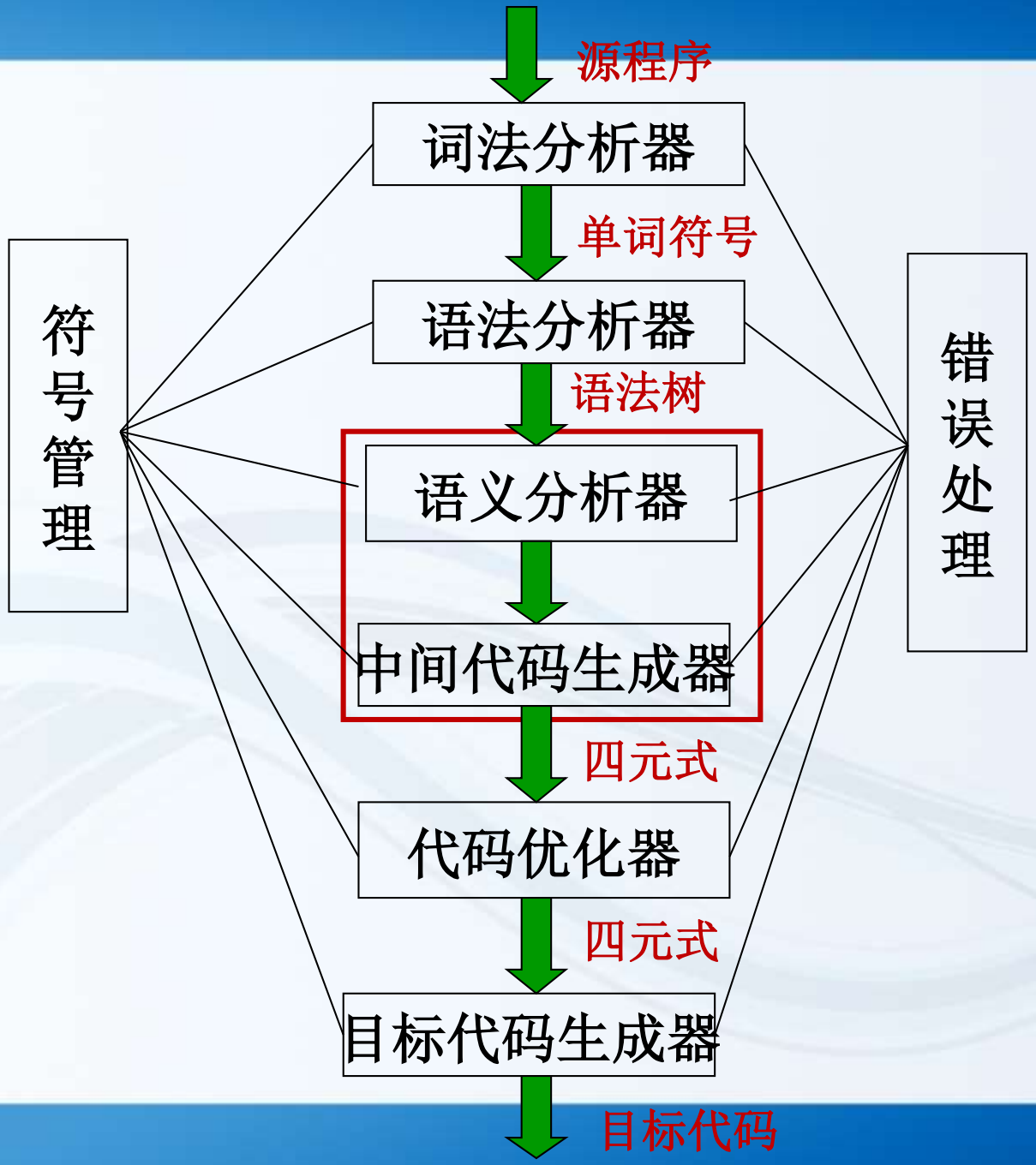
中间代码生成

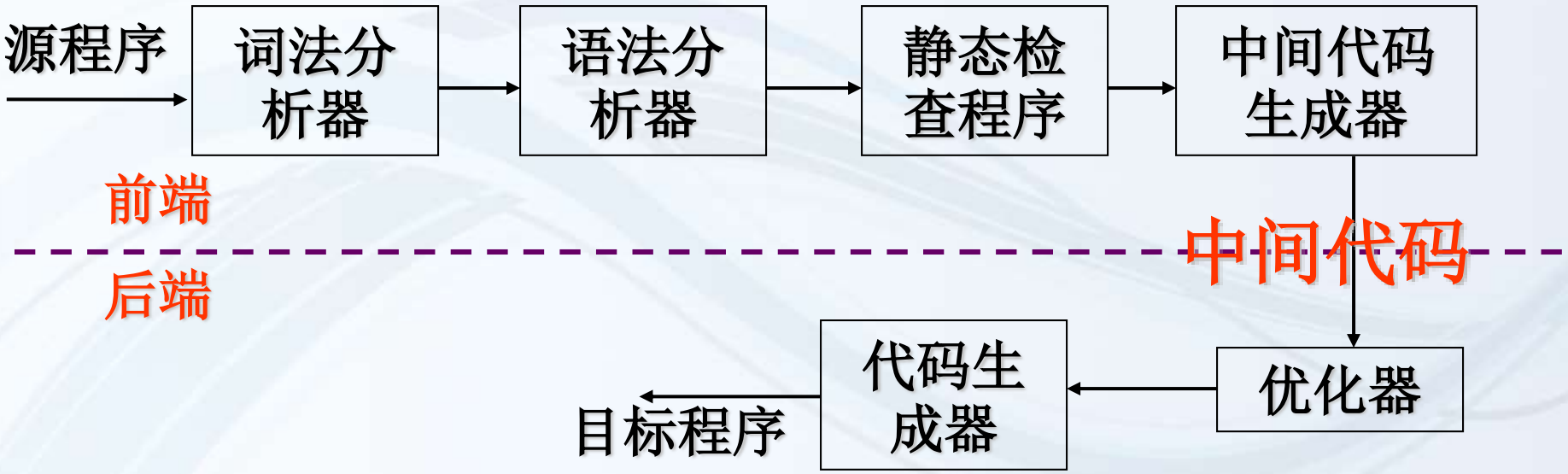
Intermediate Code Generation

编译程序总框


源程序
词法分析器
单词符号
语法分析器
语法树
语义分析器
中间代码生成器
四元式
代码优化器
四元式
目标代码生成器
目标代码
符号管理
错误处理

中间表示

- 编译器多数可以分成两部分：
  - 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



静态类型检查

✓类型检查

验证程序中执行的每个操作是否遵守语言的类型系统的过程

✓控制流检查

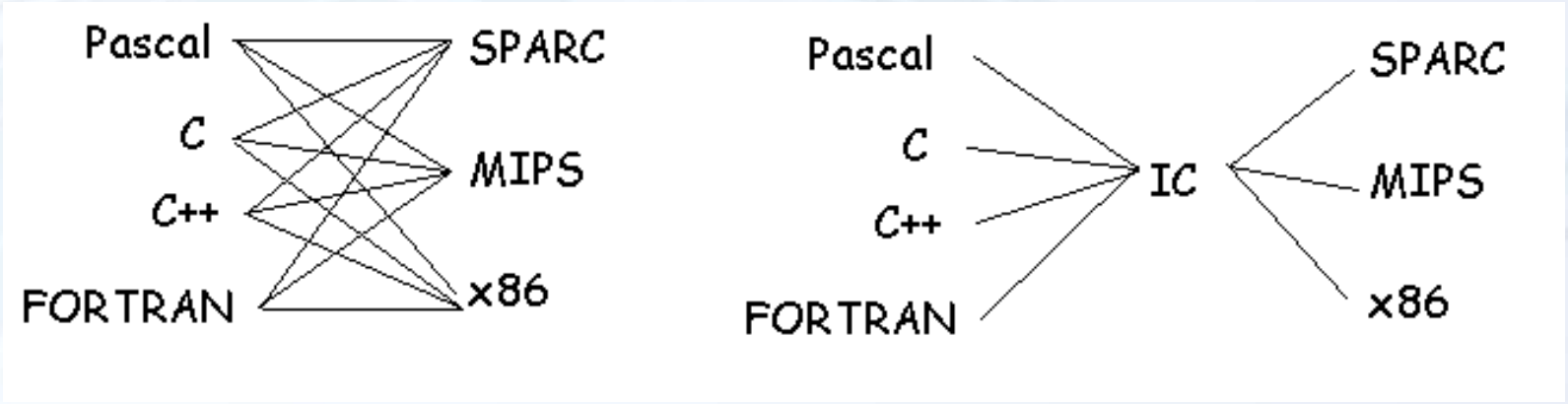
控制流语句必须使控制转移到合法的地方。例如：C语言中break语句使控制跳离循环或switch语句，如果不存在这样的语句，则报错

✓一致性检查

很多情况下对象只能被定义一次。如C语言中同一标识符在一个函数中只能被说明一次

✓名字的作用域分析

- 中间语言（表示）
  - ✓独立于机器
  - ✓复杂性介于源语言与目标语言之间
- 引入中间语言的优点
  - ✓便于进行与机器无关的代码优化工作
  - ✓易于移植
  - ✓使编译程序的结构在逻辑上更为简单明确



- 常用的中间表示：
  - 后缀式：逆波兰表示
  - 图表示：抽象语法树、DAG
  - 三地址码
    - ✓ 三元式
    - ✓四元式
    - ✓间接三元式

后缀式

- 后缀式表示法：Lukasiewicz发明的一种表示表达式的方法，又称逆波兰表示法。
- 一个表达式E的后缀形式可以如下定义
  - 如果E是一个变量或常量，则E的后缀式是E自身。
  - 如果E是E_1 op E_2形式的表达式，其中op是任何二元操作符，则E的后缀式为E_1' E_2' op，其中E_1'和E_2'分别为E_1和E_2的后缀式。
  - 如果E是(E_1)形式的表达式，则E_1的后缀式就是E的后缀式。

将表达式翻译成后缀式的语义规则

产生式	语义规则
$E \rightarrow E^{(1)} op\ E^{(2)}$	$E.code := E^{(1)}.code \ \|\| \ E^{(2)}.code \ \|\| op$
$E \rightarrow (E^{(1)})$	$E.code := E^{(1)}.code$
$E \rightarrow id$	$E.code := id$

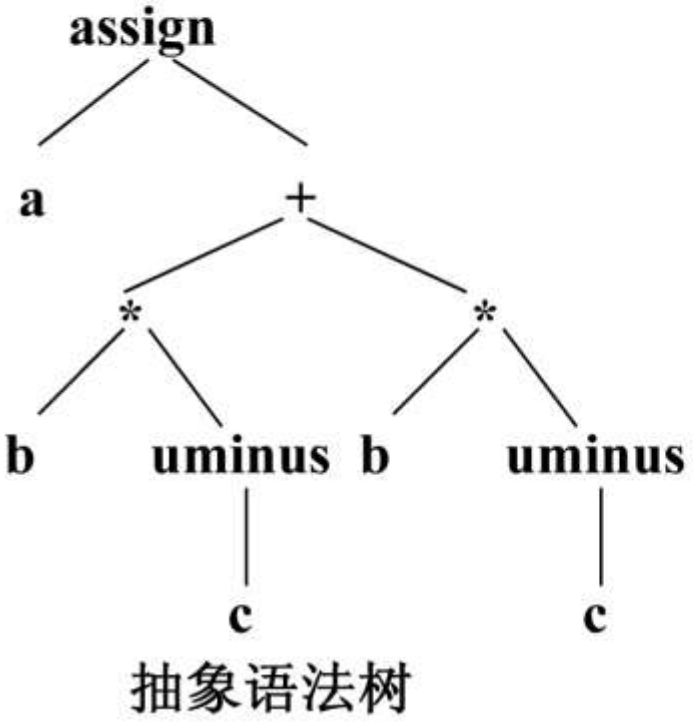
- E.code表示E后缀形式
- op表示任意二元操作符
- “||”表示后缀形式的连接

图形表示

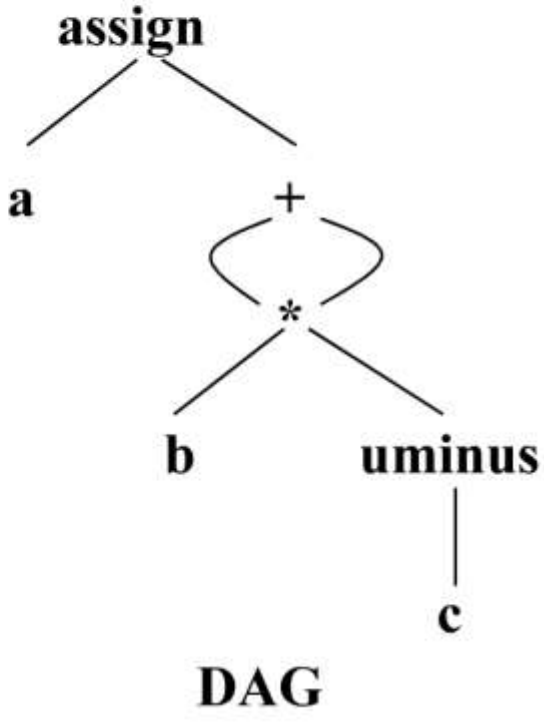
- 图形表示法: 语法树, 有向无环图
- 有向无环图（Directed Acyclic Graph， DAG）

  - 对表达式中的每个子表达式，DAG中都有一个结点
  - 一个内部结点代表一个操作符，它的孩子代表操作数
  - 在一个DAG中代表公共子表达式的结点具有多个父结点

a:=b*(-c)+b*(-c)的图表示法



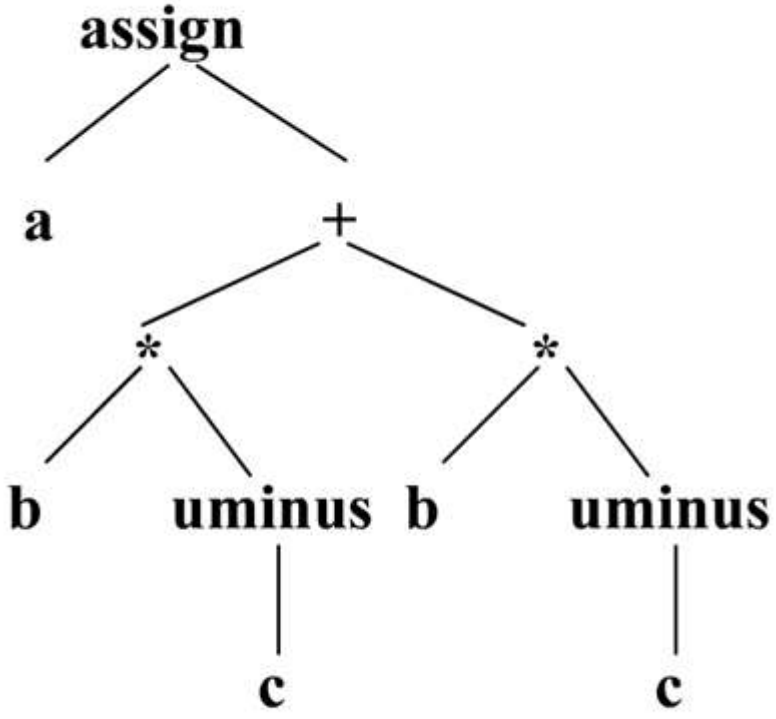
抽象语法树



DAG

后缀式是语法树的线性表示

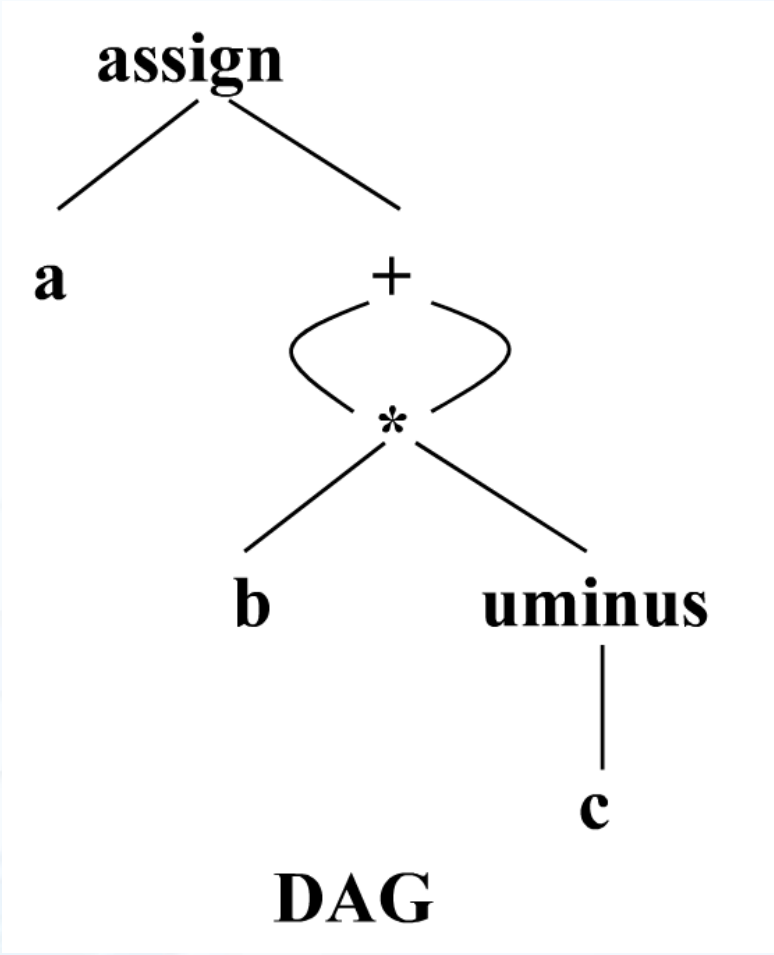
a b c uminus * b c uminus * + assign



抽象语法树

抽象语法树对应的代码：

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



DAG

DAG对应的代码：

T_1:=-c

T_2:=b*T_1

T_5:=T_2+T_2

a:=T_5

抽象语法树对应的代码：

T_1:=-c

T_2:=b*T_1

T_3:=-c

T_4:=b*T_3

T_5:=T_2+T_4

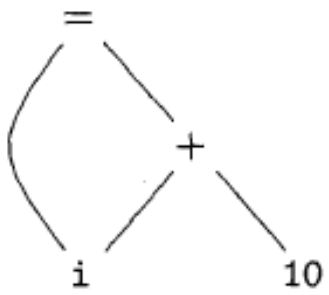
a:=T_5

建立抽象语法树的语义规则

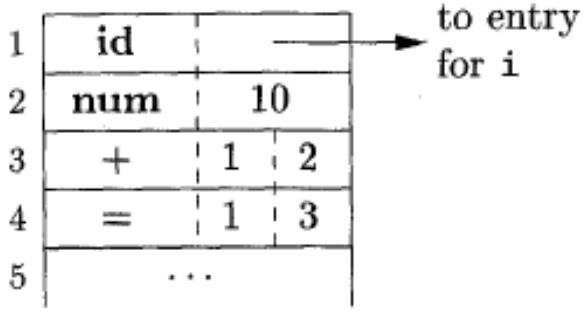
产生式	语义规则
$E \rightarrow E_1 + T$	E.*node* = **new** *node*(“+”,E_1.*node* ,T.*node*)
$E \rightarrow E_1 - T$	E.*node* = **new** *node*(“-”,E_1. *node* ,T. *node*)
$E \rightarrow T$	E.*node* = T.*node*
$T \rightarrow (E)$	T.*node* = E.*node*
$T \rightarrow id$	T.*node* = **new** *leaf*(id, id.*lexval*)
$T \rightarrow num$	T.*node* = **new** *leaf*(num, num.*val*)

构建DAG的值编码方法

- 语法树或DAG图的结点通常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
  - 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



(b) Array.

三地址码

- 三地址码（*TAC*，三地址代码）用来描述指令序列

- 三地址码可以看成是抽象语法树或DAG的一种线性表示
- 具体实现：四元式、三元式、间接三元式

a:=b*(-c)+b*(-c)的图表示法

DAG对应的三地址代码：

T_1:=-c

T_2:=b*T_1

T_5:=T_2+T_2

a:=T_5

抽象语法树对应的三地址代码：

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

常用三地址码(I)

- 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，表示为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过程，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: t_1 = i + 1
i = t_1
t_2 = i * 8
t_3 = a[t_2]
if t_3<v goto L

采用符号标号

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

采用位置号

三地址码的实现(四元式)

- 四元式是4个字段的记录结构：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

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

	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

$t_1 = - c$

$t_2 = b * t_1$

$t_3 = - c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

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

(1)

go			L

go	L		

if a<b goto L

(1)

<	a	b	L

J<	a	b	L

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

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

$t_1 = - c$

$t_2 = b * t_1$

$t_3 = - c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

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

- x[i]:=y

	op	arg1	arg2
(0)	[ ] =	x	i
(1)	assign	(0)	y

- x:=y[i]

	op	arg1	arg2
(0)	= [ ]	y	i
(1)	assign	x	(0)

三地址码的实现(间接三元式)

- 间接三元式
  - 三元式表+间接码表
  - 间接码表
    - ✓ 一张指示器表，按运算的先后次序列出有关三元式在三元式表中的位置

间接三元式示例

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

	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)

- 例如，语句

X:=(A+B)*C;
Y:=D↑(A+B)

的间接三元式表示如下表所示

间接代码
(1)
(2)
(3)
(1)
(4)
(5)

三元式表

	OP	ARG1	ARG2
(1)	+	A	B
(2)	*	(1)	C
(3)	:=	X	(2)
(4)	↑	D	(1)
(5)	:=	Y	(4)

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)

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
- 翻译时的应用
  - 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

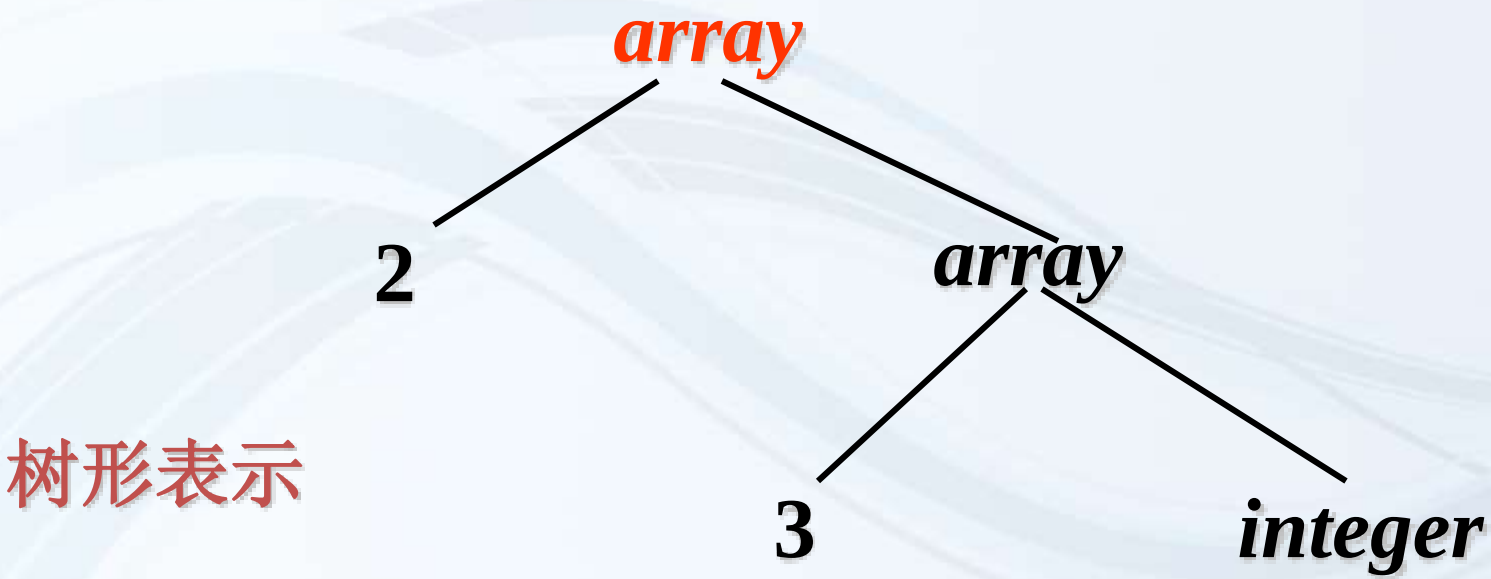
- 描述类型的结构
- 类型表达式: 类型可以是基本类型，也可以是类型构造符（类型构造算子）作用于类型而得
  - 基本类型: ***boolean, char, integer, float, void***
  - 数组类型构造符***array*(I,T)**：其中T是类型表达式，I是整数。如**int A[10]** 的类型表达式是 ***array*(10,*integer*)**
    - 均有对应的树形表示

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))



树形表示

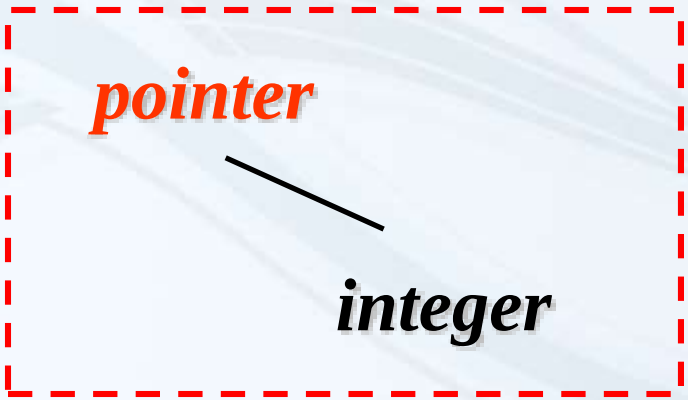
类型表达式（III）

- 指针int* aa;

  var aa:↑integer;

  pointer (*integer*)

树形表示



类型表达式（IV）

函数float divide(int i, int j)

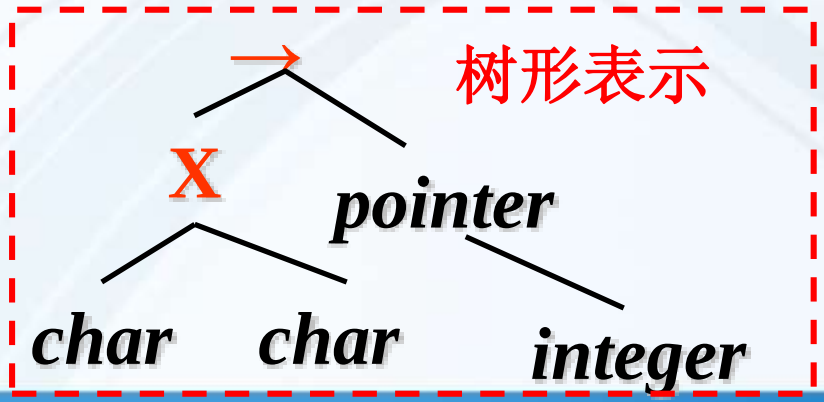
FUNCTION divide(i,j:integer):real;

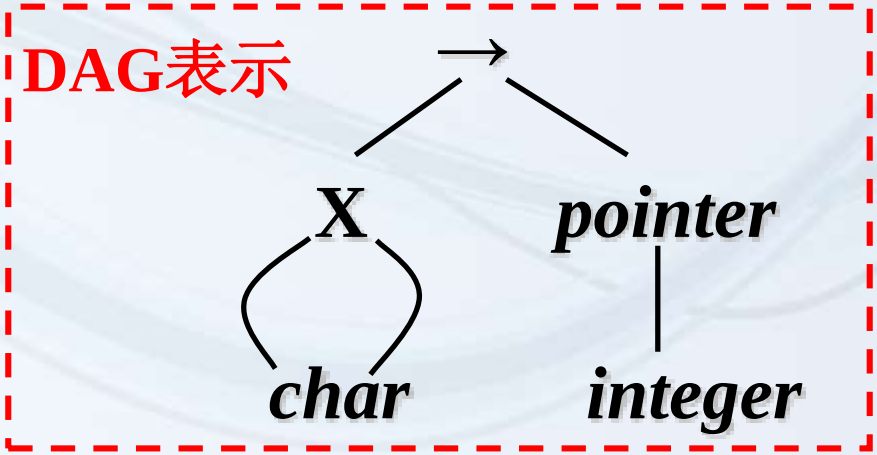
integer X integer → float

类型表达式的树形表示 int *f(char a, char b);

FUNCTION f(a,b:char): ↑ integer;

char X char → pointer(integer)





类型表达式(V)

```
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;
```

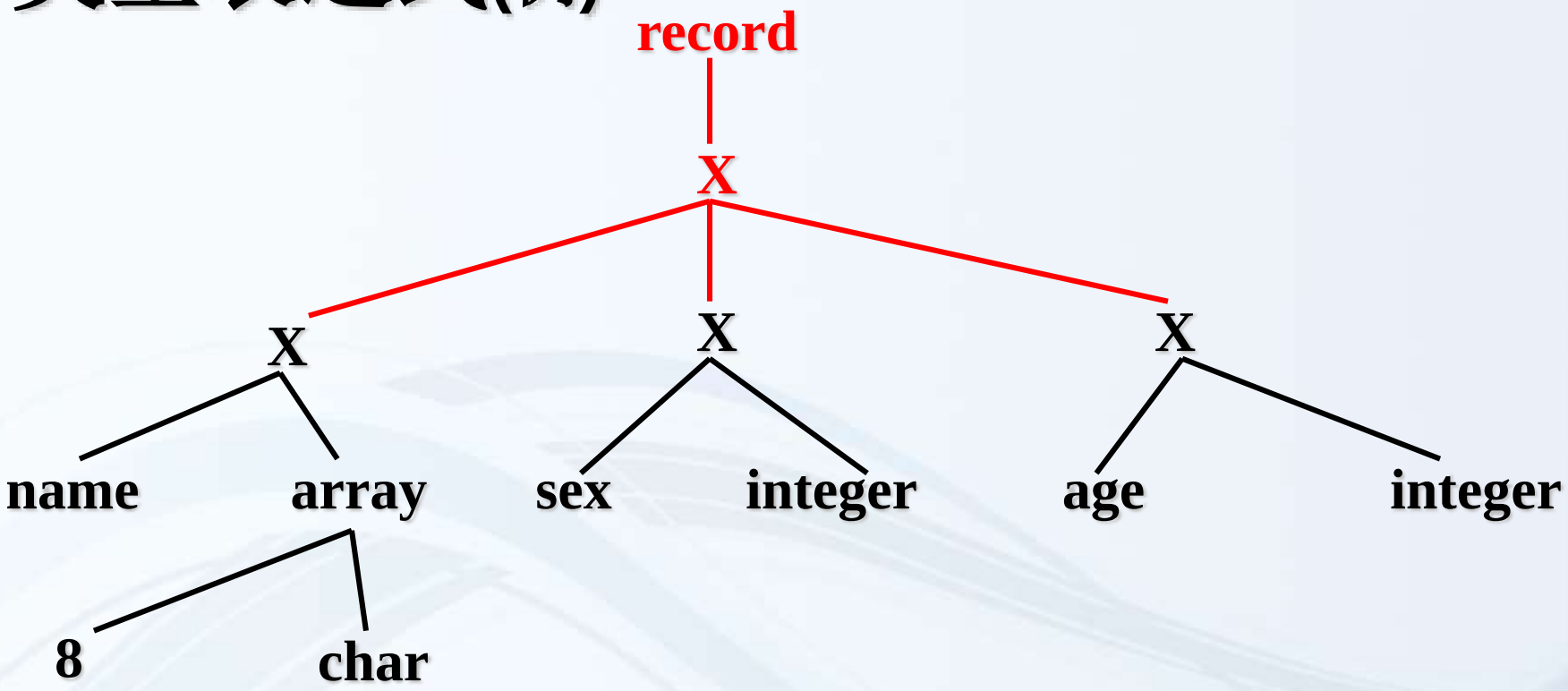
则person之类型表达式：

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

table之类型表达式：

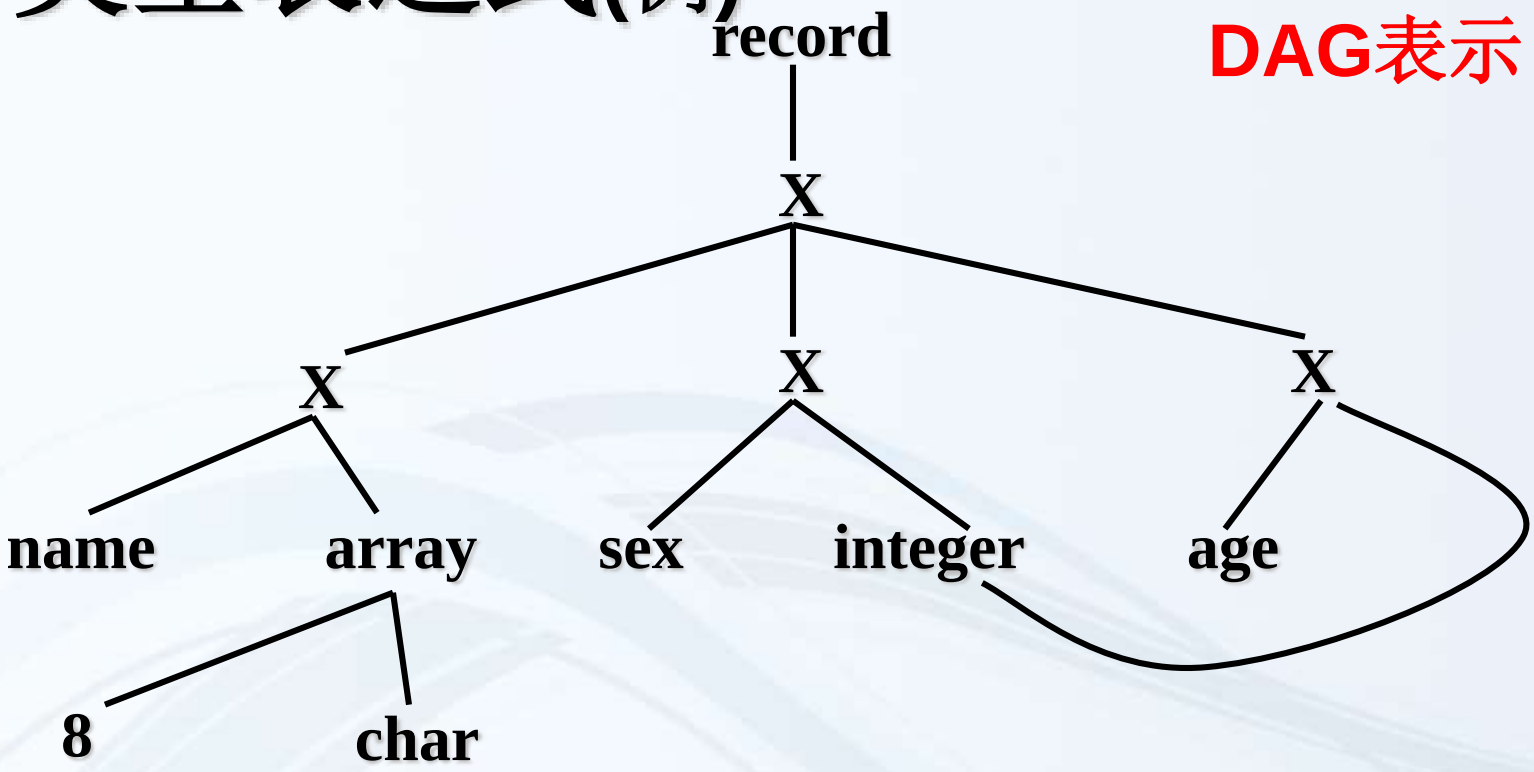
array(50,person)

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)

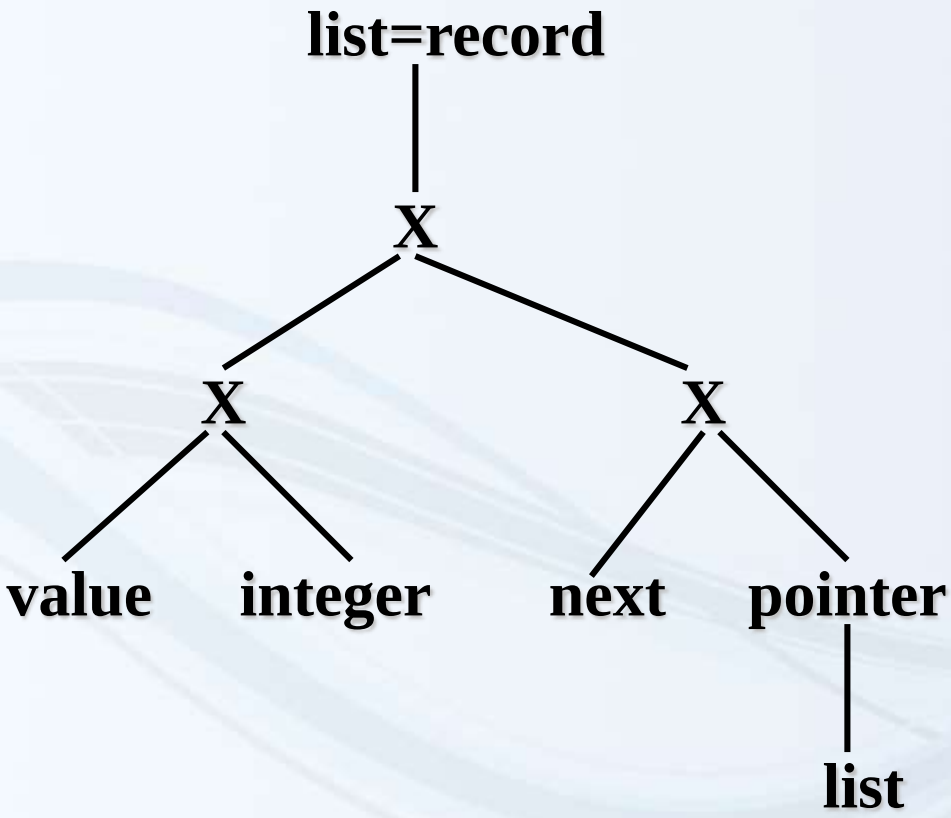


DAG表示

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

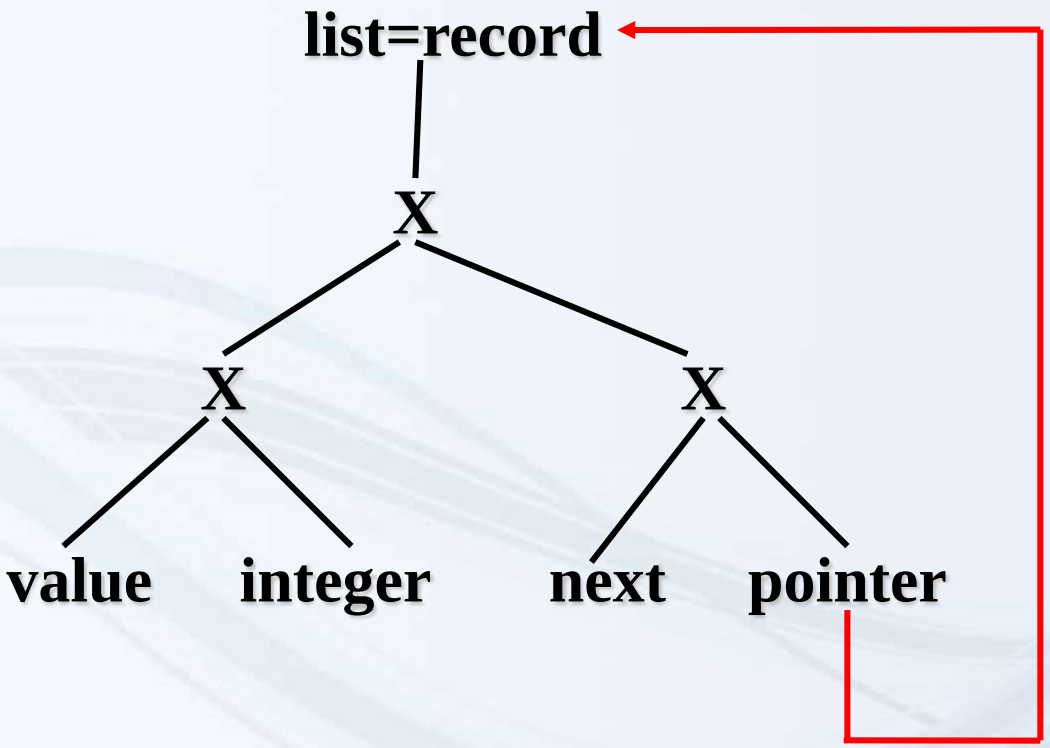
类型表达式(例)

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



类型表达式(例)

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



类型等价

- 两种等价：结构等价和名等价

➢ 结构等价：满足以下条件之一：

（1）相同的基本类型

（2）将相同类型构造算子应用于等价的类型而构建的。

（3）一个类型是另一个类型表达式的名字

➢ 名等价：满足前两个条件。

➢ 例：

```
type link =↑cell;
var p,q : link;
var r,s : ↑cell
```

p,q,r,s结构等价

p,q名等价

r,s名等价

例：

```
typedef struct{
        int age;
        char name[20];
}recA;
typedef    recA* recp;
typedef    recA* recD;
recp a,b;
recD c,d;
recA *e;
```

变量	类型表达式
a	recp
b	recp
c	recD
d	recD
e	pointer(recA)

五变量结构等价
a和b，c和d名等价

声明 (I)

- 为每个声明的名，在符号表中加入：类型、位移等等信息。
- 位移指出相对地址：*offset*
  - 全局数据的位移是指在静态数据区的位置
  - 局部数据的位移是指在局部过程的活动记录的局部数据区的位置
- 类型附加属性： *type*和*width*

T.type=float

T.width=8

声明 (II)

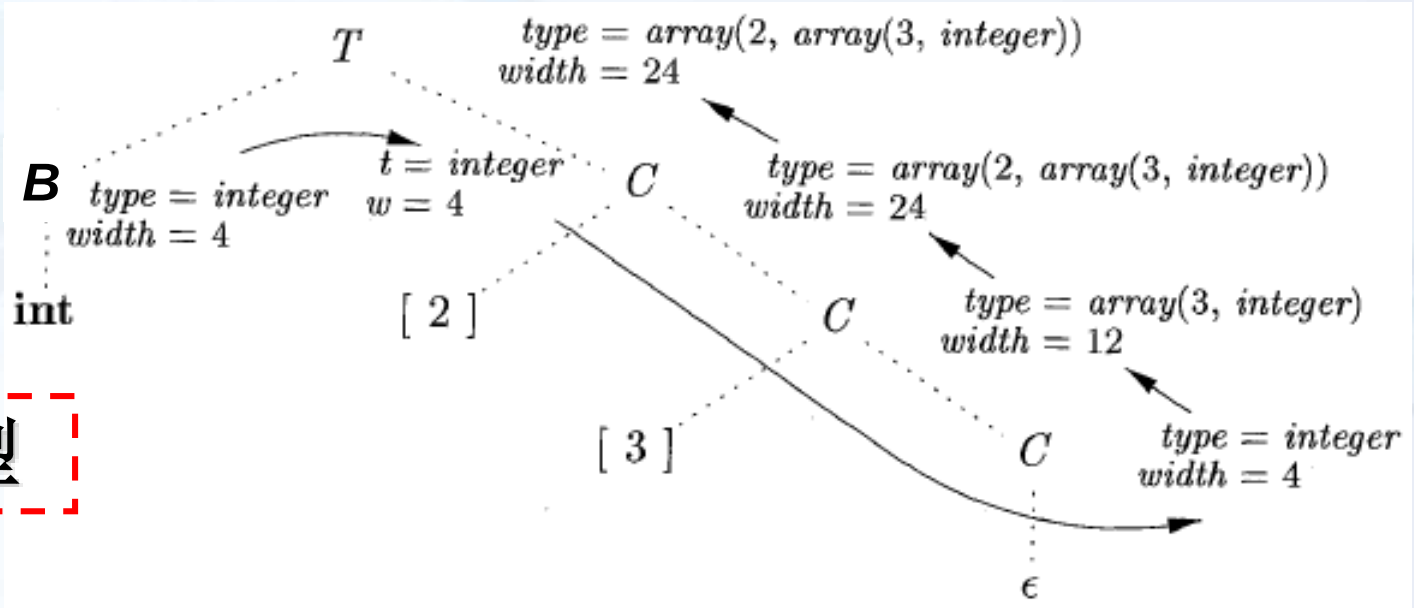
T → B {t=B.type; w=B.width;} C {T.type=C.type; T.width=C.width;}

B → int {B.type=integer;B.width=4;}

B → float {B.type=float; B.width=8;}

C→ε {C.type=t; C.width=w;}

C →[num]C_1 {C.type=array(num.value, C_1.type); C.width=num.value*C_1.width;}



先计算类型

声明 (III)

P→{offset=0;}D

D →T id ; {top.put(id.lexeme, T.type, offset); offset=offset+T.width;} D_1

D → ε

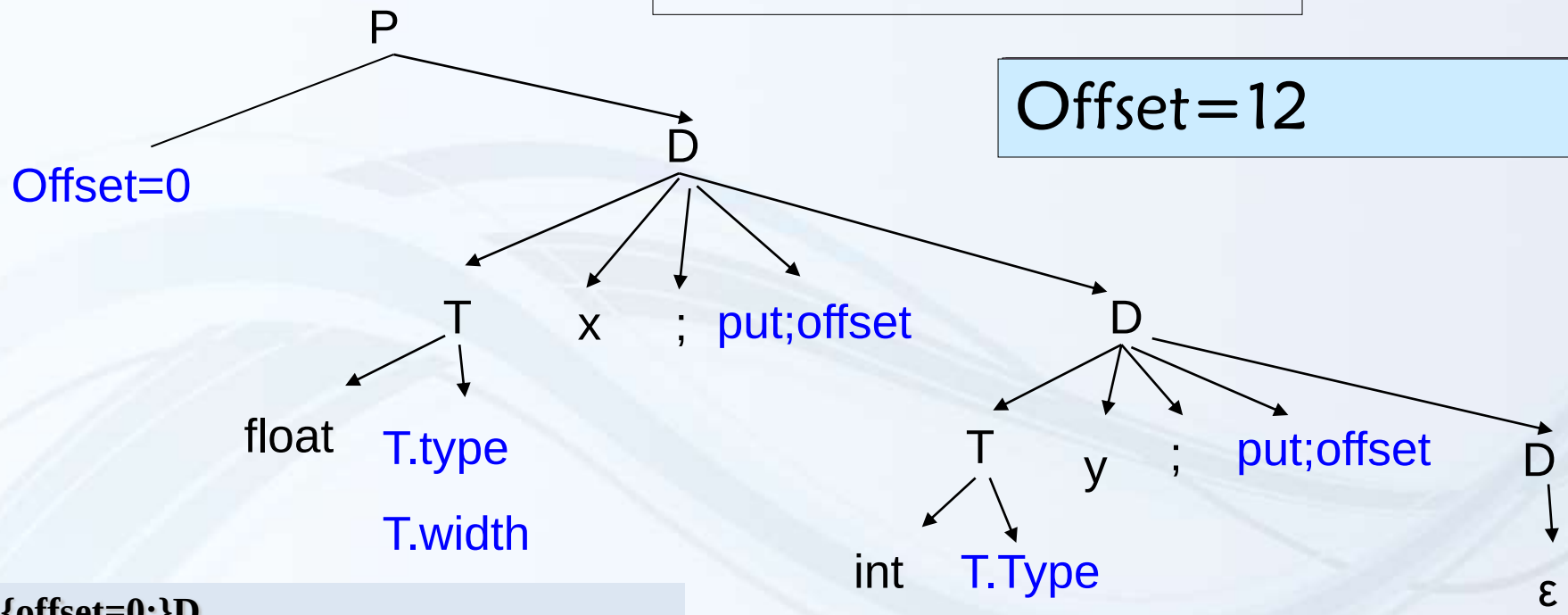
```
float x;
int y;
```

float x; int y;

T.type=int

T.width=4

Name	type	addr
X	float	0
Y	int	8
	.....	



Offset=12

P→{offset=0;}D

D →T id ; {top.put(id.lexeme, T.type, offset); offset=offset+T.width;} D_1

D → ε