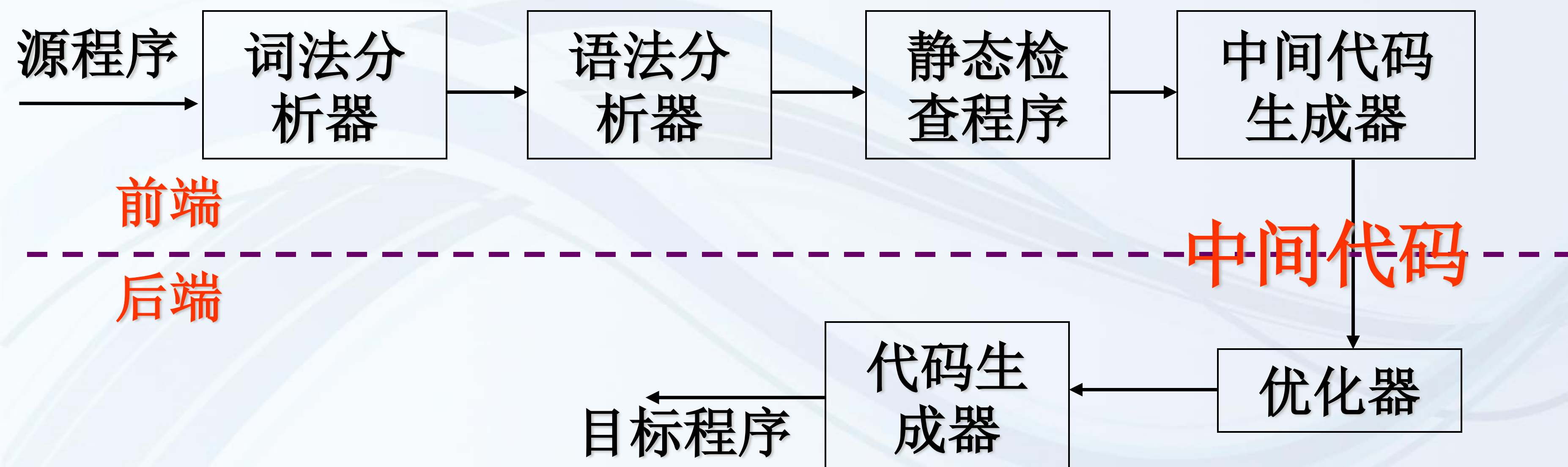
# 中间代码生成

***Intermediate Code Generation***

# 6.1 中间表示

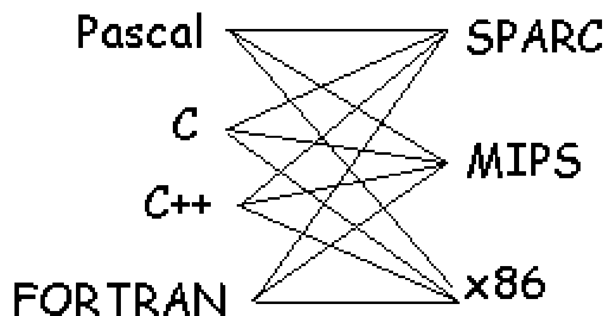
- ❖编译器可以分成两部分：
  - ➢前端将源程序转成中间表示
  - ➢后端基于中间表示产生目标代码

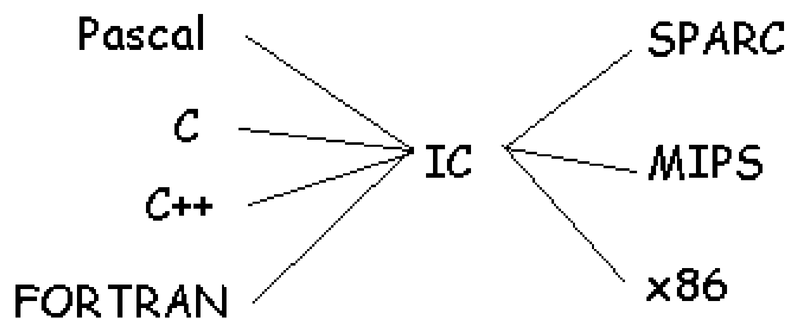


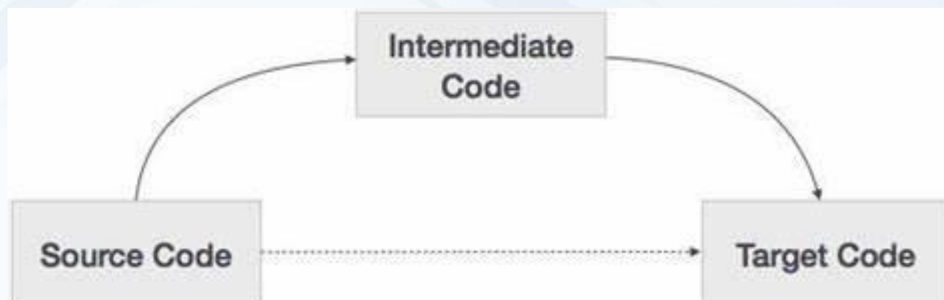
# 中间表示(II)

❖ 优点：容易为不同目标机器开发不同后端.

❖ 缺点：编译过程变慢（因为中间步骤）



# 中间表示(III)

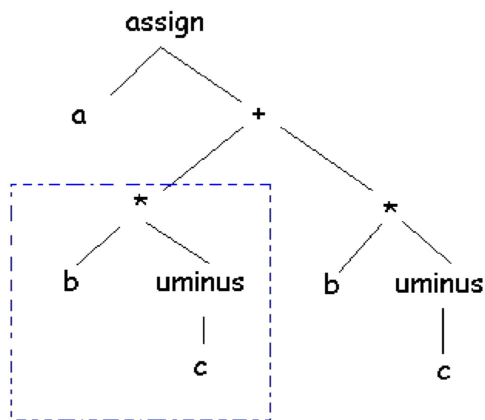
常用方法:

- 语法树
- 后缀式
- 三地址码

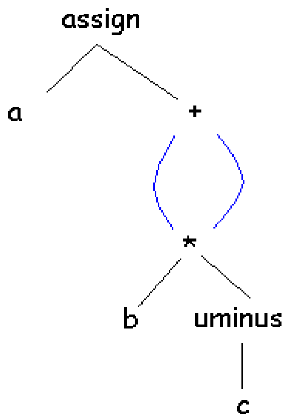
# 图形表示

❖ 图形表示法: 语法树, 有向无环图（Directed Acyclic Graphs, DAG）

a := b * -c + b * -c




(a) syntax tree




(b) DAG

a+b+a+b对应的DAG是什么？

后缀式是语法树的线性表示

**a b c uminus * b c uminus * + assign**

后根遍历 https://upimg.baike.so.com/doc/7105441-7328475.html

# 图形表示（例）

❖ 给出a+a*(b-c)+(b-c)*d对应的语法树、DAG和后缀式

语法树

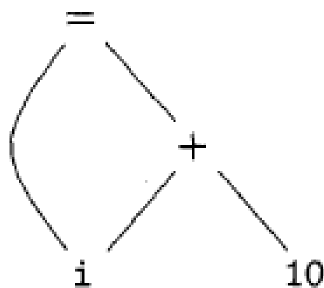
DAG

后缀式：aabc-*+bc-d*+

有没有表达式，对应的DAG有两个a?

# 值编码

- 语法树或DAG的结点存放在一个记录数组中
  - 数组的每一行是一个记录，表示一个结点。可用记录在数组中的整数下标来引用，该整数称值编码。
  - 每条记录的第一个字段是运算符代码（叶结点为记号）
  - 叶结点包含一个附加字段, 存放词法值（标识符时为指向符号表相应项目的指针，数字时是常量）
  - 内部结点（非叶结点）包含两个附加字段，指向左右子结点




(a) DAG

| | | | |
|---|---|---|---|
| 1 | id | → to entry for i | |
| 2 | num | 10 | |
| 3 | + | 1 | 2 |
| 4 | = | 1 | 3 |
| 5 | ... | | |

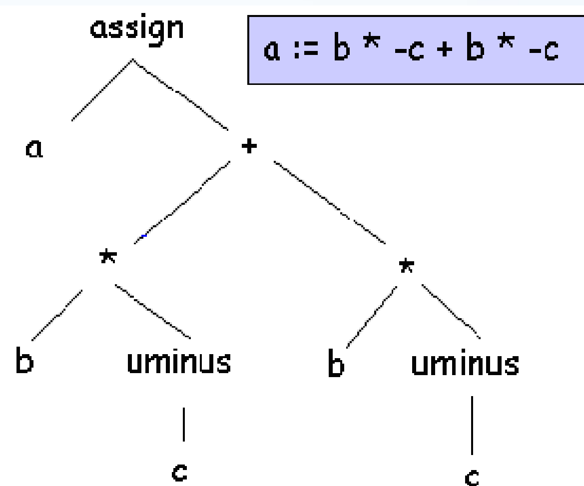
(b) Array.

# 6.2 三地址码

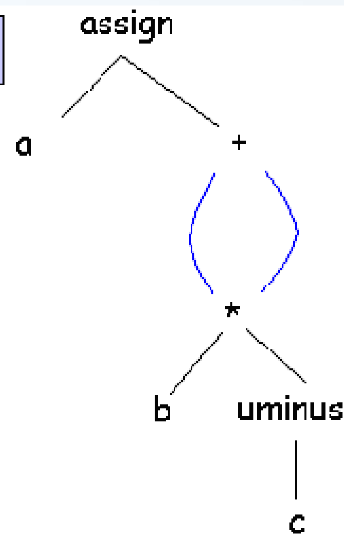
❖ 三地址码（*TAC*，*Three Address Code*）用来描述指令序列

# 三地址码(II)

- 语法树和DAG的线性表示
- 具体实现形式：四元式、三元式、间接三元式




(a) syntax tree




(b) DAG

$t_1$ := - c
$t_2$ := b * $t_1$
$t_3$ := - c
$t_4$ := b * $t_3$
$t_5$ := $t_2$ + $t_4$
a := $t_5$

(a) code for syntax tre

$t_1$ := - c
$t_2$ := b * $t_1$
$t_3$ := $t_2$ + $t_2$
a := $t_3$

(b) code for DAG

# 常用三地址码(I)

C语言里x=y[i]是什么意思，与带下标复制指令有何不同?

- ❖ 赋值指令：x=y op z
  - x,y,z是地址，op是双目运算符
  - op是单目运算符时，为x=op y
  - 复制指令：x=y
  - 带下标的复制指令
    - ✓ x=y[i]:将距离y处i个内存单元的位置中存放的值赋给x.
    - ✓ x[i]=y:将距离x处i个内存单元的位置中的内容置为y的值
  - 地址赋值指令x=&y：将x的右值设置为y的左值
  - 指针赋值指令
    - ✓ x=*y：将x的右值置为地址y的值。
    - ✓ *x=y：把y的右值赋到地址x

# 常用三地址码(II)

- 转移指令 goto L
  - 表示下一步执行带有标号L的三地址指令
  - 条件转移指令
    - if x goto L ：x 为真时转L
    - ifFalse x goto L ：x为假时转L
    - if x relop y goto L: x和y之间满足relop关系时转L

# 常用三地址码(III)

❖过程调用及返回

- param x 参数传递 :传递参数x
- call p,n 过程调用：调用p过程，实参个数为n
- y=call p,n 函数调用: 调用p函数，返回值为y

quicksort(m,n)

```
param m
param n
call quicksort, 2
```

# 例

- 翻译语句：do i = i + 1; while (a[i]<v);

```
L: t1  = i + 1
   i   =  t1
   t2  = i * 8
   t3  = a[t2]
   if t3<v goto L
```

为什么有$t_2$=i*8?

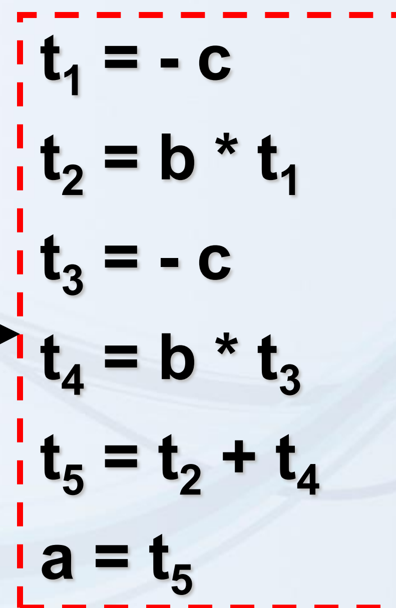
```
100: t1  = i + 1
101: i   =  t1
102: t2  = i * 8
103: t3  = a[t2]
104: if t3<v goto 100
```

采用位置号

# 三地址码的实现(四元式)

- 四元式有4个字段：op, $arg_1$, $arg_2$, 和result
- $arg_1$, $arg_2$, 和result通常指向符号表项目的入口

| | op | arg1 | arg2 | result |
|---|---|---|---|---|
| (0) | uminus | c | | $t_1$ |
| (1) | * | b | $t_1$ | $t_2$ |
| (2) | uminus | c | | $t_3$ |
| (3) | * | b | $t_3$ | $t_4$ |
| (4) | + | $t_2$ | $t_4$ | $t_5$ |
| (5) | = | $t_5$ | | a |



# 一些四元式

x[i]=y

| | | | | |
|---|---|---|---|---|
| (1) | [ ]= | x | i | $t_3$ |
| (2) | = | y | | $t_3$ |

| | | | |
|---|---|---|---|
| &= | y | $t_3$ | $t_3$ |

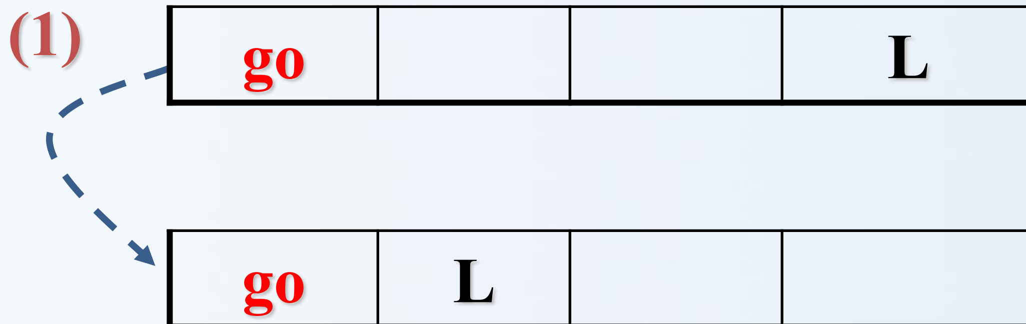
y=x[i]

| | | | | |
|---|---|---|---|---|
| (1) | =[ ] | x | i | $t_3$ |
| (2) | = | $t_3$ | | y |

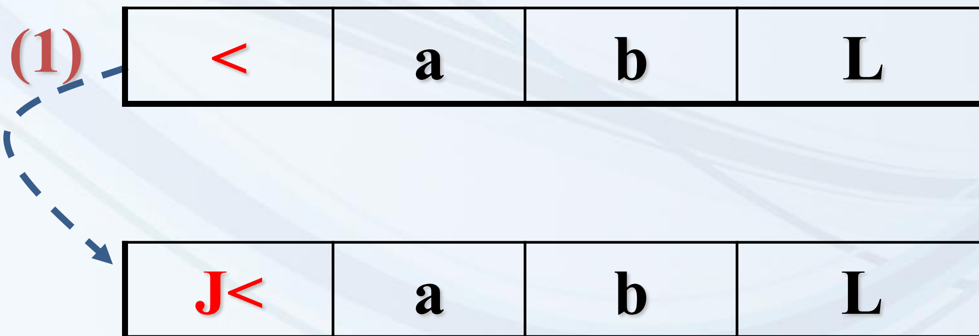
四元式的表示并不统一

# 一些四元式

goto L




if a<b goto L



# 例

- 翻译语句：do i = i + 1; while (a[i]<v);

  100: $t_1$ = i + 1

  101: i = $t_1$

  102: $t_2$ = i * 8

  103: $t_3$ = a[$t_2$]

  104: if $t_3$<v goto 100

四元式

| | | | | |
|---|---|---|---|---|
| (1) | + | i | 1 | $t_1$ |
| (2) | = | $t_1$ | | i |
| (3) | * | i | 8 | $t_2$ |
| (4) | =[ ] | a | $t_2$ | $t_3$ |
| (5) | < | $t_3$ | v | (1) |

# 三地址码的实现(三元式)

- 三元式可以避免引入临时变量
  - 使用获得变量值的位置来引用前面的运算结果

$t_1$ = - c
$t_2$ = b * $t_1$
$t_3$ = - c
$t_4$ = b * $t_3$
$t_5$ = $t_2$ + $t_4$
a = $t_5$

| | op | arg1 | arg2 |
|---|---|---|---|
| (0) | uminus | c | |
| (1) | * | b | (0) |
| (2) | uminus | c | |
| (3) | * | b | (2) |
| (4) | + | (1) | (3) |
| (5) | = | a | (4) |

# 三元式（II）

- 间接三元式
  - 包含一个指向三元式的指针列表，而不是列出三元式序列本身

| | statement |
|---|---|
| (0) | (14) |
| (1) | (15) |
| (2) | (16) |
| (3) | (17) |
| (4) | (18) |
| (5) | (19) |

| | op | arg1 | arg2 |
|---|---|---|---|
| (14) | uminus | c | |
| (15) | * | b | (14) |
| (16) | uminus | c | |
| (17) | * | b | (16) |
| (18) | + | (15) | (17) |
| (19) | = | a | (18) |

# 6.3 类型和声明

- 类型检查
  - 利用一组逻辑规则来确定程序在运行时的行为
    - ✓ 保证运算分量的类型和运算符的预期类型匹配
- 类型的应用
  - 确定一个名需要的存储空间
  - 计算一个数组元素引用的地址
  - 插入显式的类型转换
  - 选择算术运算符的正确版本

# 类型表达式(type expression)

- 描述类型的结构
- 类型表达式: 类型可以是基本类型，也可以是类型构造符（也称类型构造算子, type constructor）作用于类型而得
    - 基本类型: *boolean, char, integer, float, void*
    - 数组类型构造符*array*(I,T)：其中T是类型表达式，I是整数。如int A[10] 的类型表达式是 *array*(10,*integer*)
        - 均有相应的树形表示

# 类型表达式（II）

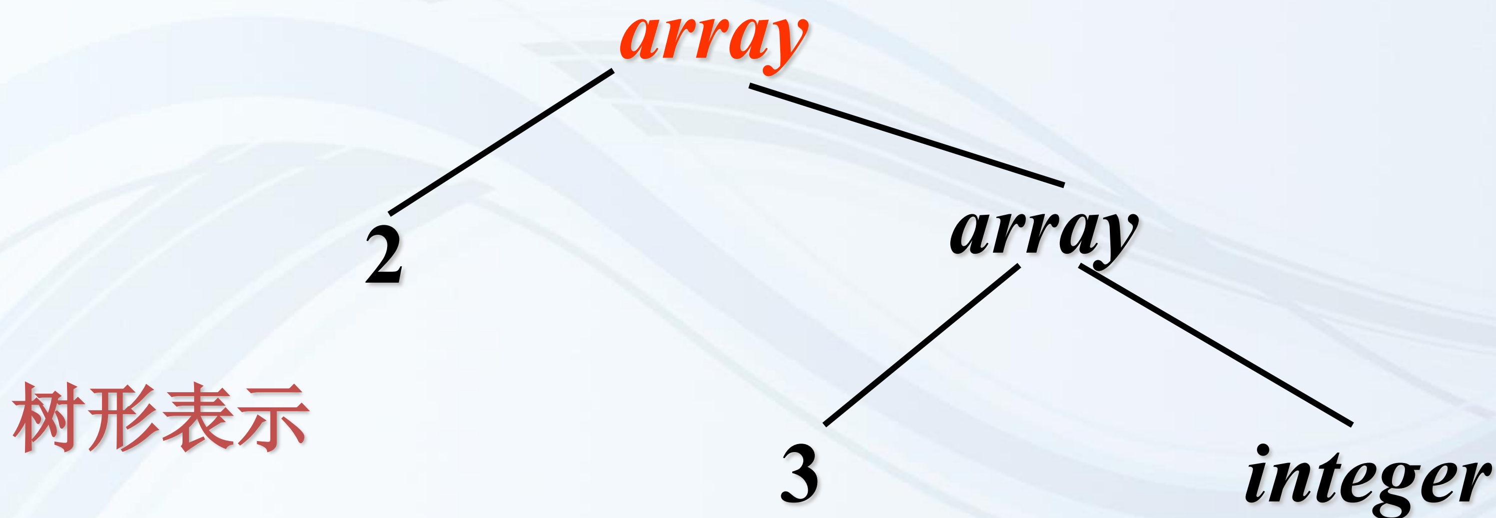
int x[2][3];

x:ARRAY[1..2, 1..3] OF integer;

类型表达式: array(2,array(3,integer))

或者：  array(1..2,array(1..3,integer))

已知float y[3][4][5];
则y的类型表达式及
其树形表示各是什么？



# 类型表达式（III 指针）

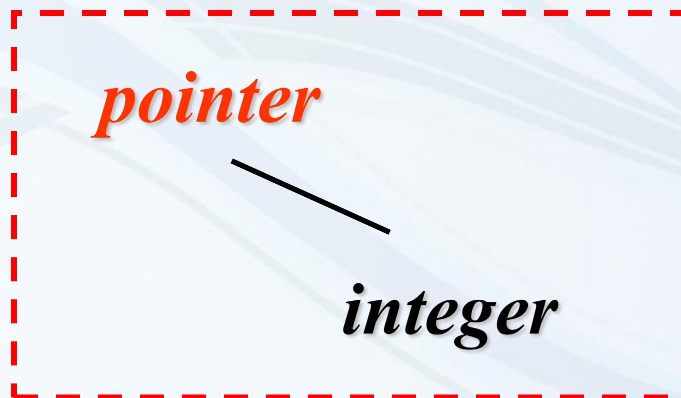
- 指针int* aa;

  var aa:↑integer;

  类型表达式 *pointer* (*integer*)

已知float* a[10];
则a的类型表达式及其树形表示各是什么？

树形表示



# 类型表达式（IV　函数）

**函数**例1 float divide(int i, int j)

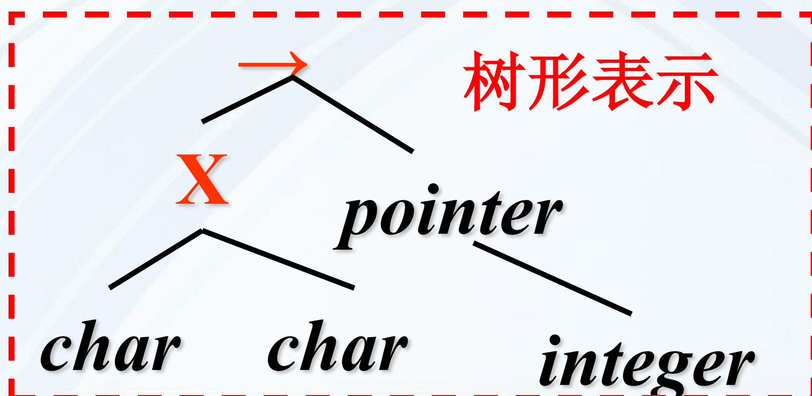
**FUNCTION  divide(i,j:integer):real;**

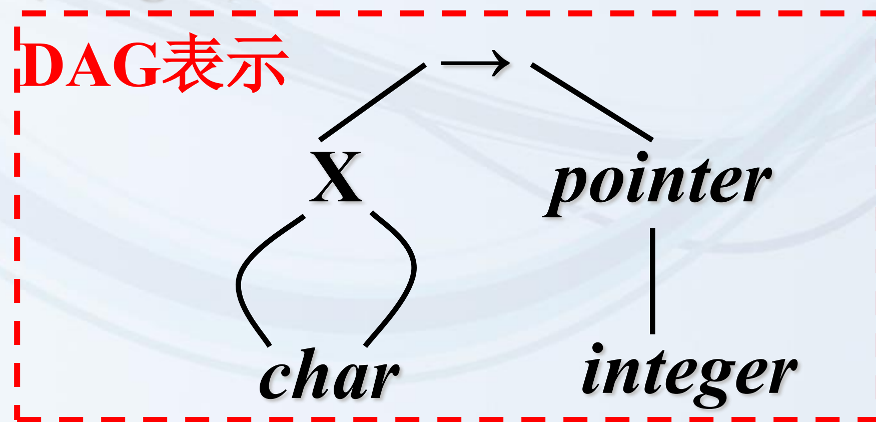
类型表达式　　*integer X integer → float*

例2  int *f(char a, char b);

**FUNCTION f(a,b:char): ↑ integer;**

类型表达式*char X char → pointer(integer)*



# 类型表达式(V 结构体)

```
typedef struct person={
        char name[8];
        int sex;
        int age;
}
   struct person table[50];
```

```
TYPE person=RECORD
        name:ARRAY[1..8] OF integer;
        sex:integer;
        age:integer;
END;
VAR table:ARRAY[1..50] OF person;
```

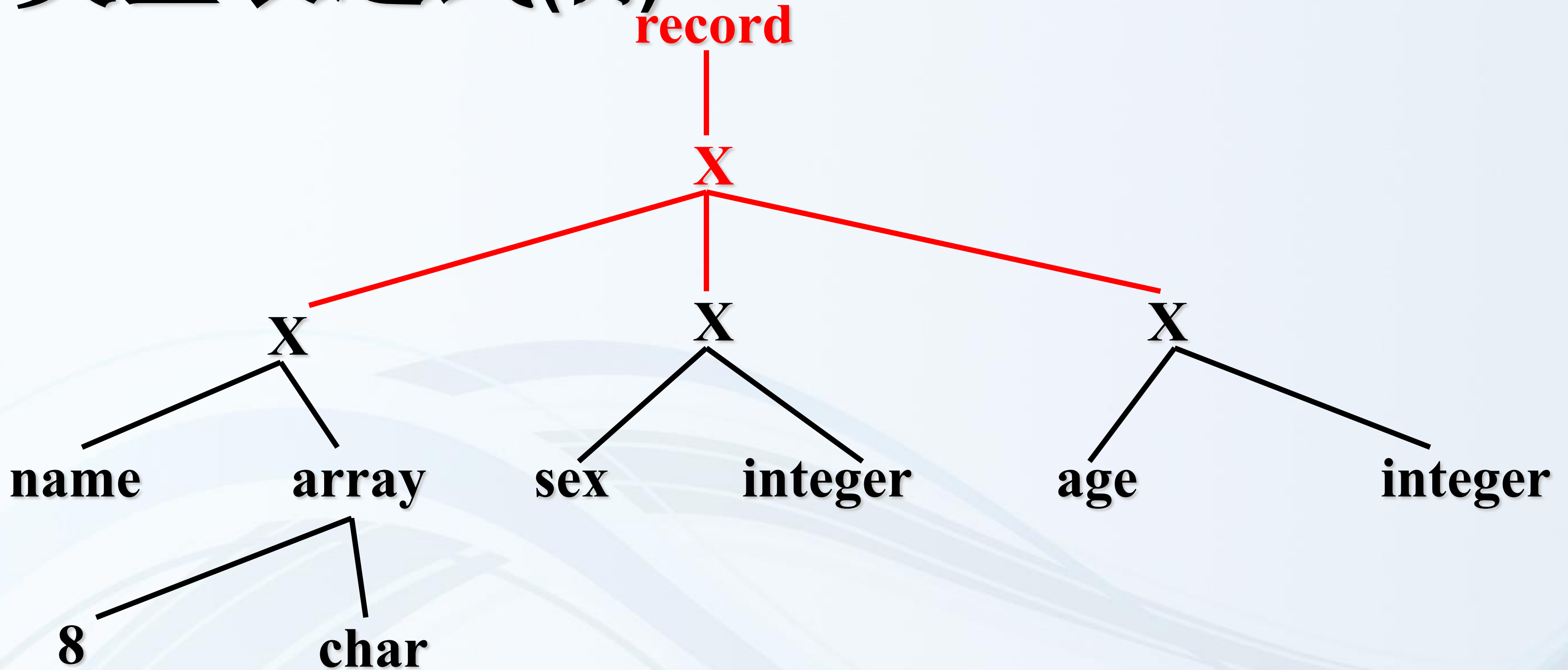
则person之类型表达式：

record((name X array(8,char)) X (sex X integer) X (age X integer))

table之类型表达式：

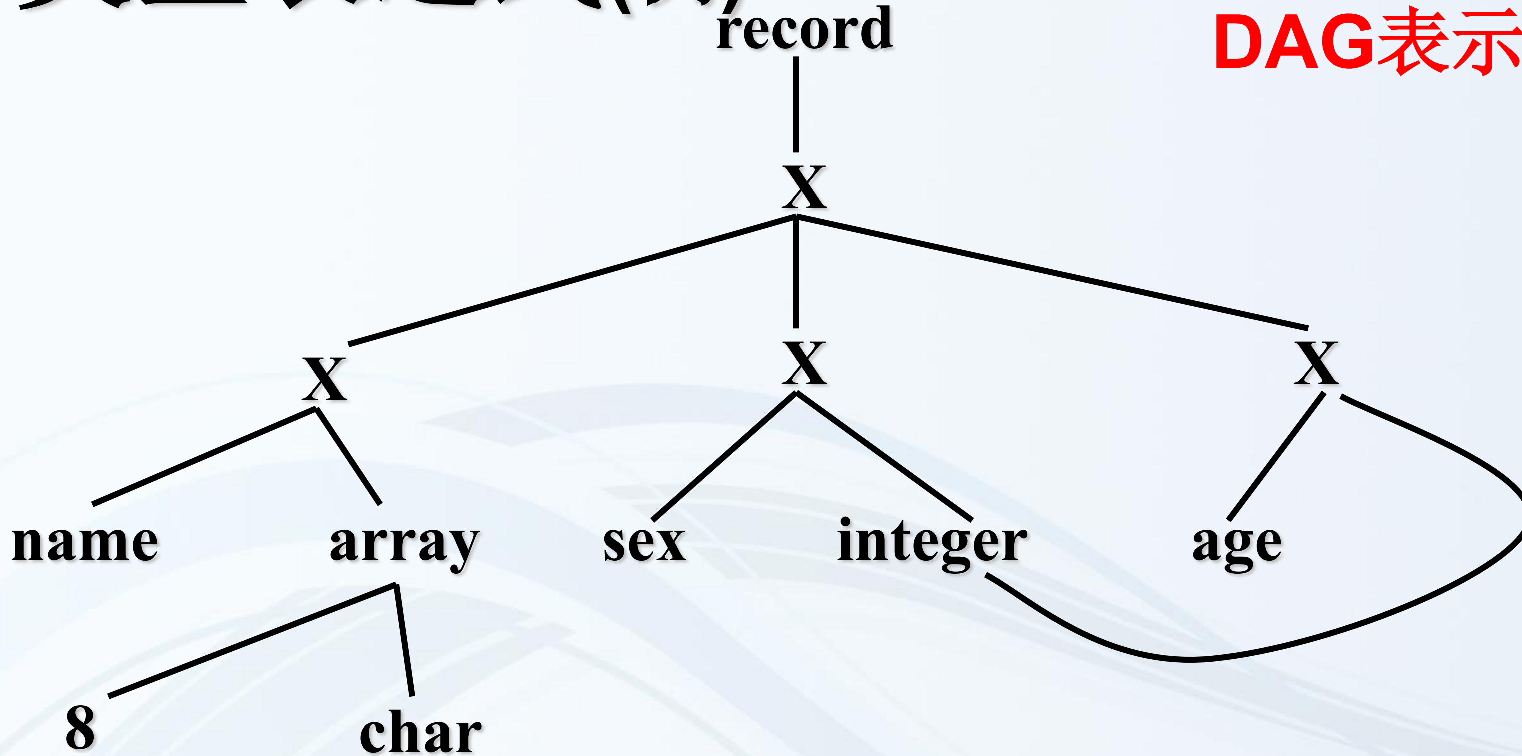
array(50, person)

# 类型表达式(例)




record((name X array(8,char)) X (sex X integer) X (age X integer))

# 类型表达式(例)

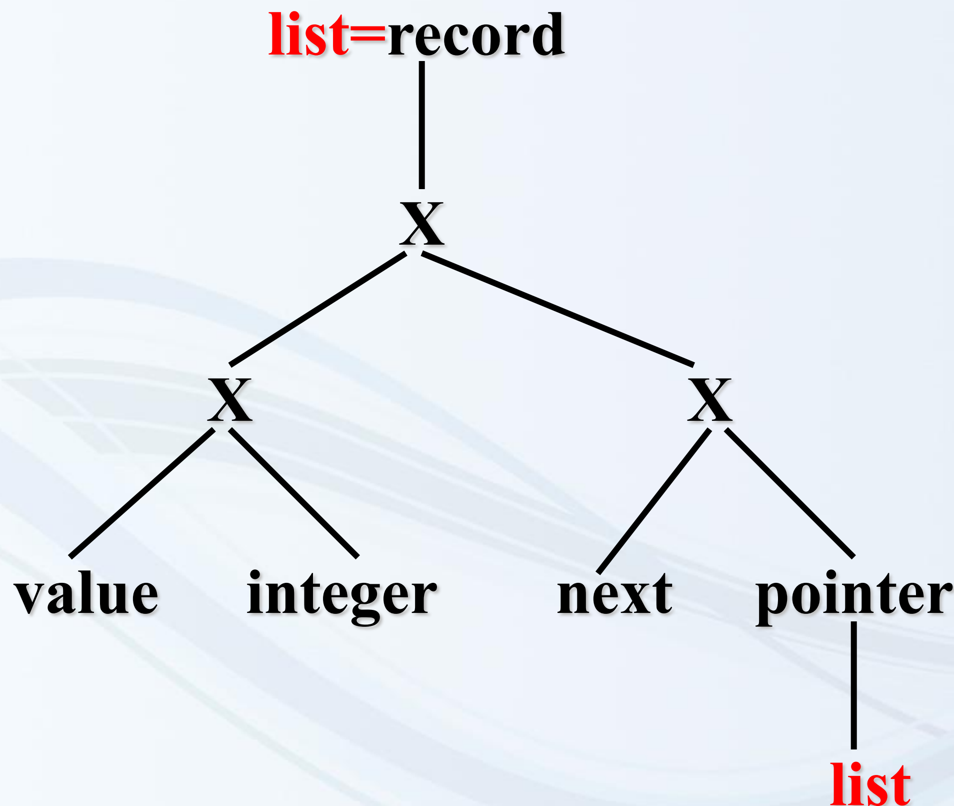



DAG表示

record((name X array(8,char)) X (sex X integer) X (age X integer))

# 类型表达式(例)

```
struct list{
    int value;
    struct list *next;
}
```



# 类型表达式(例)

```
struct list{
    int value;
    struct list *next;
}
```