个数出2个进1个),当扫描结束后，S2的栈顶就是表达式结果。后缀表达式和前缀表达式看

起来就像一对逆过程，实际上并不是这样子，因为字符读取的时候都是从左往右的，所以，前缀表达式往往需要用两个栈来计算，

其中一个栈用来预处理：将字符串倒序压进栈中。

4. 中缀表达式转换成后缀表达式

　　既然中缀表达式对于计算机的运算并不便利，而前缀后缀表达式的计算相对简单方便。因此，找到一种途径将中缀表达式

转换成前缀后缀表达式就十分重要。实际上，二者的转换算法看起来也很像一个逆过程。因此，我们着重讨论中缀转后缀。

从理论上讲，已知一棵二叉树的中序遍历序列，要求出它的后序遍历序列是不唯一的，即文法是有多义性的。但是，在这

里加上了优先级这一限制条件，转换就变得唯一了。

算法：中缀表达式转换成后缀表达式

输入：中缀表达式串

输出：后缀表达式串

PROCESS BEGIN:

1.从左往右扫描中缀表达式串s，对于每一个操作数或操作符，执行以下操作;

2.IF (扫描到的s[i]是操作数DATA)

　　　　　 将s[i]添加到输出串中;

3.IF (扫描到的s[i]是开括号'(')

将s[i]压栈;

4.WHILE (扫描到的s[i]是操作符OP)

IF (栈为空 或 栈顶为'(' 或 扫描到的操作符优先级比栈顶操作符高)

将s[i]压栈;

BREAK;

ELSE

出栈至输出串中

5.IF (扫描到的s[i]是闭括号')')

栈中运算符逐个出栈并输出，直到遇到开括号'(';

开括号'('出栈并丢弃;

6.返回第1.步

　　 7.WHILE (扫描结束而栈中还有操作符)

操作符出栈并加到输出串中

PROCESS END

4. 中缀表达式转换成前缀表达式

　　中缀表达式转换成前缀表达式和中缀表达式转换成后缀表达式十分类似，只需要将扫描方向由前往后变成由后往前，

将'('改为')',')'改为'(',注意其中一个判断优先级的地方需要由>=变成>.

　　至此，理论基础已经讲完了，用一道OJ题目来客观认识一下会比较好理解。HDU2127

Polish notation

Time Limit: 10000/1000 MS (Java/Others) Memory Limit: 32768/32768 K (Java/Others)

Total Submission(s): 112 Accepted Submission(s): 32

Problem Description

Now, give you a string of standard arithmetic expressions, please tell me the Polish notation and the value of expressions.

Input

There're have multi-case. Every case put in one line, the expressions just contain some positive integers(all less than 100, the number of integers less than 20), bi-operand operators(only have 3 kinds : +,-,\*) and some brackets'(',')'.

you can assume the expressions was valid.

Output

Each case output the Polish notation in first line, and the result of expressions was output in second line.

all of the answers are no any spaces and blank line.the answer will be not exceed the 64-signed integer.

Sample Input

1+2-3\*(4-5)

1+2\*(3-4)-5\*6

Sample Output

Case 1:

- + 1 2 \* 3 - 4 5

6

Case 2:

- + 1 \* 2 - 3 4 \* 5 6

-31

Author

威士忌

Source

HDU 2007-10 Programming Contest

解释：

　　网上很少这道题的题解，我先写一个吧，前面题目扯了一堆废话（我全删掉了），只有一句最重要的：给你中缀表达式，求出它的波兰式并计算出

结果。题目是很裸的将中缀表达式转换成前缀表达式，再进行计算出结果，可以直接套上面的算法。但是，这里有一些点易错点需要注意：

　 1.转化后的前缀表达式需要留一个空格

　　2.因为数字并不是1位的，因为数字<100，有可能有两位，所以，不能只读取一个字符做为数字，需要读取所有的数字，具体看代码对数字的处理

　　3.要注意有可能超过int的范围，一开始就想当然地用int,结果WA,还要注意将字符串转化为\_\_int64

[cpp] view plaincopy在CODE上查看代码片派生到我的代码片

#include <iostream>

#include <string>

#include <algorithm>

using namespace std;

char stack[500]; //定义顺序栈，其中top==0表示栈为空；

int top; //栈顶指针，为0表示栈为空；

char output[500], input[500]; //波兰式

int outLen;

int priority(char op) //判断运算符级别函数；其中\* /的级别为2，+ -的级别为1；

{

if (op=='+' || op=='-')

return 1;

if (op=='\*' || op=='/')

return 2;

else

return 0;

}

bool isOperator(char op) //判断输入串中的字符是不是操作符，如果是返回true

{

return (op=='+' || op=='-' || op=='\*' || op=='/');

}

void Polish(char \*s,int len) //将一个中缀串转换为前缀串，

{

memset(output,'\0',sizeof output); //输出串

outLen = 0;

for (int i=len-1; i >= 0; --i) //1）求输入串的逆序。

{

if (isdigit(s[i])) //经验见:http://blog.csdn.net/linraise/article/details/18566319#comments

{

output[outLen++] = s[i]; //3）假如是操作数，把它添加到输出串中。

while (i-1 >= 0 && isdigit(s[i-1]))

{

output[outLen++] = s[i-1];

--i;

}

output[outLen++] = ' '; //空格隔开

}

if (s[i]==')') //4）假如是闭括号，将它压栈。

{

++top;

stack[top] = s[i];

}

while (isOperator(s[i])) //5)如果是运算符，执行算法对应操作；

{ //>=很重要

if (top==0 || stack[top]==')' || priority(s[i]) >= priority(stack[top])) //空栈||或者栈顶为)||新来的元素优先级更高

{

++top;

stack[top] = s[i];

break;

}

else

{

output[outLen++] = stack[top];

output[outLen++] = ' ';

--top;

}

}

if (s[i]=='(') //6）假如是开括号，栈中运算符逐个出栈并输出，直到遇到闭括号。闭括号出栈并丢弃。

{

while (stack[top]!=')')

{

output[outLen++] = stack[top];

output[outLen++] = ' ';

--top;

}

--top; // 此时stack[top]==')',跳过)

}

//7）假如输入还未完毕，跳转到步骤2。

}

while (top!=0) //8）假如输入完毕，栈中剩余的所有操作符出栈并加到输出串中。

{

output[outLen++] = stack[top];

output[outLen++] = ' ';

--top;

}

}

char DstBuf[200];

char\* OP(char\* op1,char\* op2,char op)

{

\_\_int64 res = 0;

if (op=='+')

res = \_atoi64(op1) + \_atoi64(op2);

else if (op=='-')

res = \_atoi64(op1) - \_atoi64(op2);

else if (op=='\*')

res = \_atoi64(op1) \* \_atoi64(op2);

else if (op=='/')

res = \_atoi64(op1) / \_atoi64(op2);

\_i64toa(res,DstBuf,10);

return DstBuf;

}

char cSt1[200][80], cSt2[200][80];

\_\_int64 calc(char \*s) //波兰式需要用两个栈，逆波兰式只需要一个栈

{

int top1=0, top2=0, i;

for (i=0; s[i]; ++i)

{

if (s[i] && s[i] != ' ')

{

++top1;

sscanf(s+i,"%s",cSt1[top1]); //初始化其中一个栈

while (s[i] && s[i] != ' ')

++i;

}

}

while (top1 != 0) //栈不空

{

if (!isdigit(cSt1[top1][0])) //是操作符

{

OP(cSt2[top2], cSt2[top2-1], cSt1[top1][0]);

memcpy(cSt2[top2-1],DstBuf,sizeof DstBuf);

--top2; //操作数栈：出栈俩，进栈一

--top1; //操作符出栈

}

else

{

++top2; //数进操作数栈

memcpy(cSt2[top2],cSt1[top1],sizeof cSt1[top1]);

--top1; //操作符出栈

}

}

return \_atoi64(cSt2[1]);

}

int main()

{

#ifndef ONLINE\_JUDGE

freopen("2.txt","r",stdin);

#endif

int T = 1;

while (gets(input))

{

Polish(input, strlen(input));

reverse(output,output+outLen-1);

output[outLen-1] = '\0';

printf("Case %d:\n%s\n",T++,output);

printf("%I64d\n",calc(output));

}

return 0;

}

做完了波兰式，再看一个逆波兰式的题：HDU1237

这道是裸的逆波兰，只是简化版本的，我用上面的算法往上一交，结果WA,测试了一下这个用例1 - 3 + 4，结果输出-6,显然是优先级顺序没有处理好，

改了一下prirority就行了

更简单的做法可以看这里，本质上是简化了的逆波兰式解法。

[cpp] view plaincopy在CODE上查看代码片派生到我的代码片

#include <iostream>

#include <string>

#include <algorithm>

using namespace std;

char stack[500]; //定义顺序栈，其中top==0表示栈为空；

int top; //栈顶指针，为0表示栈为空；

char output[500], input[500]; //波兰式

int outLen;

int priority(char op) //判断运算符级别函数；其中\* /的级别为2，+ -的级别为1；

{

if (op=='+' || op=='-')

return 1;

if (op=='\*' || op=='/')

return 2;

else

return 0;

}

bool isOperator(char op) //判断输入串中的字符是不是操作符，如果是返回true

{

return (op=='+' || op=='-' || op=='\*' || op=='/');

}

void rePolish(char \*s,int len) //将一个中缀串转换为后缀串，

{

memset(output,'\0',sizeof output); //输出串

outLen = 0;

for (int i=0; i < len; ++i) //1）求输入串的逆序。

{

if (isdigit(s[i])) //经验见:http://blog.csdn.net/linraise/article/details/18566319#comments

{

output[outLen++] = s[i]; //3）假如是操作数，把它添加到输出串中。

while (i+1 < len && isdigit(s[i+1]))

{

output[outLen++] = s[i+1];

++i;

}

output[outLen++] = ' '; //空格隔开

}

if (s[i]=='(') //4）假如是闭括号，将它压栈。

{

++top;

stack[top] = s[i];

}

while (isOperator(s[i])) //5)如果是运算符，执行算法对应操作；

{

if (top==0 || stack[top]=='(' || priority(s[i]) > priority(stack[top])) //空栈||或者栈顶为)||新来的元素优先级更高

{

++top;

stack[top] = s[i];

break;

}

else

{

output[outLen++] = stack[top];

output[outLen++] = ' ';

--top;

}

}

if (s[i]==')') //6）假如是开括号，栈中运算符逐个出栈并输出，直到遇到闭括号。闭括号出栈并丢弃。

{

while (stack[top]!='(')

{

output[outLen++] = stack[top];

output[outLen++] = ' ';

--top;

}

--top; // 此时stack[top]==')',跳过)

}

//7）假如输入还未完毕，跳转到步骤2。

}

while (top!=0) //8）假如输入完毕，栈中剩余的所有操作符出栈并加到输出串中。

{

output[outLen++] = stack[top];

output[outLen++] = ' ';

--top;

}

}

double OP(double op1,double op2,char op)

{

double res = 0;

if (op=='+')

res = op1 + op2;

else if (op=='-')

res = op1 - op2;

else if (op=='\*')

res = op1 \* op2;

else if (op=='/')

res = op1 / op2;

return res;

}

double cSt1[200];

double calc(char \*s) //波兰式需要用两个栈，逆波兰式只需要一个栈

{

char dst[80];

int top1=0, i;

for (i=0; s[i]; ++i)

{

if (s[i] && s[i] != ' ')

{

sscanf(s+i,"%s",dst);

if (isdigit(dst[0]))

{

++top1;

cSt1[top1] = atof(dst); //进栈

}

else

{

cSt1[top1-1] = OP(cSt1[top1-1], cSt1[top1], dst[0]);

// memcpy(cSt1[top1-1],DstBuf,sizeof DstBuf);

--top1; //操作数栈：出栈俩，进栈一

}

while (s[i] && s[i] != ' ')

++i;

}

}

return cSt1[1];

}

int main()

{

#ifndef ONLINE\_JUDGE

freopen("2.txt","r",stdin);

#endif

int T = 1;

while (gets(input) && strcmp(input,"0"))

{

rePolish(input, strlen(input));

output[outLen-1] = '\0';

// printf("Case %d:\n%s\n",T++,output);

printf("%.2lf\n",calc(output));

}

return 0;

}

//测试用例：1 - 3 + 4

如何编写一个高效的Java表达式求值程序

当然，这个标题是有一点夺人眼球，但我确实这么做了（关于是否相信基准测试结果，这是另一个话题）。

所以，上周我一直在找一个小型、实用的计算数学表达式的类库。偶然间我在stackoverflow上看到了一个帖子，里面推荐的库（Expr）确实是很快而且基本拥有我需要的所有特性。但不幸的是，它不支持提供限制变量范围（在虚拟机里面，所有变量都位于一个全局命名空间）。

所以，我做了一件正常人不会做的事情：重新发明轮子，自己编写一个解析器和执行器。那是一个下雨的周六，我想到了用一个小型递归向下的解析器，一个简化了的、可以计算表达式的抽象语法树。抽象语法树使用一个小型变量管理助手，看起来也没什么大不了的。但它不是没有用。我做出一个初步的实现并且执行速度特别快。在进行了一些测试后，更让我充满信心，它执行的所有运算都正确无误。我想与最前面提到的类库相比，确认这个计算器到底有多快。在没有对每个内部循环和其他的执行进行优化前，我不报太大的期望，毕竟有不少类库是商业软件。所以当我看到测试结果的时候很惊讶。下面的清单展示了一个小的基准测试，使用不同的类库计算同一个表达式。 parsii 是我编写的库，测试时用的是最终版本。这个版本做了一些简化，比如预先计算了常量表达式。但是没有使用任何“黑魔法”，比如生成字节码或者其他类似的操作。

在性能评估中，一个用例是执行表达式”2 + (7 – 5) \* 3.14159 \* x^(12-10) + sin(-3.141)”。其中X的取值范围为0到1000000。测试时先运行10次，对JIT进行预热。然后再运行15次计算平均时间：

PARSII: 28.3 ms

EXPR: 37.2 ms

MathEval: 7748.5 ms

JEP: 647.0 ms

MESP: 220.8 ms

JFEP: 274.3 ms

现在我敢肯定，每一个类库都有自己的优势，所以不能直接对它们进行比较。尽管如此，令人吃惊的是一个简单实现的程序可以拥有这么好的表现。

如果读者对于编译器的原理不太了解的话，下面是一个关于编译器运行机制的简单介绍：

同其他的解析器或者编译器一样，parsii使用了传统的分词器。它将字符流转化成词法单元流，所以”4 + 38″，也就是字符数组’4′, ‘ ‘, ‘+’, ‘ ‘, ’3′ , ‘ ‘, ‘‘, ’8′被转化成：

4 (整数)

+ (符号)

3 (整数)

\* (符号)

8 (整数)

分词器取到一个字符，接着判断是一个什么类型的词法单元，然后再读入这个属于词法单元的所有字符。每一个词法单元都有类型、文本内容并且知道起始位置（行号和字符）。网上有很多深入的教程，所以在这里就不详细讲解了。你可以看一下源代码，但正如我说的，它只是一个初步的实现。

解析器用来将传入的词法单元流翻译成可以执行的AST（抽象语法树），它是一个传统的自上而下递归解析器。这是实现解析器最简单的方式，完全手写，没有利用工具生成。像这样的解析器只拥有一个包含所有语法规则的方法。

同样，关于这种类型的解析器也有很多的教程，但是如何恰当地处理错误却缺少相关的示例。除了解析表达式的速度和正确性外，优秀的错误处理机制是一个优秀解析器的最核心因素之一。正如在源代码里看到的那样，实现起来并不是太困难。因为解析器在解析表达式的过程中从来不会抛出异常，

所有的错误都被收集起来，并且继续尽可能进行解析。即使在第一个错误发生以后已经不能成功解析生成AST，重要的是要能够尽可能的继续解析。因为在一次的执行中我们需要报告尽可能多的错误。这样的方法也同样用在了分词器报告上。比如报告非法格式的词法单元，例如带有2个小数点的浮点数，放到同样的错误列表中。

执行一个解析完成后的抽象语法树非常简单。每一个抽象语法树节点都包含一个计算方法，从根节点开始到父节点会调用这个它。这里的执行结果就是表达式的结果，一个简单的例子就是算数运算，包含了+、-、\*等操作。

执行一个解析完成后的抽象语法树非常简单。每一个抽象语法树节点都包含一个计算方法，它的父亲从根节点开始调用此方法。算数运算，代表了+、-、\*等操作。

为了减少执行时间，程序里运用了3种优化措施：首先，在完成解析AST后，在根节点上进行一个简化的方法调用，并且会扩散到每一个子节点。每一个节点判断自己的子表达式中是否有简化的表达形式。例如：对于算数运算，我们检查2个操作数是不是都是常量（数字)。如果是数字，我们将计算表达式并且返回一个包含计算结果的常量。对于函数，如果所有的参数都是常量的话，也会进行此类优化。

在表达式中使用变量时会执行第二种优化。这里使用map用来在需要的时候来对变量的值进行读写。这肯定是有效的，并且会进行很多次的查找。所以我们有一个叫做Variable类，它包含了变量名称和变量值。在进行表达式解析时，变量在作用域范围内（仅是一个map）只被查找一次，之后就可以一直使用。由于每次查找都返回相同的实例，所以在计算表达式值时变量的访问就像读写字段一样廉价，因为我们刚刚获取了Variable类的值。

第三个也是最后一个优化很可能不是经常起作用。但是由于易于实现，还是应用了这种尤华。它的功能基本上和名字“延迟运算”一样，主要用于函数调用。函数不会自动计算所有参数值，并且调用函数。而“延迟运算”会检查所有的参数，自行决定哪些参数需要计算。在if函数中可以看到它应用的实例。

parsii遵循MIT许可证授权。在GitHub上可以找到所有的源代码，并且包含了预编译的jar包