VCB-Studio教程21: 后缀表达式的求值与转换

本教程旨在讲述后缀表达式的求值与设计

1. 后缀表达式简介

我们平时使用的表达式，叫做中缀表达式（infix notation），表现为1+1这样：算数 算符 算数

而计算机程序中的表达式多采用后缀表达式(suffix noation/inverse polish notation)，表现为：1 1 +， 算数，算数，算符。

为了方便起见，我们先简化假设，所有运算，都是只有两个算数参与，比如常见的四则运算。中缀表达式是依赖括号，以及既定的运算优先度来决定运算顺序。比如说 5+4\*(3+2)3，最先运算的是3+2=5，接着运算53=125，再算4\*125=500，最后算5+500=505。

这些看上去很自然的小学数学题，加了各种扩展（各种高级函数和自定义运算符），到了计算机里面就不是那么自然了。麻烦点主要是括号的处理，以及不同优先级的定义。为了解决这个，计算机普遍使用的是后缀表达式。后缀表达式没有优先级和括号，通过表达式本身的组合来决定运算规则。

将上述表达式用后缀表达式写出来（假设 pow代表乘方），效果是：

5 4 3 2 + 3 pow \* +

如何解析并计算一个后缀表达式，如何将后缀表达式与中缀表达式互转，我们首先需要了解一下计算机中常用的数据结构——堆栈(stack)的概念

2. 堆栈（stack）

堆栈是计算机科学里面一种常见的数据结构，它可以看作一个薯片桶，内部数据呈线性排列，所有数据操作都只能在桶顶部进行：



你能够对栈做这些操作：

初始化（init）：创造一个空的栈。

判断是否为空（isEmpty）：判断这个栈里面有无东西，如果没有，则为空栈。

入栈（push）：把一个数据丢入这个栈里面，只不过你只能丢到最顶端。

出栈（pop）：在栈不为空的前提下（否则报错），读取栈顶的值，并且把这个数据从栈里拿出去。

读取（top）：在栈不为空的前提下（否则报错），读取栈顶的值，但是不把数据拿出去。

下文中，为了方便码字，我们假设堆栈都是睡倒的，就是左边是栈底，右边是栈顶。比如从一个空栈开始：

Push 3

3

 Push 5

3 5

 IsEmpty？ false

3 5

 Get: 5

3 5

 a = Pop； a=3

3

 b=Pop； b=5

 Push a+b

8

 res = pop; res=8

 isEmpty? true

3. 后缀表达式的计算流程，和转为中缀表达式的方法

拿到一个后缀表达式，比如说5 4 3 2 + 3 pow \* +，标准的流程是这样的：

0. 初始化一个栈

1. 表达式从左到右读入一个值

2. 如果这个值是一个数字，把这个数字push入栈；

3. 如果这个值是一个运算符，连续两次pop栈，第一次pop的记为b，第二次pop的记为a，然后把 a 运算 b push入栈。

4. 处理完表达式所有内容后，pop栈，作为运算结果

5. 回到1，直到表达式内所有内容处理完毕

6. 如果栈为空，计算成功，否则报错。

我们来模拟一下计算过程，并且假设我们有个空栈：

读入5， push(5):

5

读入 4， push(4):

5 4

读入3， push(3):

5 4 3

读入2， push(2):

5 4 3 2

读入+， b=pop()=2, a=pop()=3, a+b=5, push(5):

5 4 5

读入3， push(3):

5 4 5 3

读入pow, b=pop()=3, a=pop()=5, a pow b = 125, push(125):

5 4 125

读入\*， b=pop()=125, a=pop()=4, a\*b=500, push(500):

5 500

读入+, b=pop()=500, a=pop()=5, a+b=505, push(505):

505

以上，表达式处理完毕，pop栈并检查是否为空，结果正确，返回505作为计算结果。

以上，我们是假定所有运算有且只有两个数参与运算，事实上还有很多运算符（或者函数）可以有一个或者三个算数。一个算数的常见有：

abs, ln, sin, cos, tan…

三个算数的，我们用到的只有 a?b:c，一般用？作为运算符。

运算结果也可以扩展一下，包括True/False（你也可以看做1/0）, 这样，大小比较 < > ≠等也可以看做运算符。

加入这些扩展后，运算规则改为：每次读入一个运算符，如果这个运算符需要k个算数，那么连续pop k次，然后进行运算。

5-8<0?log(2,8): abs(6)

对应的后缀表达式是： 5 8 - 0 < 2 8 log 6 abs ?

我们来计算一下：

读入5和8， push(5) push(8):

5 8

读入-， pop两次，5-8=-3， push(-3):

-3

读入0， push(0):

-3 0

读入<，pop两次： -3<0 = True， push(True)

True

读入2 8， push两次：

True 2 8

读入log，pop两次，log(2,8)=3， push(3):

True 3

读入6， push(6):

True 3 6

读入abs，pop一次，abs(6)=6, push(6):

True 3 6

读入？，pop三次，True?3:6=3, push(3)

3

表达式处理完毕，pop栈得到结果3，并且栈为空。

以上，就是后缀表达式的计算方法。给定一个后缀表达式，通过以上系统化的方法就可以求解。

如果给你的后缀表达式中，含有未知数，那么只要用一般代数的思路去处理就好了，就是能算则算，不能算就结果保留未知数，然后用括号括起来：

x 32768 - 1.2 \* 32768 +

读入x, 32768，push两次：

x 32768

读入-， pop两次，push("(x-32768)"):

(x-32768)

读入 1.2, push:

(x-32768) 1.2

读入\*， pop两次， push("((x-32768)\*1.2)")

((x-32768)\*1.2)

读入 32768， push:

((x-32768)\*1.2) 32768

读入 +， pop两次，push("(((x-32768)\*1.2)+32768)"):

(((x-32768)\*1.2)+32768)

处理完毕，pop之。去掉冗余括号，可见结果为：

(x-32768)\*1.2+32768

以上就是后缀表达式的代数运算，转为中缀表达式的方法。我们再看一个例子(以下省略部分冗余括号)：

x y + cos x cos y cos \* - x sin y sin \* +

读入 x, y:

x y

读入 +:

(x+y)

读入cos:

cos(x+y)

读入 x

cos(x+y) x

读入cos

cos(x+y) cos(x)

读入y

cos(x+y) cos(x) y

读入cos

cos(x+y) cos(x) cos(y)

读入\*

cos(x+y) cos(x)\*cos(y)

读入-

cos(x+y)-cos(x)\*cos(y)

读入x

cos(x+y)-cos(x)\*cos(y) x

读入sin

cos(x+y)-cos(x)\*cos(y) sin(x)

读入y

cos(x+y)-cos(x)\*cos(y) sin(x) y

读入sin

cos(x+y)-cos(x)\*cos(y) sin(x) sin(y)

读入\*

cos(x+y)-cos(x)\*cos(y) sin(x)\*sin(y)

读入+

cos(x+y)-cos(x)\*cos(y)+sin(x)\*sin(y)

所以最终结果是0（想想看为什么？）

4. 中缀表达式转后缀表达式的方法

知道如何把后缀表达式转为中缀表达式，下面我们说说怎么反过来，中缀转后缀：

1. 选择表达式中最先执行的部分

2. 依次写下所有算数，然后在后面写上运算符

3. 把写下的东西替换掉中缀表达式里的部分

4. 重复1，直到所有运算符都被处理了。

下文中，带下划线的是用后缀表达式替换的部分。比如说5+4\*(3+2) pow 3:

先算3+2，替换为3 2 +：

5+4\* 3 2 + pow 3

再算乘方，3 2 + pow 3替换为 3 2 + 3 pow:

5 + 4\* 3 2 + 3 pow

再算乘法：

5 + 4 3 2 + 3 pow \*

最后算加法：

5 4 3 2 + 3 pow \* +

所以转为后缀表达式就是5 4 3 2 + 3 pow \* +

同理，我们再看5-8<0?log(2,8): abs(6)

先算5 - 8：

5 8 - <0?log(2,8): abs(6)

再算<

5 8 - 0 <? log(2,8): abs(6)

再算log

5 8 - 0 <? 2 8 log : abs(6)

再算abs：

5 8 - 0 <? 2 8 log: 6 abs

最后算?

5 8 - 0 < 2 8 log 6 abs ?

转换完毕。

5. avs/vs中，不同表达式互转的工具

avs和vs都有帮你互转后缀和中缀表达式的滤镜，以avs为例：

Blankclip(240,1280,20)

Subtitle(mt\_infix("x 16 - 235 \* 255 /"))

就是帮你把x 16 - 235 \* 255 /这个后缀表达式转为中缀表达式

Blankclip(240,1280,20)

Subtitle(mt\_polish("255 - x"))

就是帮你把255 - x这个中缀转后缀。

vs中类似的是mvf.postfix2infix，把后缀表达式转为中缀表达式。

6. avs中的表达式计算

avs和vs中，都提供了对图像做表达式运算的功能。主要是两类滤镜，lut和expr。它们的区别是，lut是脚本初始化时候算好，把一对一的mapping记录在内存中，脚本计算的时候不运算，只查表；而Expr则是实时运算。

这类滤镜一般是假设你输入clip，yuv数值叫做x，你来设计一个带x的后缀表达式，它们帮你运算。比如说我们想把一个8bit的YUV视频，y给反一下，亮场变暗场，暗场变亮场，表达式设计是255-x，那么可以这么写：

mt\_lut("255 x -"，u=2,v=2)

你也可以用mt\_polish来写中缀表达式：

mt\_lut(mt\_polish("255 - x"),u=2,v=2)

avs中，expr太慢了没有实用性，我们一般用lut。8bit下，可以有mt\_lut, mt\_lutxy, mt\_lutxyz，后两者可以输入两个或者三个clip作为输入变量。比如说mt\_makediff(a,b)，其实就是mt\_luxy(a,b, "x y - 128 +")

avs中，lut的结果，会被clamp到0和255之间。lut还可以给yuv设计不同的表达式，这点自己去爬doc或者taro的教程。

16bit下，有dither\_lut16，实现16bit->16bit的lut。Dither tools还提供了8bit输入输出，用16bit运算精度的Dither\_lut8, Dither\_lutxy8, Dither\_lutxyz8。因为16bit下，即使是两个输入clip，要做16bit的mapping，其需要的内存高达 65536^2\*16bit = 8GB，显然内存开销太大。而8bit的mapping，就算三个输入，也只需要256^3\*8bit = 16MB，并无太大问题。

7. vs中的表达式计算

VS中提供了std.Lut，只能适用于单clip输入。理由也和avs一样；vs支持各种高精度，如果允许多维，内存吃不消。具体用法可以参见doc。它既可以通过数组来实现，也可以通过自定义函数来实现。比如上文的mt\_lut("255 x -"，u=2,v=2)，用vs可以这么写：

lut = []

for x in range(256):

 lut.append(max(min(256-x, 255), 0))

res = core.std.Lut(src8, planes=0, lut=lut)

注意，vs里的lut是不会帮你自动clamp到上下界，需要你自己确保。虽然以上例子其实不需要，但是我觉得还是有必要写一下如何手动clamp：max(min(K,255),0) 就是把K限制在0和255之间。其他上下界类似。

你也可以自定义函数来做lut:

def reverse(x):

 return max(min(255-x, 255), 0)

res = core.std.Lut(src8, planes=0, function=reverse)

除了lut，vs中的Expr也具备实用性（官方做了很多优化，速度比avs快很多）：

res = core.std.Expr(src8, ["255 x -",""])

Expr可以指定多个输入clip（通过[]来构成一个array），分别用x,y,z,a,b,c…代表。表达式也是以array给出，如果表达式个数少于平面数量，则后面的平面会使用前面的表达式。空表达式("")表示直接copy第一个输入clip的数值。比如说我们想对两个16bit的clip 做MakeDiff，chroma平面则全填0：

lumadiff = core.std.Expr([a16,b16],["x y - 32768 +","0"])

Expr可以使用format指定输出，不指定则跟第一个输入的clip一致。你可以手动指定format，比如说，下面是把一个8bit的tvrange clip，转为16bit：

res = core.std.Expr(src8,"x 256 \*",vs.YUV420P16)

Expr会先把所有输入的数字转为浮点数，所以你其实可以同时把不同bitdepth的输入喂进去，但是你得清楚那么做的后果。比如上文计算lumadiff，如果b16其实是一个YUV420P8的clip，Expr不会报错，实际效果就是类似16bit整数-8bit整数+32768这样，并不能按设计去做差。你可以在Expr中顺道为b做8->16，表达式改为：

lumadiff = core.std.Expr([a16,b8],["x y 256 \* - 32768 +","0"])

Expr运算过程是浮点数，如果输出是整数，结果先是clamp到0~255(或者0~65535)，再四舍五入到整数。自动clamp这点很好，省去了手动计算的必要性；但是如果你需要保证tvrange，你还是得自己手写限制:

res = core.std.Expr(src8,["x 16 max 235 min","x 16 max 240 min"])

这是把一个8bit的clip，通过clamp的方式，保证它符合tvrange范围。