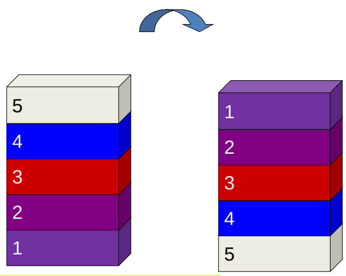
2.2 Stacks



特殊的线性表----栈

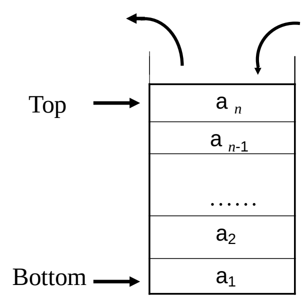
Main contents

- Definition and operations
- Implementation
- Applications

2.2.1 Definition

A stack is a list with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called top.

栈：限定仅在表尾进行插入和删除操作的线性表。
空栈：不含任何数据元素的栈。
栈顶和栈底：
允许插入（入栈、进栈、压栈）和删除（出栈、弹栈）的一端称为栈顶，另一端称为栈底。

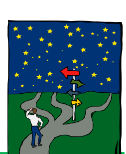


Stacks are sometimes known as last in, first out (LIFO).

Operations

The fundamental operations on a stack are push, which is equivalent to an insert, and pop, which deletes the most recently inserted element.

- 置空栈：Inistack
- 判断栈是否为空：Empty
- 入栈：Push
- 出栈：Pop
- 得到栈顶元素：GetTop



ADT Sqlist
Data
D={a_i,a_i∈ElementType,i=1,2,…,n,n≥0}
R={<a_{i-1},a_i>|a_{i-1},a_i∈D,i=2,…,n}
Operation

初始化，构造一个空的栈
InitList
Post-condition：None.

判栈空

EmptyStack(b)

Output： b: Boolean.

Post-condition： Return True if stack is empty, else return False.

入栈

Push(e)

Input： e: ElementType.

Post-condition： Add element e to top of stack.

出栈

Pop(e)

Output： Top element e of stack.

Pre-condition： Stack is not empty.

Post-condition： Remove top element from stack.

读栈顶元素

Top(e)

Output： Top element e of stack .

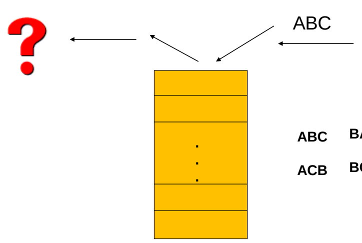
Pre-condition： Stack is not empty .

Post-condition： Return top element from stack.

End ADT

A case

已知输入序列经过栈后到达输出序列，问得到的输出序列有哪些？



ABC BAC CBA

ACB BCA ~~CAB~~

Stack

Input : ABCD

ABCD ABDC ACBD ACDB ADCB

BACD BADC BCAD BCDA BDCA

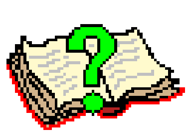
CBAD CBDA CDBA

DCBA

Number: $\frac{1}{n+1}C_{2n}^{n}$

设输入序列是123456AB，经过栈后，可以得到的输出序列有哪些

可以作为程序设计中的变量名？

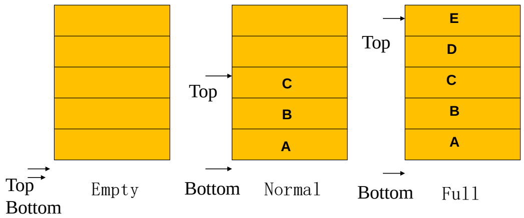


2.2.2 Implementation

1. **Array implementation**

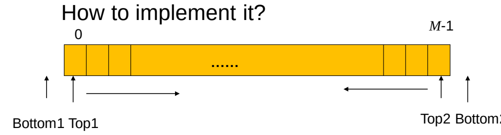
```
Const int maxstack =10;
Class Stack {
 Public:
   Stack();
   bool empty() const;
   Error_code pop();
   Error_code top(Stack_entry &item) const;
   Error_code push(const Stack_entry &item);

  private:
   int count;
   Stackentry entry[maxstack];
}
```



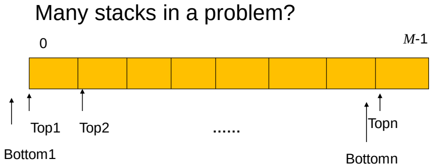
Two stacks in a problem.

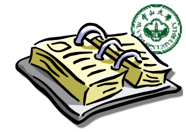
How to implement it?



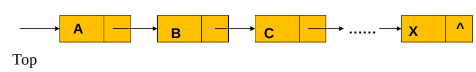
Full of stack: **Top1+1=Top2**

Many stacks in a problem?



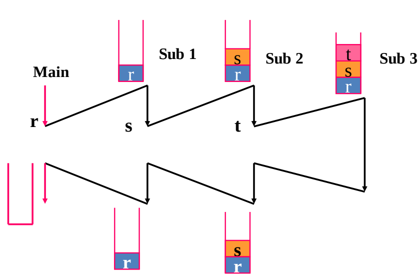


2. Linked stack



2.2.3 Applications

- Function call



1. Conversion of number system

Decimalism：

Octonary $28_{10}=3*8+4=34_8$

Quaternary $72_{10}=1*64+0*16+2*4+0=1024_4$

Binary $53_{10}=1*32+1*16+0*8+1*4+0*2+1$

$=110101_2$

159_{10}->($)_8$

8	159	余 7
8	19	余 3
8	2	余 2
	0	

2	3	7

$(159)_{10}=(237)_8$

Method:

Push: n%B，将结果入栈；

n=n/B

Until n=0

Output all element in Stack

2. Balancing symbols

Compilers check your programs for syntax errors.

Example:

2+ (3* (5+9) -44/ (6-3))

2+ (3* (5+9-44/ (6-3))

3. Expressions

表达式求值

表达式：
- 前缀表达式（波兰式）
- 中缀表达式
- 后缀表达式（逆波兰式）

例如：

(a+b)*(a－b)
- *+ab－ab
- (a+b)*(a－b)
- ab+ab－*

中缀表达式求值

4+2*3-10/5

51*(24-15/3)+6

四则运算规则：

先乘除，后加减，先括号内，后括号外。
同一运算级别，从左到右。

Symbol Priority

Compare θ_1 and θ_2

θ_1 \ θ_2	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

1) Infix expressions

- 1：操作数栈置空，操作符栈压入算符“#”
- 2：依次读入表达式的每个单词
- 3：如果是操作数，压入操作数栈
- 4：如果是操作符，将操作符栈顶元素 θ_1与读入的操作符 θ_2进行优先级比较
  - 4.1如果栈顶元素优先级低，将 θ_2压入操作符栈
  - 4.2如果相等，弹操作符栈
  - 4.3如果栈顶元素优先级高，弹出两个操作数，一个运算符，进行计算，并将计算结果压入操作数栈，重复第4步的判断
- 5:直至整个表达式处理完毕

A case

- 3*(7-2)#

步骤	操作符栈	操作数栈	输入字符	操作
1	#		3*(7-2)#	压入“3”
2	#	3	*(7-2)#	压入“*”
3	#*	3	(7-2)#	压入“(”
4	#*(	3	7-2)#	压入“7”
5	#*(	37	-2)#	压入“-”
6	#*(-	37	2)#	压入“2”
7	#*(-	372	)#	弹出“-”压入7-2
8	#*(	35	)#	弹出“(”
9	#*	35	#	计算3*5
10	#	15	#	操作符栈空，结束

```
{ SOPTR.Push("#");
  cin >>c;
  while (c!'#' ||SOPTR.top()!='#')
  {    if (!In(c,OP)) { SOPND.Push(c);
                        cin >>c; }
       else
         switch (Precede(SOPTR.top(),c))
         {   case '<':   SOPTR.Push(,c);
                         cin >>c;
                         break;
             case '=':   SOPTR.Pop(x);
                         cin >>c;
                         Break;
             case '>':   SOPTR.Pop(theta);
                         SOPND.Pop(b);
                         SOPND.Pop(a);
                         SOPND.Push(Operate(a,theta,b));
                         break;
         }
  }
  return SOPND.top();   }
```

2) Postfix expressions

4+3*5	4, 3, 5*+
2*(5+9*4/2)+6*5	2, 5, 9, 4*2/+*6, 5*+

求解算法：

设定一个操作数栈OPND；从左向右依次读入，当读到的是运算数，将其加入到运算数栈中；若读入的是运算符，从运算数栈取出两个元素，与读入的运算符进行运算，将运算结果加入到运算数栈。直到表达式的最后一个运算符处理完毕。

3) Infix to postfix conversion

Infix	Postfix
4+3*5	4, 3, 5*+
2*(5+9*4/2)+6*5	2, 5, 9, 4*2/+*6, 5*+

中缀表达式中，运算符的出现次序与计算顺序不一致；

后缀表达式中，运算符的出现次序就是计算次序。

A+B*C-D#　　　ABC*+D-

读到的符号	运算符栈	输出序列
A	#	A
+	#+	A
B	#+	AB
*	#+*	AB
C	#+*	ABC
-	#-	ABC*+
D	#-	ABC*+D
#	#	ABC*+D-

Conversion algorithm

- **1：**操作符栈压入算符“#”
- **2：**依次读入表达式的每个单词
- **3：**如果是操作数，则输出
- **4：**如果是操作符，将操作符栈顶元素 θ_1与读入的操作符 θ_2进行优先级比较
  - 4.1如果栈顶元素优先级低，将 θ_2压入操作符栈
  - 4.2如果相等，弹操作符栈
  - 4.3如果栈顶元素优先级高，弹出栈定元素并输出，重复第4步的判断
- 5:直至整个表达式处理完毕

2.2.4 Recursion

递归是算法设计中一种重要的方法，可以使许多程序结构简化，易理解，易证明。

递归定义的算法有两个部分：

递归基：直接定义最简单情况下的函数值；

递归步：通过较为简单情况下的函数值定义一般情况下的函数值。

应用条件与准则

适宜于用递归算法求解的条件是：

（1）问题具有某种可借用的类同自身的子问题描述的性质；

（2）某一有限步的子问题（也称作本原问题）有直接的解存在。

递归算法设计

递归算法既是一种有效的算法设计方法，也是一种有效的分析问题的方法。递归算法求解问题的基本思想是：对于一个较为复杂的问题，把原问题分解成若干个相对简单且类同的子问题，这样，原问题就可递推得到解。

适宜于用递归算法求解的问题的充分必要条件是：

（1）问题具有某种可借用的类同自身的子问题描述的性质；

（2）某一有限步的子问题（也称作本原问题）有直接的解存在。

当一个问题存在上述两个基本要素时，该问题的递归算法的设计方法是：

（1）把对原问题的求解设计成包含有对子问题求解的形式。

（2）设计递归出口。

递归算法举例

1. 阶乘函数

$$n! = n*(n-1)*(n-2)*......*1$$

$$n! = n*(n-1)!$$

$$f(n) = \begin{cases} 1 & n = 0 \\ n*f(n-1) & n > 0 \end{cases}$$

递归基

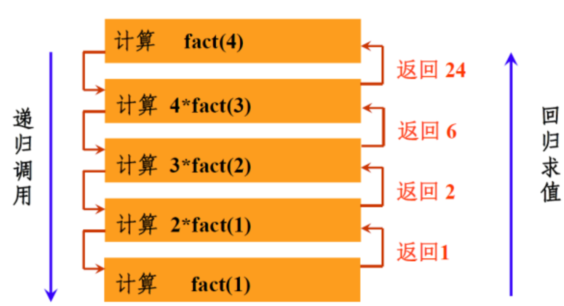
递归步

算法描述

```
Long factorial (long n)
 {
    int temp;
    if (n==0) return 1;
     else
       {
          temp=n*factorial(n-1);

          return temp;
       }
 }
```

求解过程



2.Fibonacci数列

无穷数列1，1，2，3，5，8，13，21，34，55，……，称为Fibonacci数列。

$$F(n)=\begin{cases}1 & n=0\\ 1 & n=1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

递归基

递归步

第n个Fibonacci数可递归地计算如下：

```
int fibonacci(int n)
  {
      if (n <= 1) return 1;
      return fibonacci(n-1)+fibonacci(n-2);
  }
```

3. Ackerman函数

当一个函数及它的一个变量是由函数自身定义时，称这个函数是**双递归函数**。

Ackerman函数$A(n, m)$定义如下：

$$\begin{cases}A(1,0)=2 & \\ A(0,m)=1 & m\ge 0\\ A(n,0)=n+2 & n\ge 2\\ A(n,m)=A(A(n-1,m),m-1) & n,m\ge 1\end{cases}$$

4. 多项式求值问题

有如下多项式：

$$P_n(x)=a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0$$

如果分别对每一项求值，需要$n(n+1)/2$个乘法，效率很低。

$$P_n(x)=a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0$$
$$=((\cdots(((a_n)x+a_{n-1})x+a_{n-2})x\cdots)x+a_1)x+a_0$$

递归表示：

1）递归基：$n=0, P_0=a_n$

2）递归步：对任意的k, $1\le k\le n$，如果前面 k-1步已经计算出P_{k-1}：

$$P_{k-1}=a_nx^{k-1}+a_{n-1}x^{k-2}+\cdots+a_{n-k+2}x+a_{n-k+1}$$

则有： $P_k=xP_{k-1}+a_{n-k}$

递归算法：

```
Float horner_pol(float x, float A[ ], int n)
{
   float p;
   if (n==0)
     p=A[0];
   else
     p=horner_pol(x, A,n-1)*x+A[n];
   return p;
}
```

5. 排列问题

有n个元素，编号为1,2,…,n，用一个具有n个元素的数组A来存放所生成的排列，然后输出它们。

操作步骤：

（1）数组第一个元素为1，即排列的第一个元素为1，生成后面的n-1个元素的排列。

（2）数组第一个元素与第二个元素互换，使得排列的第一个元素为2，生成后面的n-1个元素的排列。

（3）如此继续，最后数组第一个元素与第n个元素互换，使得排列的第一个元素为n，生成后面的n-1个元素的排列。

在上面的第一步中，为了生成后面n-1个元素的排列，继续采用如下步骤：

（1）数组第二个元素为2，即排列的第二个元素为2，生成后面的n-2个元素的排列。

（2）数组第二个元素与第三个元素互换，使得排列的第二个元素为3，生成后面的n-2个元素的排列。

（3）如此继续，最后数组第二个元素与第n个元素互换，使得排列的第二个元素为n，生成后面的n-2个元素的排列。

这种步骤一直继续，当排列的前n-2个元素已经确定后，为生成后面2个元素的排列，可以采用如下步骤：

（1）数组第n-1个元素为n-1，即排列的第n-1个元素为n-1，生成后面的1个元素的排列，此时数组中的n个元素已经构成一个排列。

（2）数组第n-1个元素与第n个元素互换，使得排列的第n-1个元素为n，生成后面的n-2个元素的排列。

通过上述分析，设排列算法perm(A,k,n)表示生成后面k个元素的排列，有如下递归表示：

（1）递归基：k=1，只有一个元素，已构成一个排列。

（2）递归步：对任意的k,1≤k≤n，如果可由算法perm(A,k-1,n)完成数组后面k-1个元素的排列，为完成数组后面k个元素的排列Perm(A,k,n)，逐一对数组第n-k元素与数组中第n-k~n元素进行互换，每互换一次，就执行一次perm(A,k-1,n)操作，产生一个排列。

6. 整数划分问题

将正整数n表示成一系列正整数之和：$n=n_1+n_2+\ldots+n_k$，

其中$n_1 \geq n_2 \geq \ldots \geq n_k \geq 1$，$k \geq 1$。

正整数n的这种表示称为正整数n的划分。求正整数n的不同划分个数。

例如正整数6有如下11种不同的划分：

6；

5+1；

4+2，4+1+1；

3+3，3+2+1，3+1+1+1；

2+2+2，2+2+1+1，2+1+1+1+1；

1+1+1+1+1+1。

前面的几个例子中，问题本身都具有比较明显的递归关系，因而容易用递归函数直接求解。

在本例中，如果设$p(n)$为正整数n的划分数，则难以找到递归关系，因此考虑增加一个自变量：将最大加数n_1不大于m的划分个数记作$q(n,m)$。可以建立$q(n,m)$的如下递归关系。

(1) $q(n,1)=1, n\geq 1$;
当最大加数n_1不大于1时，任何正整数n只有一种划分形式，即 $n=\overbrace{1+1+\cdots+1}^{n}$

(2) $q(n,m)=q(n,n), m\geq n$;
最大加数n_1实际上不能大于n。因此，$q(1,m)=1$。

(3) $q(n,n)=1+q(n,n-1)$;
正整数n的划分由$n_1=n$的划分和$n_1\leq n-1$的划分组成。

(4) $q(n,m)=q(n,m-1)+q(n-m,m), n>m>1$;
正整数n的最大加数n_1不大于m的划分由$n_1=m$的划分和$n_1\leq n-1$ 的划分组成。

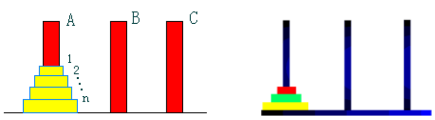
$$q(n,m)=\begin{cases} 1 & n=1, m=1 \\ q(n,n) & n<m \\ 1+q(n,n-1) & n=m \\ q(n,m-1)+q(n-m,m) & n>m>1 \end{cases}$$

正整数n的划分数$p(n)=q(n,n)$。

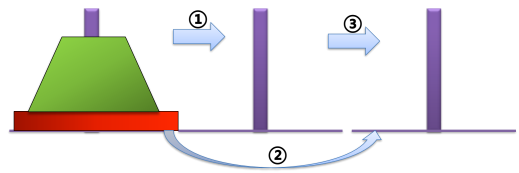
7. Hanoi塔问题

设A,B,C是3个塔座。开始时，在塔座A上有一叠共*n*个圆盘，这些圆盘自下而上，由大到小地叠在一起。各圆盘从小到大编号为1,2,…,*n*,现要求将塔座A上的这一叠圆盘移到塔座B上，并仍按同样顺序叠置。在移动圆盘时应遵守以下移动规则：

- 规则1：每次只能移动1个圆盘；
- 规则2：任何时刻都不允许将较大的圆盘压在较小的圆盘之上；
- 规则3：在满足移动规则1和2的前提下，可将圆盘移至A,B,C中任一塔座上。



- 解法
  - 为了把*n*>*1*个盘子从木桩*1*移到木桩*3*，需要把*n-1*个盘子递归的移到木桩*2*（借助*3*）；
  - 然后把最大的盘子（第*n*个）移到木桩*3*；
  - 再把*n-1*个盘子由木桩2移动到木桩*3*（借助*1*）。



如何实现移动圆盘的操作呢？

（1）**递归中止条件**：当*n*=1时，问题比较简单，只要将编号为1的圆盘从塔座*A*直接移动到塔座*C*上即可；

（2）**问题分解**：当*n*>1时，需利用塔座*B*作辅助塔座，若能设法将压在编号为*n*的圆盘上的*n*-1个圆盘从塔座*A*(依照上述原则)移至塔座*B*上，则可先将编号为*n*的圆盘从塔座*A* 移至塔座*C*上，然后再将塔座*B*上的*n*-1个圆盘(依照上述原则)移至塔座*C*上。而如何将*n*-1个圆盘从一个塔座移至另一个塔座问题是一个和原问题具有相同特征属性的问题，只是问题的规模小个1，因此可以用同样方法求解。 由此可得如下算法所示的求解*n*阶Hanoi塔问题的函数。

```
void hanoi(int n,char x,char y,char z) /*将塔座X上按直径由小到大且至上而
    下编号为1至n的n个圆盘按规则搬到塔座Z上， Y可用作辅助塔座 */
{
  if(n==1)
     move(x,1,z); /* 将编号为1的圆盘从X移动Z */
  else {
    hanoi(n-1,x,z,y); /* 将X上编号为1至n-1的圆盘移到Y,Z作辅助塔 */
    move(x,n,z); /* 将编号为n的圆盘从X移到Z */
     hanoi(n-1,y,x,z); /* 将Y上编号为1至n-1的圆盘移动到Z， X作辅助塔 */
    }
}
```

下面给出三个盘子搬动时hanoi(3, A, B, C) 递归调用过程

```
hanoi(n, x, y, z)
{
  if(n==1)
     move(x,1,z);
  else {
     hanoi(n-1,x,z,y);
     move(x,n,z);
     hanoi(n-1,y,x,z);
  }
}
```

```
hanoi(2,A,C,B):
        hanoi(1,A,B,C)  move(A->C)    1号搬到C
        move(A->B)                    2号搬到B
        hanoi(1,C,A,B)  move(C->B)    1号搬到B
move(A->C)                            3号搬到C
hanoi(2,B,A,C):
        hanoi(1,B,C,A) move(B->A)     1号搬到A
        move(B->c)  2号搬到C
        hanoi(1,A,B,C) move(A->C)     1号搬到C
```

2.2.5 Stack and recursion

对于非递归函数，调用函数在调用被调用函数前，系统要保存以下两类信息：

(1) 调用函数的返回地址；

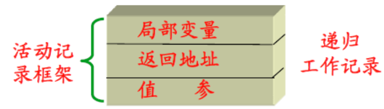
(2) 调用函数的局部变量值。

当执行完被调用函数，返回调用函数前，系统首先要恢复调用函数的局部变量值，然后返回调用函数的返回地址。

递归函数被调用时，系统要作的工作和非递归函数被调用时系统要作的工作在**形式上类同**，但**保存信息的方法不同**。

递归函数被调用时，系统需要一个运行时栈。系统的运行时栈也要保存上述两类信息。每一层递归调用所需保存的信息构成**运行时栈的一个工作记录**，在每进入下一层递归调用时，系统就建立一个新的**工作记录**，并把这个工作记录进栈成为运行时栈新的栈顶；每返回一层递归调用，就退栈一个工作记录。因为栈顶的工作记录必定是当前正在运行的递归函数的工作记录，所以栈顶的工作记录也称为**活动记录**。

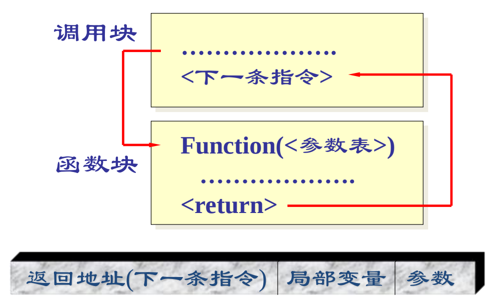
- 每一次递归调用时，需要为过程中使用的参数、局部变量和返回地址等另外分配存储空间。
- 每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织。



- 递归函数的内部执行过程
  - (1) 运行开始时，首先为递归调用建立一个工作栈，其结构包括值参、局部变量和返回地址；
  - (2) 每次执行递归调用之前，把递归函数的值参和局部变量的当前值以及调用后的返回地址压栈；
  - (3) 每次递归调用结束后，将栈顶元素出栈，使相应的值参和局部变量恢复为调用前的值，然后转向返回地址指定的位置继续执行。

工作记录的含义？意义？

函数递归时的活动记录



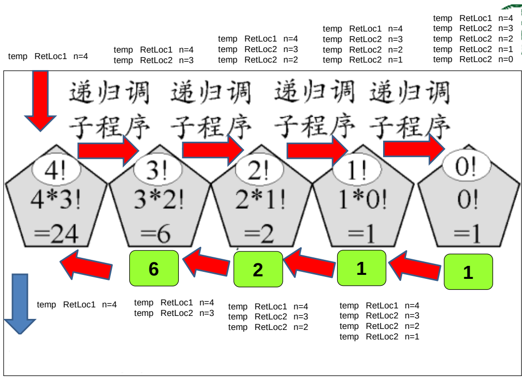

```
long Factorial ( long n ) {
    int temp;
    if ( n == 0 ) return 1;
    else temp = n * Factorial (n-1);
RetLoc2
    return temp;
}

void main ( ) {
   int n;
   n = Factorial (4);
RetLoc1
}
```

计算Fact时活动记录的内容

递归调用序列

参数	返回地址	返回时的指令
4	RetLoc1	RetLoc1 return 4*6 //返回24
3	RetLoc2	RetLoc2 return 3*2 //返回6
2	RetLoc2	RetLoc2 return 2*1 //返回2
1	RetLoc2	RetLoc2 return 1*1 //返回1
0	RetLoc2	RetLoc2 return 1 //返回1



规范化的递归转换成非递归的方法

栈的定义：

调用参数区

$p_1, p_2, \ldots, p_m$

$q_1, q_2, \ldots, q_n$

局部参数区

$m_1, m_2, \ldots, m_s$

断点地址*RT*

```
Long factorial(long n)
 {
   int temp;
   if (n==0)
       return 1;
   else
      {  temp=factorial(n-1)*n;

         return temp;
      }
 }
```

断点地址的描述

按语句标号来记录，含有*t*个调用递归过程本身的语句，则设定*t*+2个语句标号，**L0**设在第一个可执行的语句上，**Li(*i*=1..*t*)**设在*t*个递归调用的语句的返回处，**Lj(j=*t*+1)**设在过程体结束的语句上。

```
Long factorial(long n)
 {
   int temp;
  L0: if (n==0)
       return 1;
   else
      {L1: temp=factorial(n-1)*n;

         return temp;
      }
  L2:
 }
```

断点地址的标注

按语句标号来记录，含有*t*个调用递归过程本身的语句，则设定*t*+2个语句标号，**L0**设在第一个可执行的语句上，**Li(*i*=1..*t*)**设在*t*个递归调用的语句的返回处，**Lj(j=*t*+1)**设在过程体结束的语句上。

```
void hanoi(int n, char X, char Y, char Z)
{
 L0: if (n <= 1)
     move(X,Z);
 else
 {
// 最大的盘在X上不动，把X上的n－1个盘移到Y
 hanoi(n-1,X,Z,Y);
 L1: move(X,Z); //移动最大盘到Z，放好
 hanoi(n-1,Y,X,Z);//把 Y上的n－1个盘移到Z
   L2: }
L3:
}
```

规则一

在标号L0前增加语句：

S.Push(L(t+1), $p_1, p_2, \ldots, p_m, q_1, q_2, \ldots, q_n, m_1, m_2, \ldots, m_s$)

此后，所用的变量均由栈定元素指示。

```
void hanoi(int n, char X, char Y, char Z)
{
 L0: if (n <= 1)
     move(X,Z);
 else
 {
// 最大的盘在X上不动，把X上的n－1个环移到Y
 hanoi(n-1,X,Z,Y);
 L1: move(X,Z); //移动最大盘到Z，放好
 hanoi(n-1,Y,X,Z);//把 Y上的n－1个盘移到Z
 L2:
}
L3: }
```

S.Push(L3,n,x,y,z)

If (S.top(n)<=1)
Move(S.top(X),S.top(Y))

规则二

将调用递归的语句Pi(a1,a2,…a(m+n)) 改为：

S.Push(Li, a1,a2,…a(m+n)) ;
goto L0;
Li: (v1,v2,…,v(m+n))=S.Pop();

```
void hanoi(int n, char X, char Y, char Z)
{
 L0: if (n <= 1)
     move(X,Z);
 else
 {
// 最大的盘在X上不动，把X上的n－1个盘移到Y
 hanoi(n-1,X,Z,Y);
 L1: move(X,Z); //移动最大盘到Z，放好
 hanoi(n-1,Y,X,Z);//把 Y上的n－1个盘移到Z
   L2: }
L3:
}
```

S.Push(L1,n-1,x,z,y);
Goto L0;
L1: (v1,v2,v3,v4)=S.Pop();

规则三

在所有的递归出口处，增加语句 goto RT=S.Top(RT)

在最后一个标号处，是最终的函数值出栈语句：
(v1,v2,…,v(m+n))=S.Pop();

规则三

```
void hanoi(int n, char X, char Y, char Z)
{
 L0:  if (n <= 1)
     move(X,Z);
 else
 {
// 最大的盘在X上不动，把X上的n－1个盘移到Y
 hanoi(n-1,X,Z,Y);
 L1:  move(X,Z); //移动最大盘到Z，放好
 hanoi(n-1,Y,X,Z);//把 Y上的n－1个盘移到Z
     L2:  }
L3:
}
```

goto RT=S.Top(RT)

goto RT=S.Top(RT)

(v1,v2,...,v(m+n))=S.Pop();