【讲古堂】表达式求值

（[dubenju@126.com](mailto:dubenju@126.com) 2015/12/27）

在一些处理中，会使用逆波兰式对表达式求值。在这里简要地阐述一下。

什么是表达式

表达式是由数字，操作符，变量，常量等有意义地组合而成并能求得结果的式子。

例如：

32 + ( ( 9 \* Celsius ) / 5 )

4 + 2 \* 55 / 2.5

组成表达式的信息种类繁多，这里只讨论数字表达式，即表达式由以下要素构成：

数字、操作符、变量、常量。

数字

不考虑进制的话，通常指十进制数，小数的时候是有小数点的。数字通常是以被操作对象的身分出现在表达式中的，叫做操作数。

操作符

表示对操作数进行哪种操作的符号叫做操作符，被操作的值叫做操作数，对操作数进行操作的过程称为表达式求值。根据操作对象的个数分为一元操作符和二元操作符。

一元操作符，操作只应用于单一操作数。例如：23！-n。

操作应用于两个操作数的叫二元操作符。例如：6\*21、29/11、6.9+19.8、21.1-2.8

优先级

相对其他操作符，每个操作符都有一个优先级，优先级高的操作符比优先级低的操作符优先应用。一般的优先级是这样设置的：

分组操作符（）具有最高优先级。

一元操作符－＋比乘除模的优先级高。

次方乘除模比加减法的高。

操作数的类型变换

类型变换是指操作的过程中操作数的类型发生变化的现象。

比如：

1/3

1.5/0.3

变量与常量

变量是指值可以变化的量。比如x+5中的x。

常量是指一般不发生变化并代表某一数值的量。比如：pi=3.1415926。

表达式的递归

这里所谓的递归是指一个表达式的结果可以作为另一表达式的操作数。

例如：

1 + 2 \* 3

2 \* 3

1 + 6

表达式的表示方法

中缀表示

1 + x \* 3 + 4 / x

操作符是以中缀形式处于操作数的中间（例：3 + 4）。中缀表达式不容易被计算机解析，但因为它符合人们的普遍用法，被许多程序语言使用。相当于语法树的中序遍历得到的结果。

前缀表示（波兰式）

+ + 1 \* x 3 / 4 x

操作符置于操作数的前面。如果操作符的元数（arity）是固定的，则语法上不需要括号仍然能被无歧义地解析。波兰式是波兰数学家扬·武卡谢维奇1920年代引入的，用于简化命题逻辑。相当于语法树的前序遍历得到的结果。

后缀表示（逆波兰式）

1 x 3 \* + 4 x / +

操作符置于操作数的后面。逆波兰记法不需要括号来标识操作符的优先级。易于程序实现。相当于语法树的后序遍历得到的结果。

表达式的分解parse

把表达式分解成不可再细分的基本元素的过程。

记号（token）的类型

变量variable

数值number

操作符operator

字符0-9和小数点.的话则是数值型。

是预先登记的变量的话则是变量型（常量等同于变量）。

若是操作符的话则是操作符型。

上述以外的话则是不正确的类型。

表达式的分析

语法检查

语法错误就是表达式没有遵循分析程序的严格规则。多数情况下，语法错误都是人为错误，比如输入失误等，以下表达式是非法的10\*\*8(10-5)\*9)/8

表达式求值

基于中缀表示的表达式求值

分别针对操作数和操作符构建表达式，必要时采用递归的方式。由于实现起来比较复杂，多不采用，故不作细述。

基于前缀表示的表达式求值

对于解析后的记号，从后向前进行，如果扫描到操作数，则压进栈，如果扫描到操作符，则根据操作符的操作数个数从栈中弹出相应个数的操作数来进行相应的操作，并将结果压进栈，当扫描结束后，栈的栈顶就是表达式结果。

基于逆波兰表示的表达式求值

针对逆波兰，可以用一个栈来实现计算，扫描从左往右进行，如果扫描到操作数，则压进栈，如果扫描到操作符，则根据操作符的操作数个数从栈中弹出相应个数的操作数来进行相应的操作，并将结果压进栈，当扫描结束后，栈的栈顶就是表达式结果。

调度场算法

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。

所谓的调度场算法是操作数直接输出，操作符的话要和栈内的操作符进行比较，优先级高的出栈输出，然后当前操作符入栈。在最后，把栈内剩余的全部输出。这里的栈起到了调度场的作用。

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>即可。

（完）

波兰式、逆波兰式，中缀式表达式 表达式求值

如何编写一个高效的Java表达式求值程序

一个小型、实用的计算数学表达式的类库（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）。

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。