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
  （3）一个类型是另一个类型表达式的名字
- 名等价：满足前两个条件之一。

例：说明p,q,r,s四个变量间存在的类型等价。

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

```
typedef  cell*  link;
link  p, q;
cell*  r,s;
```

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

一个简单的变量声明文法

$$
\begin{array}{lcl}
D & \rightarrow & T\ \mathbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \mathbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \mathbf{int} \mid \mathbf{float} \\
C & \rightarrow & \epsilon \mid [\ \mathbf{num}\ ]\ C
\end{array}
$$

声明 (II)

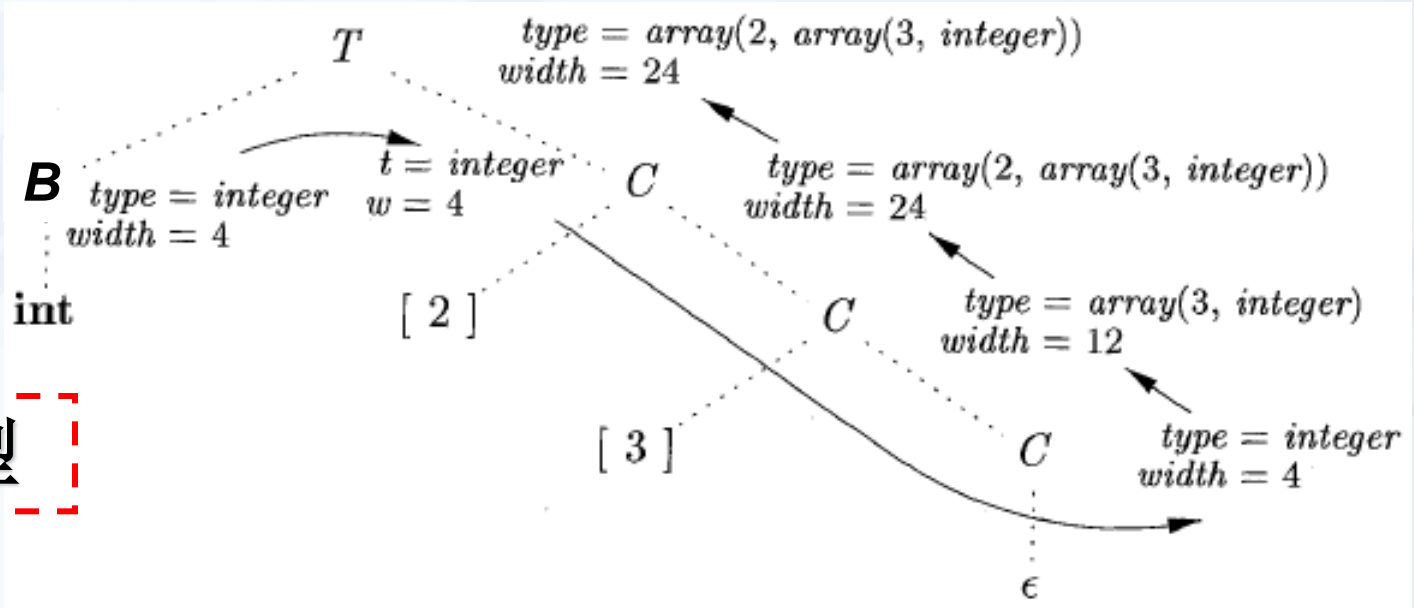
T → B {t=B.type; w=B.width;} C {T.type=C.type; T.width=C.width;}

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

记录类型见图6-18

声明 (III)

$P \rightarrow \{offset=0;\}D$

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

$D \rightarrow \varepsilon$

top是什么？

```
float x;
record { float x; float y;} p;
record {int tag; float x; float y;} q;
```

name	type	width	offset
x	float	8	0
p	record((x X float) X (y X float))	16	8
p.x	float	8	8
p.y	float	8	16
q	record((tag X integer) X (x X float) X (y X float))	20	24
q.tag	integer	4	24
q.x	float	8	28
q.y	float	8	36

6.4表达式的翻译

a = b + -c;

t_1=minus c
t_2=b+t_1
a = t_2

产生式	语义规则
$S \rightarrow id = E ;$	S.code=E.code || gen(top.get(id.lexeme) '=' E.addr)
$E \rightarrow E_1 + E_2$	E.addr = new Temp( ) E.code=E_1.code||E_2.code||gen(E.addr '=' E_1.addr '+' E_2.addr)
$E \rightarrow - E_1$	E.addr = new Temp( ) E.code = E_1.code || gen(E.addr '=' 'minus' E_1.addr)
$E \rightarrow (E_1)$	E.addr = E_1.addr E.code = E_1.code
$E \rightarrow id$	E.addr = top.get(id.lexeme) E.code = ' '
$E \rightarrow num$	E.addr = top.get(num.lexeme) E.code = ' '

为什么此处不需要new Temp( )?

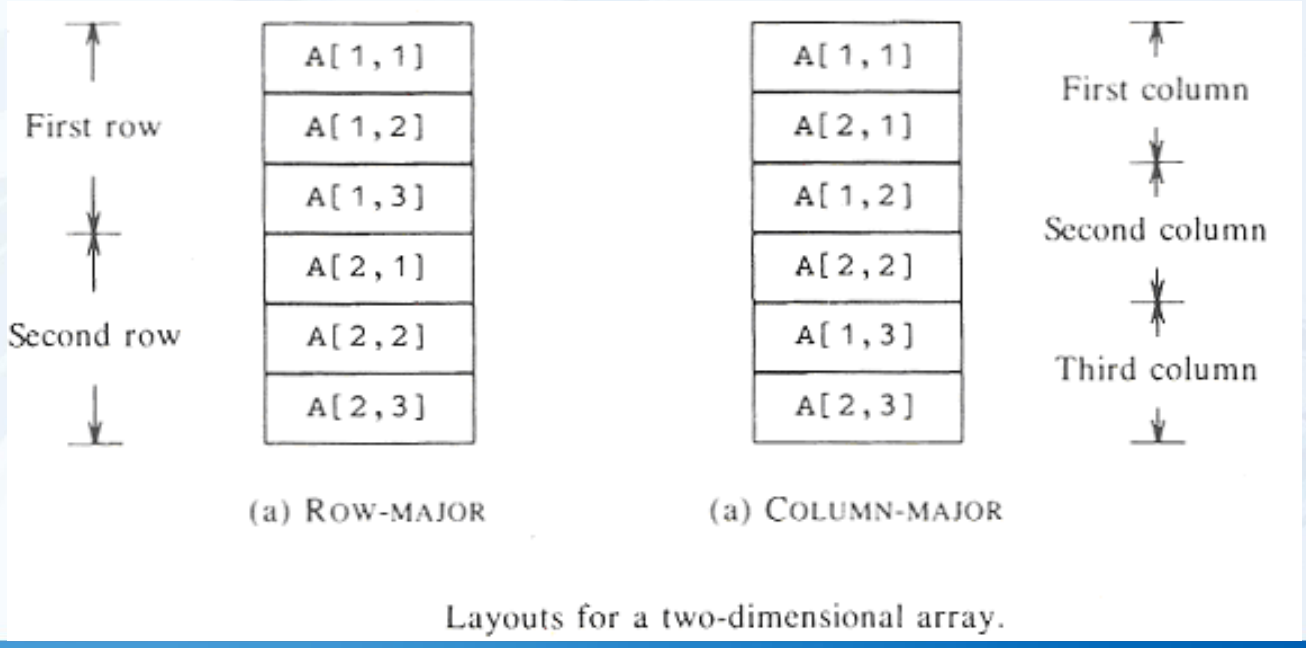
数字有没有加入符号表?

表达式的翻译(增量翻译)

产生式	语义规则
$S \rightarrow id = E\ ;$	gen(top.get(id.lexeme) '=' E.addr)
$E \rightarrow E_1 + E_2$	E.addr = new Temp( ) gen(E.addr '=' E_1.addr '+' E_2.addr)
$E \rightarrow - E_1$	E.addr = new Temp( ) gen(E.addr '=' 'minus' E_1.addr)
$E \rightarrow (E_1)$	E.addr = E_1.addr
$E \rightarrow id$	E.addr = top.get(id.lexeme)

数组元素的寻址

- 数组存储在一块连续存储空间中
- 一维数组A[*low*…*high*], 若元素宽度为*w*, 首地址为*base*, 则A中第*i*个元素开始于：$base + (i\text{-}low) \times w = i \times w + (base - low \times w) = i \times w + c$
- 二维数组可采用：*row-major* 行优先或*column-major*列优先方式存储



Layouts for a two-dimensional array.

数组元素的寻址(II)

- 在行优先存储时，$A[i_1, i_2]$ 存于

$$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w =$$

$$((i_1 \times n_2) + i_2) \times w + \underbrace{(base - ((low_1 \times n_2) + low_2) \times w)}_{\text{can be determined at compile time}}$$

- k维行优先存储的数组A，元素 $A[i_1, i_2, .., i_k]$ 存于：

$$((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3)\ldots) \times n_k + i_k) \times w +$$

$$base - ((\ldots((low_1 \times n_2 + low_2) \times n_3 + low_3)\ldots) \times n_k + low_k) \times w$$

$$base' + i_1 * w_1 + i_2 * w_2 + i_3 * w_3 + \ldots + i_k * w_k$$

$w_1 = n_2 * n_3 * \ldots * n_k * w$
$w_2 = n_3 * \ldots * n_k * w$
$\ldots$
$w_{k-1} = n_k * w$
$w_k = w$

数组引用的翻译(I)

产生式	语义规则
$L \rightarrow id[\ E\ ]$	L.array = top.get(id.lexeme); L.type = L.array.type.elem; L.addr = new Temp( ); gen(L.addr '=' E.addr '*' L.type.width);

$\mid L_1[E]$	L.array = L_1.array;
	L.type = L_1.type.elem;
	t = new Temp( ); L.addr = new Temp( );
	gen(t '=' E.addr '*' L.type.width);
	gen(L.addr '=' L_1.addr '+' t);

L.addr 指示一个临时变量，用于累加i_j * w_j *(pp245. 公式6.4)*

L.array 是指向数组名字对应条目的指针，该数组的基地址为*L.array.base*.

L.type 是L生成的子数组的类型

对于任何类型*t*, 假定*t.width*给出它的宽度，*t.elem*给出其元素的类型。

数组引用的翻译(II)

产生式	语义规则
$S \rightarrow id = E;$	gen(top.get(id.lexeme) '=' E.addr);
$S \rightarrow L = E;$	gen(L.array.base '[' L.addr ']' '=' E.addr);
$E \rightarrow E_1 + E_2$	E.addr = new Temp( ); gen(E.addr '=' E_1.addr '+' E_2.addr);
$E \rightarrow id$	E.addr = top.get(id.lexeme);
$E \rightarrow L$	E.addr = new Temp( ); gen(E.addr '=' L.array.base '[' L.addr ']');

Figure 6.22 continued

数组引用的翻译(III)

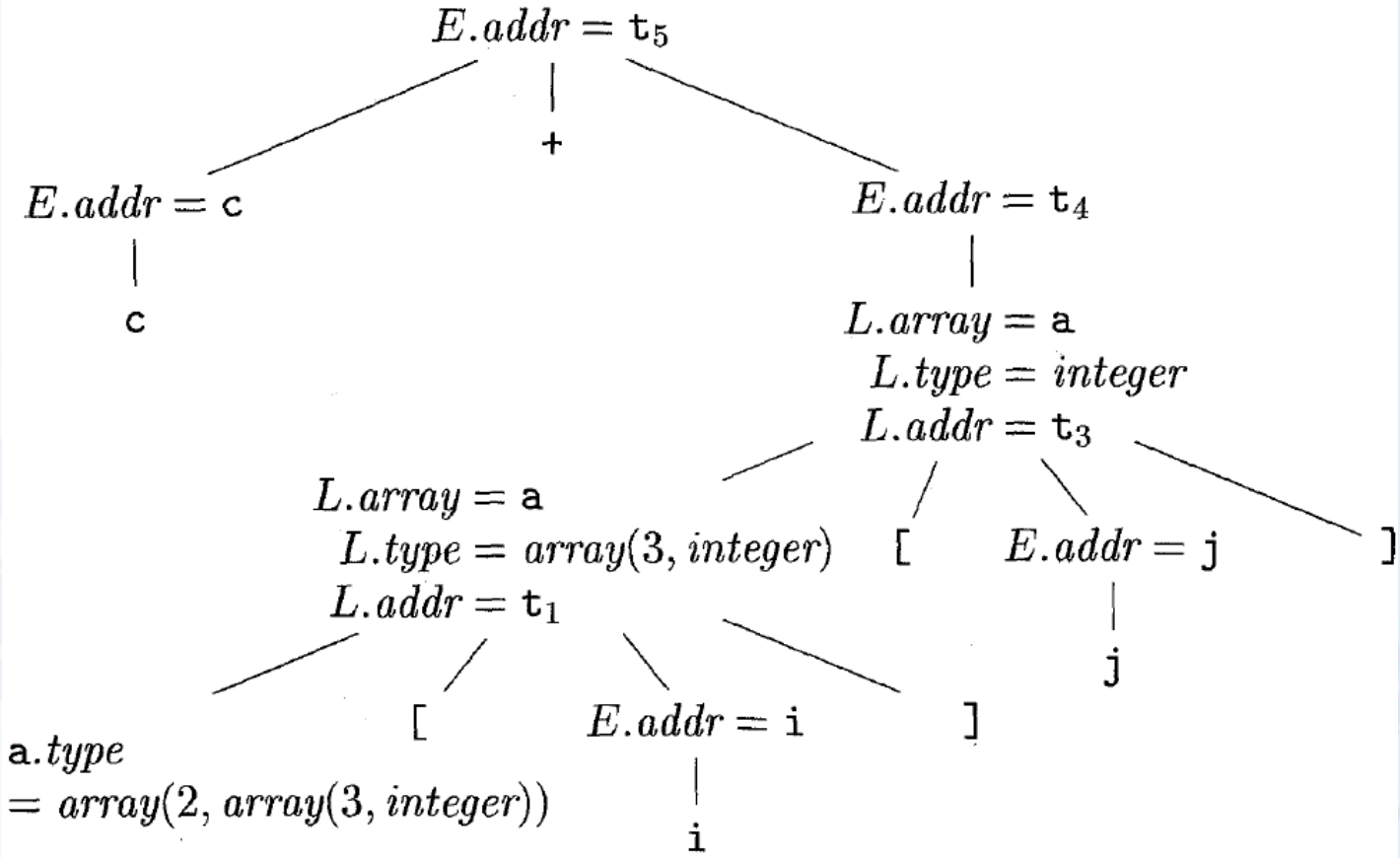
$t_1 = i * 12$

$t_2 = j * 4$

$t_3 = t_1 + t_2$

$t_4 = a[t_3]$

$t_5 = c + t_4$



令a 表示 2 x 3 整数数组, c, i, j 都是宽度为4的整数

以上给出了 c + a [i][j]的注释分析树

数组引用的翻译(IV)

假设float a[15][20], b[19][17];

翻译a[i][j]=3.5*4.7+b[k][m];

t1=i*160

t2=j*8

t3=t1+t2

t4=3.5*4.7

t5=k*136

t6=m*8

t7=t5+t6

t8=b[t7]

t9=t4+t8

a[t3]=t9

6.5类型转换

- 编译器会：
  - 拒绝一些混合类型运算
  - 进行一些自动的类型转换。
- 例: 一个含有两种类型:*integer*和 *float*的文法, *integer*在必要时，应转为 *float*

$E \rightarrow E_1 + E_2$

{if(E_1.type==integer and E_2.type==integer)

E.type = integer

else

if (E_1.type==float and E_2.type==integer)…}

2*3.14

t_1 = (float) 2

$t_2 = t_1 * 3.14$

类型转换(II)

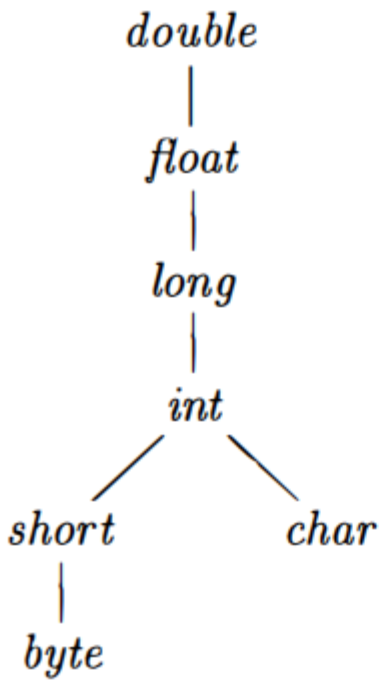
- 拓宽转换和窄化转换：
  - 拓宽：较低层的类型转换为较高层的
  - 窄化：较高层到较低层。
- 隐式转换和显式转换
  - 隐式：由编译器自动完成
  - 显式（强制类型转换）：程序员写出代码

$E \rightarrow E_1 + E_2 \{ E.type = max(E_1.type, E_2.type);$

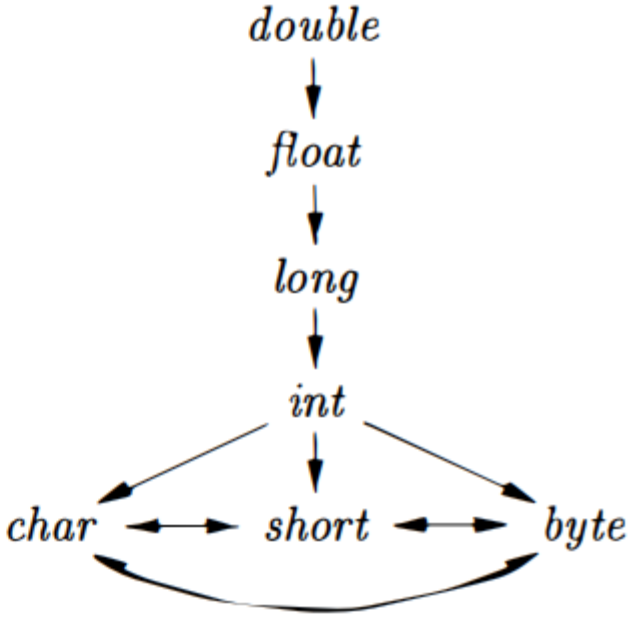
$a_1 = widen(E_1.addr, E_1.type, E.type);$

$a_2 = widen(E_2.addr, E_2.type, E.type);$

$E.addr = new\ Temp(); \ gen(E.addr\ '='\ a_1\ '+'\ a_2;\}$



(a) Widening conversions



(b) Narrowing conversions

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

类型推导

if f 可能的类型为$s_i \to t_i\ (1 \le i \le n)$，其中 $s_i \ne s_j\ (i \ne j)$

and x 的类型为$s_k\ (1 \le k \le n)$

then 表达式$f(x)$的类型为t_k