数据结构与算法

第3章 栈与队列

主讲：赵海燕
北京大学信息科学技术学院
“数据结构与算法”教学组

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6, “十一五”国家级规划教材

栈的应用

- 栈的特点：后进先出
  - 体现了元素间的透明性
- 常用来处理具有递归结构的数据
  - 深度优先搜索
  - 数制转换
  - 表达式求值
  - 行编辑处理
  - 子程序／函数调用的管理
  - 消除递归

数制转换

- 非负十进制数转换成其它进制的一种方法：
  - 例，十进制转换成八进制：$(66)_{10}=(102)_8$
    - 66/8 = 8 余 2
    - 8/8 = 1 余 0
    - 1/8 = 0 余 1

  结果为余数的逆序：102
- 特点： 先求得的余数写出结果时最后写出，最后求出的余数最先写出，符合栈的先入后出性质，故可用栈来实现数制转换
  - 同理，一个非负十进制整数 N 转换为另一个等价的B进制数，可以采用“除B取余法”来解决

表达式计算

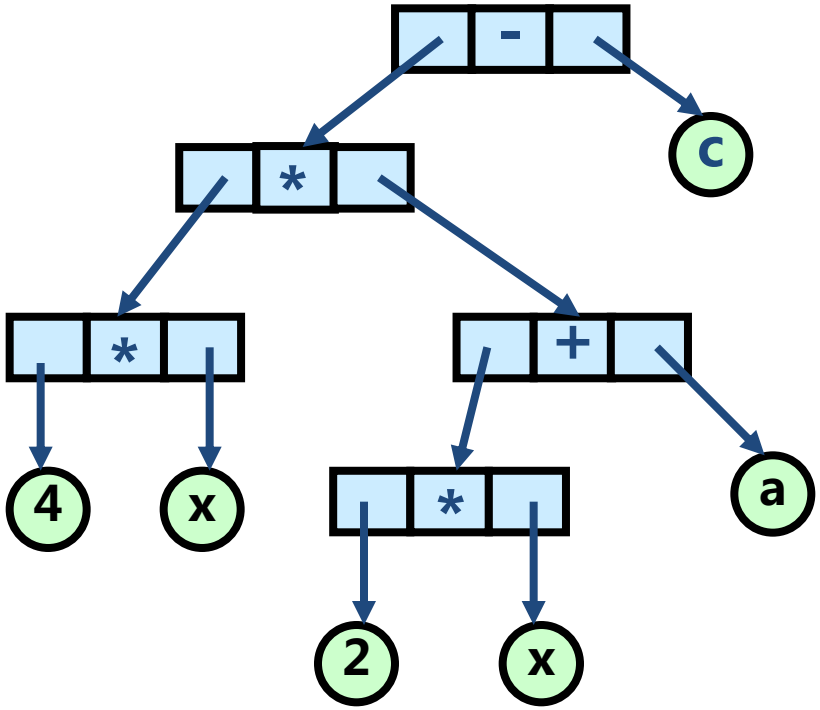
- 表达式的递归定义
  - 基本符号集：{0，1，…，9，+，-，*，/，（，）}
  - 语法成分集：{<表达式>，<项>，<因子>，<常数>，<数字>}
  - 语法公式集
- 中缀表达式

  23+(34*45)/(5+6+7)
- 后缀表达式

  23 34 45 * 5 6 + 7 + / +

中缀表达式

4 * x * (2 * x + a) – c

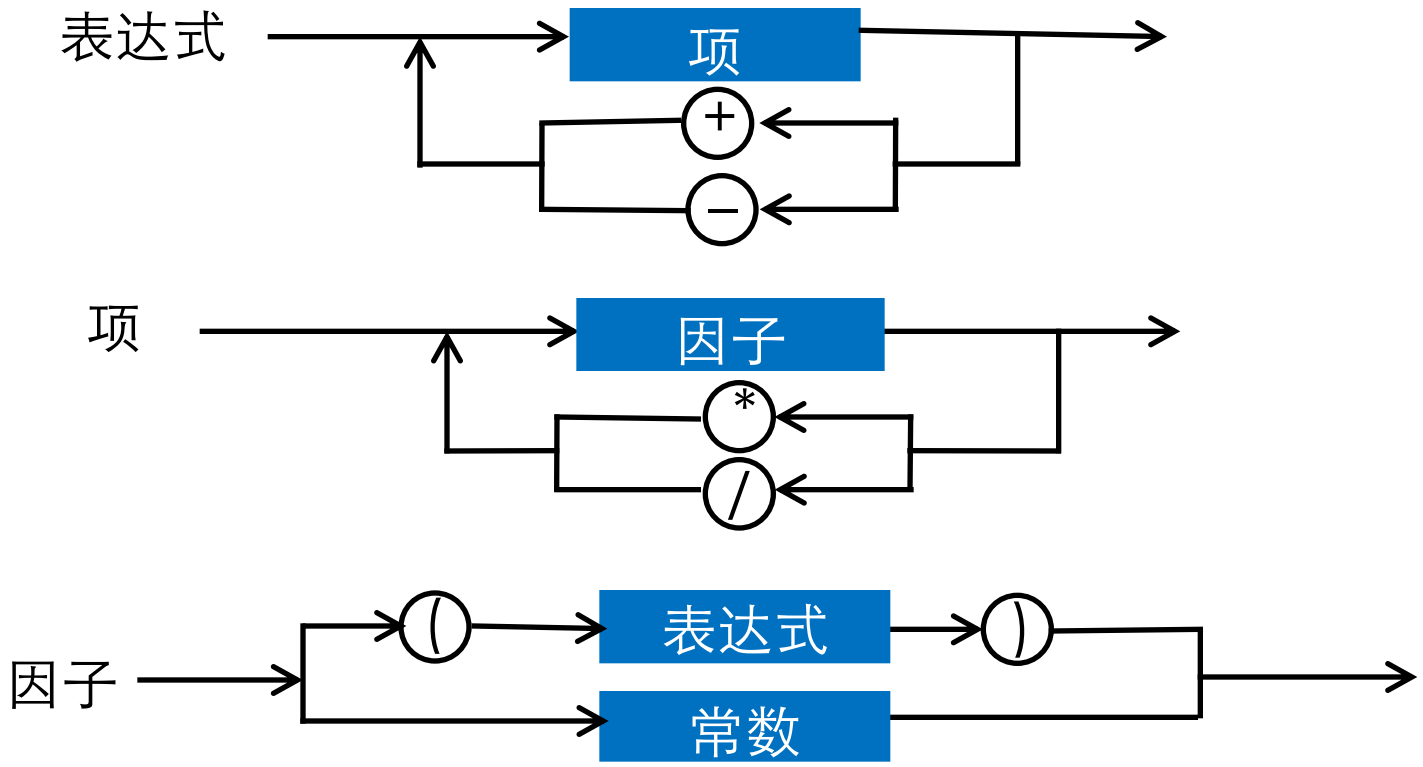


- 特点
  - 运算符在中间
  - 需要括号改变优先级

中缀表达法的语法公式

<表达式> ::= <项> + <项>
| <项> – <项>
| <项>
<项> ::= <因子> * <因子>
| <因子> / <因子>
| <因子>
<因子> ::= <常数>
| (<表达式>)
<常数> ::= <数字>
| <数字> <常数>
<数字> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

中缀表达式的递归图示



中缀表达式的计算

1. 若有括号，先执行**括号内**的计算，后执行括号外的计算。具有多层括号时，按层次反复地脱括号，左右括号必须配对
2. 无括号或同层括号时，先乘(*) 、除(／)，后加(+)、减(-)
3. 同一个层次，若有多个乘除（*、／）或 加减 (+，-) 的运算，按**自左至右次序**执行

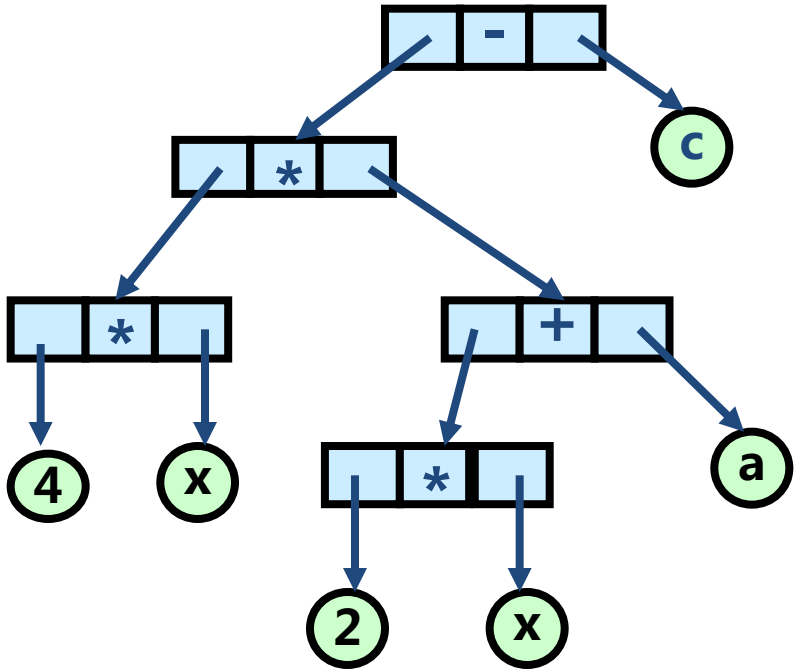
23+(34*45)/(5+6+7) ＝？

计算特点?

后缀表达式

4 x * 2 x * a + * c –

- 特点
  - 运算符在后面
  - 完全不需要括号



后缀表达式的语法公式

```
<表达式> ::= <项> <项> +
       | <项> <项> -
       | <项>
<项>   ::= <因子> <因子> *
       | <因子> <因子> /
       | <因子>
<因子>  ::= <常数>
<常数>  ::= <数字>
       | <数字> <常数>
<数字>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

后缀表达式的计算

23 34 45 * 5 6 + 7 + / + =?

计算特点?

中缀和后缀表达式的主要异同？

23 + (34*45) / (5+6+7) = ?

23 34 45 * 5 6 + 7 + / + = ?

后缀表达式求值

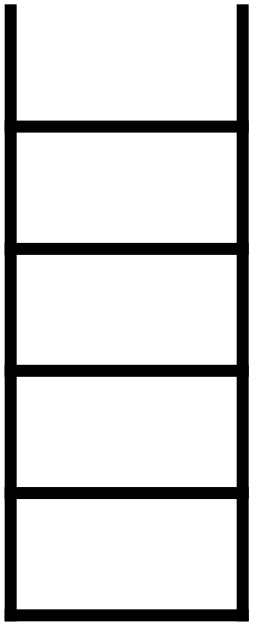
1. 循环：依次顺序读入表达式的符号序列（假设以＝作为输入序列的结束），并根据读入的元素符号逐一分析：
   - 操作数，则将其压入栈顶；
   - 运算符，则从栈中两次取出栈顶，按照运算符对这两个操作数进行计算。将计算结果压入栈顶
2. 如此继续，直到遇到符号＝，此时栈顶的值即为输入表达式的值

后缀表达式求值

待处理的后缀表达式

23 34 45 * 5 6 + 7 + / +

栈状态的变化



计算 结果

后缀计算器的类定义

```
class Calculator {
private:
    Stack<double> s;        // 这个栈用于压入保存操作数
    // 从栈顶弹出两个操作数opd1和opd2
    bool GetTwoOperands(double& opd1,double& opd2);
    // 取两个操作数，并按op对两个操作数进行计算
    void Compute(char op);
public:
    Calculator(void){} ;   // 创建计算器实例，开辟一个空栈
    void Run(void);         // 读入后缀表达式，遇“=”符号结束
    void Clear(void);       // 清除计算器，为下一次计算做准备
};
```

后缀计算器的类定义

```
template <class ELEM>
bool Calculator<ELEM>::GetTwoOperands(ELEM& opnd1, ELEM& opnd2)
{
    if (S.IsEmpty()) {
        cerr << "Missing operand!" <<endl;
        return false;
    }
    opnd1 = S.Pop();          // 右操作数
    if (S.IsEmpty()) {
        cerr << "Missing operand!" <<endl;
        return false;
    }
    opnd2 = S.Pop();          // 左操作数
    return true;
}
```

后缀计算器的类定义

```
template <class ELEM> void Calculator<ELEM>::Compute(char op) {
    bool result;  ELEM operand1, operand2;
    result = GetTwoOperands(operand1, operand2);
    if (result == true)
        switch(op) {
            case '+' : S.Push(operand2 + operand1); break;
            case '-' : S.Push(operand2 - operand1); break;
            case '*' : S.Push(operand2 * operand1); break;
            case '/' : if (operand1 == 0.0) {
                       cerr << "Divide by 0!" << endl;
                       S.ClearStack();
                   } else S.Push(operand2 / operand1);
                   break;
        }
    else S.ClearStack();
}
```

后缀计算器的类定义

```
template <class ELEM> void Calculator<ELEM>::Run(void) {
    char c; ELEM newoperand;
    while (cin >> c, c != '=') {
        switch(c) {
            case '+': case '-': case '*': case '/':
                Compute(c);
                break;
            default:
                cin.putback(c); cin >> newoperand;
                Enter(newoperand);
                break;
        }
    }
    if (!S.IsEmpty())
        cout << S.Pop() << endl;          // 印出求值的最后结果
}
```

中缀表达式 vs 后缀表达式

中缀和后缀表达式的主要异同?

23+(34*45)/(5+6+7) = ?

23 34 45 * 5 6 + 7 + / + = ?

用 同 化 异 ?

中缀表达式 to 后缀表达式

- **从左到右扫描中缀表达式。用栈存放表达式中的操作符、开括号以及在此开括号后暂不确定计算次序的其他符号：**

(1) 当输入为**操作数**时，直接输出到后缀表达式序列；

(2) 当遇到**开括号**时，将其入栈；

(3) 当输入遇到**闭括号**时，先判断栈是否为空，若为空（括号不匹配），应该作为错误异常处理，清栈退出。若非空，则把栈中元素依次弹出，直到遇到第一个开括号为止，将弹出的元素输出到后缀表达式序列中（弹出的开括号不放到序列中），若没有遇到开括号，说明括号不配对，做异常处理，清栈退出；

中缀表达式 to 后缀表达式

(4) 当输入为**运算符**op（四则运算 + - * / 之一）时

(a) 循环 当（栈非空 and 栈顶不是开括号 and 栈顶运算符的优先级不低于输入的运算符的优先级）时，反复

将栈顶元素弹出，放到后缀表达式序列中；

(b) 将输入的运算符压入栈内；

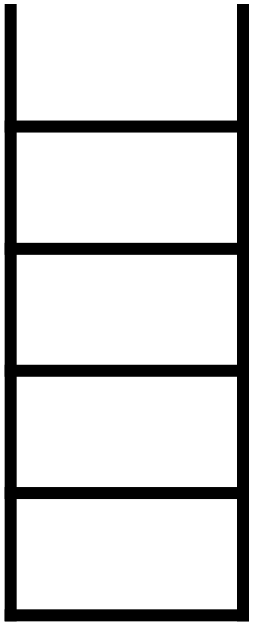
(5) 当中缀表达式的符号全部读入时，若栈内仍有元素，将其全部依次弹出，置于后缀表达式序列的尾部。若弹出的元素遇到开括号，则说明括号不匹配，做异常处理，清栈退出

中缀表达式 to 后缀表达式

待处理中缀表达式

23 + (34 * 45) / (5 + 6 + 7)

栈状态的变化



输出的后缀表达式

中缀表达式求值

- 中缀表达式 → 后缀表达式
- 直接计算　how to？

栈的应用

- 栈的特点：后进先出
  - 体现了元素间的透明性
- 常用于处理具有递归结构的数据
  - 深度优先搜索
  - 数制转换
  - 表达式求值
  - 行编辑处理
  - 子程序／函数调用的管理
  - 消除递归

栈与递归

- 函数的递归定义
- 主程序和子程序的参数传递
- 栈在实现函数递归调用中的作用

递归

- **作为数学和计算机科学的基本概念，递归是解决复杂问题的一个有力手段，可使问题的描述和求解变得简洁、清晰、易懂**
  - 符合人类解决问题的思维方式，能够描述和解决很多问题，许多程序设计语言提供支持机制
  - 常常比非递归算法更易设计，尤当问题本身或所涉及的数据结构就是递归定义时，
- **计算机内（编译程序）是如何将程序设计中的便于人类理解的递归程序转换为计算机可处理的非递归程序的?**
  - 计算机则只能按照指令集顺序执行

递归

- **直接或间接调用自己的函数或过程**
  1. **递归步骤：将规模较大的原问题分解为一个或多个规模更小、但具有类似于原问题特性的子问题**
     - 即，较大的问题递归地用较小的子问题来描述，解原问题的方法同样可用来解这些子问题
  2. **递归出口：确定一个或多个无须分解、可直接求解的最小子问题（称为递归的终止条件）**
- **相关概念**
  - **迭代**
  - **归纳**

递归示例： 阶乘函数

- 阶乘 n! 的递归定义如下：

$$\text{factiorial}(n) = \begin{cases} 1, \text{if } n \le 1 \\ n \times \text{factiorial}(n-1), \text{if } n > 1 \end{cases}$$

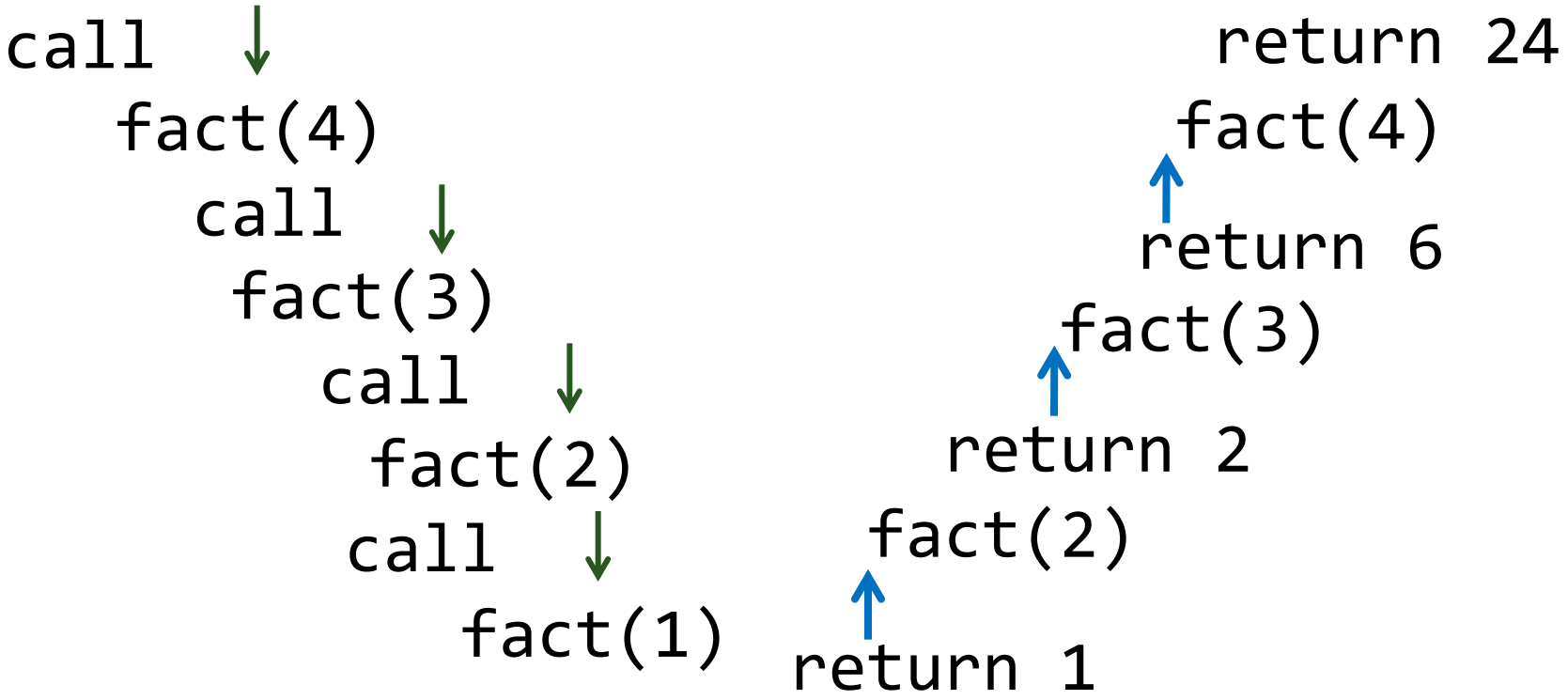
- 递归定义由两部分组成
  - 递归基础（递归出口）
  - 递归规则
- 正确性证明?

递归示例： 阶乘函数的程序实现

```
// 递归定义的计算阶乘n!的函数
long fact(long n) {
     if (n <= 1)
         return 1;
     else
         return n * fact (n-1) ;    // 递归调用
}
```

if (n == 1)?

函数执行过程图解：阶乘


call
fact(4)
call
fact(3)
call
fact(2)
call
fact(1)
return 1
fact(2)
return 2
fact(3)
return 6
fact(4)
return 24

递归示例：一个数学递归公式

$$f(n) = \begin{cases} n+1, \text{if } n < 2 \\ f(\lfloor n/2 \rfloor) \times f(\lfloor n/4 \rfloor), \text{if } n \geq 2 \end{cases}$$

相应的递归函数实现

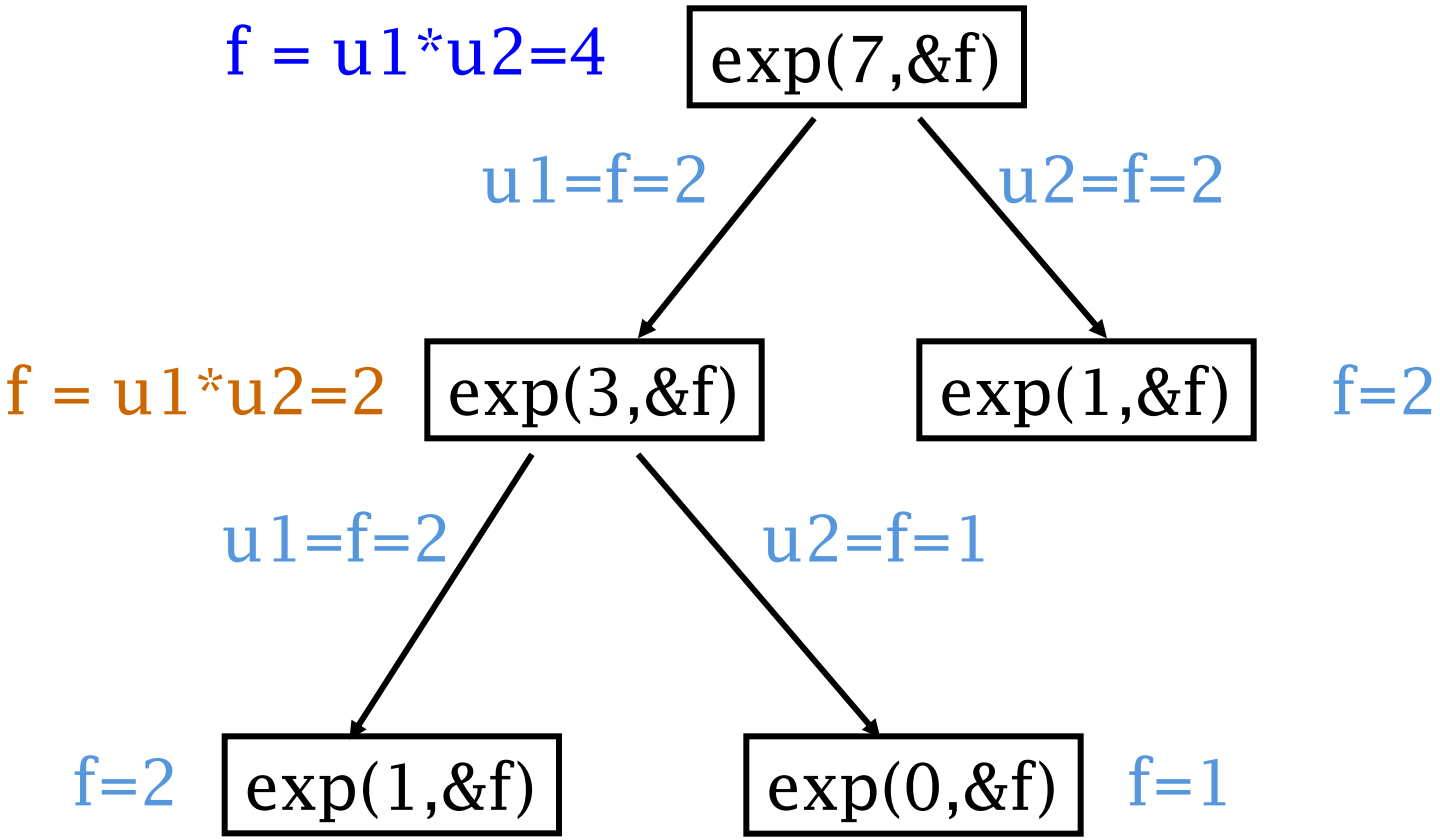
```
int f(int n) {
   if (n<2)
        return n+1;
   else
        return f(n/2) * f(n/4);
}
```

递归示例： 简单转换

```
int f(int n) {
    if (n<2)
        return n+1;
    else
        return f(n/2) * f(n/4);
}
```

```
void exp(int n, int& f) {
    int u1, u2;
    if (n<2)
        f = n+1;
    else {
        exp((int)(n/2), u1);
        exp((int)(n/4), u2);
        f = u1*u2;
    }
}
```

函数执行过程图解

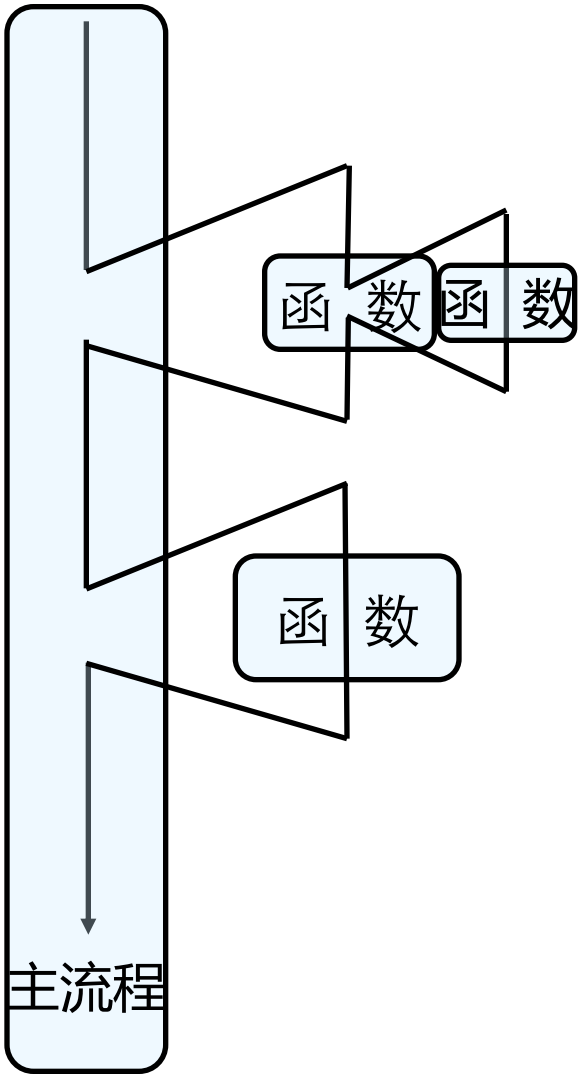

f = u1*u2=4
exp(7,&f)
u1=f=2
u2=f=2
f = u1*u2=2
exp(3,&f)
exp(1,&f)
f=2
u1=f=2
u2=f=1
f=2
exp(1,&f)
exp(0,&f)
f=1

函数调用与递归的实现

- **函数调用**
  - 静态分配：数据区的分配可在程序运行前进行，直到整个程序运行结束才释放。函数的调用和返回处理比较简单，不需每次分配和释放被调函数的数据区
  - 动态分配：递归调用时，被调函数的局部变量等不能提前静态分配到固定单元，而须每调用一次就分配一份，以存放当前所使用的数据，当返回时随即释放。故其存储分配只能在执行调用时才能进行
- **程序运行时环境**
  - 目标计算机上用来管理存储器并保存执行过程所需的信息的寄存器及存储器的结构

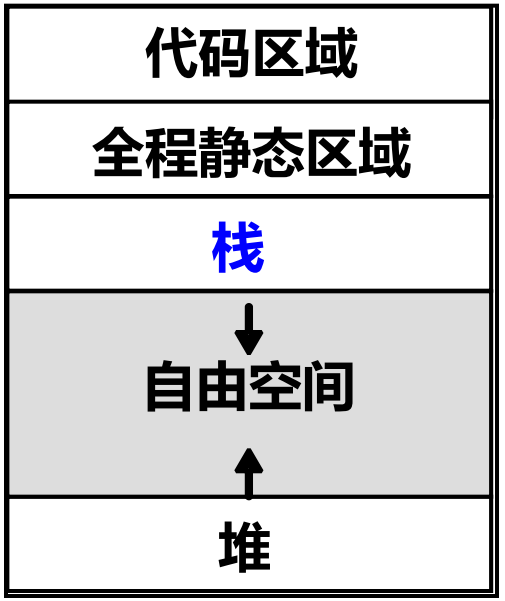
函数调用及返回的步骤

- **调用**
  - 保存调用信息（参数，返回地址）
  - 分配数据区（局部变量）
  - 控制转移给被调函数的入口
- **返回**
  - 保存返回信息
  - 释放数据区
  - 控制转移到上级函数（主调用函数）



函数运行时的动态存储分配

- **运行栈（stack）**用于分配后进先出LIFO的数据
  - e.g.， 函数调用
- **堆（heap）**用于不符合LIFO的数据
  - e.g.，指针所指向空间的分配



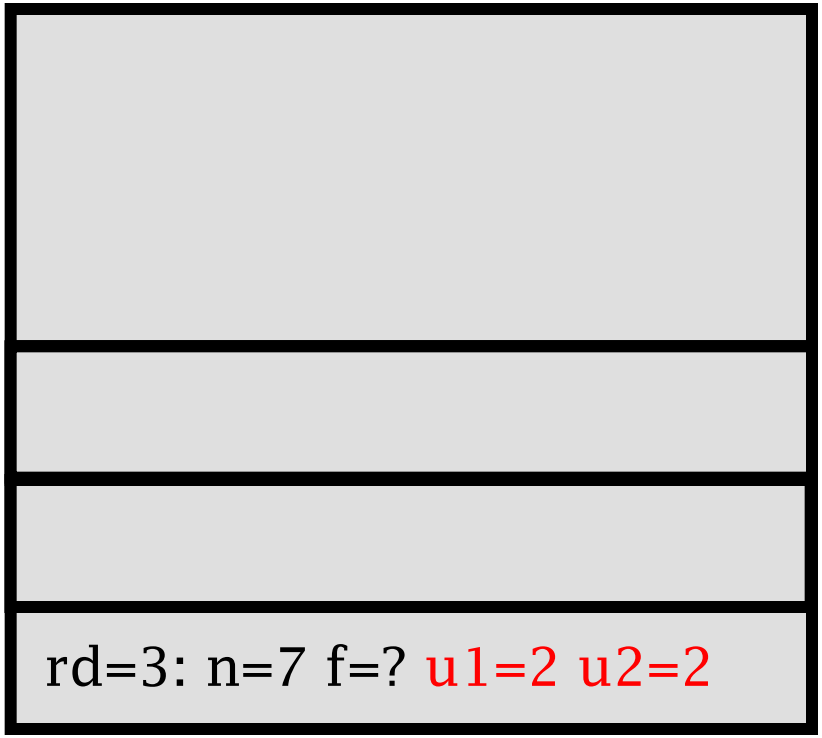
运行栈中的活动记录

- **函数活动记录（activation record）是动态存储分配中一个重要的单元**
  - 当调用或激活一个函数时，相应的活动记录包含了为其局部数据分配的存储空间

运行栈中的活动记录

- **运行栈随程序执行时的调用链而生长/缩小**
  - 每调用一个函数，执行一次进栈操作，把被调函数的活动记录入栈，即每个新的活动记录都分配在栈的顶部
  - 每次从函数返回时，执行一次出栈操作，释放本次的活动记录，恢复到上次调用所分配的数据区中
  - 被调函数中变量地址全部采用相对于栈顶的相对地址来表示
- **一个函数可有多个不同的活动记录各代表一次调用**
  - 递归的深度决定递归函数在运行栈中活动记录的数目
  - 递归函数中同一个局部变量，在不同的递归层次被分配给不同的存储空间，放在内部栈的不同位置

用栈模拟递归调用过程




```
void exp(int n, int& f) {
    int u1, u2;
    if (n<2)
        f = n+1;
    else {
        exp((int)(n/2), u1);
        exp((int)(n/4), u2);
        f = u1*u2;
    }
}
```

递归算法的非递归实现

- 以阶乘为例，非递归方式
  - 建立迭代
  - 递归转换为非递归

阶乘的迭代实现

```
// 使用循环迭代方法，计算阶乘n!的一种程序
long fact (long n) {
   int  m = 1;
   int  i ;
   if (n > 1)
       for ( i = 1;  i  <= n;  i++ )
              m  = m * i ;
   return m ;
}
```

递归的再研究

- 递归
  - 递归出口　　意味?
  - 递归规则　　意味?

阶乘的一种非递归实现

```
// 使用栈方法，计算阶乘n!的程序
long fact(long n) {
   Stack s;
   int  m = 1;

   while (n>1)                          // ?
        s.push(n--);
   while (!isEmpty(s))                  // ?
        m *= s.pop(s);
   return m;
}
```

非递归程序的实现原理

- **与函数调用原理相同，只不过把由系统负责的保存工作信息变为由程序自己保存**
  - 减少数据的冗余(主要是节省了局部变量的空间)，提高存储效率
  - 减少函数调用的处理以及冗余的重复计算，提高时间效率
- **程序要完成的工作分成两类**
  - **手头工作** 程序正在做的工作，须有其结束条件，不能永远做下去
  - **待完成工作** 某些不能一步完成的工作，必须暂缓完成，保存在栈中，必须含有完成该项工作的所有必要信息
- **程序须有序地完成各项工作**
  - **手头工作**和**待完成工作**可互相切换
  - **待完成工作**必须转换成手头工作才能处理

递归到非递归的转换

- **机械方法**
  1. 设置一**工作栈**保存当前工作记录
  2. 设置 t+2个**语句标号**
  3. 增加**非递归入口**
  4. **替换**第 i (i = 1, …, t)个递归规则
  5. 所有递归出口处增加语句： goto label t+1;
  6. 标号为t+1的语句的格式
  7. 改写循环和嵌套中的递归
  8. 优化处理

1. 设置一工作栈保存当前工作记录

- 在函数中出现的所有参数和局部变量都必须用栈中相应的数据成员代替
  - 返回语句标号域 (t+2个数值)
  - 函数参数(值参、引用型)
  - 局部变量

```
class elem {                    // 栈数据元素类型
      int rd;                   // 返回语句的标号
      Datatypeofp1 p1;                  // 函数参数
      …
      Datatypeofpm pm;
      Datatypeofq1 q1;                  // 局部变量
      …
      Datatypeofqn qn;
   };
   Stack<elem> S;
```

2. 设置t+2个语句标号

- label*0*：　　第一个可执行语句
- label*t+1*：　　设在函数体结束处
- label*i*　(1<=i<=t)：　第i个递归返回处

3. 增加非递归入口

```
// 入栈
elem tmp;
tmp.rd = t+1;
tmp.p1=p1;
…
tmp.pm = pm;
tmp.q1 = q1;
…
tmp.qn = qn;

S.push(tmp);
```

4. 替换第i (1,…,t)个递归规则

- 假设函数体中第 i (i=1, …, t)个递归调用语句为：

  rf(a1, a2, …, am);

  则用以下语句替换：

  ```
      S.push(i, a1, …, am);                    // 实参进栈
      goto label0;
      …..
  labeli:  x = S.pop();
  ```

  /* 退栈，然后根据需要，将x中某些值赋给栈顶的工作记录S.top ()：相当于把引用型参数的值回传给局部变量 */

5. 所有递归出口处增加语句

goto label*t+1*;

6. 标号为t+1的语句

```
switch ((x=S.top ()).rd) {
    case 0 :    goto label 0;
                break;
    case 1 :    goto label 1;
                break;
    …….
    case t+1 :  item = S.top(); S.pop();   // 返回处理
                break;
    default :   break;
}
```

7. 改写循环和嵌套中的递归

- 对于循环中的递归，改写成等价的goto型循环
- 对于嵌套递归调用

  譬如，　rf (… rf())

  改为：

```
exp1 = rf ( );
exp2 = rf (exp1);
…
expk = rf (expk-1)
```

  然后应用**规则 4** 解决

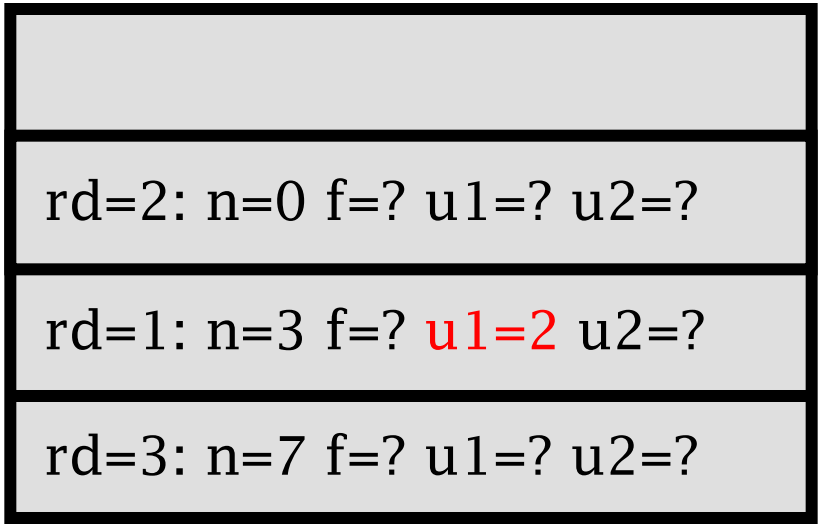
8. 优化处理

- 上述转换可得到带goto语句的非递归程序
- 优化
  - 去掉冗余的进栈/出栈
  - 根据流程图找出相应的循环结构，消去goto语句

转换示例

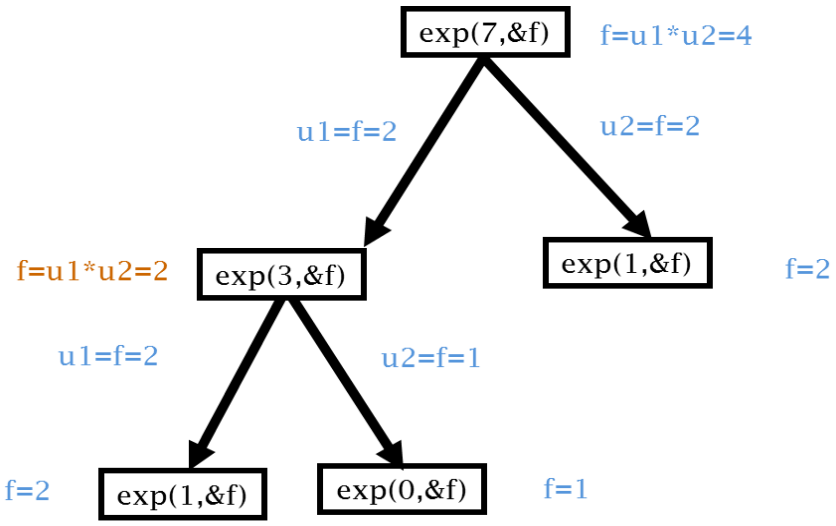
$$f(n) = \begin{cases} n+1, \text{if } n < 2 \\ f(\lfloor n/2 \rfloor) \times f(\lfloor n/4 \rfloor), \text{if } n \geq 2 \end{cases}$$

```
typedef struct elem {
    int rd, pn, pf, q1, q2;
} ELEM;
class nonrec {
    private:
      stack <ELEM> S;
    public:
      nonrec(void) { // constuctor
      void replace1(int n, int& f);
};
```



递归入口

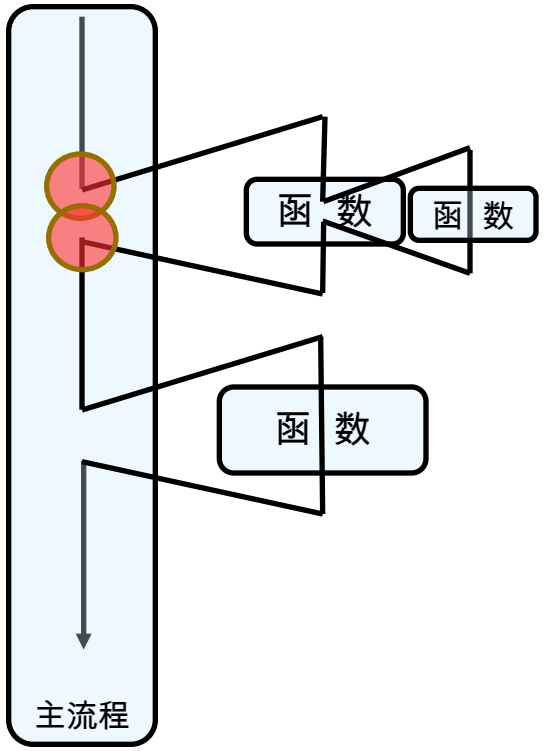
```
void nonrec::replace1(int n, int& f)  {
   ELEM x, tmp
   x.rd = 3;     x.pn = n;
   S.push(x);     // 压在栈底作”监视哨”
label0: if ((x = S.top()).pn < 2)  {
          S.pop( );
          x.pf = x.pn + 1;
          S.push(x);
          goto label3;
        }
```



第一个递归语句

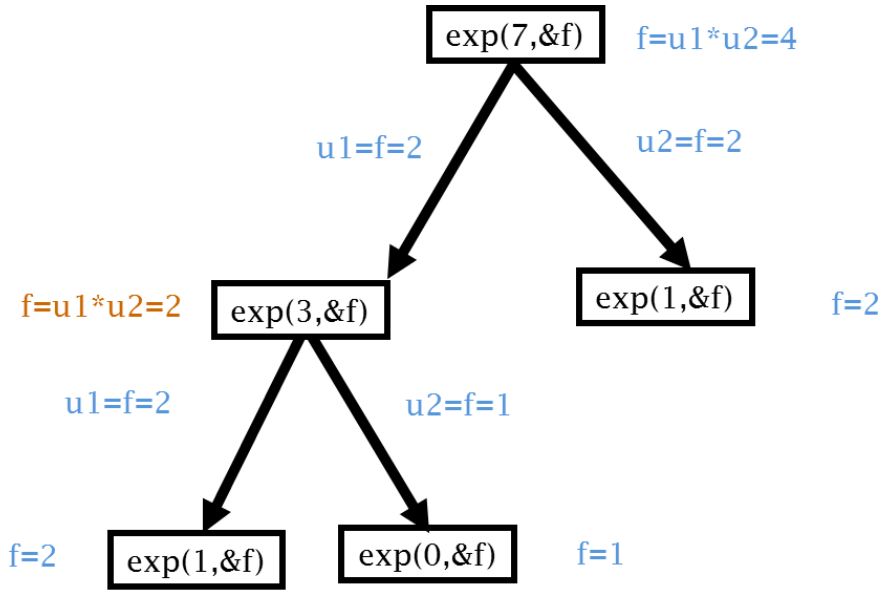
$$f(n) = \begin{cases} n+1, \text{if } n < 2 \\ f(\lfloor n/2 \rfloor) \times f(\lfloor n/4 \rfloor), \text{if } n \geq 2 \end{cases}$$

```
        x.rd = 1;        // 第一处递归
        x.pn = (int)(x.pn/2);
        S.push(x);
        goto label0;
label1: tmp = S.top(); S.pop();
        x = S.top(); S.pop();
        x.q1 = tmp.pf; // 修改u1=pf
        S.push(x);
```

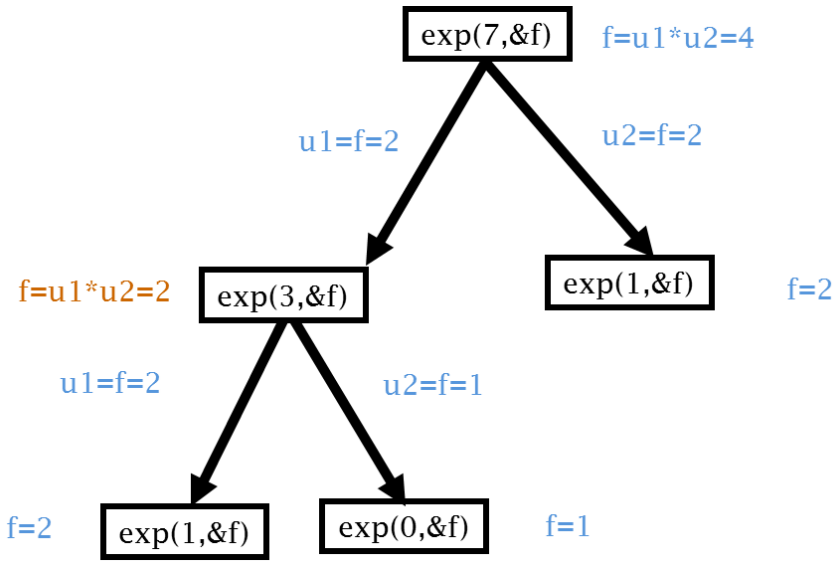


第二个递归语句

```
x.pn = (int)(x.pn/4);
        x.rd = 2;
        S.push(x);
        goto label0;
label2: tmp = S.top(); S.pop();
        x = S.top(); S.pop();
        x.q2 = tmp.pf;
        x.pf = x.q1 * x.q2;
        S.push(x);
```



标号为t+1的语句




```
label3: x = S.top();
        switch(x.rd)  {
            case 1 : goto label1;
                     break;
            case 2 : goto label2;
                     break;
            case 3 : tmp = S.top(); S.pop();
                     f = tmp.pf;              //计算结束
                     break;
            default : cerr << "error label number in stack";
                      break;
        }
}
```

优化后的非递归算法（1）

```
void nonrec::replace2(int n, int& f)  {
    ELEM x, tmp;
    x.rd = 3;   x.pn = n;   S.push(x);        // 入口信息
    do {                                      // 沿左边入栈
        while ((x=S.top()).pn >= 2){
            x.rd = 1;
            x.pn = (int)(x.pn/2);
            S.push(x);
        }
        x = S.top(); S.pop();                 // 原出口，n <= 2
        x.pf = x.pn + 1;
        S.push(x);
        while ((x = S.top()).rd==2) {// 如果是从第二个递归返回，则上升
          tmp = S.top(); S.pop();
          x = S.top(); S.pop();
          x.pf = x.q * tmp.pf;
          S.push(x);
       }
```

优化后的非递归算法(2)

```
        if ((x = S.topValue()).rd == 1) {
          tmp = S.top(); S.pop();
          x = S.top(); S.pop();
          x.q = tmp.pf;    S.push(x);
          tmp.rd = 2;  // 进入第二个递归
          tmp.pn = (int)(x.pn/4);
          S.push(tmp);
        }
    } while ((x = S.top()).rd != 3);
      x = S.top(); S.pop();
      f = x.pf;
}
```

递归的非递归实现

- 请大家思考，用机械的转换方法
  - 2阶斐波那契函数

    $f_0=0, f_1=1, f_n = f_{n-1}+ f_{n-2}$
  - 背包问题
  - Hanio Tower

背包问题

- 简化版：设有一背包可放入的物品重量为S，现有n件物品，重量分别为w_0，w_1，…，w_{n-1}，问能否从这n件物品中选择若干件放入背包，使其重量之和正好为S。

  若存在一种符合上述要求的选择，则称此背包问题有解，否则无解。

背包问题 ： 递归实现

$$knap(S,n)=\begin{cases}true, & 当S=0\\ false, & 当S<0或S>0且n<1\\ knap(S-w_{n-1},n-1)\,||\,knap(S,n-1), & 当S>0且n\geq 1\end{cases}$$

```
bool knap(int s, int n) {     // 递归实现
        if (s == 0)   return true;
        if ((s < 0) || (s>0 && n <1))
                return false;
        if (knap (s-w[n-1], n-1)) {
                cout << w[n-1];
                return true;
        }
        else return knap(s, n-1);
}
```

背包问题： 递归规则

- **规则**
  1. 若w[n-1]包含在解中，求解knap(s-w[n-1], n-1)
  2. 若w[n-1]不包含在解中，求解knap(s, n-1)
- **递归出口，共有3种返回地址**:
  1. 计算knap(s, n)完毕返回调用它的函数
  2. 计算knap(s-w[n-1], n-1)完毕，返回到本调用函数继续计算
  3. 计算knap(s, n-1)完毕，返回到本调用函数继续计算

背包问题 ：转换规则

1. 凡调用语句knap(s, n)均代换成

(1) tmp.s = s;
tmp.n = n;
tmp.rd = 返回地址编号
(2) stack.push(tmp);
(3) 转向递归入口

2. 将调用出口的返回处理统一为

(1) stack.pop(&tmp);
(2) 分以下情况执行
i) tmp.rd = 0 时，算法结束
ii) tmp.rd = 1时，转规则1返回处理
iii) tmp.rd = 2时，转规则2返回处理

递归转非递归的性能实验

快排的时间对照（单位ms）

方法 \ 数据量	10000	100000	1000000	10000000
递归快排	4.5	29.8	268.7	2946.7
机械方法的非递归快排	1.6	23.3	251.7	2786.1
非机械方法的非递归快排	1.6	20.2	248.5	2721.9
STL中的sort	4.8	59.5	629.8	7664.1

注：测试环境

Intel Core Duo CPU T2350

内存 512MB

操作系统 Windows XP SP2

编程环境 Visual C++ 6.0

递归转非递归的性能实验

- 递归与非递归处理问题的规模比较

  e.g., 递归求f(x)

```
int f (int x) {
    if (x==0) return 0;
    return f(x-1) + 1;
}
```

  - 在默认设置下，当 *x* 超过 11,772 时会出现堆栈溢出

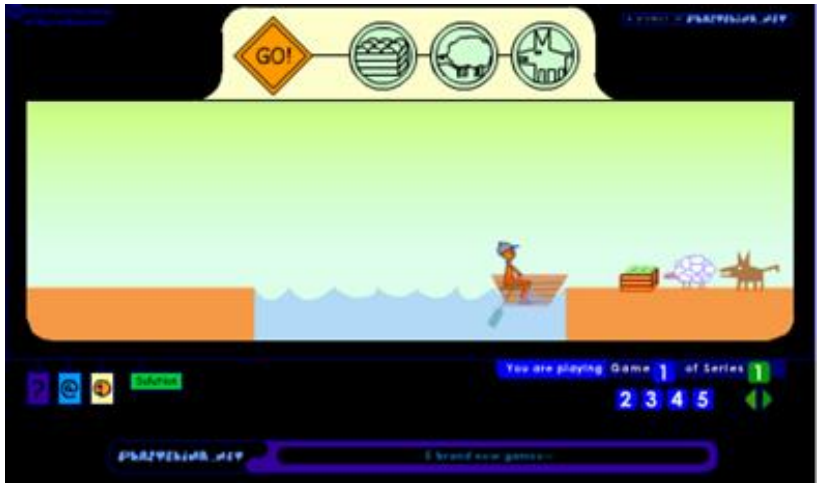
- 非递归求f(x)，栈中元素记录当前 *x* 值和返回值
  - 在默认设置下，当 *x* 超过32,375,567时会出现错误

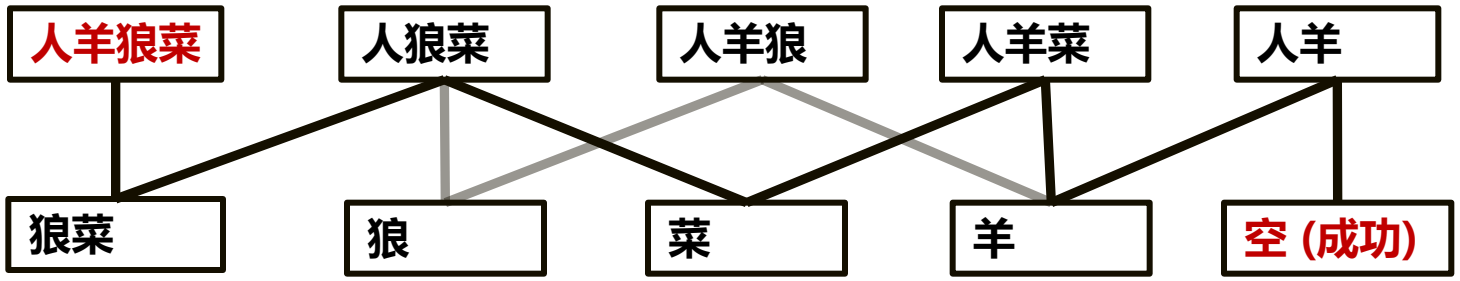
队列的应用

- 队列的特点：先进先出
  - 体现了元素间的公平性
- 常用于满足先来先服务特性的应用
  - 调度或缓冲
    - 消息缓冲器
    - 邮件缓冲器
    - 计算机的硬设备之间的通信数据缓冲
    - 操作系统的资源管理
  - 宽度优先搜索

队列应用：农夫过河

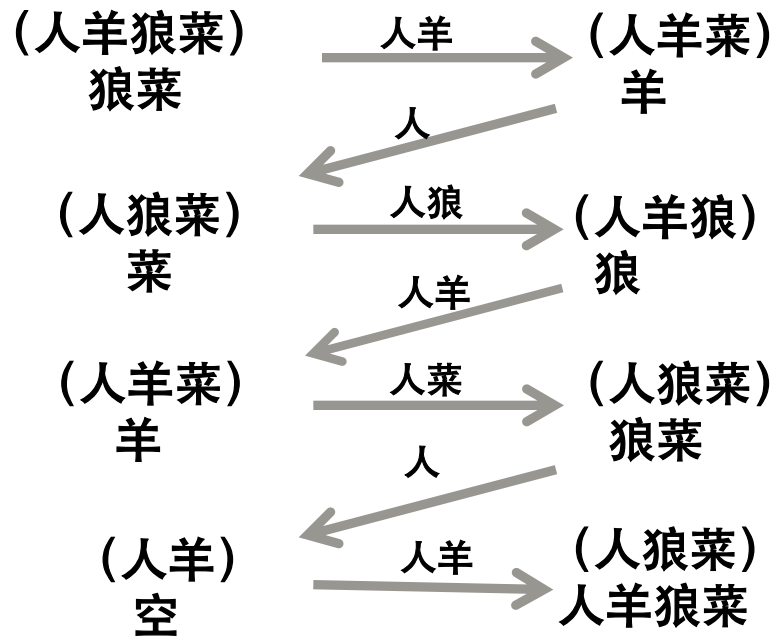
- **问题抽象**：人狼羊菜乘船过河
  - 只有人能撑船
  - 船只有两个位置（包括人）
  - 狼羊、羊菜不能在没有人时共处





队列应用：农夫过河

- **数据抽象**
  - 不合理状态：狼羊、人菜；羊菜、人狼；狼羊菜、人
  - 顶点表示“原岸状态”—10种（包括“空”）
  - 边：一次合理的渡河操作实现的状态转变



队列应用：农夫过河问题

- **数据抽象**
  - 每个角色的位置进行描述
    - 农夫、狼、菜和羊，四个目标各用一位（按农夫、狼、白菜、羊次序），目标在原岸位置：0，目标岸：1

0	1	0	1

    - e.g., 0101 表示农夫、白菜在原岸，而狼、羊在目标岸（此状态为不安全状态）

队列应用：农夫过河问题

- **数据表示**
  - 用整数 status 表示上述人、物的状态
    - 整数 0x08 表示的状态

1	0	0	0

    - 整数 0x0F 表示的状态

1	1	1	1

  - 如何从上述状态中得到每个角色所在位置？函数返回值为
    - 真（1），表示所考察人或物在目标岸
    - 否则，表示所考察人或物在起始岸

队列应用：农夫过河问题

- 确定每个角色位置的函数

```
bool farmer(int status) {
    return ((status & 0x08)  != 0);
}

bool wolf(int status) {
    return ((status & 0x04)  != 0);
}

bool cabbage(int status) {
    return ((status & 0x02)  != 0);
}

bool goat(int status) {
    return ((status & 0x01)  != 0);
}
```

人	狼	菜	羊
1	x	x	x
x	1	x	x
x	x	1	x
x	x	x	1

队列应用：农夫过河问题

- **安全状态的判断**

人	狼	菜	羊
0	1	0	1

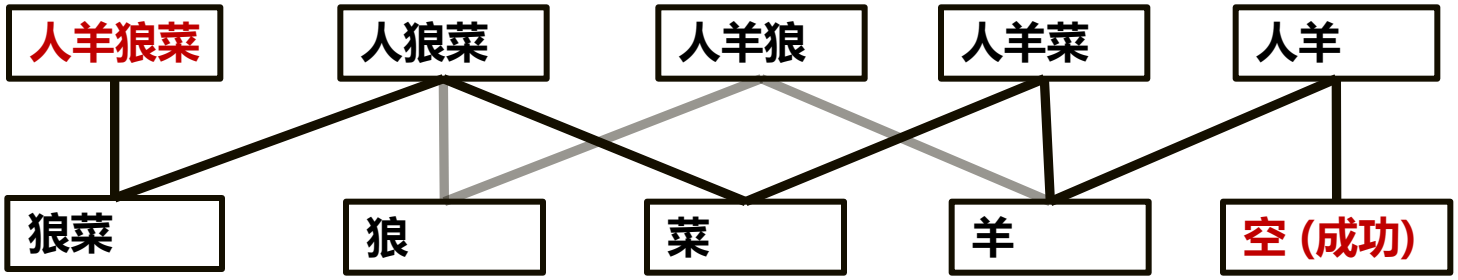
```
bool safe(int status)  {                    // 返回 true:安全，false:不安全
   if ((goat(status) == cabbage(status)) &&
       (goat(status) != farmer(status)))
           return(false);                   // 羊吃白菜
   if ((goat(status) == wolf(status)) &&
       (goat(status) != farmer(status)))
           return(false);                   // 狼吃羊
   return(true);                            // 其它状态为安全
}
```

队列应用：农夫过河问题

- **算法抽象**

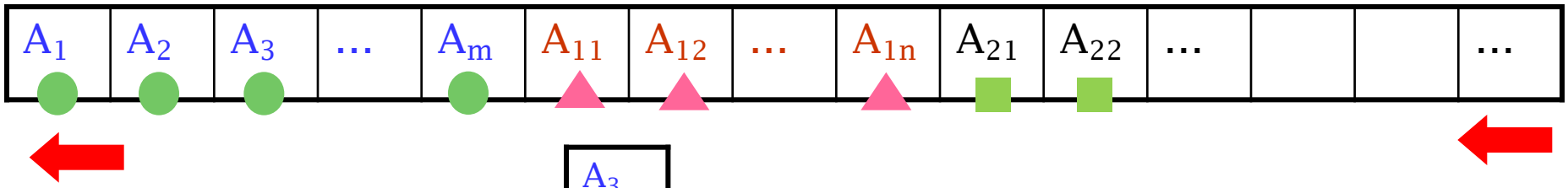
  从初始状态 0000（**人羊狼菜**，整数0）出发，寻找全部由安全状态构成的状态序列，以 状态1111（空，整数15）为最终目标

  - 状态序列中 每个 状态都可以从前一状态通过农夫（可以带一样东西）划船过河的动作到达
  - 序列中不能出现 重复 状态

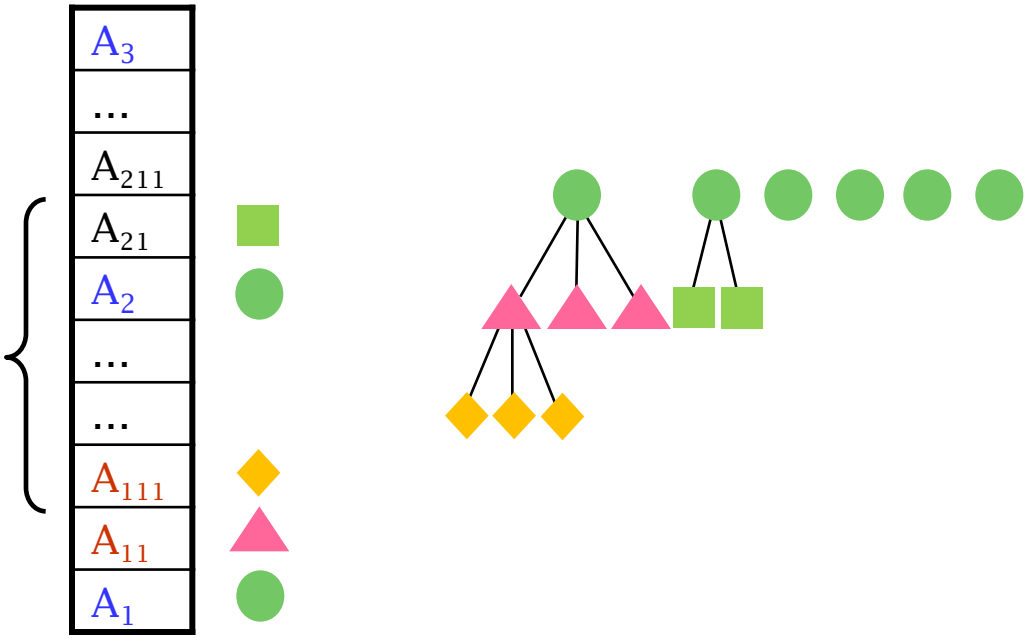


队列应用：农夫过河问题

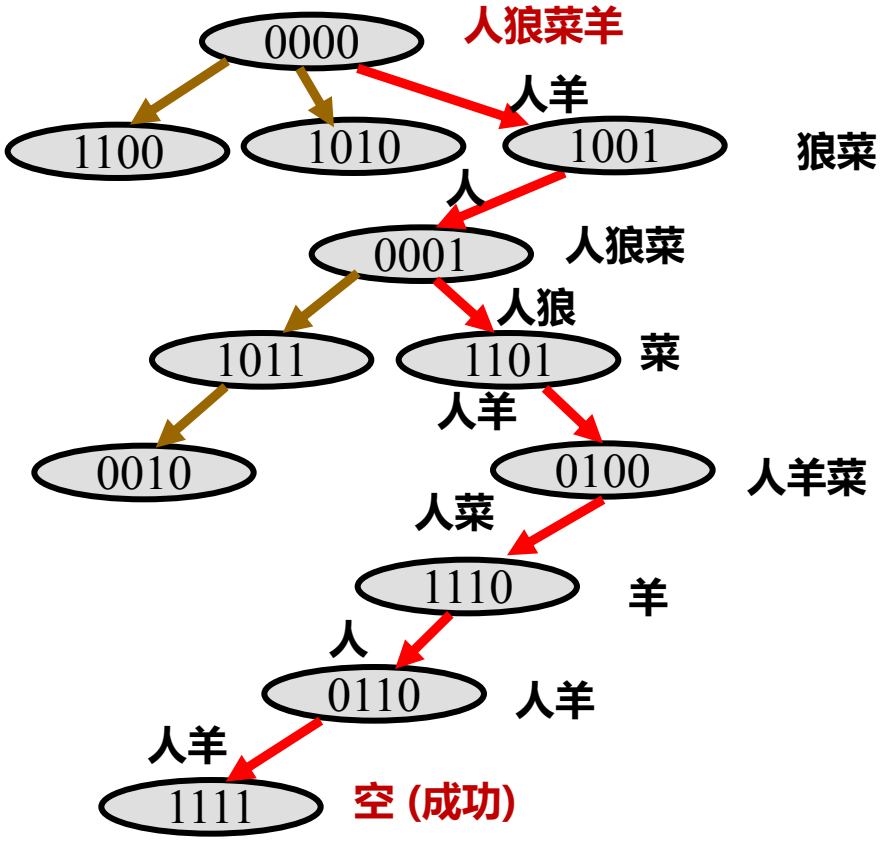
广度优先：(m 种状态)

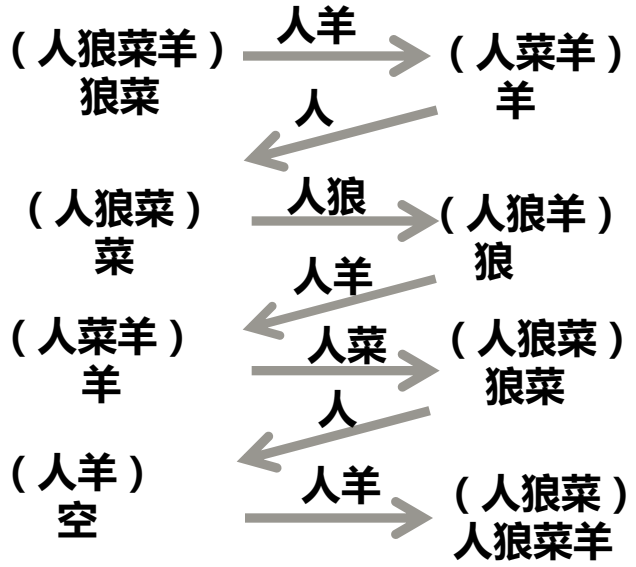


深度优先：(m 种状态)



队列应用：农夫过河问题


0000
人狼菜羊
人羊
1100
1010
1001
狼菜
人
0001
人狼菜
人狼
1011
1101
菜
人羊
0010
0100
人羊菜
人菜
1110
羊
人
0110
人羊
人羊
1111
空 (成功)


(人狼菜羊)
狼菜
人羊
(人菜羊)
羊
人
(人狼菜)
菜
人狼
(人狼羊)
狼
人羊
(人菜羊)
羊
人菜
(人狼菜)
狼菜
人
(人羊)
空
人羊
(人狼菜)
人狼菜羊

队列应用：农夫过河问题

- **算法设计**
  - 定义一个整数队列 moveTo，它的每个元素表示一个可以安全到达的中间状态
  - 定义一个数据结构 记录已被访问过的各个状态，以及已被发现的能够到达当前这个状态的路径
    - 用顺序表 route 的第 i 个元素记录状态i是否已被访问过
    - 若 route[i] 已被访问过，则在这个顺序表元素中记入前驱状态值； -1表示未被访问
    - route 的大小（长度）为 16

队列应用：农夫过河问题

- **算法实现**

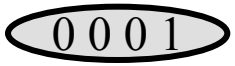
```
void solve() {
    int movers, i, location, newlocation;
    vector<int> route(END+1, -1);
                    // 记录已考虑的状态路径
    queue<int> moveTo;

    // 准备初值
    moveTo.push(0x00);
    route[0] = 0;
```

队列应用：农夫过河问题

- **算法实现**

人狼菜羊




```
while (!moveTo.empty() && route[15] == -1) {
    // 得到现在的状态
    status = moveTo.front();
    moveTo.pop();
    for (movers = 1; movers <= 8; movers <<= 1) {
    // 农夫总是在移动，随农夫移动的也只能是在农夫同侧的东西
        if (farmer(status)  == (bool)(status & movers)) {
            newstatus = status ^ (0x08 | movers);
            // 安全的，并且未考虑过的走法
            if (safe(newstatus) && (route[newstatus] == -1)) {
                route[newstatus] = status;
                moveTo.push(newstatus); }
        }
    }
}
```

队列应用：农夫过河问题

- **算法实现**

```
    // 反向打印出路径
    if (route[15] != -1) {
        cout << "The reverse path is :  " << endl;
        for (int status = 15; status >= 0; status = route[status]) {
            cout << "The status is :  " << status << endl;
            if (status == 0) break;
        }
    }
    else
        cout << "No solution. " << endl;
}
```