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 $5^3=125$，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。

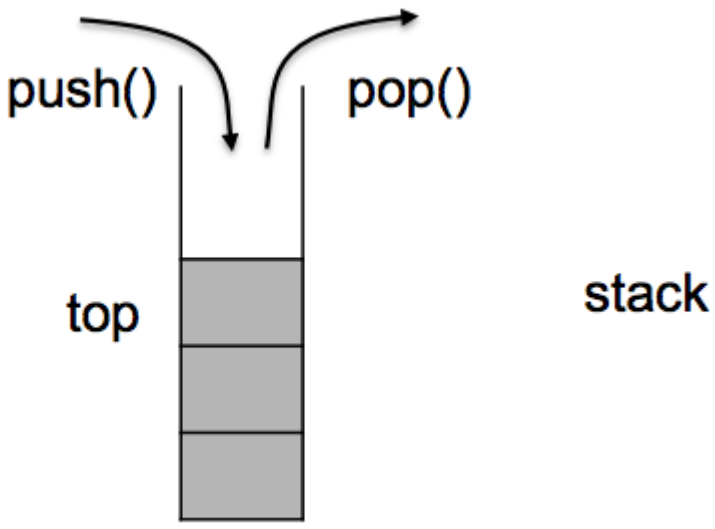
将上述表达式用后缀表达式写出来（假设 pow 代表乘方），效果是：

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。