中国科学技术大学
University of Science and Technology of China

USTC, CHINA
ADSLAB
先进数据系统实验室

《编译原理与技术》
中间代码生成 II
（与类型相关部分）

计算机科学与技术学院
李 诚
18/11/2019

通知

- 本周日上午10:00-11:30，东校区高性能计算中心503期中考试查卷

与类型相关的中间代码生成

中国科学技术大学
University of Science and Technology of China

- 符号表的组织
- 声明语句的翻译
- 数组寻址的翻译
- 类型转换

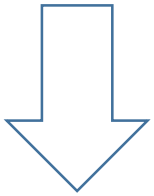
符号表 (Symbol table)

- 符号表的使用和修改伴随编译的全过程
- 存储entity的各种信息
  - 如variable names, function names, objects, classes, interfaces 等
  - 如类型信息、所占用内存空间、作用域
- 用于编译过程中的分析与合成
  - 语义分析：如使用前声明检查、类型检查、确定作用域等
  - 合成：如类型表达式构造、内存空间分配等

符号表 (Symbol table)

代码片段：

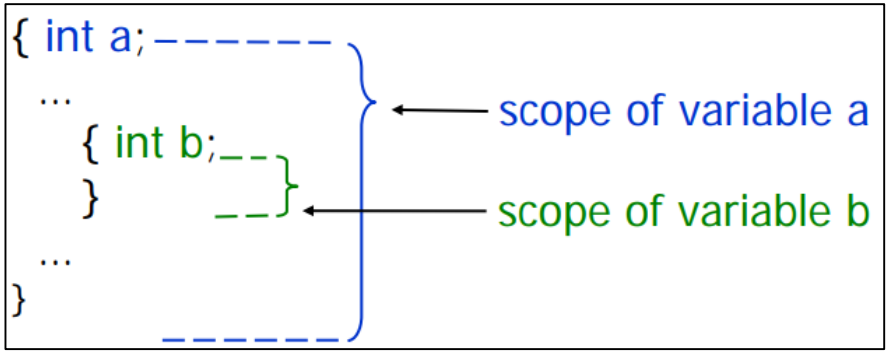
```
extern bool foo(auto int m, const int n);
const bool tmp;
```



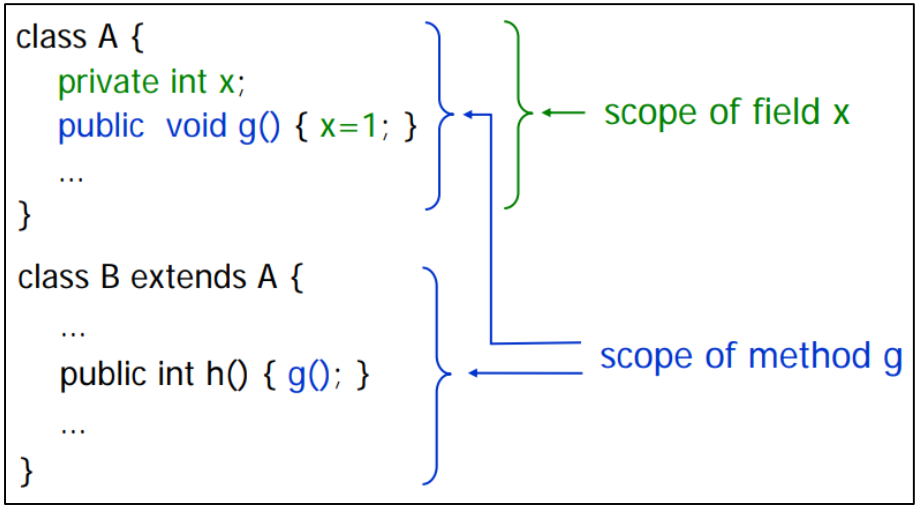
NAME	KIND	TYPE	OTHER
foo	fun	int x int → bool	extern
m	par	int	auto
n	par	int	const
tmp	var	bool	const

符号表

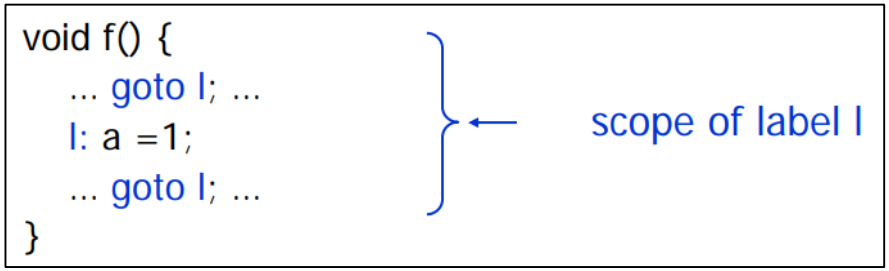
符号表——作用域



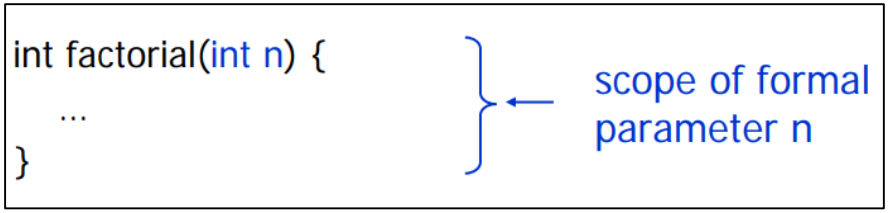
程序块中



对象中的field和methods



语句标号



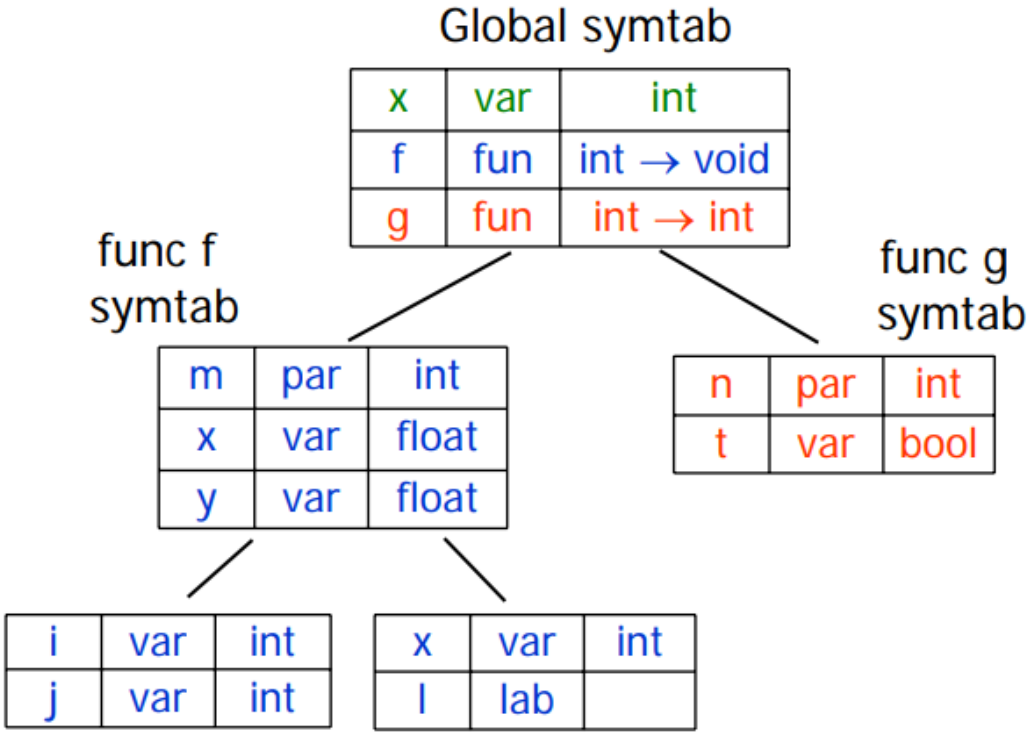
过程或函数定义中的参数

符号表 (Symbol table)


```
int x;

void f(int m) {
    float x, y;
    …
    { int i, j; …; }
    { int x; l: …; }
}

int g(int n) {
    bool t;
    …;
}
```



注：l代表label

符号表 (Symbol table)

- It is built in lexical and syntax analysis phases, used by compiler to achieve compile time efficiency.
  - **Lexical Analysis:** Creates new table entries in the table, example like entries about token.
  - **Syntax Analysis:** Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.
  - **Semantic Analysis:** Uses table info to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.

符号表 (Symbol table)

- It is built in lexical and syntax analysis phases, used by compiler to achieve compile time efficiency.
  - **Intermediate Code generation:** Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.
  - **Code Optimization:** For machine dependent optimization.
  - **Target Code generation:** Generates code by using address information of identifier present in the table.

符号表的实现

❑具体的操作

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

❑数据结构

- ❖数组、链表、hash表等

与类型相关的中间代码生成

中国科学技术大学
University of Science and Technology of China

- 符号表的组织
- 声明语句的翻译
- 数组寻址的翻译
- 类型转换

声明语句翻译的要点

- 分配存储单元
  - 名字、类型、字宽、偏移
- 作用域的管理
  - 过程调用
- 记录类型的管理
- 不产生中间代码指令，但是要更新符号表

声明语句的翻译

- 例：文法G_1如下：

$P \rightarrow D ; S$

$D \rightarrow D ; D$

$D \rightarrow id : T$

$T \rightarrow integer \mid real \mid array [num] of T_1 \mid \uparrow T_1$

声明语句的翻译

❑ 有关符号的属性

T.type - 变量所具有的类型，如

整型 INT

实型 REAL

数组类型 array（元素个数，元素类型）

指针类型 pointer（所指对象类型）

T.width - 该类型数据所占的字节数

offset - 变量的存储偏移地址

声明语句的翻译

T.type		T.width
整型	INT	4
实型	REAL	8
数组	array（num, T_1）	num.val * T_1.width
指针	pointer（T_1）	4

enter(name, type, offset)—将类型type和偏移offset填入符号表中name所在的表项。

声明语句的翻译

计算被声明名字的类型和相对地址

P→ {*offset = 0*} D ; S

相对地址初始化为0

D→ D ; D

D→ id : T {*enter(* id.*lexeme, T.type, offset) ;*

offset = offset + T.width }

更新符号表信息

T→integer {*T.type = integer; T.width = 4*}

T→real {*T.type = real; T.width = 8*}

类型=>字宽

T→array [number] of T_1

{*T.type = array(num.val,* T_1*.type);*

*T.width = num.val * *T_1*.width*}

T→ ↑ T_1 {*T.type = pointer(*T_1*.type); T.width = 4*}

声明语句翻译的要点

- 分配存储单元
  - 名字、类型、宽、偏移
- 作用域的管理
  - 过程调用
- 记录类型的管理
- 不产生中间代码指令，但是要更新符号表

允许自定义过程时的翻译

- 所讨论语言的文法

$P \rightarrow D; S$

$D \rightarrow D\ ;\ D \mid \text{id} : T \mid$

proc id ; D ; S

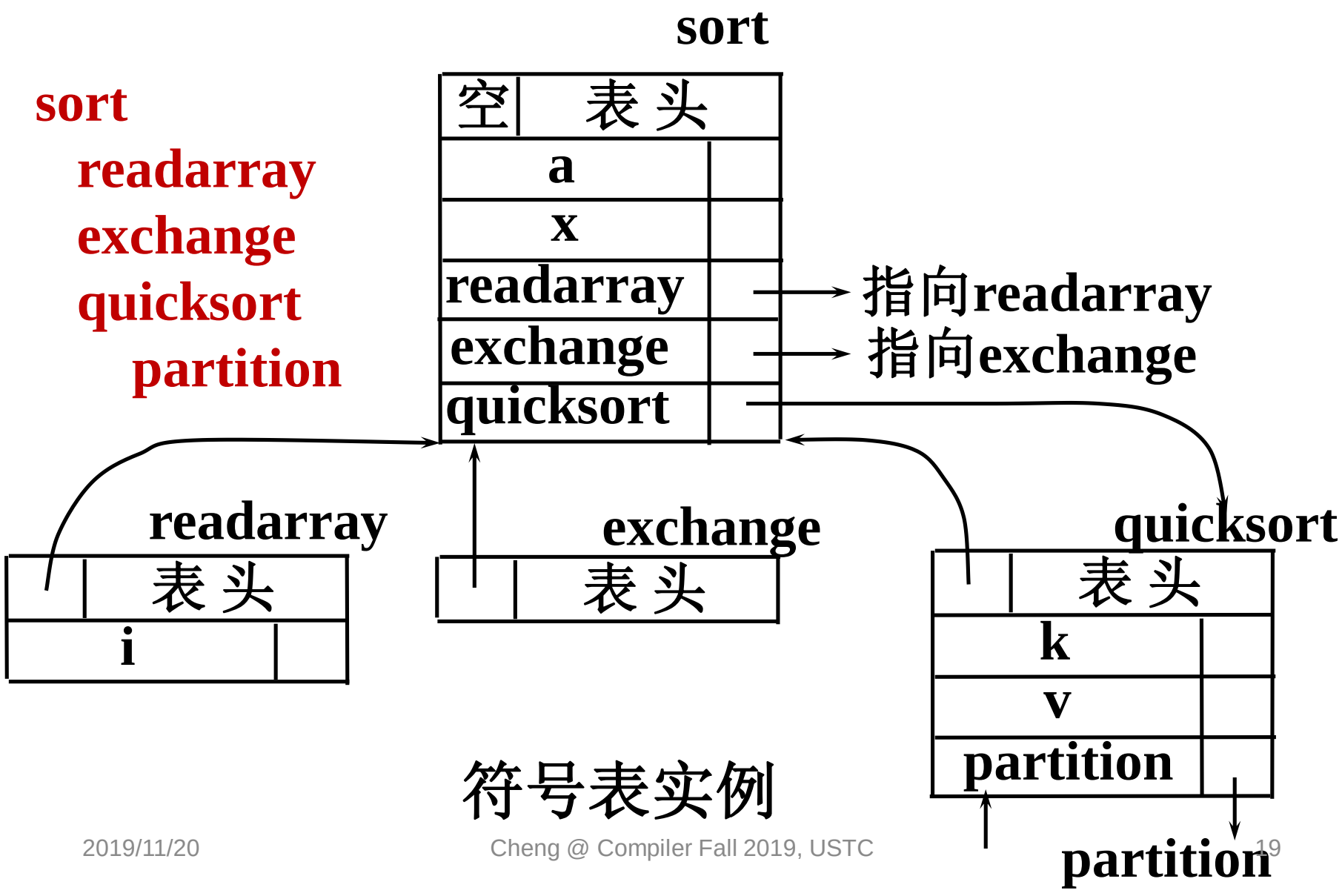
- 管理作用域(过程嵌套声明)
  - 每个过程内声明的符号要置于该过程的符号表中
  - 方便地找到子过程和父过程对应的符号

```
sort
    var a:…; x:…;
    readarray
        var i:…;
    exchange
    quicksort
        var k, v:…;
        partition
            var i, j:…;
```

教科书186页图6.14

过程参数被略去

各过程的符号表



符号表实例

❑符号表的特点及数据结构

- ❖各过程有各自的符号表：哈希表
- ❖符号表之间有双向链
  - ➢父→子：过程中包含哪些子过程定义
  - ➢子→父：分析完子过程后继续分析父过程
- ❖维护符号表栈(*tblptr*)和地址偏移量栈(*offset*)
  - ➢保存尚未完成的过程的符号表指针和相对地址

❑语义动作用到的函数

/* 建立新的符号表，其表头指针指向父过程符号表*/

1. mkTable(parent-table)

/* 将所声明变量的类型、偏移填入当前符号表*/

2. enter(current-table, name, type, current-offset)

/* 在父过程符号表中建立子过程名的条目*/

3. enterProc(parent-table, sub-proc-name, sub-table)

/*在符号表首部添加变量累加宽度，可利用符号表栈tblptr和偏移栈offset（栈顶值分别表示当前分析的过程的符号表及可用变量偏移位置）*/

4. addWidth(table, width)

声明语句的处理

$P \rightarrow M\ D;\ S$

$M \rightarrow \varepsilon$

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow \text{proc id}\ ;\ N\ D_1;\ S$

$D \rightarrow \text{id}\ :T$

$N \rightarrow \varepsilon$

声明语句的处理

$P \rightarrow M\ D;\ S$

tblptr：符号表栈
offset：偏移量栈

$M \rightarrow \varepsilon$ {*t = mkTable (nil);*
push(t, tblptr); push (0, offset) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1;\ S$

$D \rightarrow$ id : T

$N \rightarrow \varepsilon$

建立主程序（最外围）的符号表偏移从0开始

声明语句的处理

$P \rightarrow M\ D;\ S$

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblptr*); *push* (0, *offset*) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1;\ S$

$D \rightarrow$ id : T {*enter*(*top*(*tblptr*), id.*lexeme*, *T.type*, *top*(*offset*));
top(*offset*) = *top*(*offset*) + *T.width* }

$N \rightarrow \varepsilon$

将变量name的有关属性填入当前符号表

声明语句的处理

$P \rightarrow M\ D;\ S$

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblptr*); *push* (0, *offset*) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1;\ S$

$D \rightarrow$ id :T {*enter*(*top*(*tblptr*), id.*lexeme*, *T.type*, *top*(*offset*));
top(*offset*) = *top*(*offset*) + *T.width* }

$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblptr*));
push(*t*, *tblptr*); *push*(0, *offset*) }

建立子过程的符号表和偏移从0开始

声明语句的处理

$P \rightarrow M\ D;\ S$

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblptr*); *push* (0, *offset*) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1;\ S$ {*t* = *top*(*tblptr*);
addWidth(*t*, *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*);
enterProc(*top*(*tblptr*), id.*lexeme*, *t*) }

$D \rightarrow$ id :*T* {*enter*(*top*(*tblptr*), id.*lexeme*, *T.type*, *top*(*offset*));
top(*offset*) = *top*(*offset*) + *T.width* }

$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblptr*));
push(*t*, *tblptr*); *push*(0, *offset*) }

保留当前过程声明的总空间；弹出符号表和偏移栈顶（露出父过程的符号表和偏移；在父过程符号表中填写子过程名有关条目

声明语句的处理

$P \rightarrow M\ D;\ S$ {*addWidth (top (tblptr), top (offset));*
pop(tblptr); pop (offset) }

$M \rightarrow \varepsilon$ {*t = mkTable (nil);*
push(t, tblptr); push (0, offset) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1;\ S$ {*t = top(tblptr);*
addWidth(t, top(offset)); pop(tblptr); pop(offset);
enterProc(top(tblptr), id.*lexeme, t)* }

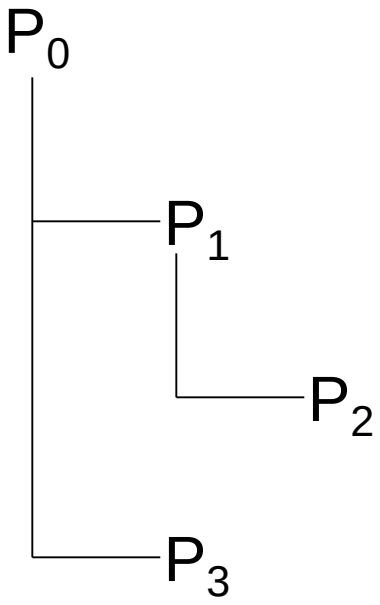
$D \rightarrow$ id :T {*enter(top(tblptr),* id.*lexeme, T.type, top(offset));*
top(offset) = top(offset) + T.width }

$N \rightarrow \varepsilon$ {*t = mkTable(top(tblptr));*
push(t, tblptr); push(0, offset) }

修改变量分配空间大小并清空符号表和偏移栈

举例：过程嵌套声明


```
i : int;  j : int ;
PROC P1 ;
   k : int;  f : real ;
   PROC P2;
          l : int ;
         a1 ;
   a2;
PROC P3;
   temp : int ;  max : int ;
   a3;
```



过程声明层次图

举例：过程嵌套声明

❑初始：M→ε

null	总偏移：

P_0

举例：过程嵌套声明

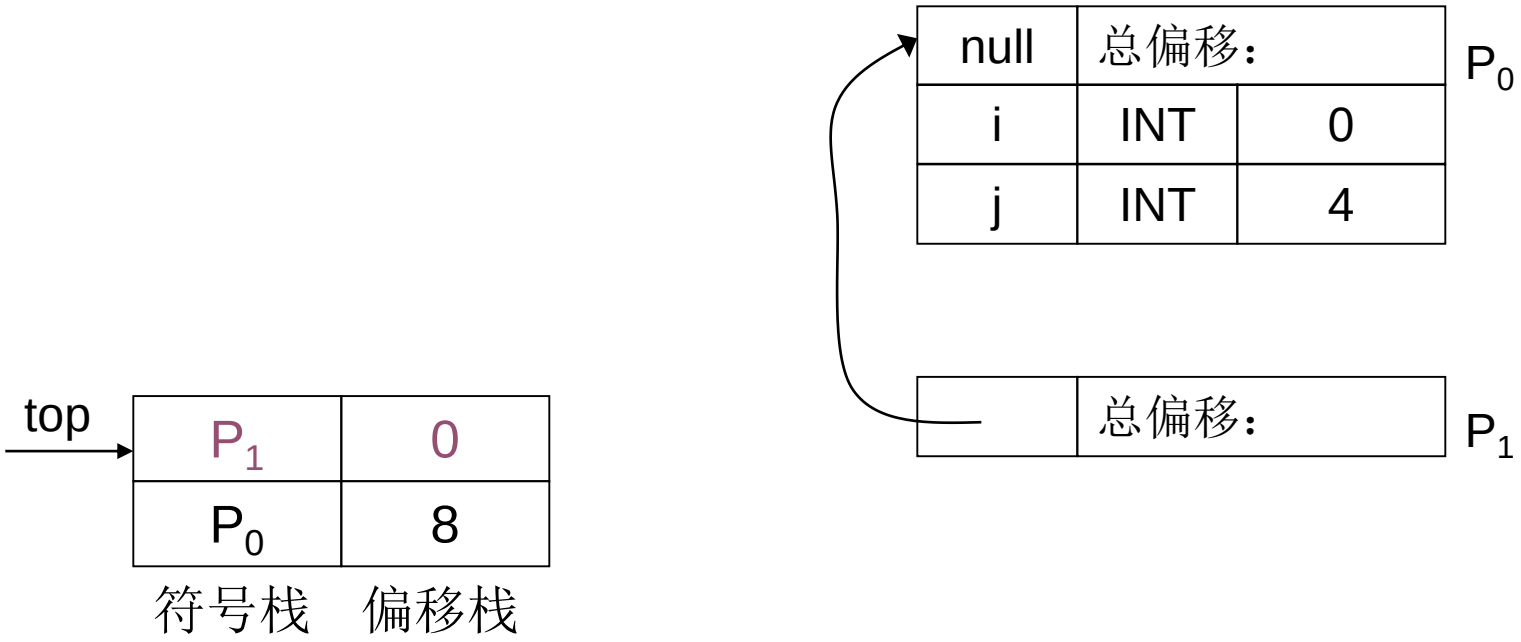
- i : int ; j : int ;

null	总偏移：	
i	INT	0
j	INT	4

P_0

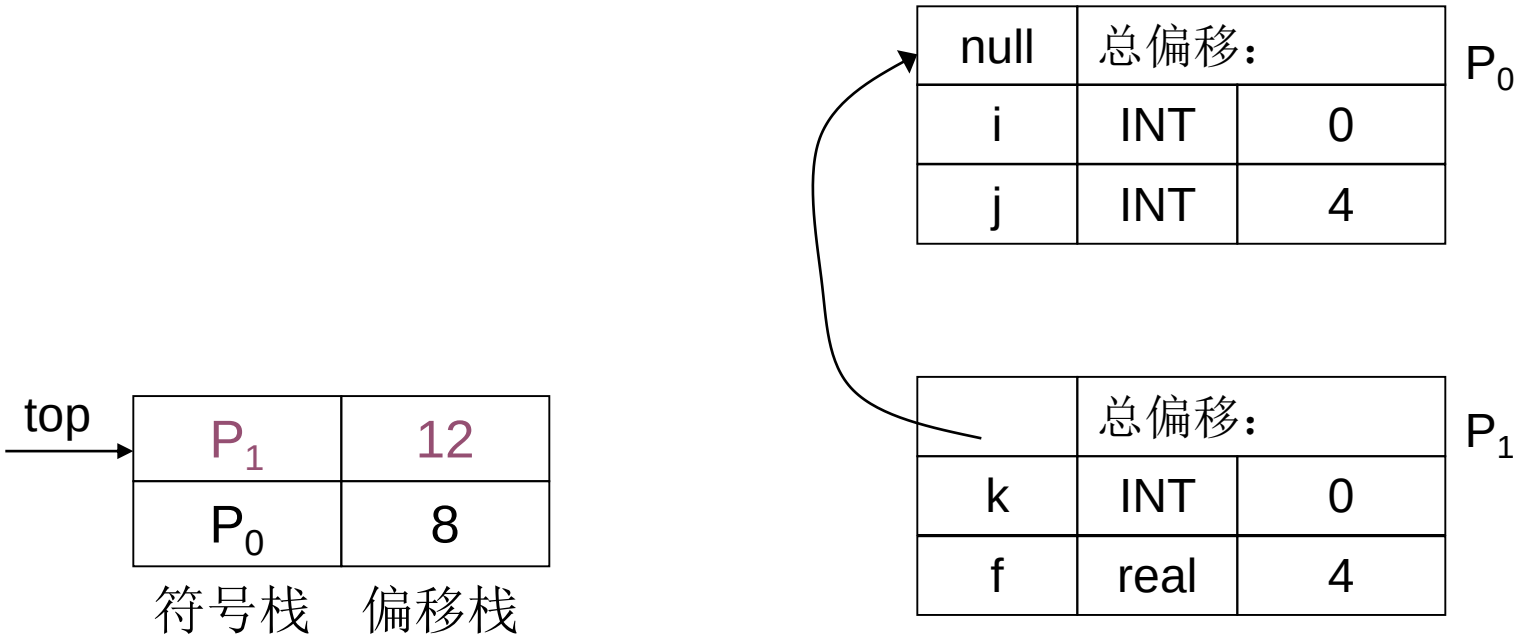
举例：过程嵌套声明

- PROC P_1；（N→ε）



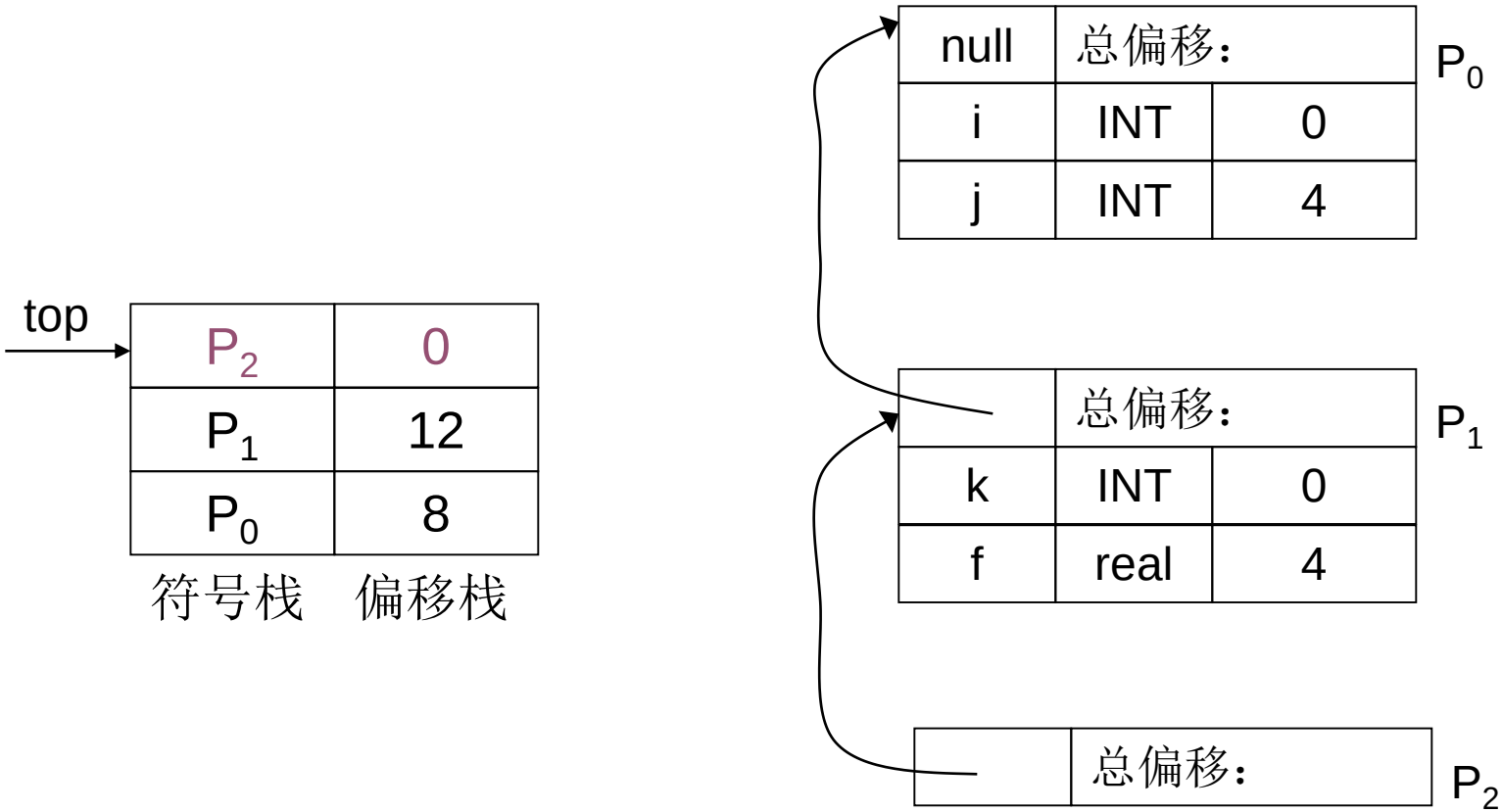
举例：过程嵌套声明

- k : int ; f : real ;



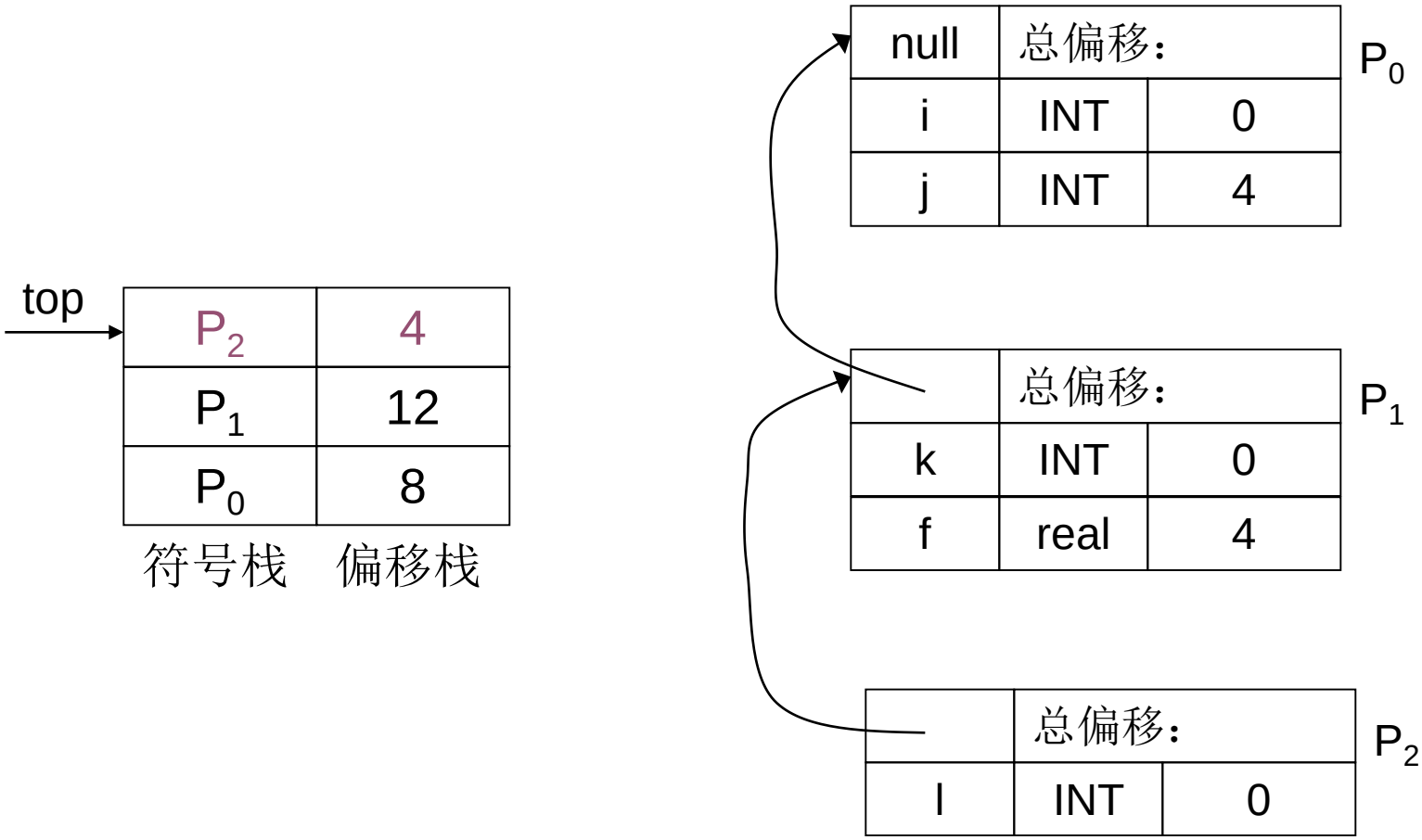
举例：过程嵌套声明

- PROC P_2；（N→ε）



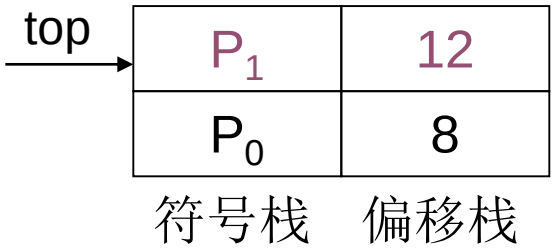
举例：过程嵌套声明

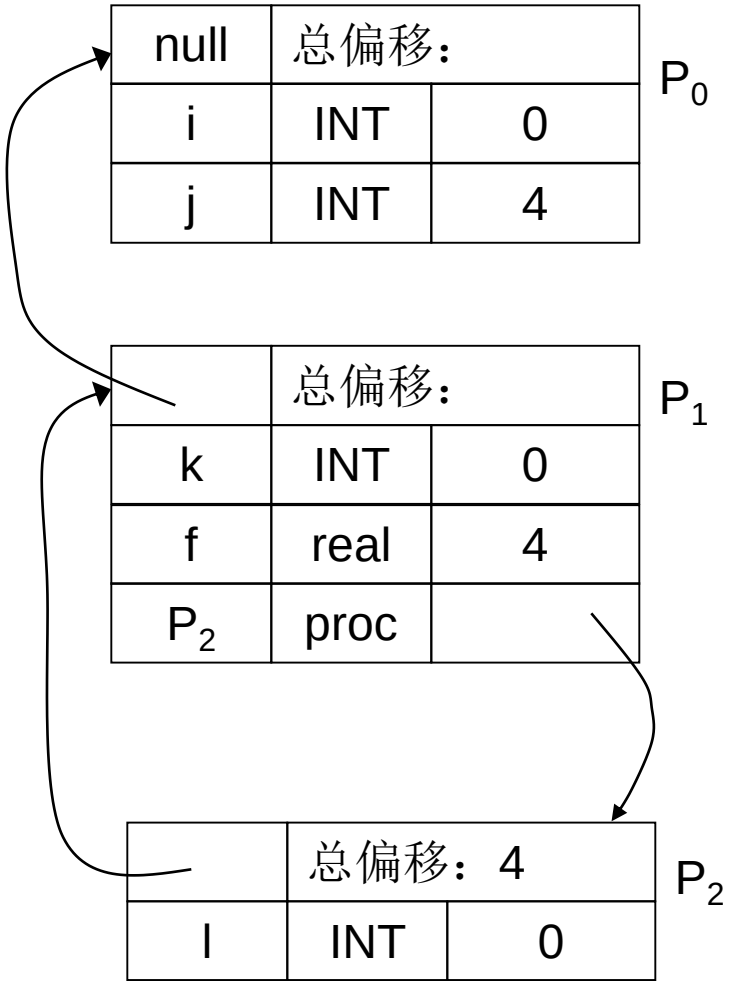
- l : int ;



举例：过程嵌套声明

- a_1；

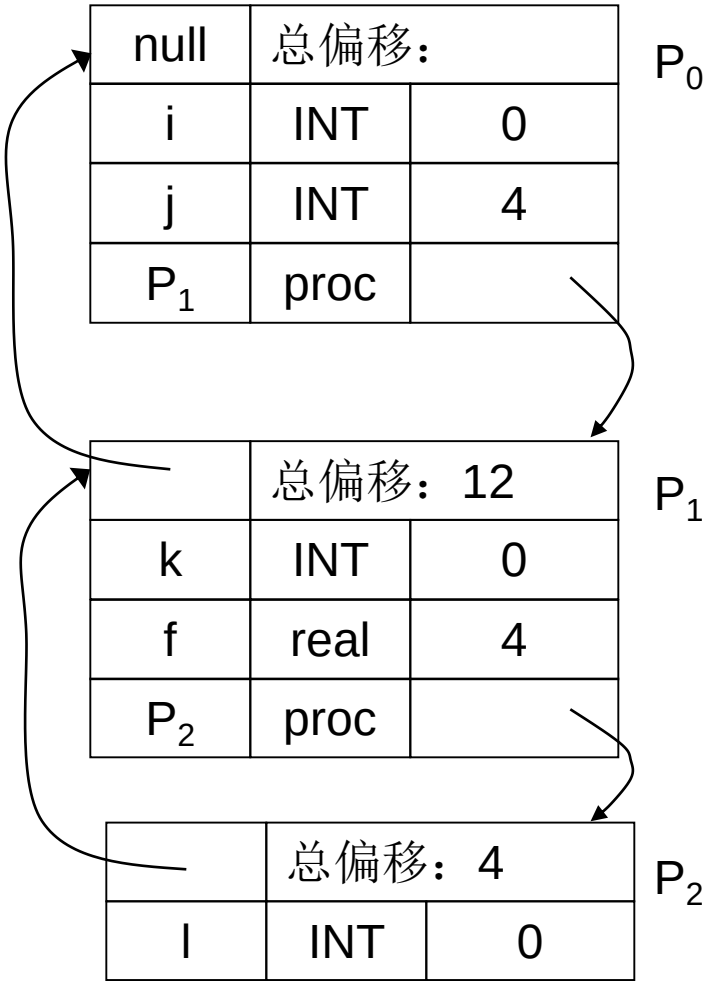

top
P_1
12
P_0
8
符号栈
偏移栈


null
总偏移：
i
INT
0
j
INT
4
P_0
总偏移：
k
INT
0
f
real
4
P_2
proc
P_1
总偏移：4
l
INT
0
P_2

举例：过程嵌套声明

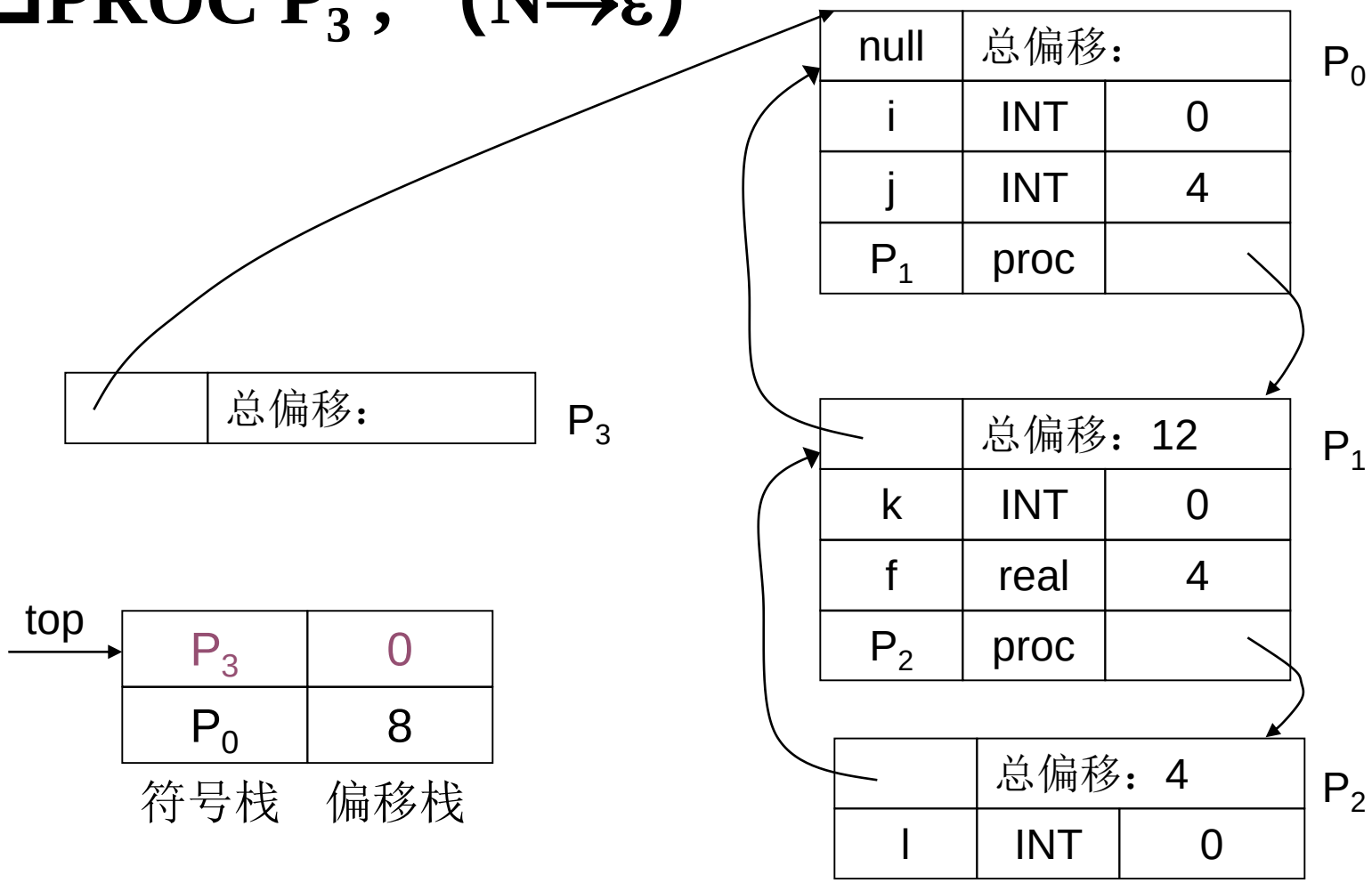
- a_2 ;


top
P_0
8
符号栈
偏移栈


null
总偏移：
i
INT
0
j
INT
4
P_1
proc
P_0
总偏移：12
k
INT
0
f
real
4
P_2
proc
P_1
总偏移：4
l
INT
0
P_2

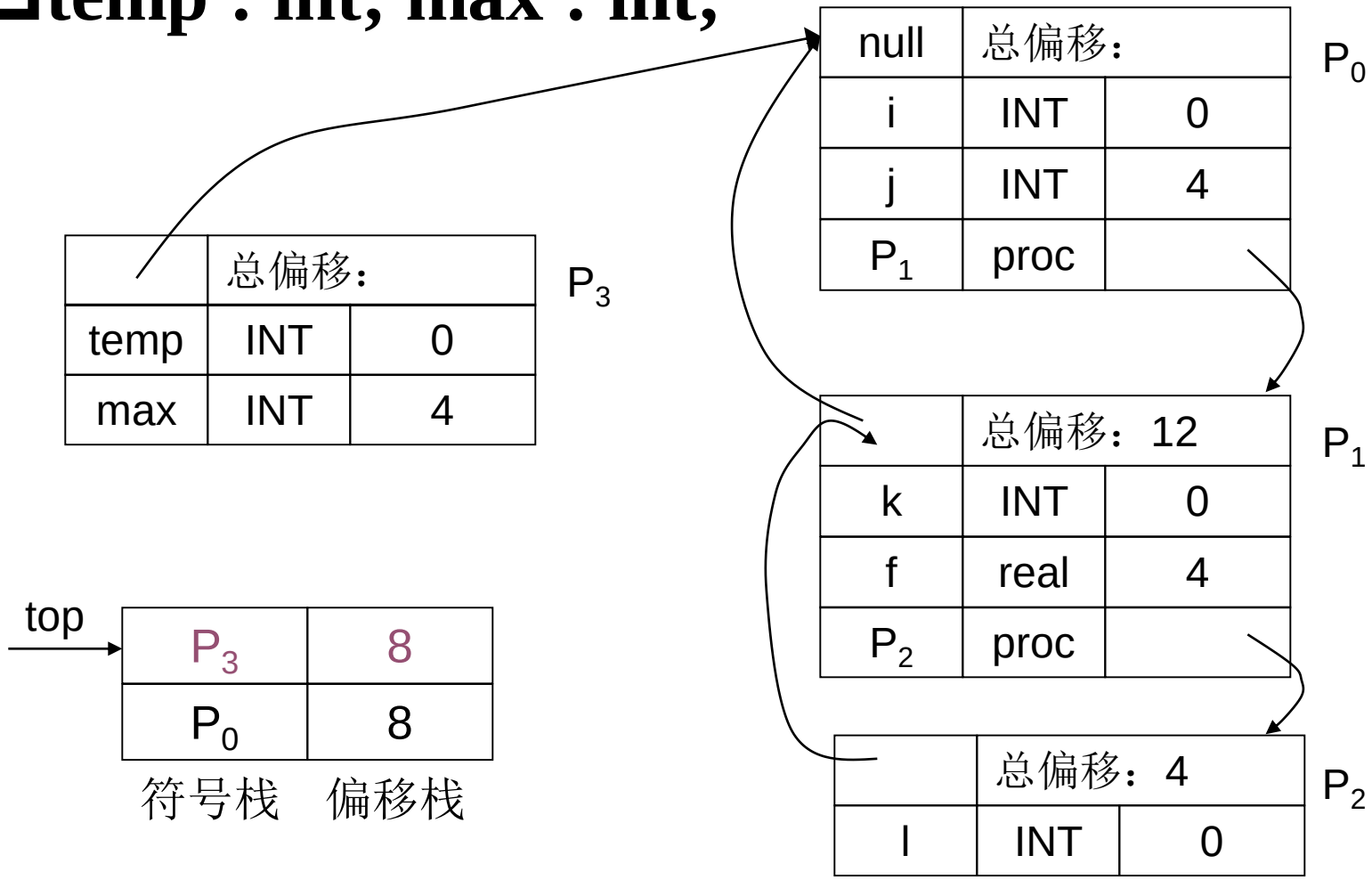
举例：过程嵌套声明

- PROC P_3；（N→ε）



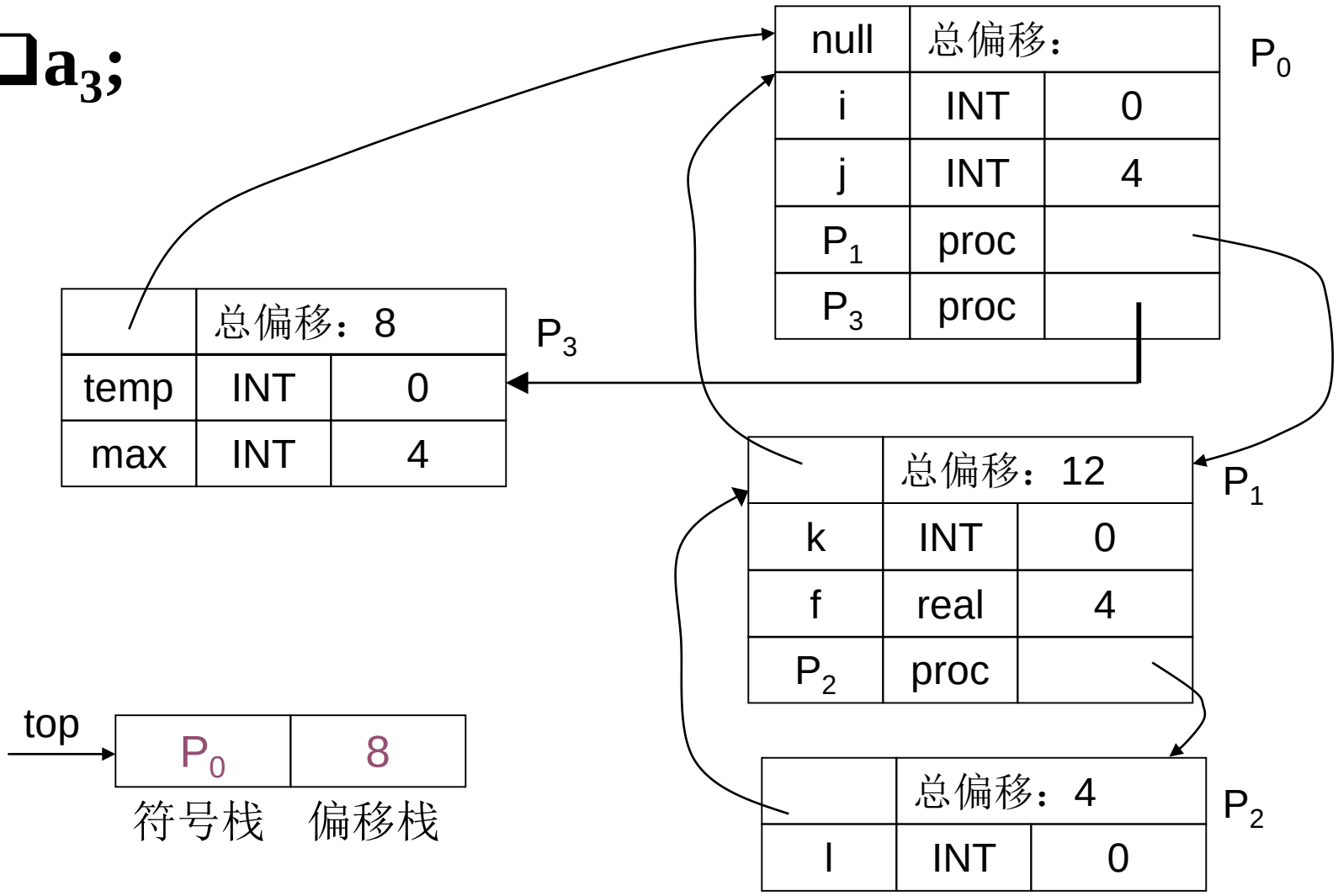
举例：过程嵌套声明

- temp : int; max : int;



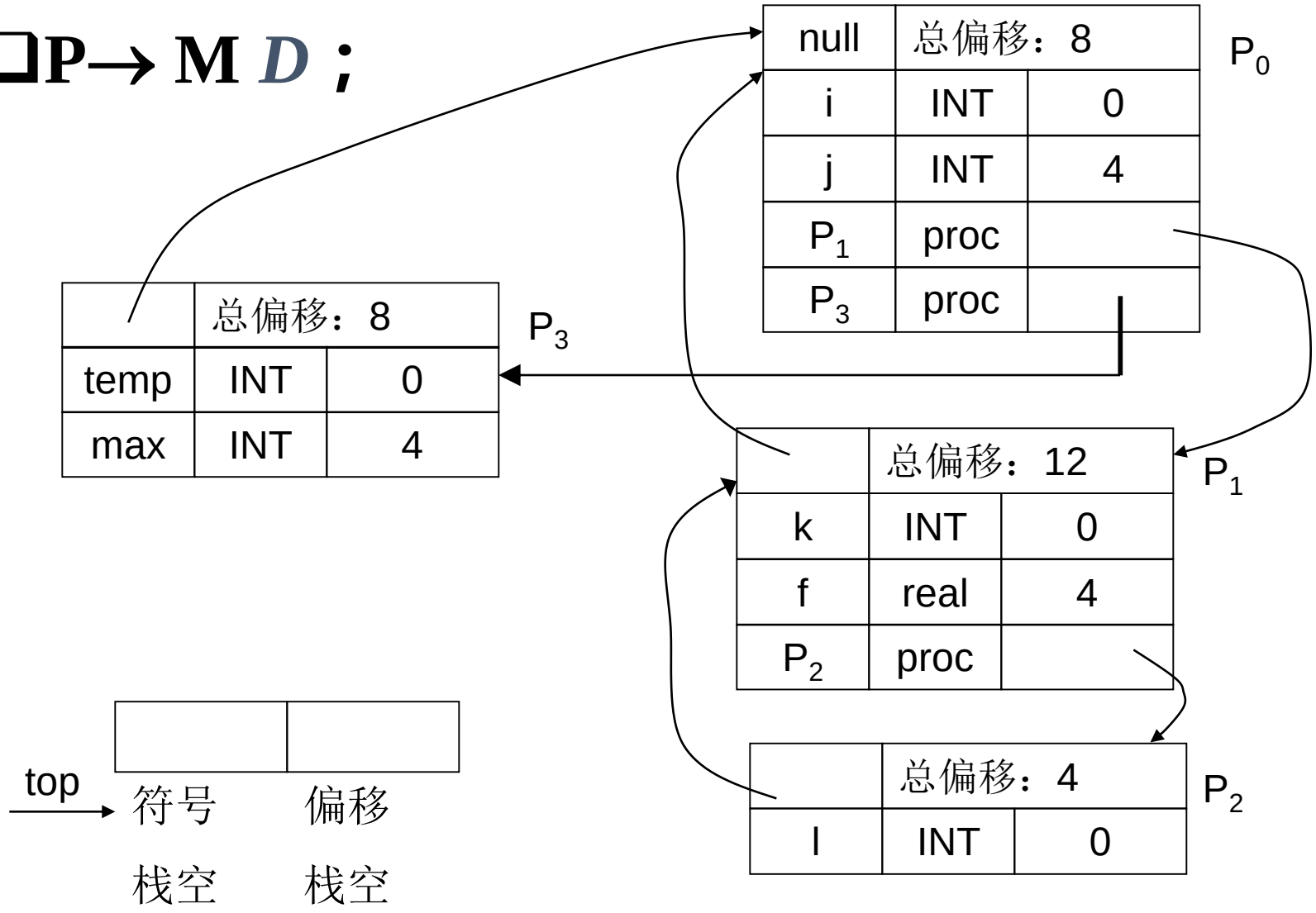
举例：过程嵌套声明

- a_3;


null
总偏移：
i
INT
0
j
INT
4
P1
proc
P3
proc
P0
总偏移：8
temp
INT
0
max
INT
4
P3
总偏移：12
k
INT
0
f
real
4
P2
proc
P1
总偏移：4
l
INT
0
P2
top
P0
8
符号栈
偏移栈

举例：过程嵌套声明

❑P→ M *D* ;



声明语句翻译的要点

- 分配存储单元
  - 名字、类型、宽、偏移
- 作用域的管理
  - 过程调用
- **记录类型的管理**
- **不产生中间代码指令，但是要更新符号表**

记录的域名管理

❑描述记录的文法

$T \rightarrow$ record D end

记录类型单独建符号表，域的相对地址从0开始

$T \rightarrow$ record L D end

$L \rightarrow \varepsilon$

```
record
    a :…;
    r : record
            i :…;
            . . .
        end;
    k : …;
end
```

记录的域名管理

❑描述记录的文法

$T \rightarrow$ record D end

记录类型单独建符号表，域的相对地址从0开始

$T \rightarrow$ record L D end

```
record
    a :…;
    r : record
            i :…;
            . . .
        end;
    k : …;
end
```

$L \rightarrow \varepsilon$ { *t = mkTable(nil);*

push(t, tblptr); push(0, offset) }

建立符号表，进入作用域

记录的域名管理

❑描述记录的文法

$T \rightarrow$ record D end

记录类型单独建符号表，域的相对地址从0开始

$T \rightarrow$ record L D end

{*T.type = record (top(tblptr));*

T.width = top(offset);

pop(tblptr); pop(offset) }

$L \rightarrow \varepsilon$ { *t = mkTable(nil);*

push(t, tblptr); push(0, offset) }

```
record
    a :…;
    r : record
            i :…;
            . . .
        end;
    k : …;
end
```

设置记录的类型表达式和宽度，退出作用域

记录的域名管理

❑描述记录的文法

$T \rightarrow$ record D end

记录类型单独建符号表，域的相对地址从0开始

$T \rightarrow$ record L D end

{*T.type = record (top(tblptr));*

T.width = top(offset);

pop(tblptr); pop(offset) }

$L \rightarrow \varepsilon$ { *t = mkTable(nil);*

push(t, tblptr); push(0, offset) }

D的翻译同前

```
record
    a :…;
    r : record
            i :…;
            . . .
        end;
     k : …;
end
```

举例：记录域的偏移

- 有2个C语言的结构定义如下：

```
struct A {
    char c1;
    char c2;
    long l;
    double d;
} S1;
```

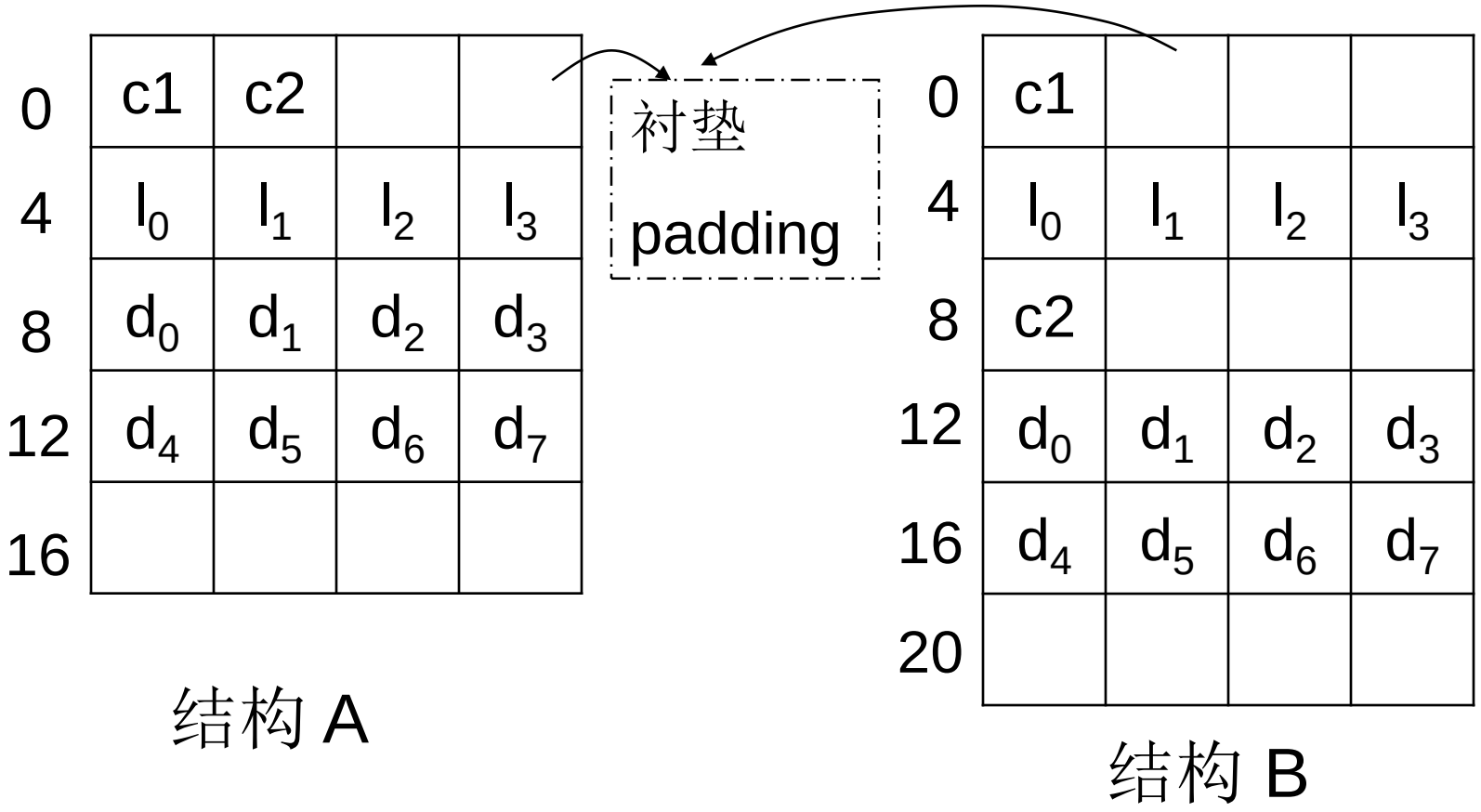
```
struct B {
    char c1;
    long l;
    char c2;
    double d;
} S2;
```

举例：记录域的偏移

- 数据（类型）的对齐 - alignment
- 在 X86-Linux下：
  - char：对齐1，起始地址可分配在任意地址
  - int，long，double：对齐4，即从被4整除的地址开始分配
- 注*：其它类型机器，double可能对齐到8
  - 如sun－SPARC

举例：记录域的偏移

- 结构A 和 B的大小分别为16和20字节(Linux)



举例：记录域的偏移

❑2个结构中域变量的偏移如下：

```
struct A {
  char c1;   0
  char c2;   1
  long l;    4
  double d;  8
} S1;
```

```
struct B {
    char c1;    0
    long l;     4
    char c2;    8
    double d;  12
} S2;
```

与类型相关的中间代码生成

中国科学技术大学
University of Science and Technology of China

- 声明语句的翻译
- **数组寻址的翻译**
- **类型转换**

数组元素的翻译

❑数组类型的声明

e.g. Pascal的数组声明，

A : array[low_1..$high_1$,…,low_n..$high_n$] of integer ;

数组元素：A[i , j, k,…] 或 A[i][j][k]…

（下界）$low_1 \le i \le high_1$（上界） , …

e.g. C的数组声明，

int A [100][100][100];

数组元素：A[i][30][40] $0 \le i \le (100-1)$

数组元素的翻译

- 翻译的主要任务
  - 输出(Emit)地址计算的指令
  - “基址[偏移]”相关的中间指令：t =b[o]，b[o]=t

数组元素的地址计算

❑一维数组A的第*i*个元素的地址计算

$$base + (i - low) \times w$$

base: 整个数组的基地址

low: 下标的下界

w: 每个数组元素的宽度

数组元素的地址计算

- 一维数组A的第i个元素的地址计算

$$base + (i - low) \times w$$

$base$: 整个数组的基地址

low: 下标的下界

w: 每个数组元素的宽度

可以变换成

$$i \times w + (base - low \times w)$$

$low \times w$**是常量，编译时计算，减少了运行时计算**

数组元素的地址计算

❑二维数组

A: array[1..2, 1..3] of T

❖列为主

A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3]

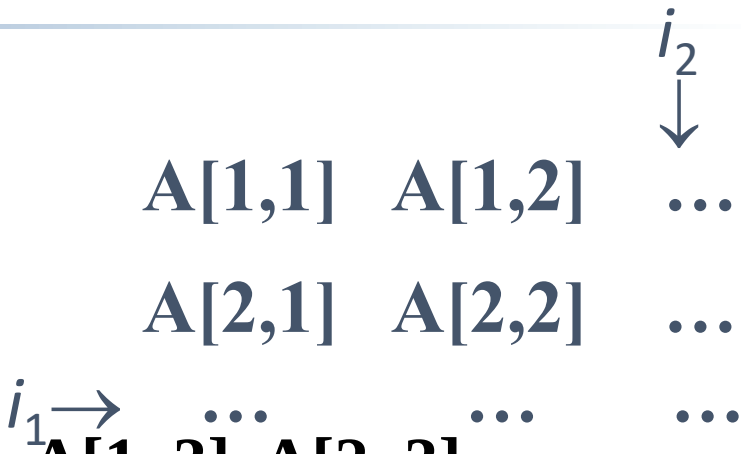
❖ 行为主

A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], A[2, 3]

数组元素的地址计算

❑二维数组

A: array[1..2, 1..3] of T



❖列为主

A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3]

❖行为主

A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], A[2, 3]

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

（A[i_1, i_2]的地址，其中$n_2 = high_2 - low_2 + 1$）

变换成 $((i_1 \times n_2) + i_2) \times w +$

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

数组元素的地址计算

- 多维数组下标变量A[i_1, i_2, ..., i_k]的地址表达式
  - 以行为主

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

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

数组元素的地址计算

- 多维数组下标变量A[i_1, i_2, ..., i_k]的地址表达式
  - 以行为主

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

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

红色部分是数组访问翻译中的最重要的内容

数组元素地址计算翻译方案

- 下标变量访问的产生式

$S \rightarrow L := E$　　$L \rightarrow \text{id} [\ Elist\ ] \mid \text{id}$

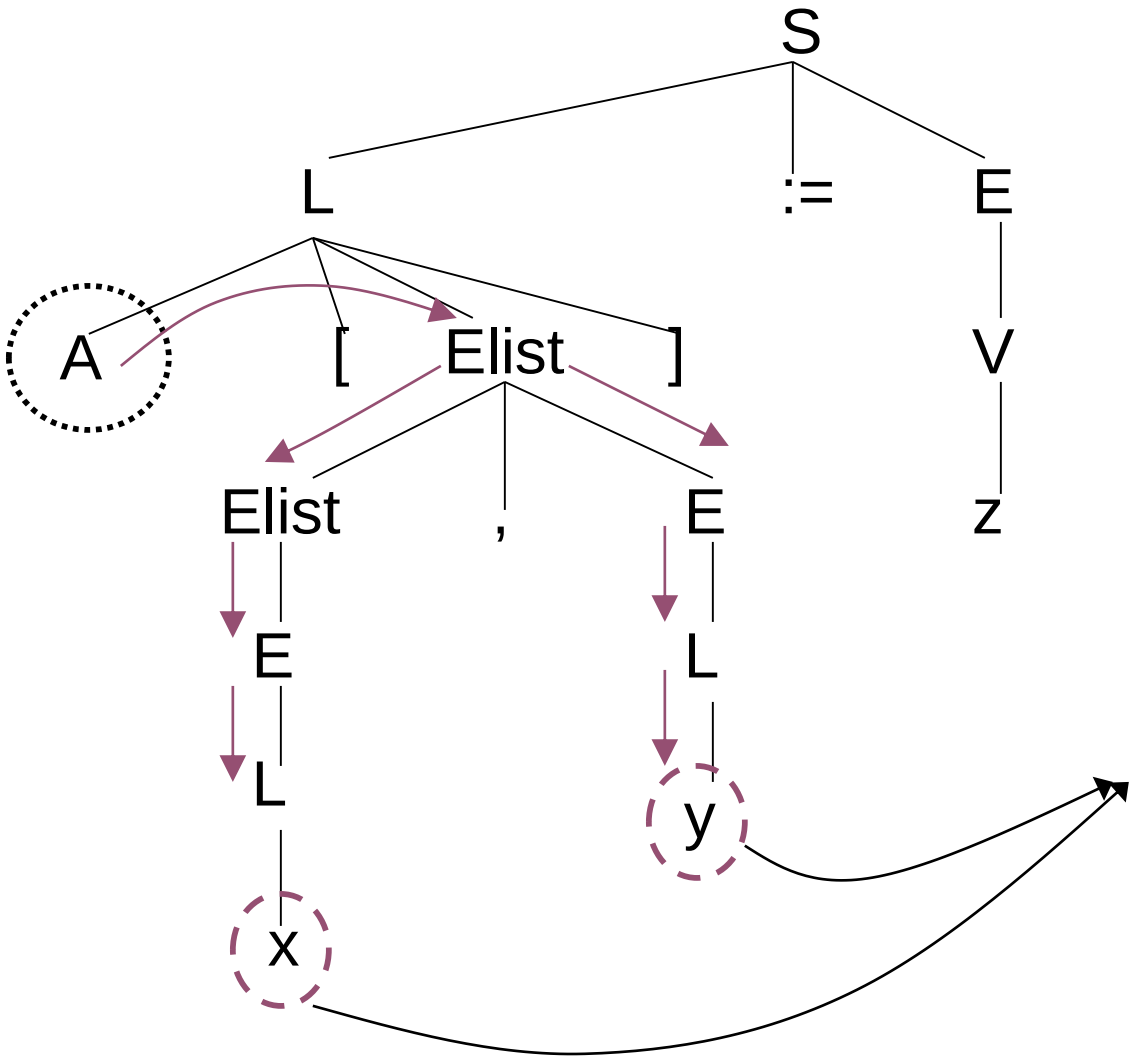
$Elist \rightarrow Elist, E \mid E$　　$E \rightarrow \text{L} \mid \ldots$

- 采用语法制导的翻译方案时存在的问题

$Elist \rightarrow Elist, E \mid E$

由Elist的结构只能得到各维的下标值，但无法获得数组的信息（如各维的长度）

A[x,y] := z的分析树



当分析到下标（表达式）x和y时，要计算地址中的“可变部分”。这时需要知晓数组A的有关的属性，如n_m，类型宽度w等，而这些信息存于在结点A处。若想使用必须定义有关继承属性来传递之。但在移进—归约分析不适合继承属性的计算！

修改文法

❑所有产生式

$S \rightarrow L := E$

$E \rightarrow E + E$

$E \rightarrow (E)$

$E \rightarrow L$

$L \rightarrow Elist\ ]$

$L \rightarrow \mathrm{id}$

$Elist \rightarrow Elist, E$

$Elist \rightarrow \mathrm{id}\ [\ E$

修改文法，使数组名id成为Elist的子结点（类似于前面的类型声明），从而避免继承属性的出现

相关符号属性定义：

L.place，L.offset ：

- 若L是简单变量，L.place为其“值”的存放场所，而L.offset为空（null）；
- 当L表示数组元素时，L.place是其地址的“常量值”部分；而此时L.offset为数组元素地址中可变部分的“值”存放场所，数组元素的表示为：

  L.place [L.offset]

相关符号属性定义：

Elist.place : “可变部分”的值，即下标计算的值

Elist.array : 数组名条目的指针

Elist.ndim: 当前处理的维数

limit(array, j) : 第j维的大小

width(array) : 数组元素的宽度

invariant(array) : 静态可计算的值

数组元素的翻译

- 翻译时重点关注三个表达式：
  - $Elist \rightarrow \text{id} [E$ ： 计算第1维
  - $Elist \rightarrow Elist_1, E$ ： 传递信息
  - $L \rightarrow Elist]$ ： 计算最终结果

数组元素的翻译

$S \rightarrow L := E$ {if *L.offset* == null then /* *L*是简单变量 */

emit (*L.place*, '= ', *E.place*)

else

/*取数组元素的左值*/

emit (*L.place* , '[', *L.offset*, ']', '=', *E.place*) }

数组元素的翻译

$Elist \rightarrow \text{id} [E$ {*Elist.place* = *E.place*;
/*第一维下标*/
Elist.ndim = 1;
Elist.array = id.*place* }

数组元素的翻译

$Elist \rightarrow Elist_1, E$

{

$t = newTemp();$

/*维度增加1*/

$m = Elist_1.ndim + 1;$

/* 第m维的大小*/

$n_m = limit(Elist_1.array, m);$

/*计算公式7.6 $e_{m-1} * n_m$ */

$emit(t, '=', Elist_1.place, '*', n_m);$

/*计算公式7.6 $e_m = e_{m-1} * n_m + i_m$*/

$emit(t, '=', t, '+', E.place);$

$Elist.array = Elist_1.array;$

$Elist.place = t;$

$Elist.ndim = m$

}

数组元素的翻译

$L \rightarrow Elist$] { $L.place = newTemp()$;

/*获取数组元素地址的常量值*/

emit (*L.place*, '=', *base*(*Elist.array*), '–',
invariant (*Elist.array*));

L.offset = *newTemp*();

/*获取数组元素地址的可变部分*/

emit (*L.offset*, '=', *Elist.place*, '*',
width(*Elist.array*))}

数组元素的翻译

$L \rightarrow$ id {*L.place* = id.*place*; *L.offset* = null }

$E \rightarrow L$ { if *L.offset* == null then /* *L*是简单变量 */

E.place = *L.place*

else begin *E.place* = *newTemp*();

emit (*E.place*, '=', *L.place*, '[', *L.offset*, ']') end }

$E \rightarrow E_1 + E_2$ {*E.place* = *newTemp*();

emit (*E.place*, '=', E_1.*place* , '+', E_2.*place*) }

$E \rightarrow (E_1)$ {*E.place* = E_1.*place* }

其他翻译同前

数组元素的翻译-举例

- 数组A的定义为：A[1…10, 1…20] of integer
- 数组的下界为1，即low为1
- 为赋值语句 x := A[y, z]生成中间代码

举例： x := A[y, z]

L.place = x
L.offset = null
|
x

A[1…10, 1…20] of integer

举例： x := A[y, z]

L.place = x
L.offset = null
|
x

:=

A[1…10, 1…20] of integer

举例： x := A[y, z]

L.place = x
L.offset = null

:=

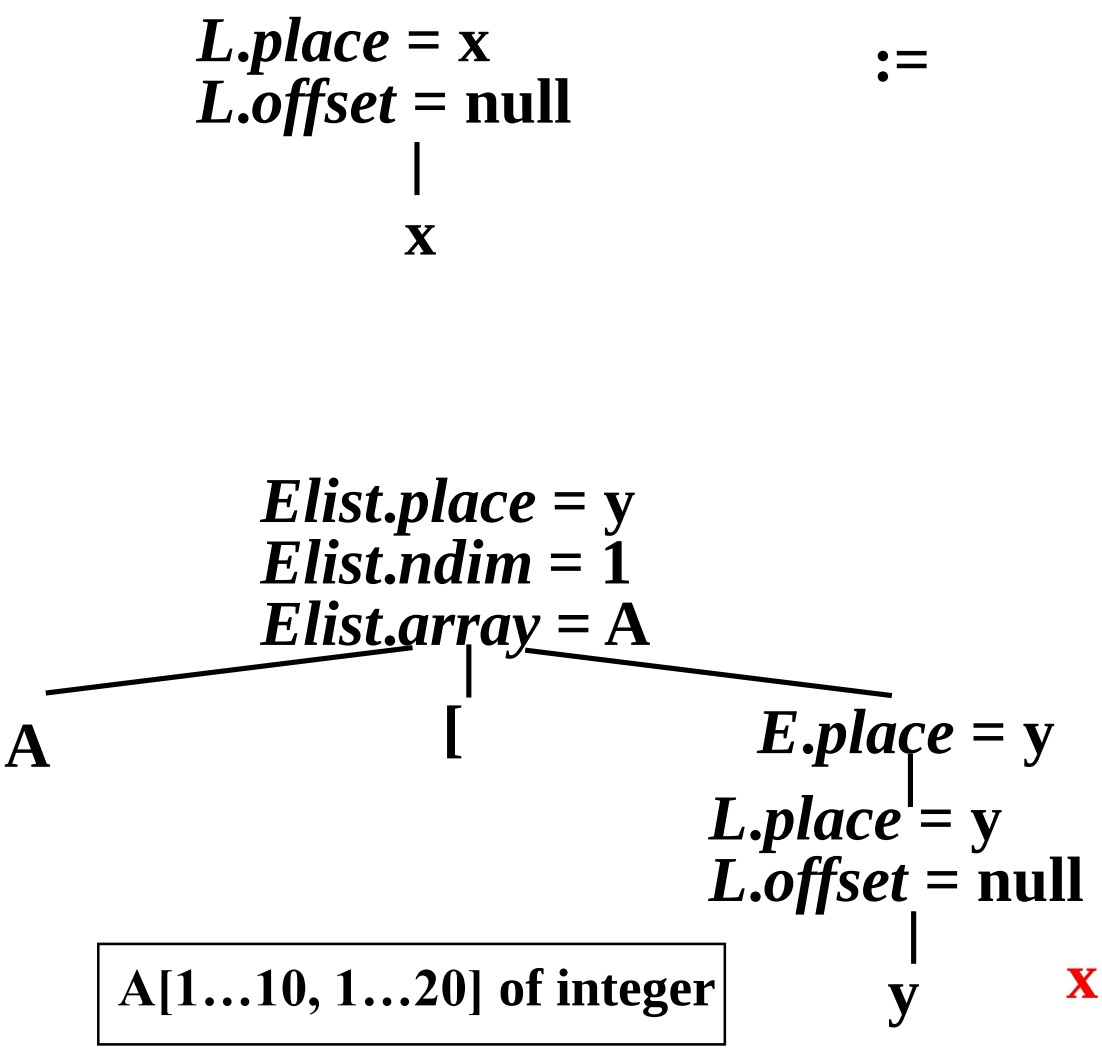
A [*E.place* = y

L.place = y
L.offset = null

A[1…10, 1…20] of integer

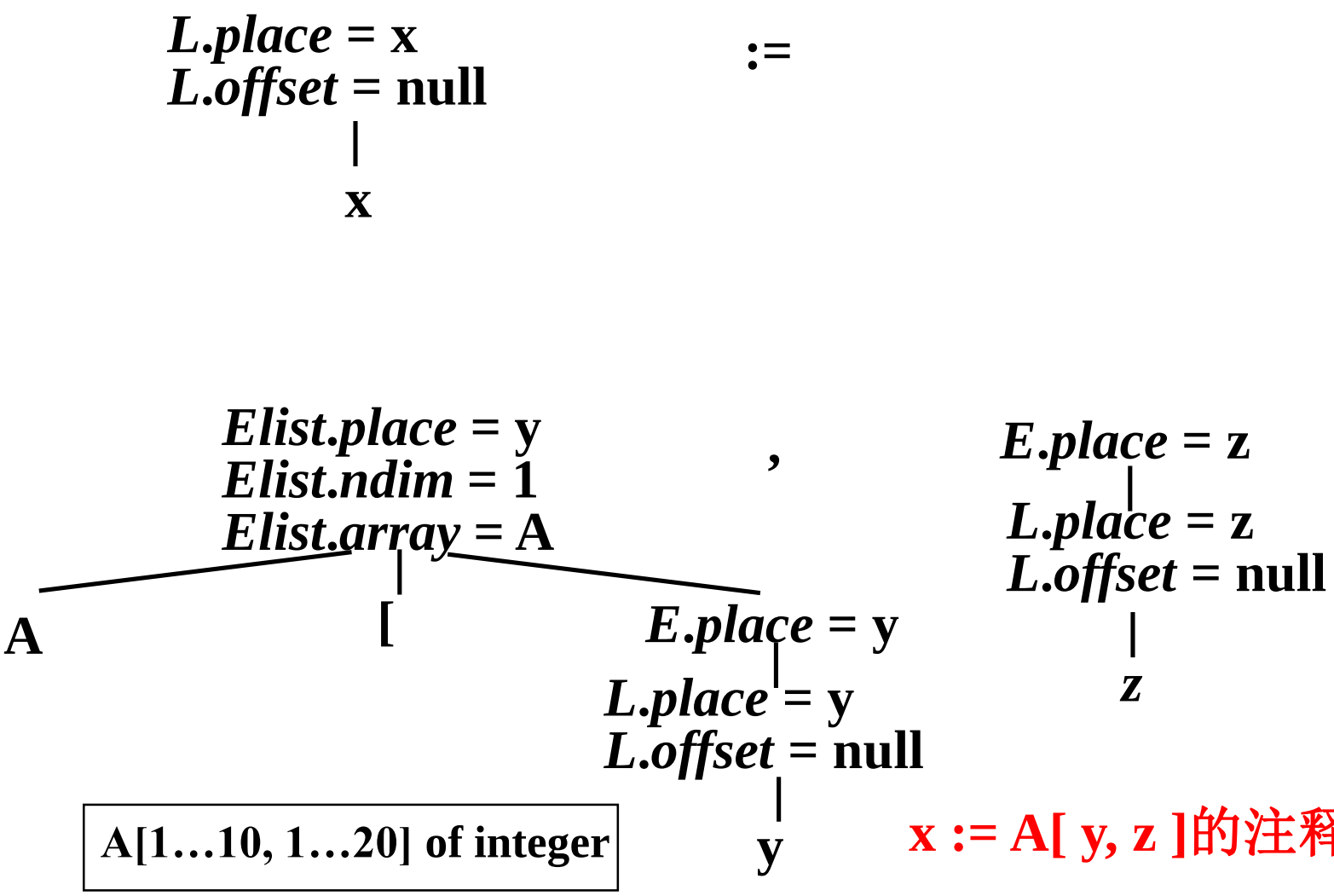
y

举例： x := A[y, z]



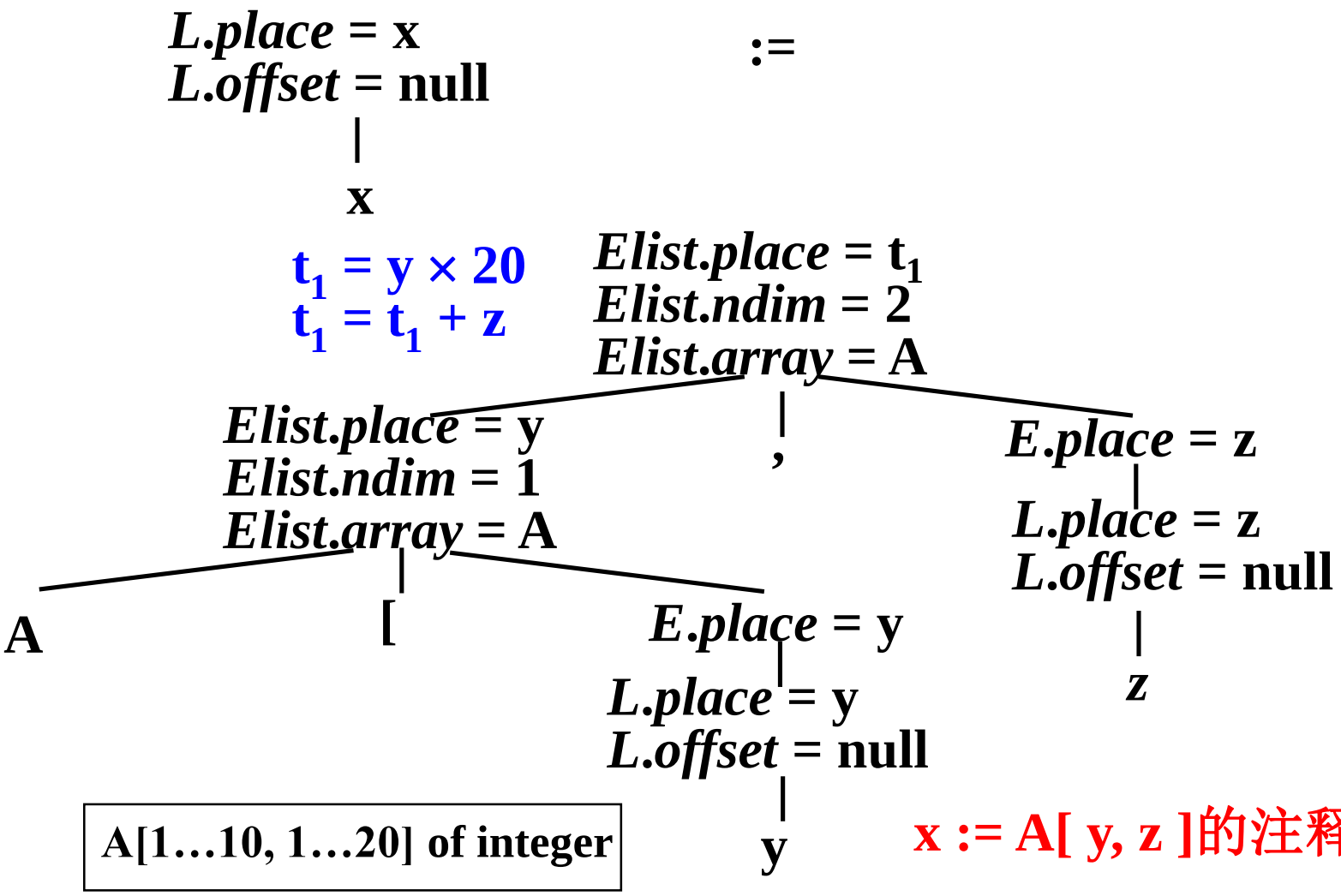
x := A[y, z]的注释分析树

举例： x := A[y, z]



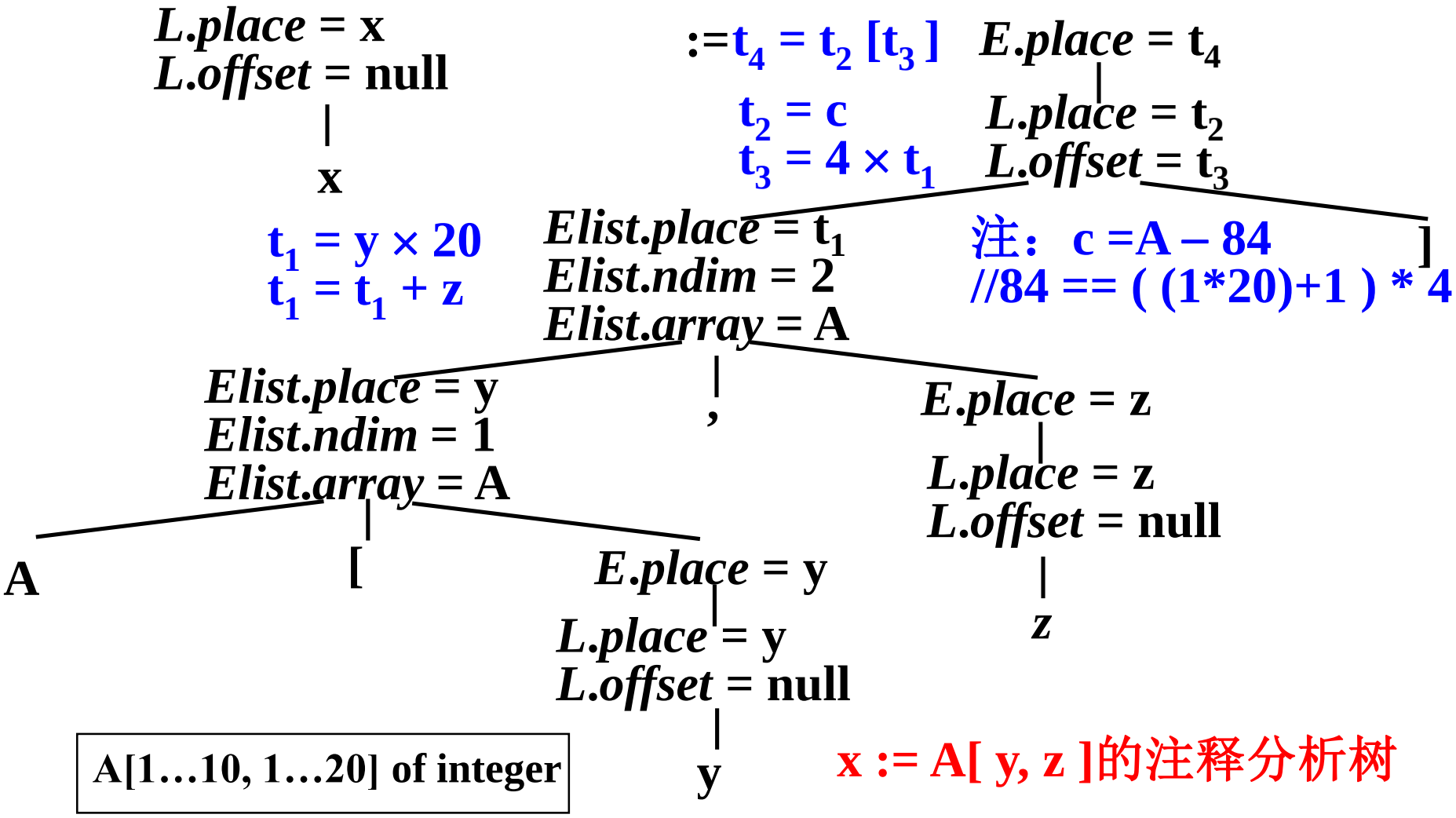
x := A[y, z]的注释分析树

举例： x := A[y, z]



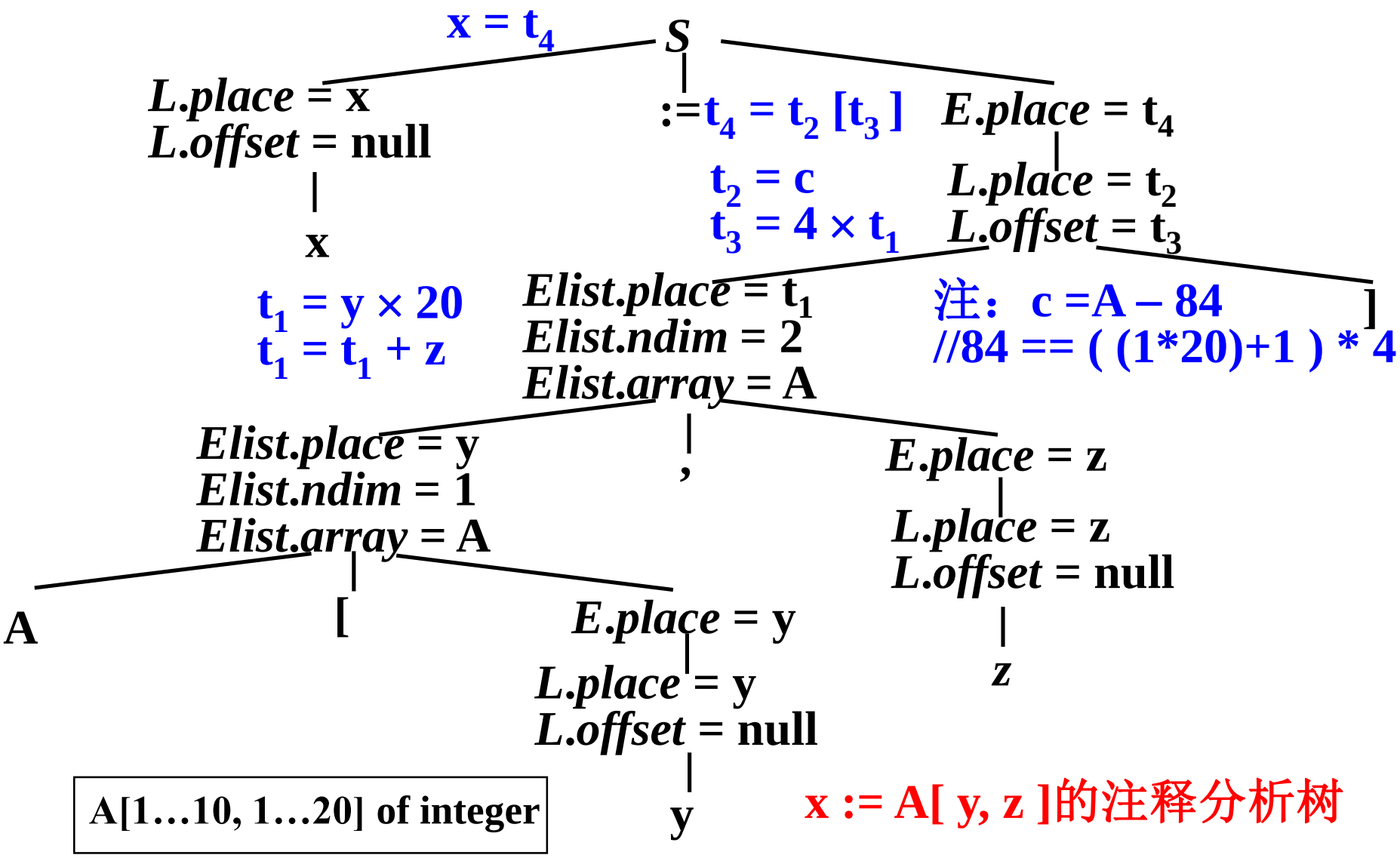
x := A[y, z]的注释分析树

举例： x := A[y, z]



x := A[y, z]的注释分析树

举例： x := A[y, z]



x := A[y, z]的注释分析树

举例：A[i, j] := B[i, j] * k

- 数组A：A[1..10, 1..20] of integer;

  数组B：B[1..10, 1..20] of integer;

  w ：4 （integer）

- TAC如下：

  (1) t_1 := i * 20

  (2) t_1 := t_1 + j

  (3) t_2 := A - 84 // 84 == ((1*20)+1) * 4

  (4) t_3 := t_1 * 4 // 以上A[i, j]的（左值）翻译

举例：A[i, j] := B[i, j] * k

TAC如下（续）：

(5) t_4 := i * 20

(6) t_4 := t_4 + j

(7) t_5 := B - 84

(8) t_6 := t_4 * 4

(9) t_7 := t_5[t_6]

//以上计算B[i,j]的右值

TAC如下（续）：

(10) t_8 := t_7 * k

//以上整个右值表达

//式计算完毕

(11) t_2[t_3] := t_8

// 完成数组元素的赋值

与类型相关的中间代码生成

中国科学技术大学
University of Science and Technology of China

- 声明语句的翻译
- 数组寻址的翻译
- **类型转换**

❑例 x = y + i * j
（x和y的类型是real，i和j的类型是integer）

中间代码

t_1 = i **int×** j

t_2 = inttoreal t_1

t_3 = y **real+** t_2

x = t_3

int× 和 real+ 不是类型转换，而是算符

目标机器的运算指令是区分整型和浮点型的

高级语言中的重载算符=>中间语言中的多种具体算符

类型转换的处理

- 以$E \rightarrow E_1 + E_2$为例说明
  - 判断E1 和E2的类型，看是否要进行类型转换;若需要，则分配存放转换结果的临时变量并输出类型转换指令

```
{ E.place = newTemp();
if (E1.type == integer && E2.type == integer) then begin
  emit (E.place, '=', E1.place, 'int+', E2.place);
  E.type = integer
end
else if (E1.type == integer && E2.type == real) then
  begin
  u = newTemp(); emit (u, '=', 'inttoreal', E1.place);
  emit (E.place, '=', u, 'real+', E2.place); E.type = real;
end
  . . .}
```

中国科学技术大学
University of Science and Technology of China
USTC, CHINA
ADSLAB
先进数据系统实验室

《编译原理与技术》
中间代码生成 I

TBA