

## 04-F 中缀表达式求值

#数据结构邓神

把玩

典型应用场合	
逆序输出	<ul style="list-style-type: none"><li>• conversion</li><li>• 输出次序与处理过程颠倒；递归深度和输出长度不易预知</li></ul>
递归嵌套	<ul style="list-style-type: none"><li>• stack permutation + parenthesis</li><li>• 具有自相似性的问题可递归描述，但分支位置和嵌套深度不固定</li></ul>
延迟缓冲	<ul style="list-style-type: none"><li>• evaluation</li><li>• 线性扫描算法模式中，在预读足够长之后，方能确定可处理的前缀</li></ul>
栈式计算	<ul style="list-style-type: none"><li>• RPN</li><li>• 基于栈结构的特定计算模式</li></ul>

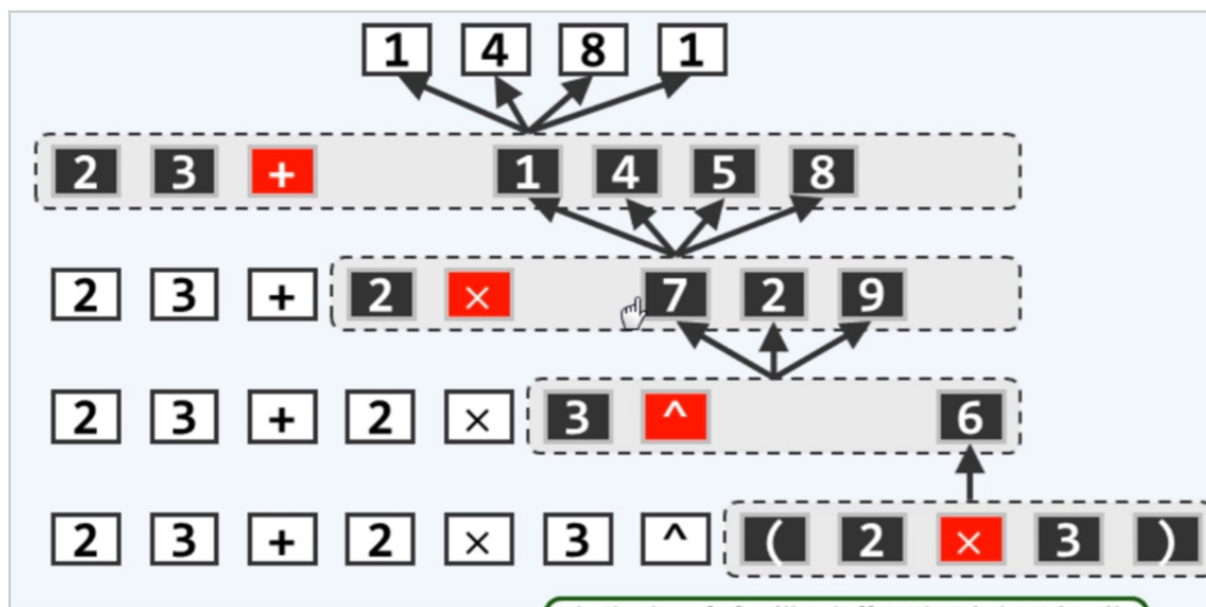
表达式求值：

**表达式求值**

- ❖ 给定语法正确的算术表达式s，计算与之对应的数值
- ❖ \$ echo \$( ( 0 + ( 1 + 23 ) / 4 \* 5 \* 67 - 8 + 9 ) )
- ❖ \> set /a ( !0 ^<< ( 1 - 2 + 3 \* 4 ) ) - 5 \* ( 6 ^ 7 ) / ( 8 ^ 9 )
- ❖ PostScript  
GS> 0 1 23 add 4 div 5 mul 67 mul add 8 sub 9 add =  $\Leftarrow$  RPN
- ❖ Excel: = COS(0) + 1 - ( 2 - POWER( ( FACT(3) - 4 ), 5 ) ) \* 67 - 8 + 9
- ❖ Word: = NOT(0) + 12 + 34 \* 56 + 7 + 89
- ❖ calc: 0 ! + 12 + 34 \* 56 + 7 + 89 =
- ❖ calc: 0 ! + 1 - ( 2 - ( 3 ! - 4 ) ^ 5 ) \* 67 - 8 + 9 =  
y

Data Structures & Algorithms (Fall 2013), Tsinghua University

构思



每次找到一个优先的表达式，然后计算将其表达式转换为一个值

实现减而治之的算法。

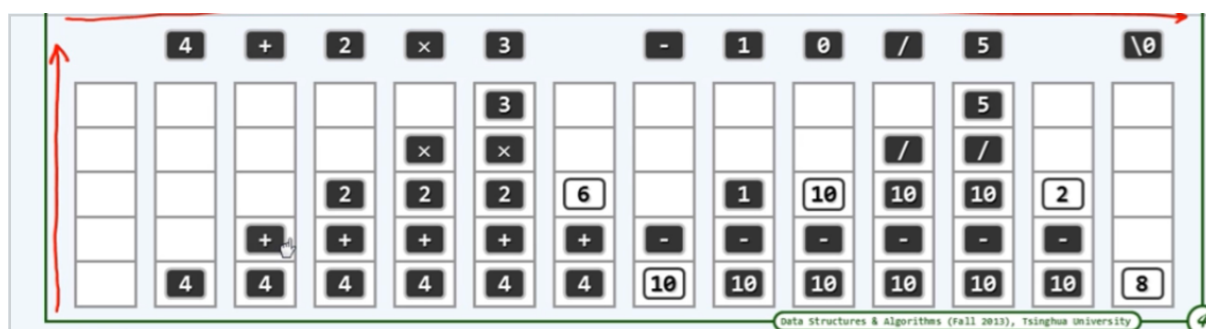
如果用线性扫描的算法，每次扫描到一个运算符都无法确定这个运算符是否可以优先计算，换句话说我们扫描到次序无法与计算次序相对应

## 借助栈结构



我们把所有扫描过的运算符存为一个栈，done表示经过判断可以处理的，而Buffer就是待处理的

## 实例



我们发现如果在一个高优先级的运算符的后面出现一个低（或者同级别运算符，或者结束）优先级的运算符就表示之前的运算符可以计算，

这种算法可以线性完成的表达式计算  
最后在栈内唯一的元素就是我们需要的值。

我们很难自然检测出需要的表达式，比如说  $2*3$  或者  $10/5$   
那么应该如何处理呢，就是把运算符和数值区别对待  
也就是两个栈结构

## 算法框架

### 实现：主算法

```
❖ float evaluate( char* S, char* & RPN ) { //中缀表达式求值
    Stack<float> opnd; Stack<char> optr; //运算数栈、运算符栈
    optr.push('\0'); //尾哨兵'\0'也作为头哨兵首先入栈
    while ( !optr.empty() ) { //逐个处理各字符，直至运算符栈空
        if ( isdigit( *S ) ) //若当前字符为操作数，则
            readNumber( S, opnd ); //读入（可能多位的）操作数
        else //若当前字符为运算符，则视其与栈顶运算符之间优先级的高低
            switch( orderBetween( optr.top(), *S ) ) { /* 分别处理 */ }
    } //while
    return opnd.pop(); //弹出并返回最后的计算结果
}
```

```
float evaluate(char * S,char* & RPN){
    stack<float> opnd;
    stack<char> optr;
    optr.push('\0');
    while(!optr.empty()){
        if (isdigit(*S)){
            readNumer(S,opnd);
        }
        else {
            switch(orderBetween(optr.top(),*S)){
                /* 分别处理 */
            }
        }
    }
    return opnd.top();
}
```

如何判定优先级关系：表格

### 实现：优先级表

```
const char pri[N_OPTR][N_OPTR] = { //运算符优先等级 [栈顶][当前]
//      |----- 当前运算符 -----|
//      +   -   *   /   ^   !   (   )   \0
/* -- + */ '>', '>', '<', '<', '<', '<', '<', '>', '>',
/* | - */ '>', '>', '<', '<', '<', '<', '<', '>', '>',
/* 栈 * */ '>', '>', '>', '>', '<', '<', '<', '>', '>',
/* 顶 / */ '>', '>', '>', '>', '<', '<', '<', '>', '>',
/* 运 ^ */ '>', '>', '>', '>', '>', '<', '<', '>', '>',
/* 算 ! */ '>', '>', '>', '>', '>', '>', ' ', '>', '>',
/* 符 ( */ '<', '<', '<', '<', '<', '<', '<', '=', ' ',
/* | ) */ ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
/* -- \0 */ '<', '<', '<', '<', '<', '<', '<', ' ', '=',
}
```

Data Structures & Algorithms (Fall 2013)

一共四种可能 > < = '' (空字符)

### 实现：不同优先级处理方法

```
❖ switch( orderBetween( optr.top(), *S ) ) {
    case '<': //栈顶运算符优先级更低
        optr.push( *S ); S++; break; //计算推迟，当前运算符进栈
    case '=': //优先级相等 (当前运算符为右括号，或尾部哨兵'\0')
        optr.pop(); S++; break; //脱括号并接收下一个字符
    case '>': { //栈顶运算符优先级更高，实施相应的计算，结果入栈
        char op = optr.pop(); //栈顶运算符出栈，执行对应的运算
        if ( '!' == op ) opnd.push( calcu( op, opnd.pop() ) ); //一元运算符
        else { float p0pnd2 = opnd.pop(), p0pnd1 = opnd.pop(); //二元运算符
                opnd.push( calcu( p0pnd1, op, p0pnd2 ) ); //实施计算，结果入栈
            } //为何不直接：opnd.push( calcu( opnd.pop(), op, opnd.pop() ) )?
        break;
    } //case '>'
}
```

Data Structures & Algorithms (Fall 2013), Tsinghua University

为什么不直接：会有歧义，不确定两个pop的执行次序，会导致某些非常隐蔽的问题

```
switch(orderBetween(optr.top(),*S)){
    case '<' : // 栈顶部的运算符符号比扫描到的运算符符号优先级低，高的先入栈，继续等待
        optr.push(*S);
        S++;
        break;
```

`case '=' :` // 等于只有两种情况（扫描到元素为右括号，或者尾部哨兵），我们可以将开始的 \0 视为一个左括号，后面的 /0，视为一个有右括号，如果两者相遇就代表完全处理完毕

```
optr.pop();
```

```
S++;
```

```
break;
```

`case '>':` // 如果扫描到运算符比扫描到运算符优先级更高，就说明栈顶运算符可以进行运算

```
char op = optr.top();
```

```
if ('!' == op){ // 处理一元运算符
```

```
    opnd.push(calcu(op,opnd.top()));
```

```
    opnd.pop();
```

```
}else { // 处理二元
```

```
    // 取出两个操作数
```

```
    float p0pnd2 = opnd.top();
```

```
    opnd.pop();
```

```
    float p0pnd1 = opnd.top();
```

```
    opnd.pop();
```

```
    // 将操作数结果回到栈中
```

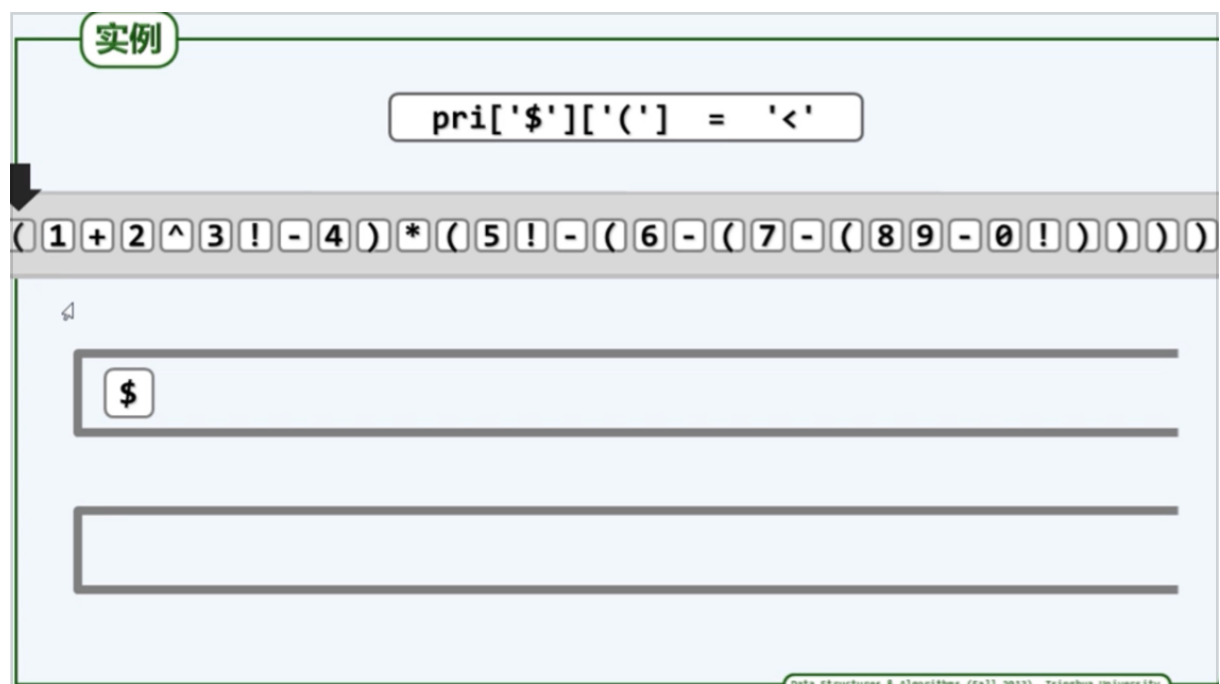
```
    opnd.push(calcu(p0pnd1,op,p0pnd2));
```

```
}
```

```
break;
```

```
}
```

## 实例



是一种线性扫描的过程