它们都是对表达式的记法，因此也被称为前缀记法、中缀记法和后缀记法。它们之间的区别在于运算符相对与操作数的位置不同：前缀表达式的运算符位于与其相关的操作数之前；中缀和后缀同理。

举例：

(3 + 4) × 5 - 6 就是中缀表达式

- × + 3 4 5 6 前缀表达式

3 4 + 5 × 6 - 后缀表达式

**中缀表达式（中缀记法）**

中缀表达式是一种通用的算术或逻辑公式表示方法，操作符以中缀形式处于操作数的中间。中缀表达式是人们常用的算术表示方法。

虽然人的大脑很容易理解与分析中缀表达式，但对计算机来说中缀表达式却是很复杂的，因此计算表达式的值时，通常需要先将中缀表达式转换为前缀或后缀表达式，然后再进行求值。对计算机来说，计算前缀或后缀表达式的值非常简单。

**前缀表达式（前缀记法、波兰式）**

前缀表达式的运算符位于操作数之前。

**前缀表达式的计算机求值：**

从右至左扫描表达式，遇到数字时，将数字压入堆栈，遇到运算符时，弹出栈顶的两个数，用运算符对它们做相应的计算（栈顶元素 op 次顶元素），并将结果入栈；重复上述过程直到表达式最左端，最后运算得出的值即为表达式的结果。

例如前缀表达式“- × + 3 4 5 6”：

(1) 从右至左扫描，将6、5、4、3压入堆栈；

(2) 遇到+运算符，因此弹出3和4（3为栈顶元素，4为次顶元素，注意与后缀表达式做比较），计算出3+4的值，得7，再将7入栈；

(3) 接下来是×运算符，因此弹出7和5，计算出7×5=35，将35入栈；

(4) 最后是-运算符，计算出35-6的值，即29，由此得出最终结果。

可以看出，用计算机计算前缀表达式的值是很容易的。

**将中缀表达式转换为前缀表达式：**

遵循以下步骤：

(1) 初始化两个栈：运算符栈S1和储存中间结果的栈S2；

(2) 从右至左扫描中缀表达式；

(3) 遇到操作数时，将其压入S2；

(4) 遇到运算符时，比较其与S1栈顶运算符的优先级：

(4-1) 如果S1为空，或栈顶运算符为右括号“)”，则直接将此运算符入栈；

(4-2) 否则，若优先级比栈顶运算符的较高或相等，也将运算符压入S1；

(4-3) 否则，将S1栈顶的运算符弹出并压入到S2中，再次转到(4-1)与S1中新的栈顶运算符相比较；

(5) 遇到括号时：

(5-1) 如果是右括号“)”，则直接压入S1；

(5-2) 如果是左括号“(”，则依次弹出S1栈顶的运算符，并压入S2，直到遇到右括号为止，此时将这一对括号丢弃；

(6) 重复步骤(2)至(5)，直到表达式的最左边；

(7) 将S1中剩余的运算符依次弹出并压入S2；

(8) 依次弹出S2中的元素并输出，结果即为中缀表达式对应的前缀表达式。

例如，将中缀表达式“1+((2+3)×4)-5”转换为前缀表达式的过程如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 扫描到的元素 | S2(栈底->栈顶) | S1 (栈底->栈顶) | 说明 |
| 5 | 5 | 空 | 数字，直接入栈 |
| - | 5 | - | S1为空，运算符直接入栈 |
| ) | 5 | - ) | 右括号直接入栈 |
| 4 | 5 4 | - ) | 数字直接入栈 |
| × | 5 4 | - ) × | S1栈顶是右括号，直接入栈 |
| ) | 5 4 | - ) × ) | 右括号直接入栈 |
| 3 | 5 4 3 | - ) × ) | 数字 |
| + | 5 4 3 | - ) × ) + | S1栈顶是右括号，直接入栈 |
| 2 | 5 4 3 2 | - ) × ) + | 数字 |
| ( | 5 4 3 2 + | - ) × | 左括号，弹出运算符直至遇到右括号 |
| ( | 5 4 3 2 + × | - | 同上 |
| + | 5 4 3 2 + × | - + | 优先级与-相同，入栈 |
| 1 | 5 4 3 2 + × 1 | - + | 数字 |
| 到达最左端 | 5 4 3 2 + × 1 + - | 空 | S1中剩余的运算符 |

因此结果为“- + 1 × + 2 3 4 5”。

**后缀表达式（后缀记法、逆波兰式）**

后缀表达式与前缀表达式类似，只是运算符位于操作数之后。

**后缀表达式的计算机求值：**

与前缀表达式类似，只是顺序是从左至右：

从左至右扫描表达式，遇到数字时，将数字压入堆栈，遇到运算符时，弹出栈顶的两个数，用运算符对它们做相应的计算（次顶元素 op 栈顶元素），并将结果入栈；重复上述过程直到表达式最右端，最后运算得出的值即为表达式的结果。

例如后缀表达式“3 4 + 5 × 6 -”：

(1) 从左至右扫描，将3和4压入堆栈；

(2) 遇到+运算符，因此弹出4和3（4为栈顶元素，3为次顶元素，注意与前缀表达式做比较），计算出3+4的值，得7，再将7入栈；

(3) 将5入栈；

(4) 接下来是×运算符，因此弹出5和7，计算出7×5=35，将35入栈；

(5) 将6入栈；

(6) 最后是-运算符，计算出35-6的值，即29，由此得出最终结果。

**将中缀表达式转换为后缀表达式：**

与转换为前缀表达式相似，遵循以下步骤：

(1) 初始化两个栈：运算符栈S1和储存中间结果的栈S2；

(2) 从左至右扫描中缀表达式；

(3) 遇到操作数时，将其压入S2；

(4) 遇到运算符时，比较其与S1栈顶运算符的优先级：

(4-1) 如果S1为空，或栈顶运算符为左括号“(”，则直接将此运算符入栈；

(4-2) 否则，若优先级比栈顶运算符的高，也将运算符压入S1（注意转换为前缀表达式时是优先级较高或相同，而这里则不包括相同的情况）；

(4-3) 否则，将S1栈顶的运算符弹出并压入到S2中，再次转到(4-1)与S1中新的栈顶运算符相比较；

(5) 遇到括号时：

(5-1) 如果是左括号“(”，则直接压入S1；

(5-2) 如果是右括号“)”，则依次弹出S1栈顶的运算符，并压入S2，直到遇到左括号为止，此时将这一对括号丢弃；

(6) 重复步骤(2)至(5)，直到表达式的最右边；

(7) 将S1中剩余的运算符依次弹出并压入S2；

(8) 依次弹出S2中的元素并输出，结果的逆序即为中缀表达式对应的后缀表达式（转换为前缀表达式时不用逆序）。

例如，将中缀表达式“1+((2+3)×4)-5”转换为后缀表达式的过程如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 扫描到的元素 | S2(栈底->栈顶) | S1 (栈底->栈顶) | 说明 |
| 1 | 1 | 空 | 数字，直接入栈 |
| + | 1 | + | S1为空，运算符直接入栈 |
| ( | 1 | + ( | 左括号，直接入栈 |
| ( | 1 | + ( ( | 同上 |
| 2 | 1 2 | + ( ( | 数字 |
| + | 1 2 | + ( ( + | S1栈顶为左括号，运算符直接入栈 |
| 3 | 1 2 3 | + ( ( + | 数字 |
| ) | 1 2 3 + | + ( | 右括号，弹出运算符直至遇到左括号 |
| × | 1 2 3 + | + ( × | S1栈顶为左括号，运算符直接入栈 |
| 4 | 1 2 3 + 4 | + ( × | 数字 |
| ) | 1 2 3 + 4 × | + | 右括号，弹出运算符直至遇到左括号 |
| - | 1 2 3 + 4 × + | - | -与+优先级相同，因此弹出+，再压入- |
| 5 | 1 2 3 + 4 × + 5 | - | 数字 |
| 到达最右端 | 1 2 3 + 4 × + 5 - | 空 | S1中剩余的运算符 |

因此结果为“1 2 3 + 4 × + 5 -”（注意需要逆序输出）。

编写Java程序将一个中缀表达式转换为前缀表达式和后缀表达式，并计算表达式的值。其中的toPolishNotation()方法将中缀表达式转换为前缀表达式（波兰式）、toReversePolishNotation()方法则用于将中缀表达式转换为后缀表达式（逆波兰式）：

注：

(1) 程序很长且注释比较少，但如果将上面的理论内容弄懂之后再将程序编译并运行起来，还是比较容易理解的。有耐心的话可以研究一下。(2) 此程序是笔者为了说明上述概念而编写，仅做了简单的测试，不保证其中没有Bug，因此不要将其用于除研究之外的其他场合。

**[java]** [view plain](http://blog.csdn.net/antineutrino/article/details/6763722/#) [copy](http://blog.csdn.net/antineutrino/article/details/6763722/#)

**package** qmk.simple\_test;

**import** java.util.Scanner;

**import** java.util.Stack;

/\*\*

 \* Example of converting an infix-expression to

 \* Polish Notation (PN) or Reverse Polish Notation (RPN).

 \* Written in 2011-8-25

 \* @author QiaoMingkui

 \*/

**public** **class** Calculator {

      **public** **static** **final** String USAGE = "== usage ==\n"

            + "input the expressions, and then the program "

            + "will calculate them and show the result.\n"

            + "input 'bye' to exit.\n";

      /\*\*

       \* @param args

       \*/

      **public** **static** **void** main(String[] args) {

            System.out.println(USAGE);

            Scanner scanner = **new** Scanner(System.in);

            String input = "";

            **final** String CLOSE\_MARK = "bye";

            System.out.println("input an expression:");

            input = scanner.nextLine();

            **while** (input.length() != 0

                  && !CLOSE\_MARK.equals((input))) {

                  System.out.print("Polish Notation (PN):");

                  **try** {

                        toPolishNotation(input);

                  } **catch** (NumberFormatException e) {

                        System.out.println("\ninput error, not a number.");

                  } **catch** (IllegalArgumentException e) {

                        System.out.println("\ninput error:" + e.getMessage());

                  } **catch** (Exception e) {

                        System.out.println("\ninput error, invalid expression.");

                  }

                  System.out.print("Reverse Polish Notation (RPN):");

                  **try** {

                        toReversePolishNotation(input);

                  } **catch** (NumberFormatException e) {

                        System.out.println("\ninput error, not a number.");

                  } **catch** (IllegalArgumentException e) {

                        System.out.println("\ninput error:" + e.getMessage());

                  } **catch** (Exception e) {

                        System.out.println("\ninput error, invalid expression.");

                  }

                  System.out.println("input a new expression:");

                  input = scanner.nextLine();

            }

            System.out.println("program exits");

      }

      /\*\*

       \* parse the expression , and calculate it.

       \* @param input

       \* @throws IllegalArgumentException

       \* @throws NumberFormatException

       \*/

      **private** **static** **void** toPolishNotation(String input)

                  **throws** IllegalArgumentException, NumberFormatException {

            **int** len = input.length();

            **char** c, tempChar;

            Stack<Character> s1 = **new** Stack<Character>();

            Stack<Double> s2 = **new** Stack<Double>();

            Stack<Object> expression = **new** Stack<Object>();

            **double** number;

            **int** lastIndex = -1;

            **for** (**int** i=len-1; i>=0; --i) {

                  c = input.charAt(i);

                  **if** (Character.isDigit(c)) {

                        lastIndex = readDoubleReverse(input, i);

                        number = Double.parseDouble(input.substring(lastIndex, i+1));

                        s2.push(number);

                        i = lastIndex;

                        **if** ((**int**) number == number)

                              expression.push((**int**) number);

                        **else**

                              expression.push(number);

                  } **else** **if** (isOperator(c)) {

                        **while** (!s1.isEmpty()

                                    && s1.peek() != ')'

                                    && priorityCompare(c, s1.peek()) < 0) {

                              expression.push(s1.peek());

                              s2.push(calc(s2.pop(), s2.pop(), s1.pop()));

                        }

                        s1.push(c);

                  } **else** **if** (c == ')') {

                        s1.push(c);

                  } **else** **if** (c == '(') {

                        **while** ((tempChar=s1.pop()) != ')') {

                              expression.push(tempChar);

                              s2.push(calc(s2.pop(), s2.pop(), tempChar));

                              **if** (s1.isEmpty()) {

                                    **throw** **new** IllegalArgumentException(

                                          "bracket dosen't match, missing right bracket ')'.");

                              }

                        }

                  } **else** **if** (c == ' ') {

                        // ignore

                  } **else** {

                        **throw** **new** IllegalArgumentException(

                                    "wrong character '" + c + "'");

                  }

            }

            **while** (!s1.isEmpty()) {

                  tempChar = s1.pop();

                  expression.push(tempChar);

                  s2.push(calc(s2.pop(), s2.pop(), tempChar));

            }

            **while** (!expression.isEmpty()) {

                  System.out.print(expression.pop() + " ");

            }

            **double** result = s2.pop();

            **if** (!s2.isEmpty())

                  **throw** **new** IllegalArgumentException("input is a wrong expression.");

            System.out.println();

            **if** ((**int**) result == result)

                  System.out.println("the result is " + (**int**) result);

            **else**

                  System.out.println("the result is " + result);

      }

      /\*\*

       \* parse the expression, and calculate it.

       \* @param input

       \* @throws IllegalArgumentException

       \* @throws NumberFormatException

       \*/

      **private** **static** **void** toReversePolishNotation(String input)

                  **throws** IllegalArgumentException, NumberFormatException {

            **int** len = input.length();

            **char** c, tempChar;

            Stack<Character> s1 = **new** Stack<Character>();

            Stack<Double> s2 = **new** Stack<Double>();

            **double** number;

            **int** lastIndex = -1;

            **for** (**int** i=0; i<len; ++i) {

                  c = input.charAt(i);

                  **if** (Character.isDigit(c) || c == '.') {

                        lastIndex = readDouble(input, i);

                        number = Double.parseDouble(input.substring(i, lastIndex));

                        s2.push(number);

                        i = lastIndex - 1;

                        **if** ((**int**) number == number)

                              System.out.print((**int**) number + " ");

                        **else**

                              System.out.print(number + " ");

                  } **else** **if** (isOperator(c)) {

                        **while** (!s1.isEmpty()

                                    && s1.peek() != '('

                                    && priorityCompare(c, s1.peek()) <= 0) {

                              System.out.print(s1.peek() + " ");

                              **double** num1 = s2.pop();

                              **double** num2 = s2.pop();

                              s2.push(calc(num2, num1, s1.pop()));

                        }

                        s1.push(c);

                  } **else** **if** (c == '(') {

                        s1.push(c);

                  } **else** **if** (c == ')') {

                        **while** ((tempChar=s1.pop()) != '(') {

                              System.out.print(tempChar + " ");

                              **double** num1 = s2.pop();

                              **double** num2 = s2.pop();

                              s2.push(calc(num2, num1, tempChar));

                              **if** (s1.isEmpty()) {

                                    **throw** **new** IllegalArgumentException(

                                          "bracket dosen't match, missing left bracket '('.");

                              }

                        }

                  } **else** **if** (c == ' ') {

                        // ignore

                  } **else** {

                        **throw** **new** IllegalArgumentException(

                                    "wrong character '" + c + "'");

                  }

            }

            **while** (!s1.isEmpty()) {

                  tempChar = s1.pop();

                  System.out.print(tempChar + " ");

                  **double** num1 = s2.pop();

                  **double** num2 = s2.pop();

                  s2.push(calc(num2, num1, tempChar));

            }

            **double** result = s2.pop();

            **if** (!s2.isEmpty())

                  **throw** **new** IllegalArgumentException("input is a wrong expression.");

            System.out.println();

            **if** ((**int**) result == result)

                  System.out.println("the result is " + (**int**) result);

            **else**

                  System.out.println("the result is " + result);

      }

      /\*\*

       \* calculate the two number with the operation.

       \* @param num1

       \* @param num2

       \* @param op

       \* @return

       \* @throws IllegalArgumentException

       \*/

      **private** **static** **double** calc(**double** num1, **double** num2, **char** op)

                  **throws** IllegalArgumentException {

            **switch** (op) {

            **case** '+':

                  **return** num1 + num2;

            **case** '-':

                  **return** num1 - num2;

            **case** '\*':

                  **return** num1 \* num2;

            **case** '/':

                  **if** (num2 == 0) **throw** **new** IllegalArgumentException("divisor can't be 0.");

                  **return** num1 / num2;

            **default**:

                  **return** 0; // will never catch up here

            }

      }

      /\*\*

       \* compare the two operations' priority.

       \* @param c

       \* @param peek

       \* @return

       \*/

      **private** **static** **int** priorityCompare(**char** op1, **char** op2) {

            **switch** (op1) {

            **case** '+': **case** '-':

                  **return** (op2 == '\*' || op2 == '/' ? -1 : 0);

            **case** '\*': **case** '/':

                  **return** (op2 == '+' || op2 == '-' ? 1 : 0);

            }

            **return** 1;

      }

      /\*\*

       \* read the next number (reverse)

       \* @param input

       \* @param start

       \* @return

       \* @throws IllegalArgumentException

       \*/

      **private** **static** **int** readDoubleReverse(String input, **int** start)

                  **throws** IllegalArgumentException {

            **int** dotIndex = -1;

            **char** c;

            **for** (**int** i=start; i>=0; --i) {

                  c = input.charAt(i);

                  **if** (c == '.') {

                        **if** (dotIndex != -1)

                              **throw** **new** IllegalArgumentException(

                                    "there have more than 1 dots in the number.");

                        **else**

                              dotIndex = i;

                  } **else** **if** (!Character.isDigit(c)) {

                        **return** i + 1;

                  } **else** **if** (i == 0) {

                        **return** 0;

                  }

            }

            **throw** **new** IllegalArgumentException("not a number.");

      }

      /\*\*

       \* read the next number

       \* @param input

       \* @param start

       \* @return

       \* @throws IllegalArgumentException

       \*/

      **private** **static** **int** readDouble(String input, **int** start)

      **throws** IllegalArgumentException {

            **int** len = input.length();

            **int** dotIndex = -1;

            **char** c;

            **for** (**int** i=start; i<len; ++i) {

                  c = input.charAt(i);

                  **if** (c == '.') {

                        **if** (dotIndex != -1)

                              **throw** **new** IllegalArgumentException(

                              "there have more than 1 dots in the number.");

                        **else** **if** (i == len - 1)

                              **throw** **new** IllegalArgumentException(

                              "not a number, dot can't be the last part of a number.");

                        **else**

                              dotIndex = i;

                  } **else** **if** (!Character.isDigit(c)) {

                        **if** (dotIndex == -1 || i - dotIndex > 1)

                              **return** i;

                        **else**

                              **throw** **new** IllegalArgumentException(

                              "not a number, dot can't be the last part of a number.");

                  } **else** **if** (i == len - 1) {

                        **return** len;

                  }

            }

            **throw** **new** IllegalArgumentException("not a number.");

      }

      /\*\*

       \* return true if the character is an operator.

       \* @param c

       \* @return

       \*/

      **private** **static** **boolean** isOperator(**char** c) {

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

      }

}

下面是程序运行结果（绿色为用户输入）：

== usage ==

input the expressions, and then the program will calculate them and show the result.

input 'bye' to exit.

input an expression:

3.8+5.3

Polish Notation (PN):+ 3.8 5.3

the result is 9.1

Reverse Polish Notation (RPN):3.8 5.3 +

the result is 9.1

input a new expression:

5\*(9.1+3.2)/(1-5+4.88)

Polish Notation (PN):/ \* 5 + 9.1 3.2 + - 1 5 4.88

the result is 69.88636363636364

Reverse Polish Notation (RPN):5 9.1 3.2 + \* 1 5 - 4.88 + /

the result is 69.88636363636364

input a new expression:

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

Polish Notation (PN):- + 1 \* + 2 3 4 5

the result is 16

Reverse Polish Notation (RPN):1 2 3 + 4 \* + 5 -

the result is 16

input a new expression:

bye

program exits