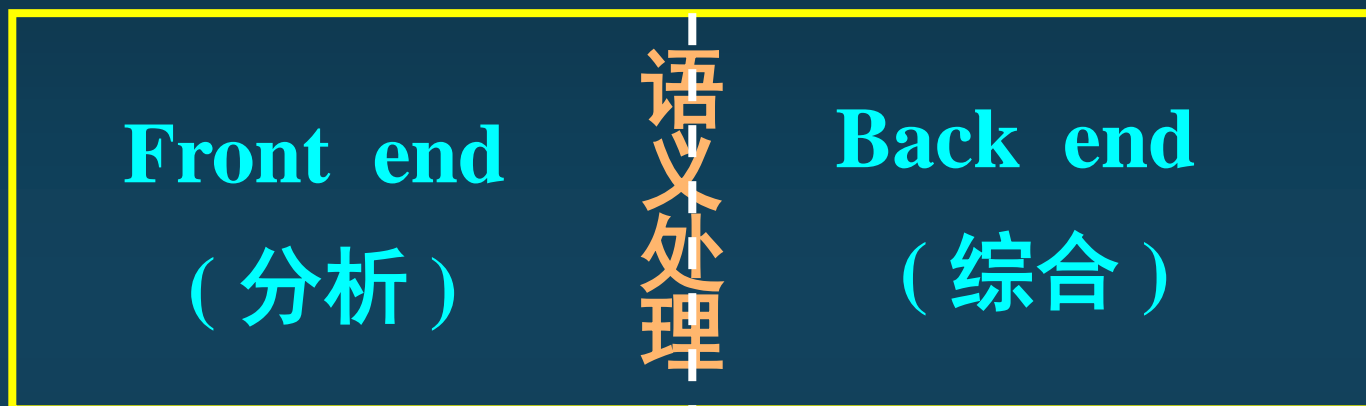


## ■ 语义分析的地位



- 编译程序最实质性的工作;
- 第一次对源程序的语义作出解释, 引起源程序质的变化。

## ■ 语义分析的任务

按照语法分析器识别的语法范畴进行**语义检查**和**处理**，产生相应的**中间代码**或目标代码。



## ■ 中间代码

介于源语言和目标代码之间一种代码。

## ■ 引入中间代码的目的

1. 方便生成目标代码;
2. 便于优化;
3. 便于移植。

## ■ 语法制导翻译

为文法的每一个产生式配一个相应的语义子程序（或语义规则描述的语义动作），并在语法分析的同时调用它。

$[ \$ L_2 , \text{翻译动作序列} ]$



对偶集合的相互映射

## ■ 属性翻译文法

把语义引入文法，给产生式中的文法符号附加“属性”，构成涉及语义的翻译文法。



文法符号的语义性质(语义属性)

## ■ 形式定义 $A = (G, V, F)$

其中：

G：一般为二型文法；

V：属性的有穷集；

F：关于属性的断言或谓词的有穷集。

# ■ 属性表示 数、符号串、类型、存储空间...


$N.t$

( $N$ 是 $G$ 非终结符,  $t$ 是 $N$ 的属性)

| 产生式                          | 语义规则   |
|------------------------------|--|
| $L \rightarrow E$            | <code>print( E.val )</code>                    |
| $E \rightarrow E_1 + T$      | <code>E.val = E<sub>1</sub>.val + T.val</code> |
| $E \rightarrow T$            | <code>E.val = T.val</code>                     |
| $T \rightarrow T_1 * F$      | <code>T.val = T<sub>1</sub>.val × F.val</code> |
| $T \rightarrow F$            | <code>T.val = F.val</code>                     |
| $F \rightarrow (E)$          | <code>F.val = E.val</code>                     |
| $F \rightarrow \text{digit}$ | <code>F.val = digit.lexval</code>              |


## ■ 继承属性

分析树中, 如果一个结点的属性值是由该结点的父结点和(或)兄弟结点的属性定义的称为继承属性。

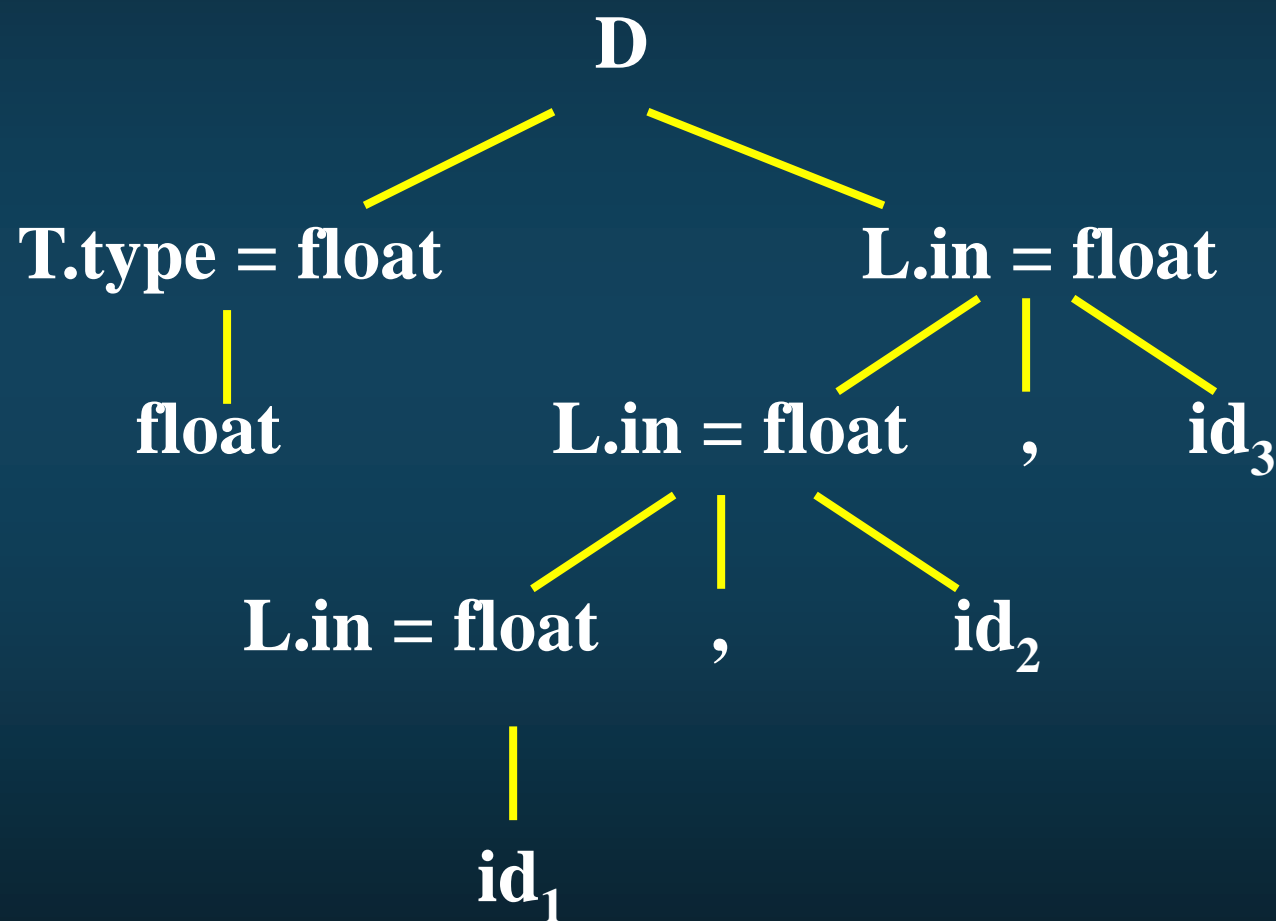


## ■ 综合属性

分析树中, 如果一个结点的属性值是通过子结点的属性值计算得到则称为综合属性。

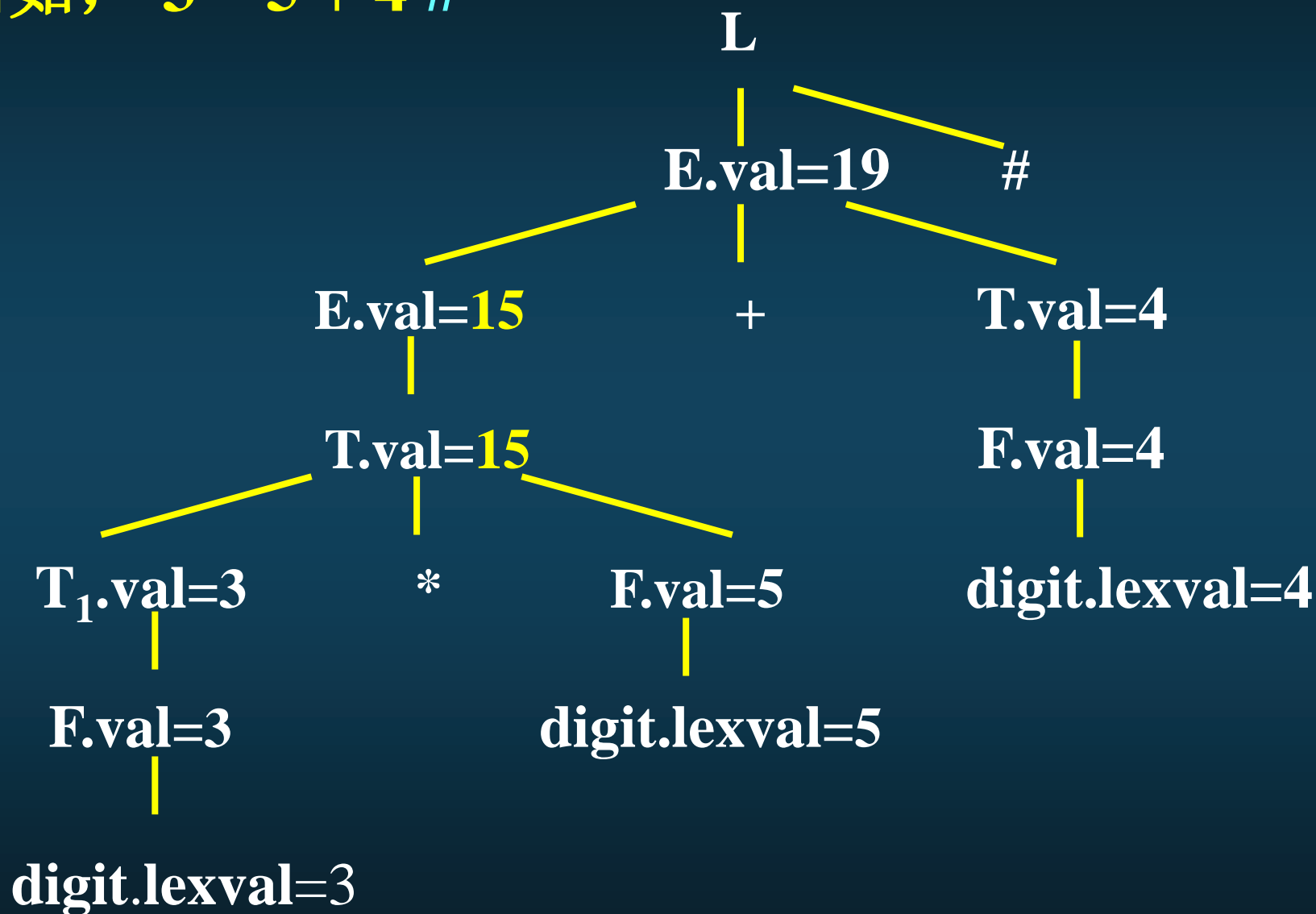


例如,     float id<sub>1</sub>, id<sub>2</sub>, id<sub>3</sub> ;





例如,  $3 * 5 + 4 \#$



中间语言( 中间代码)是语义程序的输出。

中间语言的设计与应用既要考虑从源语言到目标语言的翻译跨度又要考虑目标机的指令集特点。

中间语言 { 逆波兰式  
N元式(三元式、间接三元式四元式)  
图

# 一. 逆波兰式 (中缀表达式 $\longrightarrow$ 后缀式)

形式定义:

$$\underbrace{e_1 e_2 \dots e_n}_{\text{运算对象}} \theta \quad (n \geq 1)$$

$\theta$  为 运算符

例如,

$$a * b \longrightarrow ab*$$

$$-a \longrightarrow a@ \quad (@ \text{表示单目} \\ \text{“}_- \text{”})$$

$$a+b*(c+d)*(e+f) \longrightarrow abcd+*ef+*+$$

IF <expr> **then** <s1> **else** <s2>

Label1 ▼ ▼ Label2



<expr> <lable1> BZ <s1> <lable2> BR <s2>

**BZ**、**BR**: 引入的两个操作符。

**BZ**是二目操作符，如果<expr>的计算结果为假，则产生一个到<lable1>的转移。 <lable1>是<s2>的头一个符号。

**BR**则是一个单目操作符，它产生一个到<lable2>的转移。

## 例6.1 设有如下C程序片段

```
{  
    int k;  
    i=j=0;  
h:   k=100;  
    if (k>i+j) { k--; i++; }  
        else k=i*2-j*2;  
    goto h;  
}
```



(1) **block**  
(2)  $i\ j\ 0 ==$   
(7)  $h:$   
(8)  $k\ 100 =$   
(11)  $k\ i\ j + > (28) \text{ BZ}$   
(18)  $k\ k\ 1 - = i\ i\ 1 + = (37) \text{ BR}$   
(29)  $k\ i\ 2 * j\ 2 * - =$   
(38) **(7) BR**  
(39) **blockend**



addr

## 二. N元式      N个域的记录结构


( D<sub>1</sub>, D<sub>2</sub>, D<sub>3</sub>, ... , D<sub>n</sub> )

OP域

操作对象域

### ■ 常见N元式

三元式、间接三元式  双地址机指令形式

四元式  三地址机指令形式

## 三元式形式定义:

$$\text{NO. ( OP, ARG}_1, [\text{ARG}_2] )$$

其中:

**NO.:** 为产生的三元式的顺序编号;

**OP:** 是操作符;

**ARG1**和**ARG2**为第一操作数和第二操作数。(也可以是前面某一个三元式的编号, 代表该三元式的计算结果被作为操作数)

## 间接三元式形式定义:

间接码表 + 三元式



例如，语句  $X=A+B*C$  的三元式表示为

| NO. | OP | ARG1 | ARG2 |
|-----|----|------|------|
| (1) | *  | B    | C    |
| (2) | +  | A    | (1)  |
| (3) | =  | (2)  | X    |

**if  $X > Y$  then  $Z = X$  else  $Z = Y + 1$**

的三元式可以表示为

| NO. | OP | ARG1 | ARG2 |
|-----|----|------|------|
| (1) | —  | X    | Y    |
| (2) | BZ | (1)  | (5)  |
| (3) | =  | X    | Z    |
| (4) | BR |      | (7)  |
| (5) | +  | Y    | 1    |
| (6) | =  | (5)  | Z    |
| (7) |    |      |      |

$$(a+b)^2 + (a+b)^3$$

|   |                 |
|---|-----------------|
| ① | <u>+ , a, b</u> |
| ② | ↑ , ① , 2       |
| ③ | <u>+ , a, b</u> |
| ④ | ↑ , ③ , 3       |
| ⑤ | +, ② , ④        |

其中：“↑”表示幂运算。

# $(a+b)^2 + (a+b)^3$ 的间接三元式

间接码表

|   |
|---|
| ① |
| ② |
| ① |
| ③ |
| ④ |

控制三元式代码执行顺序



三元式

|   |         |
|---|---------|
| ① | +, a, b |
| ② | ↑, ①, 2 |
| ③ | ↑, ①, 3 |
| ④ | +, ②, ③ |

## 四元式形式定义:

NO. ( OP, [ARG<sub>1</sub>], [ARG<sub>2</sub>], Result )

|   |                  |                  |                |
|---|------------------|------------------|----------------|
| + | a,               | b,               | T <sub>1</sub> |
| ↑ | T <sub>1</sub> , | 2,               | T <sub>2</sub> |
| ↑ | T <sub>1</sub> , | 3,               | T <sub>3</sub> |
| + | T <sub>1</sub> , | T <sub>2</sub> , | T <sub>4</sub> |

\*\* 注: T<sub>i</sub>是临时变量。

### 三. 图（树）

一个三元式  $\longleftrightarrow$  一棵子树

OP  $\longleftrightarrow$  子树根

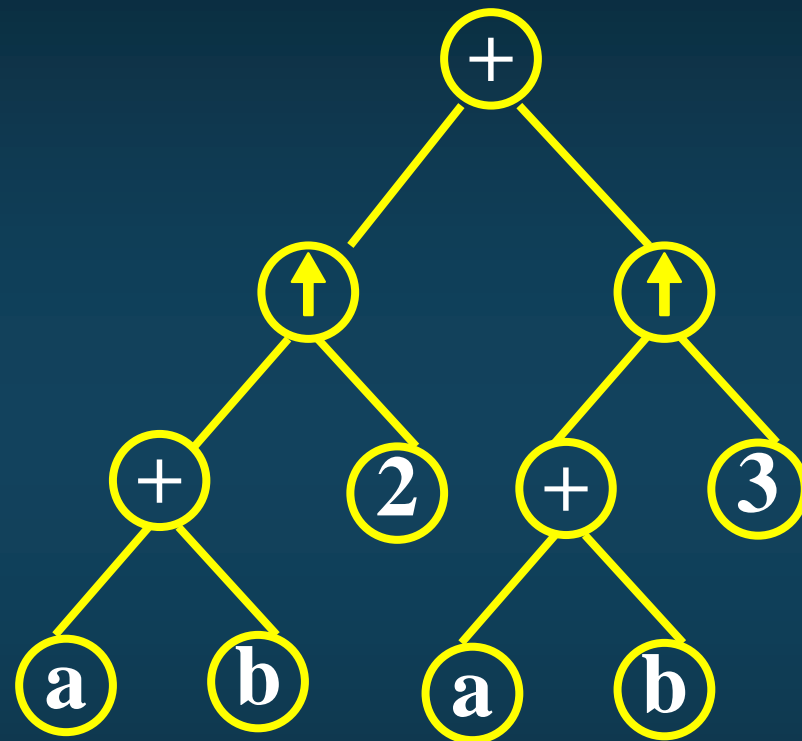
ARG1  $\longleftrightarrow$  子树左叶节点

ARG2  $\longleftrightarrow$  子树右叶节点

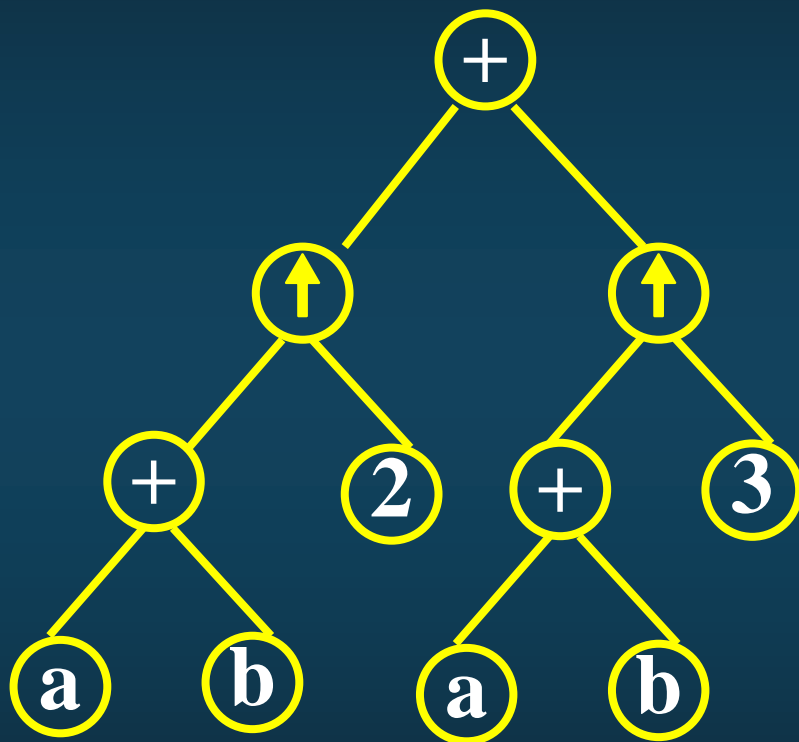
$$(a+b)^2 + (a+b)^3$$

### 三元式

|   |         |
|---|---------|
| ① | +, a, b |
| ② | ↑, ①, 2 |
| ③ | ↑, ①, 3 |
| ④ | +, ②, ③ |



## 几种表示间关系



三元式

↑ 线性化  
树

↓ 后序遍历

$ab+2 \uparrow ab+3 \uparrow +$



## ■ 源程序流基本组成单位特点

关键字: 表示语句性质, 反映语句结构;

标识符: 表示程序中各种实体;

如, 变量名; 常量名; 过程名; 函数名; 文件名, 数组名 ...

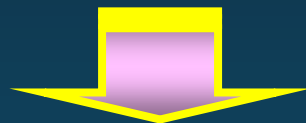


**V:** name , type , addr , level ...

**array:** name , type , addr , dim , level , size ...

语句标号: name, addr, 定义否 ...

过程、函数: name, addr, 形参 ...



反映了标识符的语义特征属性，是翻译的依据。



记录于符号表中 (namelist)

整个编译过程中动态地采集、记录、变更、引用

例如，设有C程序片段

```
      :  
      int i, a[4];  
      { ...  
        i=a[2];  
        ...  
      }
```

词法语法分析

语义分析

类型检查;

数组越界检查;

回填addr; ...

i, a  $\Rightarrow$  符号表

i.type  $\Rightarrow$  符号表

a.type  $\Rightarrow$  符号表

a.维数  $\Rightarrow$  符号表

a.每维大小  $\Rightarrow$  符号表

.....

## ■ 符号表

存放源程序中有关标识符的属性信息的数据结构。

## ■ 符号表结构

| 名字域 | 属性信息域 |
|-----|-------|
|-----|-------|

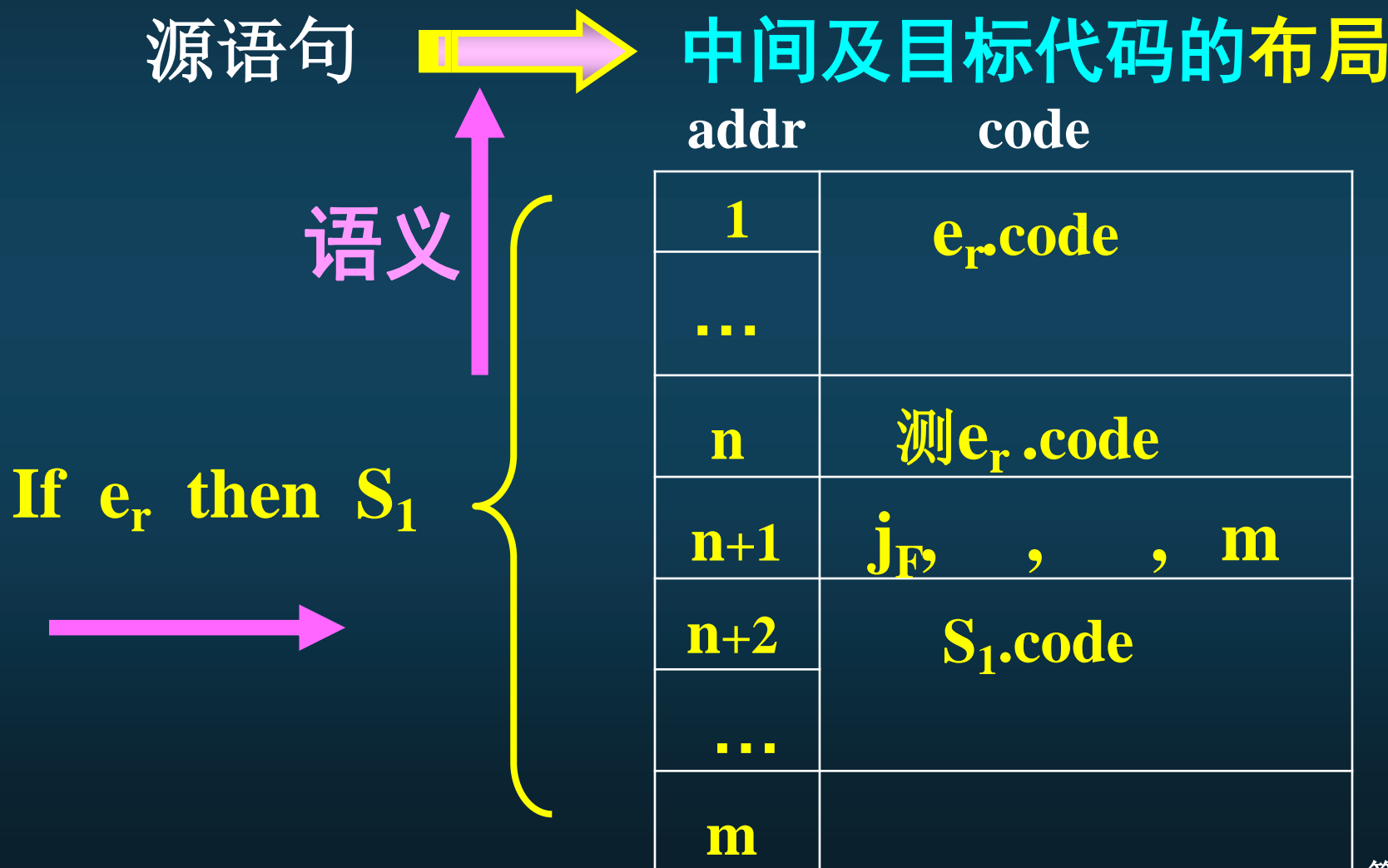
- 符号表作用 {
  - 收集标识符属性信息;
  - 语义检查依据;
  - 代码生成时地址分配依据。

## ■ 常见标识符类的主要属性与作用

1. 标识符。作用：查重（考虑作用域和可视性前提）
2. 类型。作用：存储空间分配；可施加运算的检查等
3. 存储类别。作用：提供语义处理、检查、存储分配的依据，还决定量的作用域、可视性，动态生存期等问题。
4. 作用域、可视性。作用：动态活动环境支持，提供量所在层次；
5. 其他。

## ■ 语句翻译设计要点

### 1. 确定语句的目标结构；



## ■ 语句翻译设计要点

1. 确定语句的目标结构；
2. 确定中间代码；
3. 根据目标结构和语义规则，构造语义子程序（转换翻译程序）；
4. 涉及的实现技术；

## ■ 说明类语句

语言中定义性信息，一般不产生目标代码，其作用是辅助完成编译。

例如， 变量说明 类型说明， 对象说明， 标号说明 .....

## ■ 说明类语句的处理

相关说明的属性信息填入符号表，提供语义检查和存储分配的依据。



# 一. 常量定义语句的翻译

#define 标识符 常量

CONST pi=3.1416; true=1;

CONST pi=3.1415926;

namelist

| name | kind | type | addr |     |
|------|------|------|------|-----|
| pi   | CONS | R    |      |     |
| true | CONS | B    |      |     |
| ...  |      |      |      | ... |

constlist

| ord | value     |
|-----|-----------|
| 1   | 3.1416    |
| 2   | 1         |
|     | 3.1415926 |

## ■ 常量说明语句的语义子程序

**CONST\_DEF  $\rightarrow$  CONST <con\_list>;**

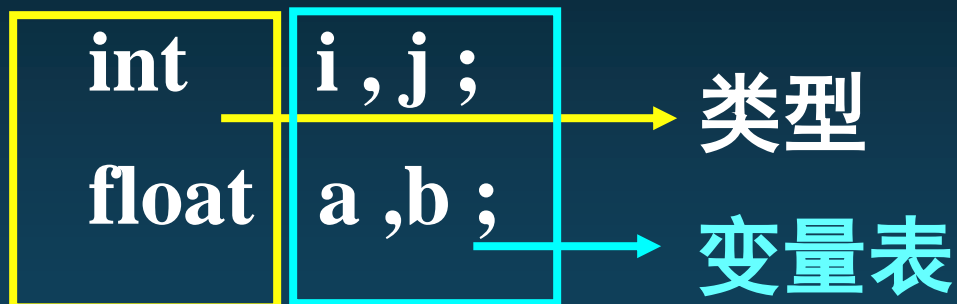
**<con\_list>  $\rightarrow$  <con\_list>;CD**

**<con\_list>  $\rightarrow$  CD**

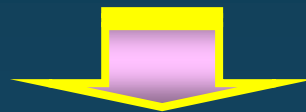
**CD  $\rightarrow$  id=num**

```
{ num.ord=look_con_table(num.lexval);
  id.ord=num.ord;
  id.type= num.type;
  id.kind=constant;
  add(id.entry;id.ord; id.type; id.kind) }
```

## 二. 简单说明类语句



**namelist**



| name | kind | type | addr | ... |
|------|------|------|------|-----|
| i    | V    | I    | 0    |     |
| j    | V    | I    | 4    |     |
| a    | V    | R    | 8    |     |
| b    | V    | R    | 16   |     |
| ...  |      |      |      |     |

## ■ 翻译方案和语义子程序

```

D → int id    { fill(ENTRY(id), int); D.AT=int }
D → float id  { fill(ENTRY(id), float); D.AT=float }
D → D, id;    { fill(ENTRY(id), D1 .AT);
                  D.AT=D1 .AT }
  
```

\* 其中：

**D.AT**：设为非终结符D的语义变量，它记录说明语句所规定的量的某种性质。

**fill(P, A)**：函数。完成把性质A填入P所指的符号表入口的相应数据项中。

**ENTRY(i)**：函数。给出i所代表的量在符号表中的入口。

### 三. 复合类型说明语句

$$T \rightarrow \text{struct } L\{D\}[V];$$
$$L \rightarrow \text{id} \mid \varepsilon$$
$$D \rightarrow D;F \mid F$$
$$F \rightarrow \text{type } V;$$
$$V \rightarrow V, \text{id} \mid \text{id}$$

其中:

**L**: 结构类型名;    **D**: 结构成员;    **V**: 变量表;

**F**: 结构成员项;    **id**: 标识符;


例如,


struct date { int year, month, day; } today, yesterday;

## ■ 语义处理涉及

- (1) 结构成员与该结构相关;
- (2) 结构的存储: 一个结构的所有成员项连续存放(简单方式);
- (3) 结构的引用是结构成员的引用, 不能整体引用; (成员信息须单独记录)

例如,

```
struct k1  L
{
    int a;
    float b;
    int c[10];
    char d;
}
```

 {D}

V空

## namelist

| name | kind   | ... | addr |
|------|--------|-----|------|
| k1   | struct |     |      |
| ...  |        |     |      |

## 结构成员分表

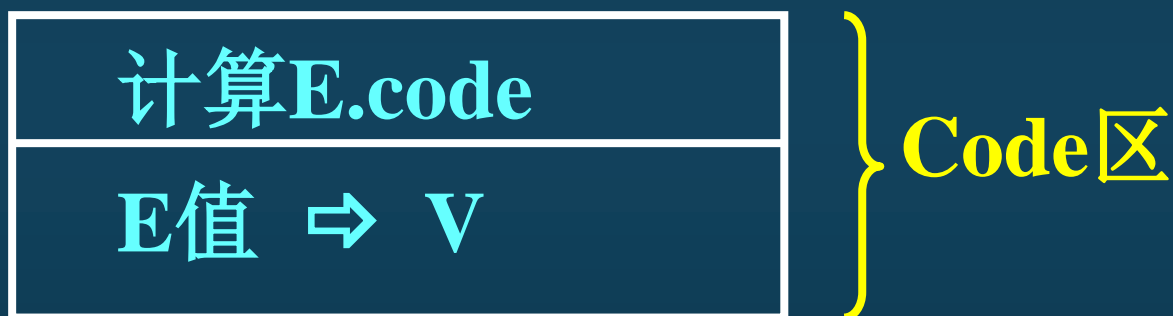
| name | kind  | type | LEN | OFFSET |
|------|-------|------|-----|--------|
| a    | V     | I    | 4   | 0      |
| b    | V     | R    | 8   | 4      |
| c    | array | I    | 40  | 12     |
| d    | V     | C    | 1   | 52     |
| ...  |       |      |     |        |



## ■ 赋值语句形式定义

$$A \rightarrow V = E$$

## ■ 赋值语句目标结构



\*\* 赋值语句的处理集中在表达式的处理上

## ■ 赋值语句翻译语义子程序

$A \rightarrow i = E$

{ GEN(=, E.PLACE, \_, ENTRY(i) }

其中:

**GEN:** 函数。

把四元式(OP, ARG1, ARG2, RESULT)  
填入四元式表。

**E. PLACE:** 表示存放E值的变量名在符号表的  
入口地址。

函数**ENTRY(i)**同前。

## ■ 表达式形式定义

$$E \rightarrow E_1 \text{ op } E_2 \mid \text{op } E_1 \mid \text{id}$$

其中：

**OP**: 为算术运算符；

$E_1, E_2, \text{id}$ : 运算对象。

## ■ 表达式语义处理

(1) 表达式处理(产生表达式的中间代码)；

(2) “=” 的处理： “=” 左右部类型相容性  
检查

和转换；

## ■ 表达式翻译的语义子程序

(1)  $E \rightarrow E_1 \text{ OP } E_2$

{ **E.PLACE=NEWTEMP;**

**GEN(OP, E<sub>1</sub>.PLACE, E<sub>2</sub>.PLACE, E.PLACE)** }

(2)  $E \rightarrow \text{OP } E_1$

{ **E.PLACE=NEWTEMP;**

**GEN(OP, E<sub>1</sub>.PLACE, \_, E.PLACE)** }

(3)  $E \rightarrow \text{id}$

{ **E.PLACE=ENTRY(i)** }

- **控制流语句**: 改变程序执行顺序, 引起程序执行发生跳转的语句。
  - 程序设计语言中出现频繁的语句;
  - 为可执行语句, 要产生相应的目标代码;
  - 控制流程的变换, 依靠代码中的跳转指令与对应跳转的语句标号。

## ■ 控制流语句特点（动态）

改变程序执行顺序，引起程序执行发生跳转（向前或向后）。

## ■ 跳转目标与依据

语句标号 { **显式：** 位于源语句之前；  
（如，**L1:** goto L2;）  
**隐式：** 内含于源语句之中且在源程序中未标识的；

If (e<sub>r</sub>) S<sub>1</sub>; else S<sub>2</sub>

e<sub>r</sub>=T

跳转目标

e<sub>r</sub>=F

跳转目标

跳出if

继续

for ( e<sub>1</sub>; e<sub>2</sub>; e<sub>3</sub>) S ;

循环体

终值判别

循环体

开始

跳出循

环体继续

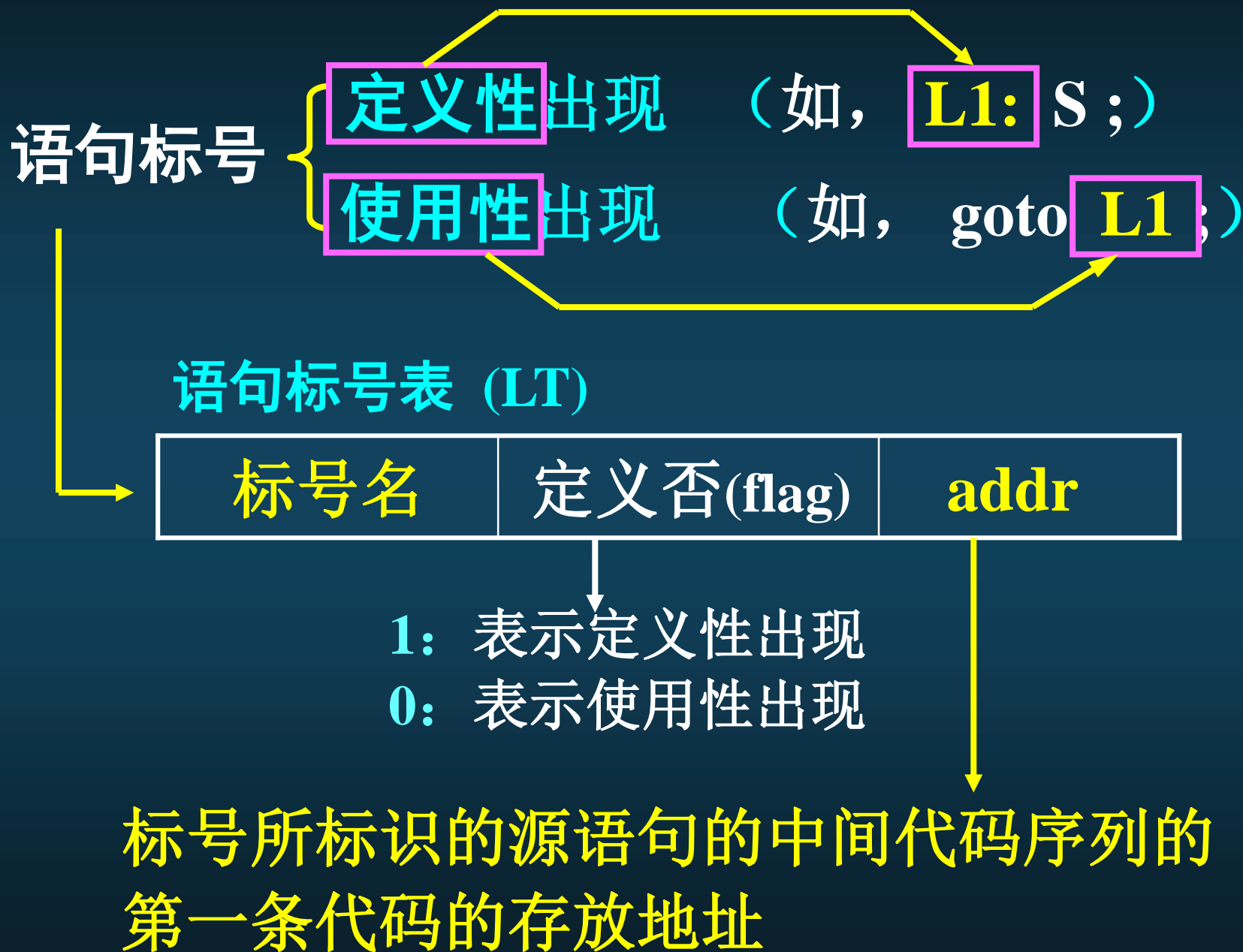
## ■ 语句标号处理与拉链-返填技术

控制流类语句处理面对의公共问题和实现技术;

适用于一遍扫描的编译器;

- 语句标号处理 {
  - 先定义后引用
  - 先引用后定义

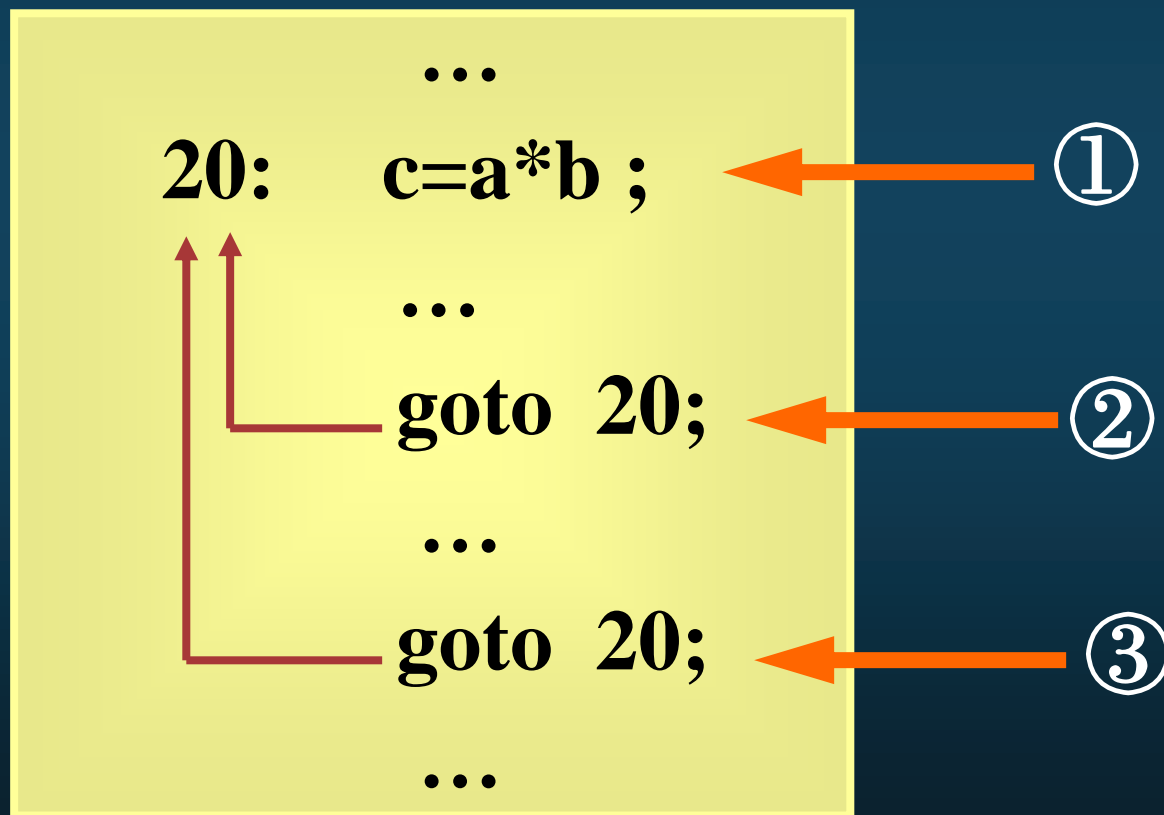


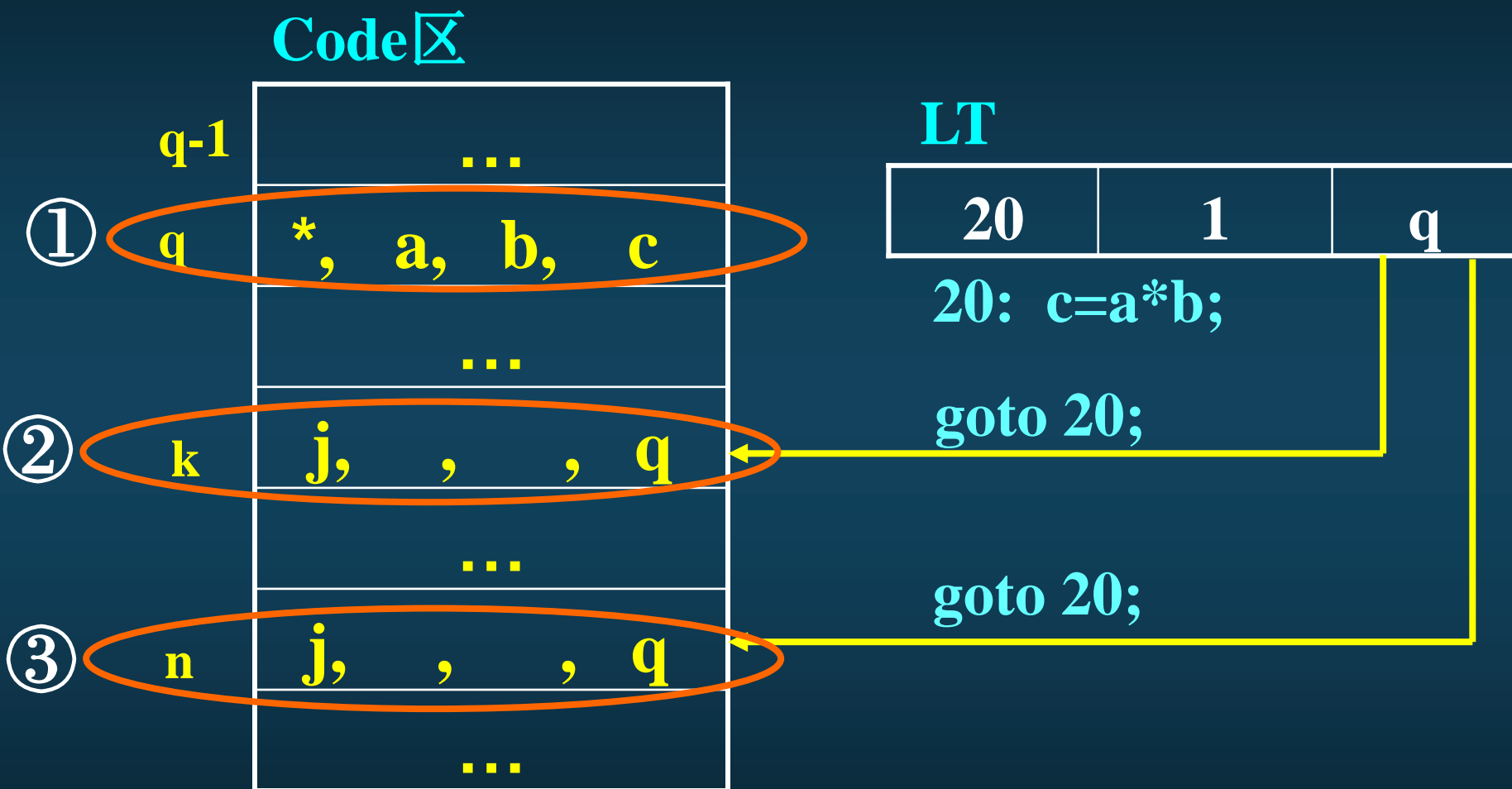


## ■ 对多遍扫描的编译器

视为一种情况(先定义后引用) 处理。

例如,

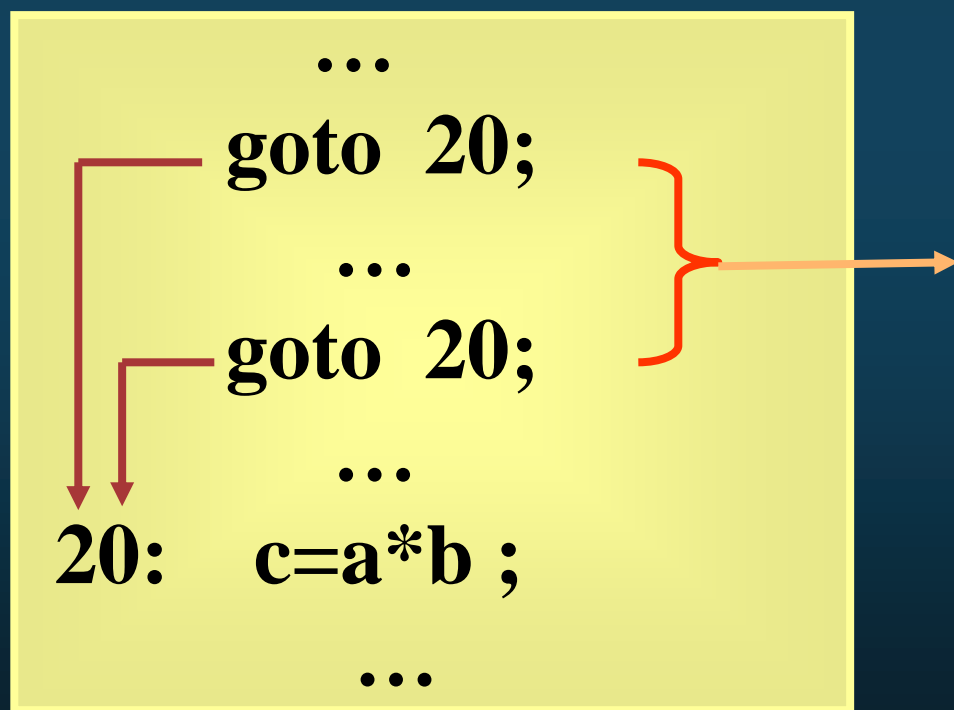




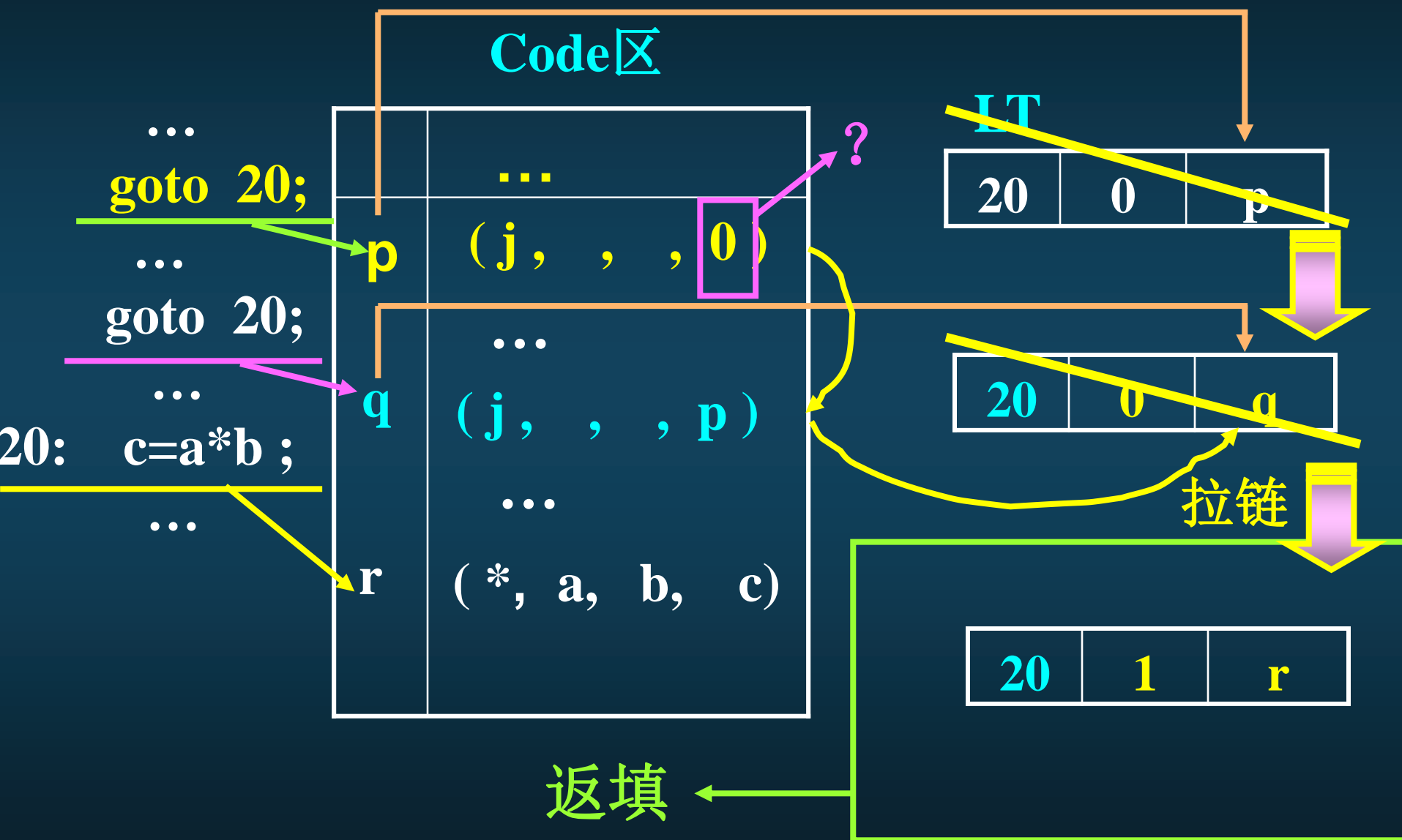
## ■ 对一遍扫描的编译器

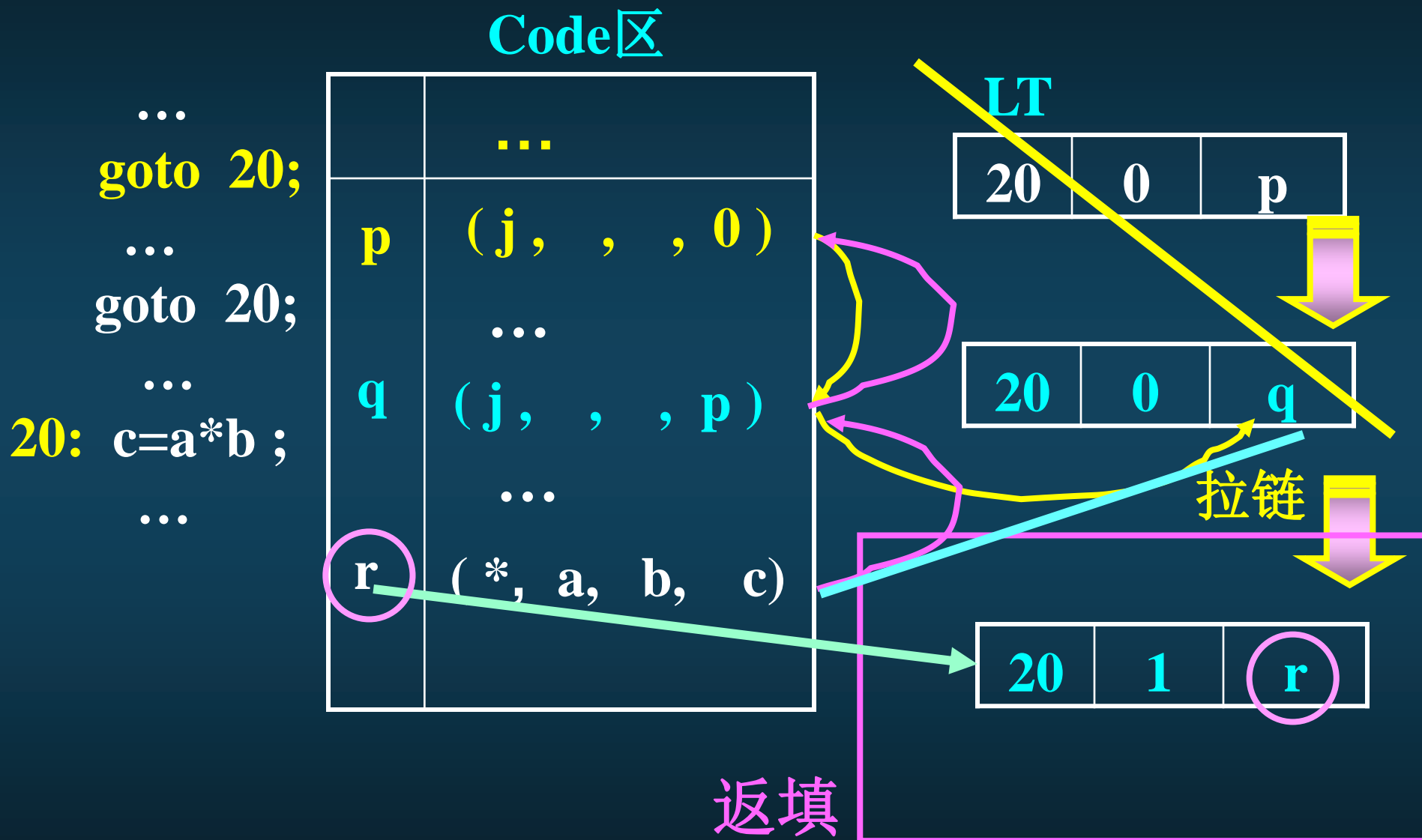
采用**拉链—返填**技术。

例如，



产生这2条语句的代码时，还未扫描到20号语句，即没有生成该语句的代码，因此转向的目标地址未定。





## Code区

```

...
goto 20;
...
goto 20;
...
20: c=a*b ;
...

```

|   |              |
|---|--------------|
|   | ...          |
| p | (j, , , r)   |
|   | ...          |
| q | (j, , , r)   |
|   | ...          |
| r | (*, a, b, c) |

## LT

|    |   |   |
|----|---|---|
| 20 | 0 | p |
|----|---|---|



|    |   |   |
|----|---|---|
| 20 | 0 | q |
|----|---|---|

拉链



|    |   |   |
|----|---|---|
| 20 | 1 | r |
|----|---|---|

## 🔥 注意:

1. 链尾在代码区的四元式中，链头在标号表中，链在与同一语句标号相关的跳转代码中；

2. **拉链次序**，链头在代码区的四元式中：

$p \longrightarrow q$  ( 在LT的addr )

3. **返填次序**：

$q \longrightarrow p$  ( 在代码区跳转指令中(addr=0) )





## 二. 条件语句的翻译 (P184~P185)

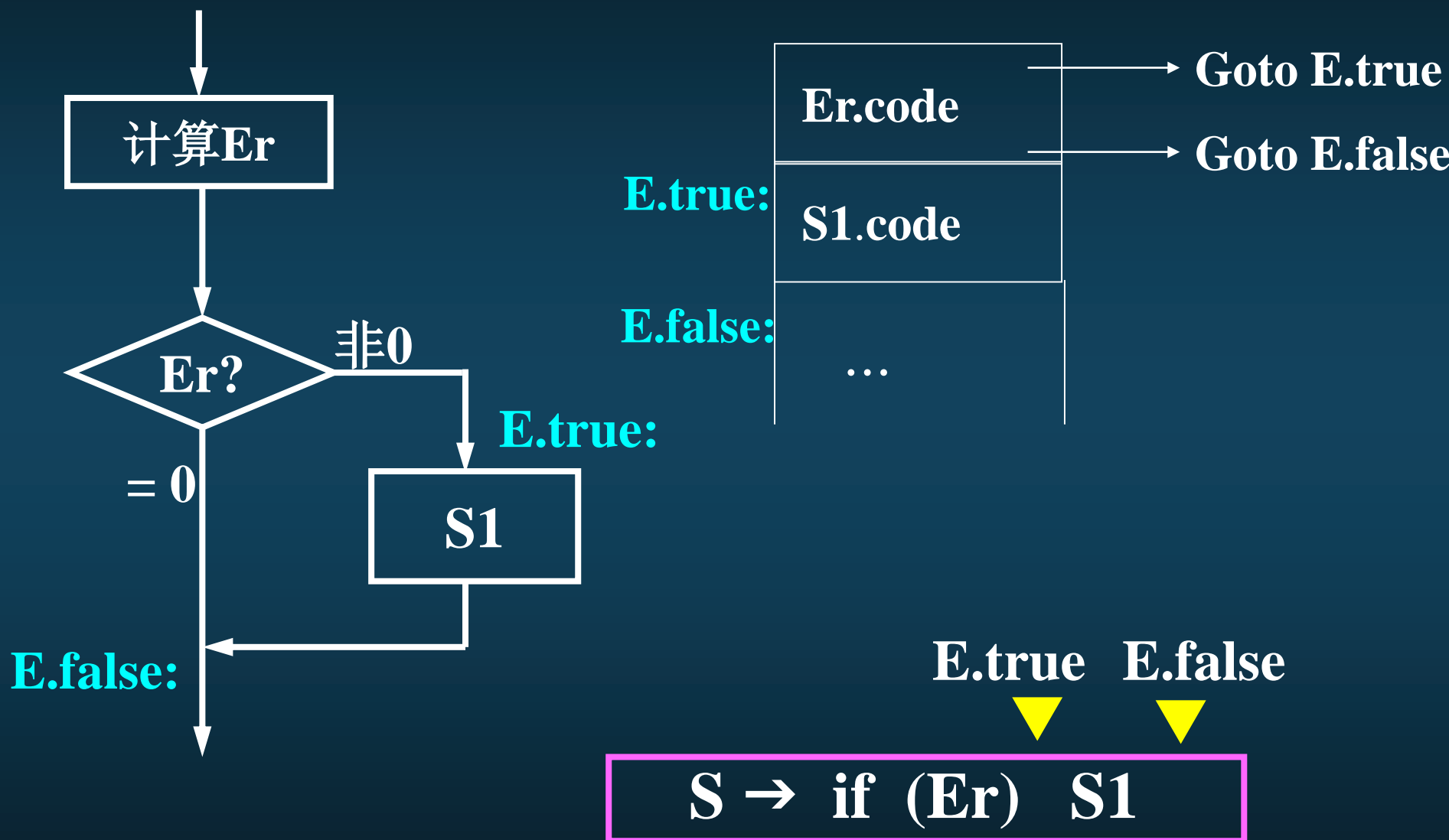
### IF Statement G :

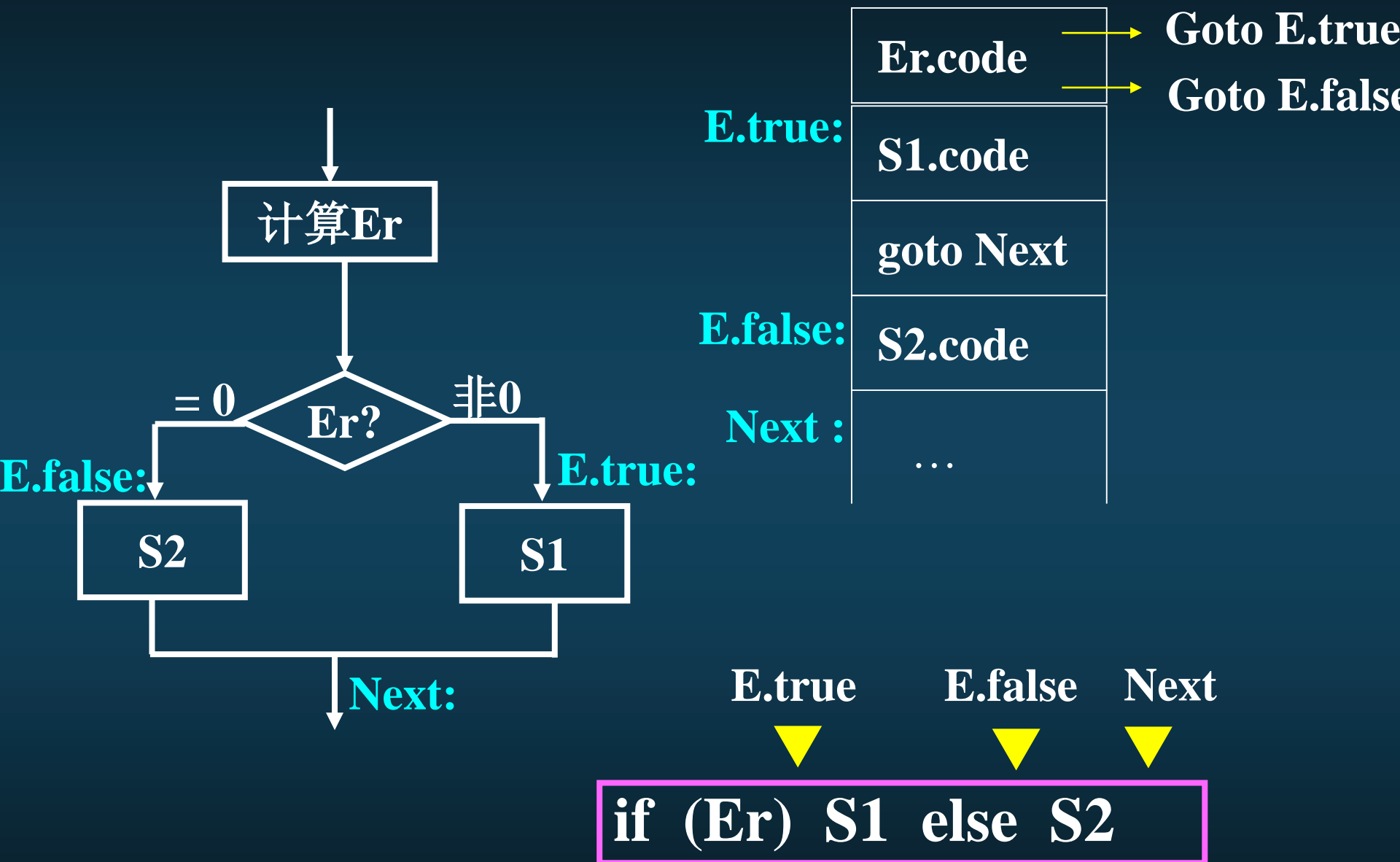
$S \rightarrow$  if (Er) S1 |  
if (Er) S1 else S2 |  
while (Er) S1

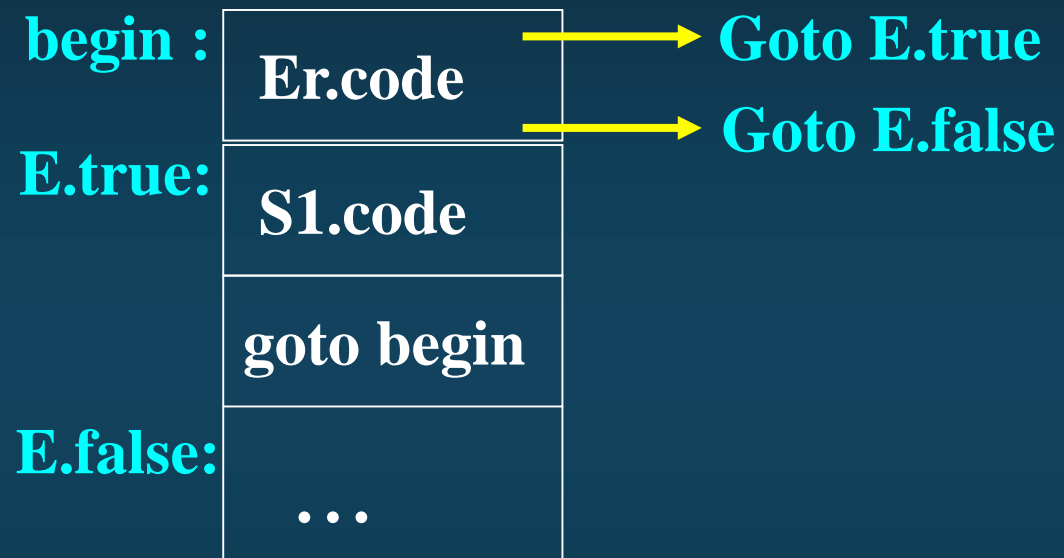
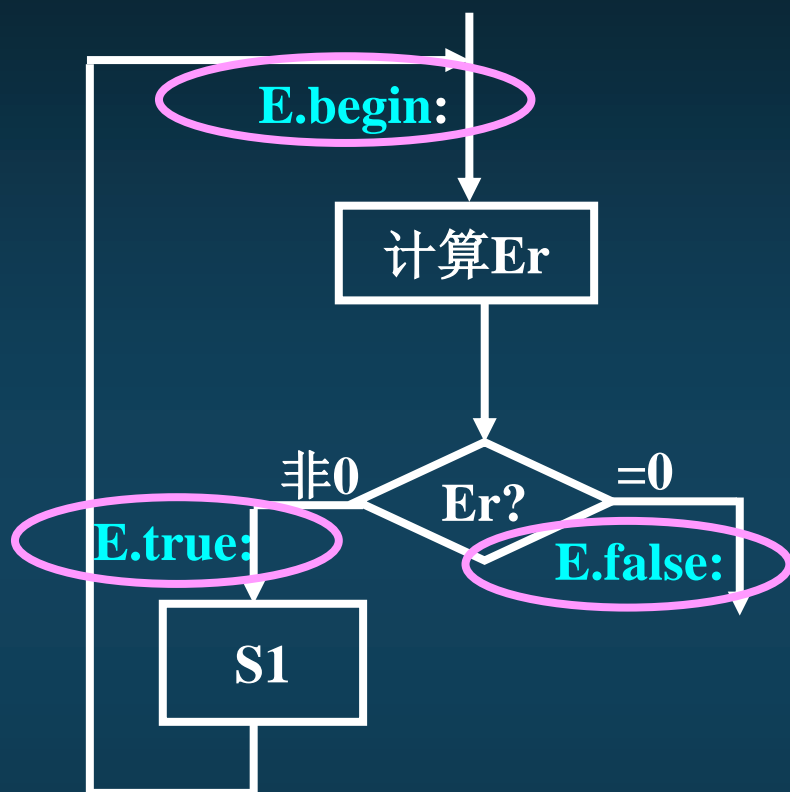
其中:

**Er:** 条件表达式 (有语义值 Er.True和  
Er.False);

**S1, S2:** 语句;







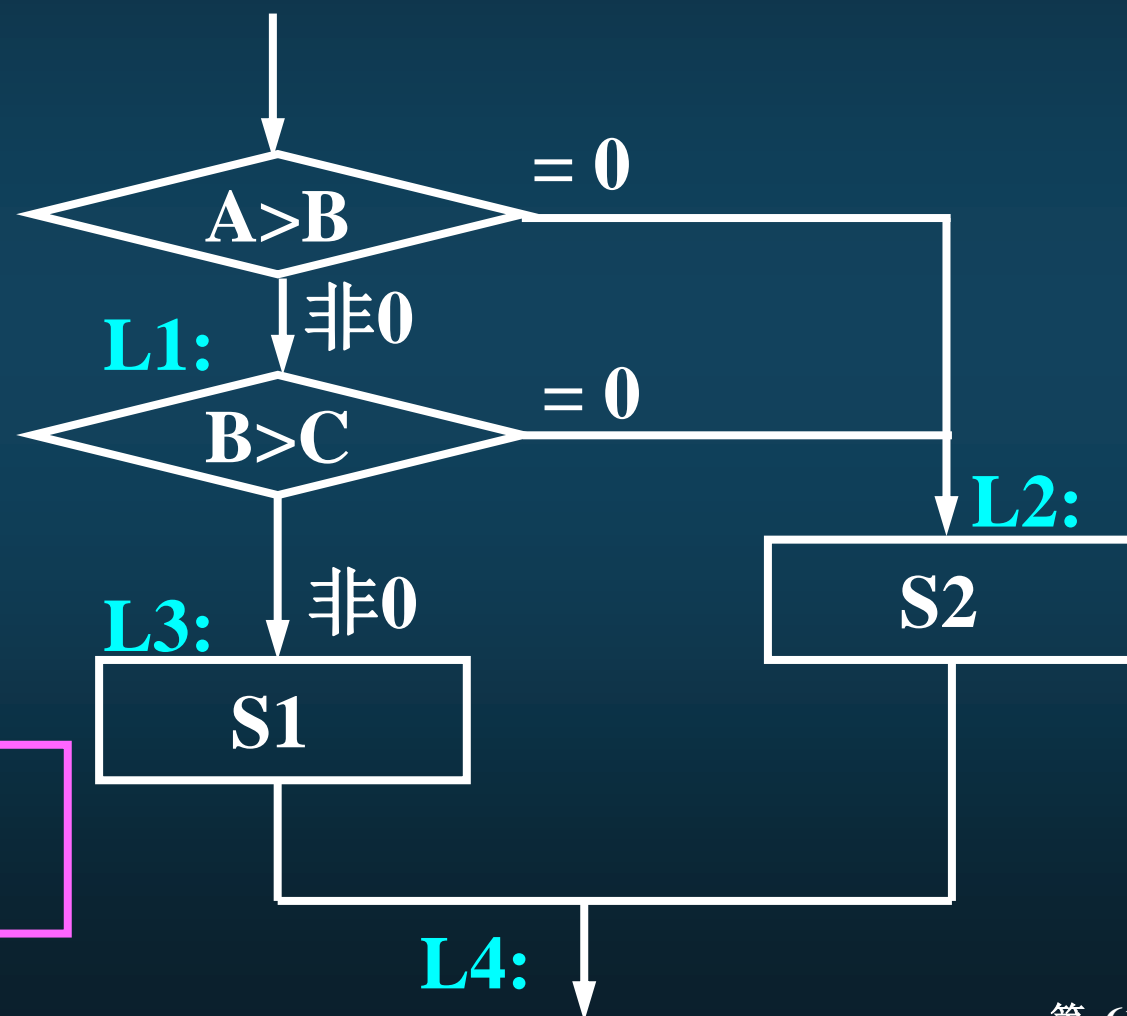
begin   E.true   E.false



**while (Er) S1**

**例6.2**    **if** (A>B && B>C) S1 **else** S2

**L1**
**L3**
**L2**
**L4**



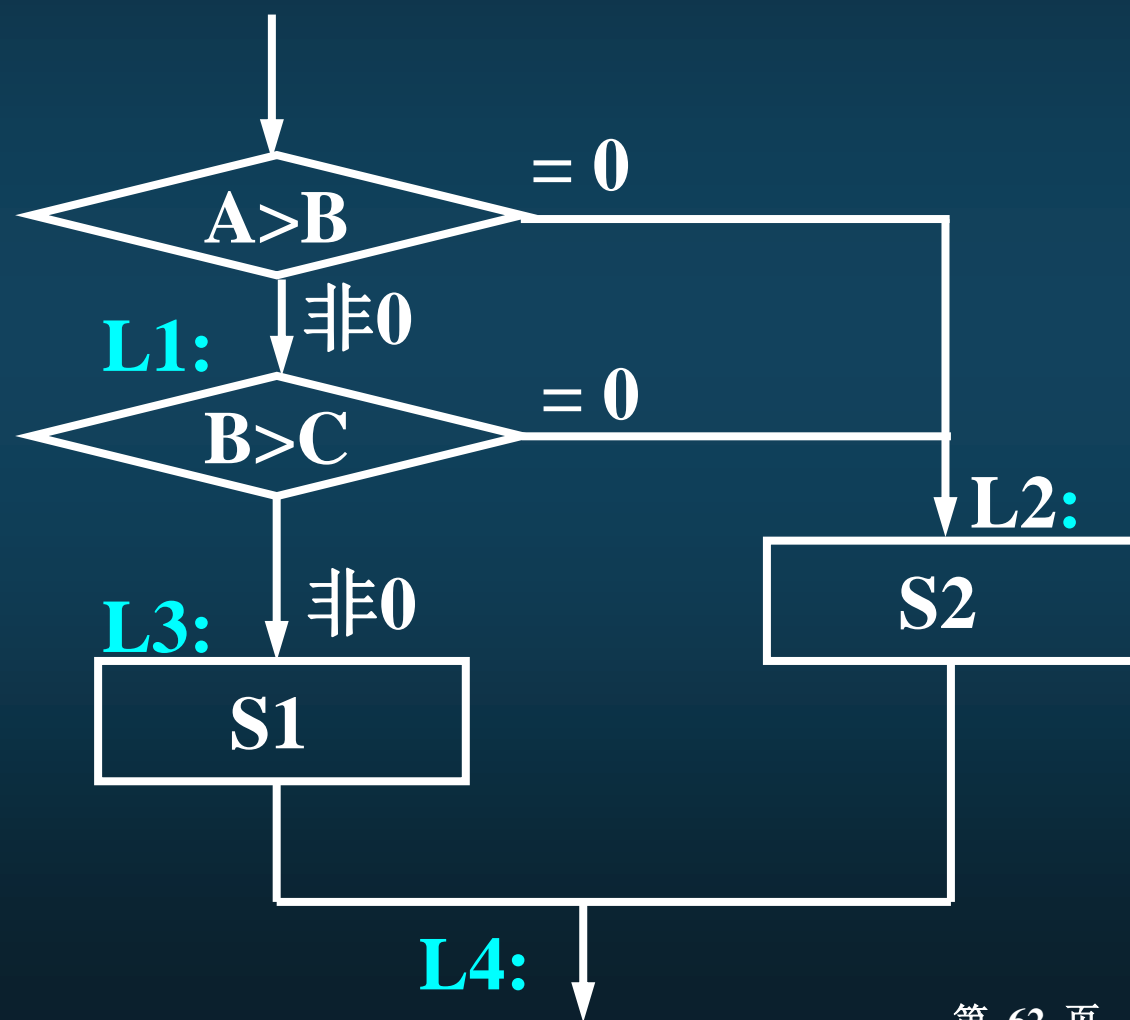
标号产生顺序:

**L1; L3; L2; L4;**

**例6.2**    `if (A>B && B>C) S1 else S2`





**L1**      **L3**
**L2**
**L4**

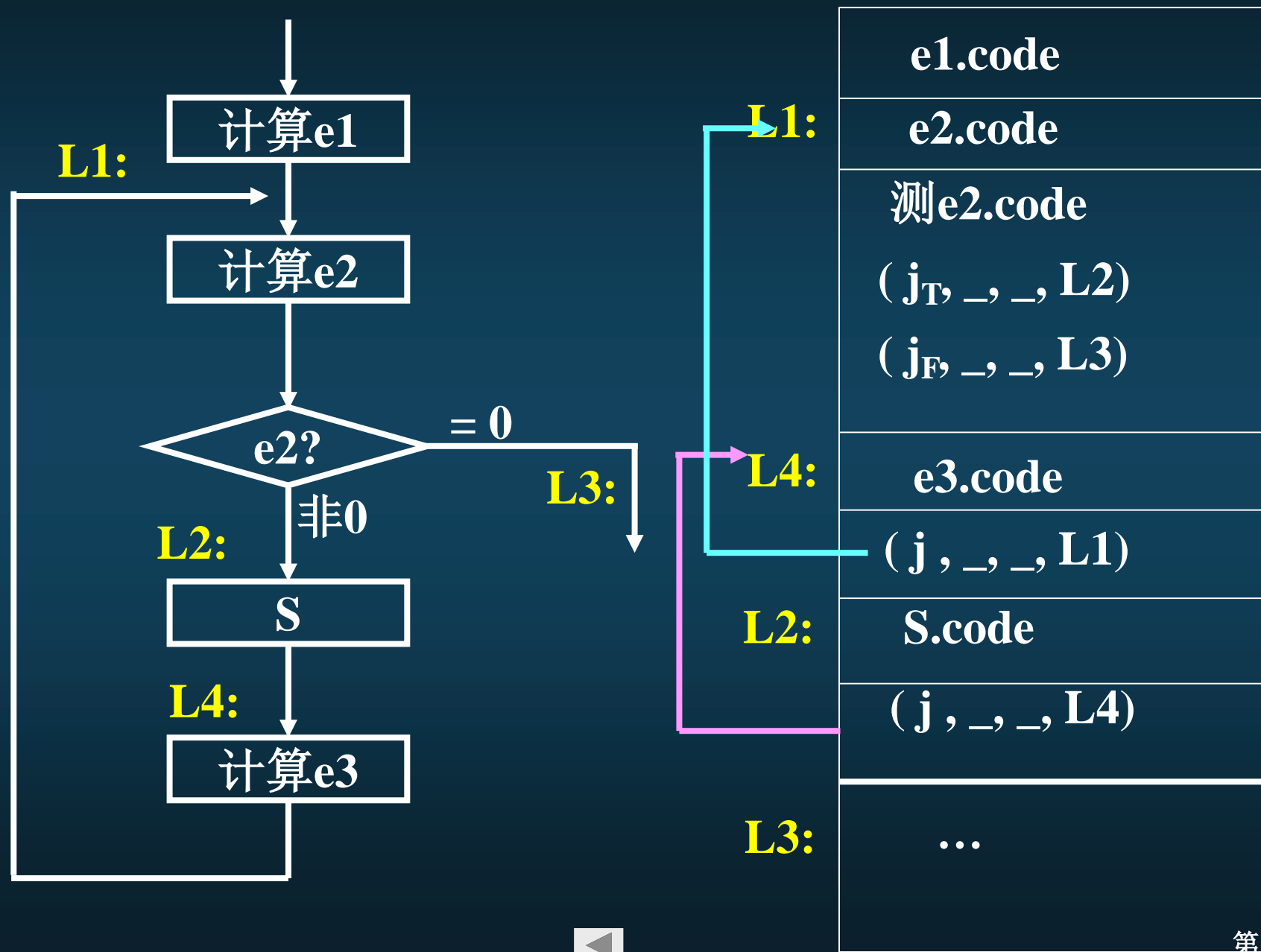
|   |  |
|---|--|
| 1 | (>, A, B, T <sub>1</sub> )               |
| 2 | (j <sub>F</sub> , T <sub>1</sub> , , 7 ) |
| 3 | (>, B, C, T <sub>2</sub> )               |
| 4 | (j <sub>F</sub> , T <sub>2</sub> , , 7 ) |
| 5 | <b>S1.code</b>                           |
| 6 | (j, , , 8 )                              |
| 7 | <b>S2.code</b>                           |
| 8 |  |



### 三. 循环语句的翻译 (P185)

$A \rightarrow \text{for} ( e1 ; e2 ; e3 ) S$

     
L1      L4      L2      L3





## 例6.3

给出如下C程序段的目标代码结构：

...

for ( e1; e2; e3;)

L1 L2

for ( t1; t2; t3;)

L3 L4 L5

if ( e<sub>r</sub> ) S<sub>1</sub>; L7<sub>for</sub> // S<sub>1</sub>是C语句

L6

L8<sub>for</sub> L9 S<sub>2</sub>;

// S<sub>2</sub>是C语句

...



|    |  |
|----|--|
| L1 | e1.code  |
|    | e2.code  |
| L2 | 测e2值<br>(j <sub>T</sub> , , , L3 )<br>(j <sub>F</sub> , , , L9 ) |
|    | e3.code  |
| L3 | (j, , , L1 )   |
| L4 | t1.code  |
|    | t2.code  |
| L5 | 测t2值<br>(j <sub>T</sub> , , , L6 )<br>(j <sub>F</sub> , , , L8 ) |
|    |  |

|    |  |
|----|--|
| L5 | t3.code  |
| L6 | (j, , , L4 )   |
|    | e <sub>r</sub> .code                                       |
| L7 | 测e <sub>r</sub> 值<br>(j <sub>F</sub> , , , L7 )<br>S1.code |
|    | (j, , , L5 )   |
| L8 | (j, , , L2 )   |
| L9 | S2.code  |
|    |  |

## 四. 多分支语句的翻译

$A \rightarrow \text{switch } (e)$

{ case  $c_1$  :  $S_1$  break ;

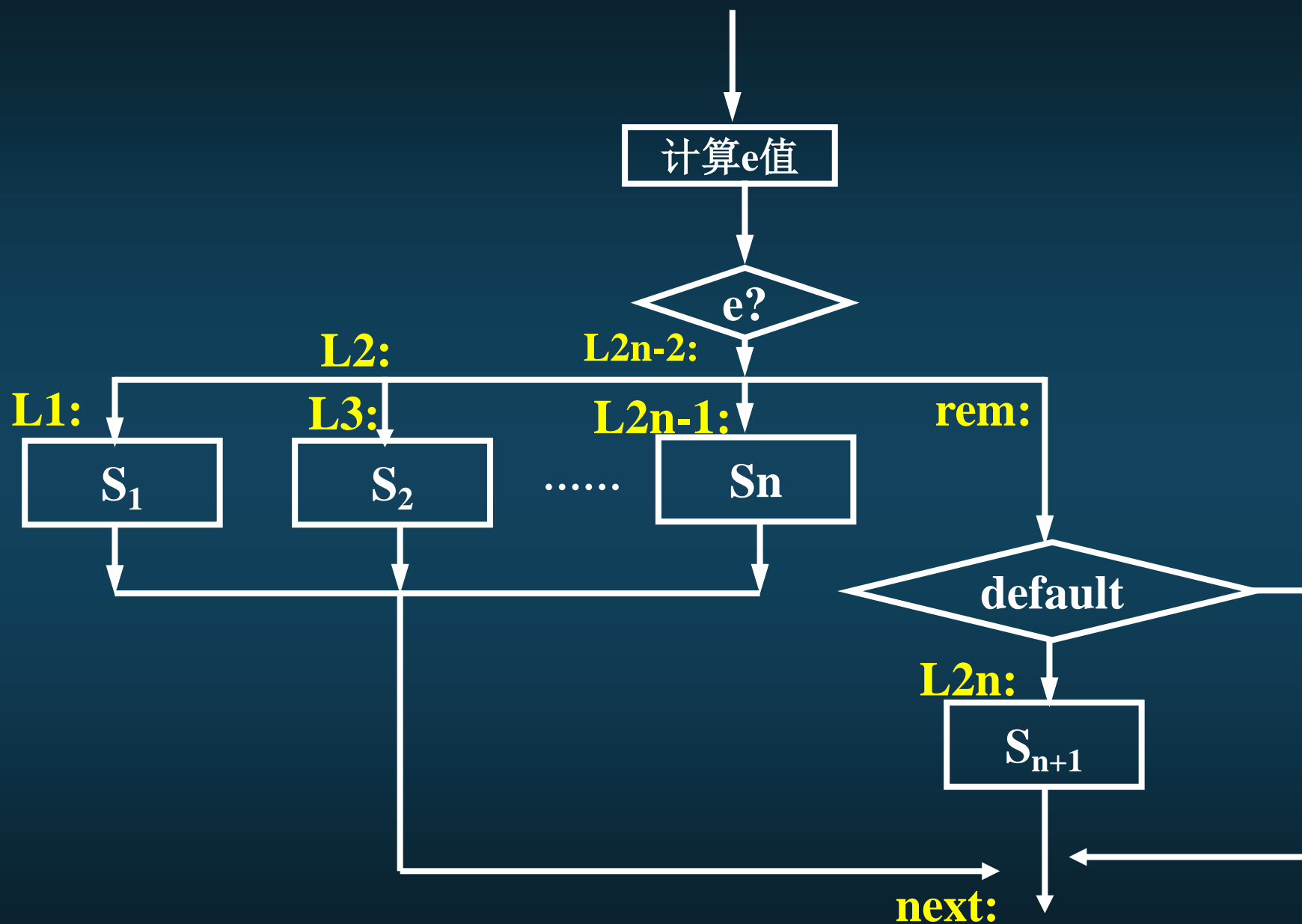
case  $c_2$  :  $S_2$  break ;

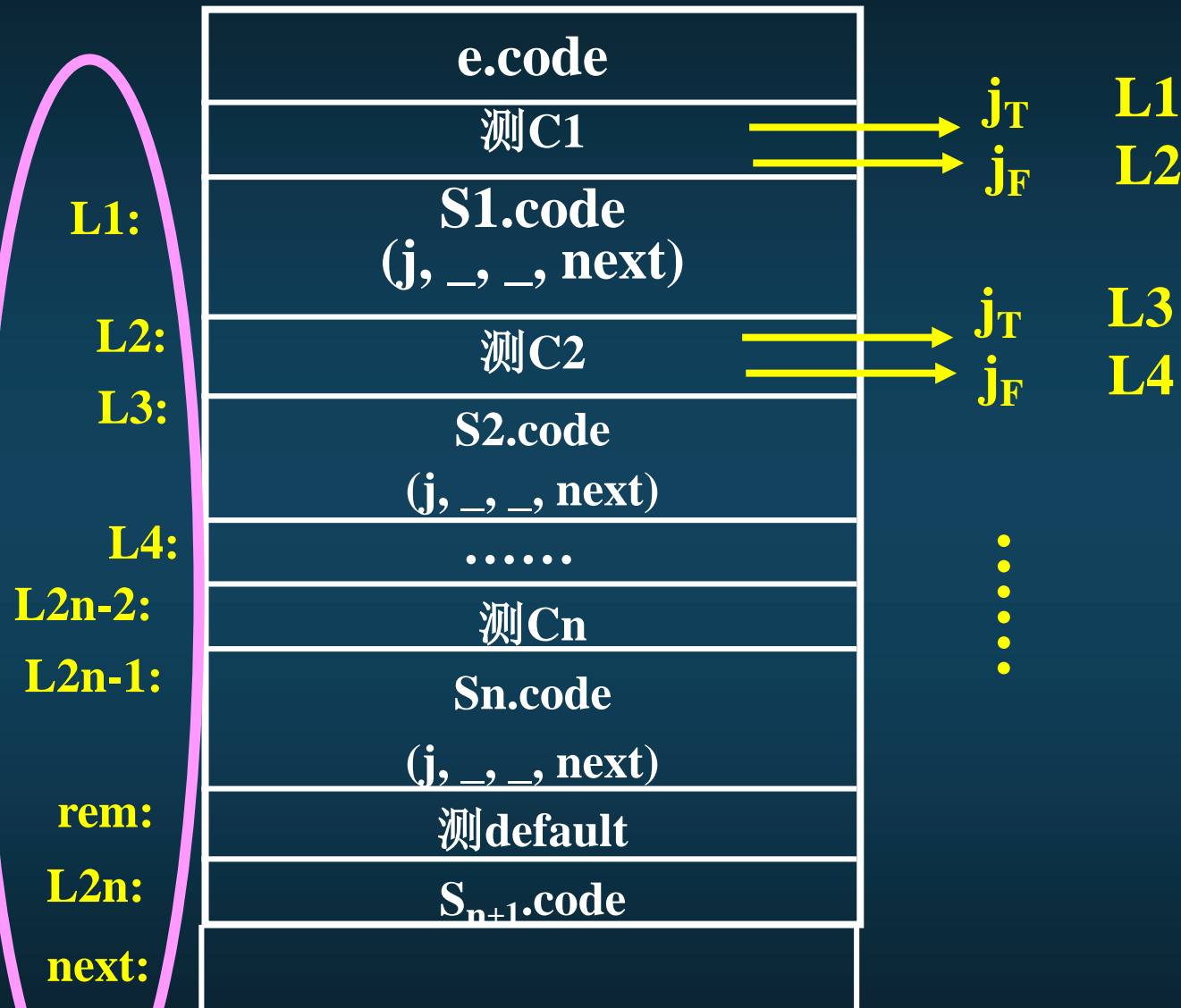
.....

case  $c_n$  :  $S_n$  break ;

default :  $S_{n+1}$

}





**A  $\rightarrow$  switch (e )**

**{ case  $c_1$  :  $S_1$  break ;**

**case  $c_2$  :  $S_2$  break ;**

**.....**

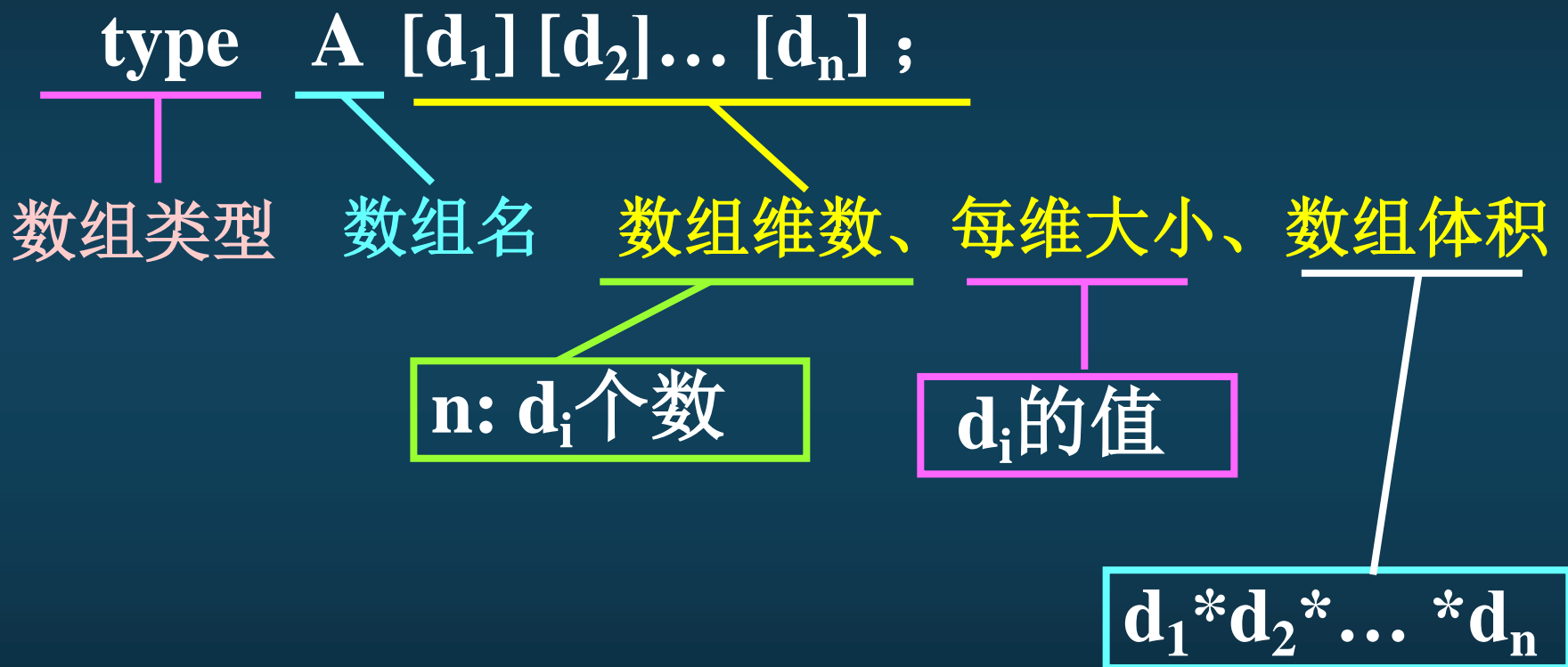
**case  $c_n$  :  $S_n$  break ;**

**default :  $S_{n+1}$**

**}**

|       |   |       |
|-------|---|-------|
|       | e.code  |       |
|       | (j, _, _, test)   |       |
| L1:   | S1.code   |       |
|       | (j, _, _, next)   |       |
| L2:   | S2.code   |       |
|       | (j, _, _, next)   |       |
|       | .....   |       |
| Ln:   | Sn.code   |       |
|       | (j, _, _, next)   |       |
| Ln+1: | Sn+1.code   |       |
|       | (j, _, _, next)   |       |
| test: | If $e=c_1$ goto L1<br>If $e=c_2$ goto L2<br>.....<br>If $e=c_n$ goto Ln<br>If default goto Ln+1 | } 测试表 |
| next: |   |       |

# 一. 数组说明



type A [d<sub>1</sub> .. d<sub>1</sub>'] [d<sub>2</sub> .. d<sub>2</sub>'] ... [d<sub>n</sub> .. d<sub>n</sub>'] ;



## 数组说明的语义处理

填表，登记数组的属性信息。

type

name

dim

dim\_value

vol

符号表

公共、等长信息  
(符号表)

与计算数组元素地址  
有关、不等长信息  
(信息向量表)

## 例7.4 设有说明

```
int a[2][2];
```

```
float b[4];
```

namelist

| name | kind  | type | addr |
|------|-------|------|------|
| a    | array | I    |      |
| b    | array | R    |      |

信息向量表

|                |   |
|----------------|---|
| 2              | a |
| 2              |   |
| 2              |   |
| 2*2            |   |
| C <sub>a</sub> |   |
| a <sub>0</sub> | b |
| 4              |   |
| 1              |   |
| C <sub>b</sub> |   |
| b <sub>0</sub> |   |

## 信息向量表

|                   |                                |
|-------------------|--------------------------------|
| A <sub>0</sub> -C | 第一维大小                          |
|                   | 第二维大小                          |
|                   | ...                            |
|                   | 第n维大小                          |
|                   | 维数                             |
|                   | 体积 = $d_1 * d_2 * \dots * d_n$ |
|                   | 计算数元地址不变部分                     |
|                   | 数组第一个元素地址                      |
|                   | ...                            |

若为上下界需计算:

$$a(2..10) \longrightarrow 10-2+1=9$$

何时计算?

如果为动态数组如何  
处理?

## 二. 数组元素引用

type A [d<sub>1</sub>] [d<sub>2</sub>]... [d<sub>n</sub>] ;      数组说明

A [i<sub>1</sub>] [i<sub>2</sub>]... [i<sub>n</sub>]      数组引用

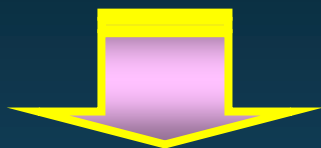
**\*\* 不能整体引用，仅对单一数组元素引用。**

### ➤ 数组元素引用的语义处理

语义检查： 类型匹配； 下标越界检查；

产生代码： 数组元素地址计算的中间代码。

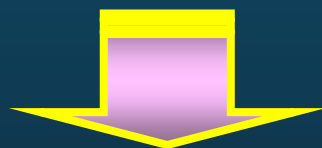
数组元素地址计算编译时能否完成？



一般不能

∵  $A[i_1][i_2] \dots [i_n]$  中  $i_k$  多为表达式

如,  $a[i+j-1][j++]$       运行时得到下标值



涉及数组元素地址计算的有关知识

- 数组存储方式 (线性连续)
  - 按行存储
  - 按列存储

例如, `int a[2][2];`

**按行**

|       |                      |
|-------|----------------------|
| $a_0$ | <code>a[0][0]</code> |
|       | <code>a[0][1]</code> |
|       | <code>a[1][0]</code> |
|       | <code>a[1][1]</code> |

**按列**

|       |                      |
|-------|----------------------|
| $a_0$ | <code>a[0][0]</code> |
|       | <code>a[1][0]</code> |
|       | <code>a[0][1]</code> |
|       | <code>a[1][1]</code> |

$$a[0][1]_{\text{add}} = a_0 + 1 * \text{int\_size}$$

$$a[0][1]_{\text{add}} = a_0 + 2 * \text{int\_size}$$

## ■ 数组元素地址计算

据存储方式通过规则计算数组元素位置

$$(\text{顺序号}) + a_0$$

■ 对一维数组 `int a[n];` // 默认下标下界=0

则  $a[i]_{\text{addr}} = a_0 + i$

( 若数组下标下限=1, 则  $a[i] = a_0 + (i-1)$  )

■ 对二维数组 `int a[n][m];`

则 
$$a[i][j]_{\text{addr}} = a_0 + i * m + j$$

若数组下标下限=1，则

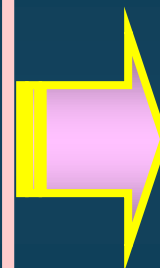
$$a[i][j]_{\text{add}} = a_0 + (i-1) * m + (j-1)$$



■ 对n维数组  $\text{int } a[d_1][d_2] \dots [d_n];$

则  $a[i_1][i_2] \dots [i_n]$  add

$$\begin{aligned}
 &= a_0 + i_1 * d_2 * d_3 * \dots * d_n \\
 &+ i_2 * d_3 * d_4 * \dots * d_n \\
 &+ \dots \\
 &+ i_{n-1} * d_n + i_n
 \end{aligned}$$



含  $i_k$ , 是可变部分, 程序运行时方可知。

进一步考虑更一般的情况: 若数组下标下限  $\neq 0$

则  $a[i_1][i_2] \dots [i_n]_{\text{add}}$

$$= a_0 + (i_1 - 1) * d_2 * d_3 * \dots * d_n$$

$$+ (i_2 - 1) * d_3 * d_4 * \dots * d_n$$

+ ...

$$+ (i_{n-1} - 1) * d_n + (i_n - 1)$$

变换

V

$$= a_0 + i_1 d_2 d_3 \dots d_n + i_2 d_3 d_4 \dots d_n + \dots + i_{n-1} d_n + i_n - (d_2 d_3 \dots d_n + d_3 d_4 \dots d_n + \dots + d_n + 1)$$

不变

C

$$a[i_1] [i_2] \dots [i_n]_{\text{add}}$$

$$V$$

$$= a_0 + i_1 d_2 d_3 \dots d_n + i_2 d_3 d_4 \dots d_n + \dots + i_{n-1} d_n + i_n$$

$$- (d_2 d_3 \dots d_n + d_3 d_4 \dots d_n + \dots + d_n + 1)$$

不变

$$C$$

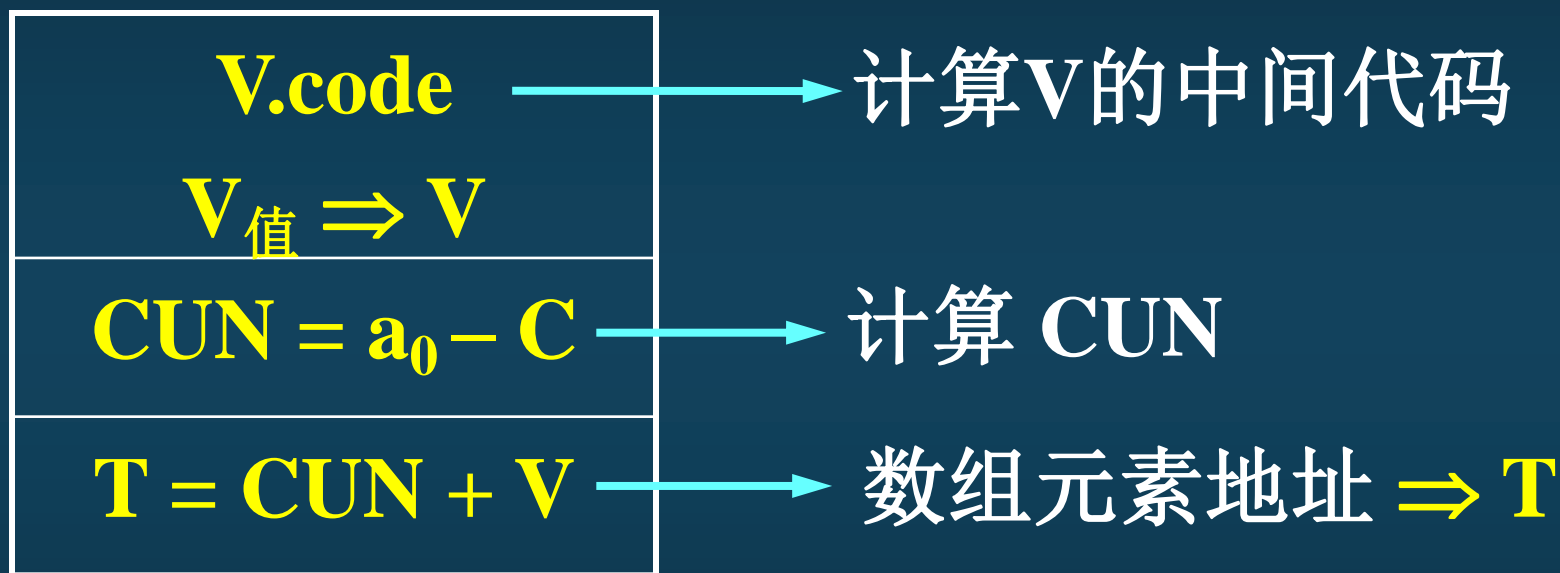
$$= \frac{a_0 - C}{\text{conspart}} + \frac{V}{\text{varpart}}$$

$$= CUN + V$$

编译时计算,填入内情向量表

据数元引用情况,产生计算的中间代码,程序运行时算出。

## ■ 数组元素引用目标结构



## 例6.5 设有说明

```

...
int a[10][20]
...
s = a[x][y];
...

```

namelist

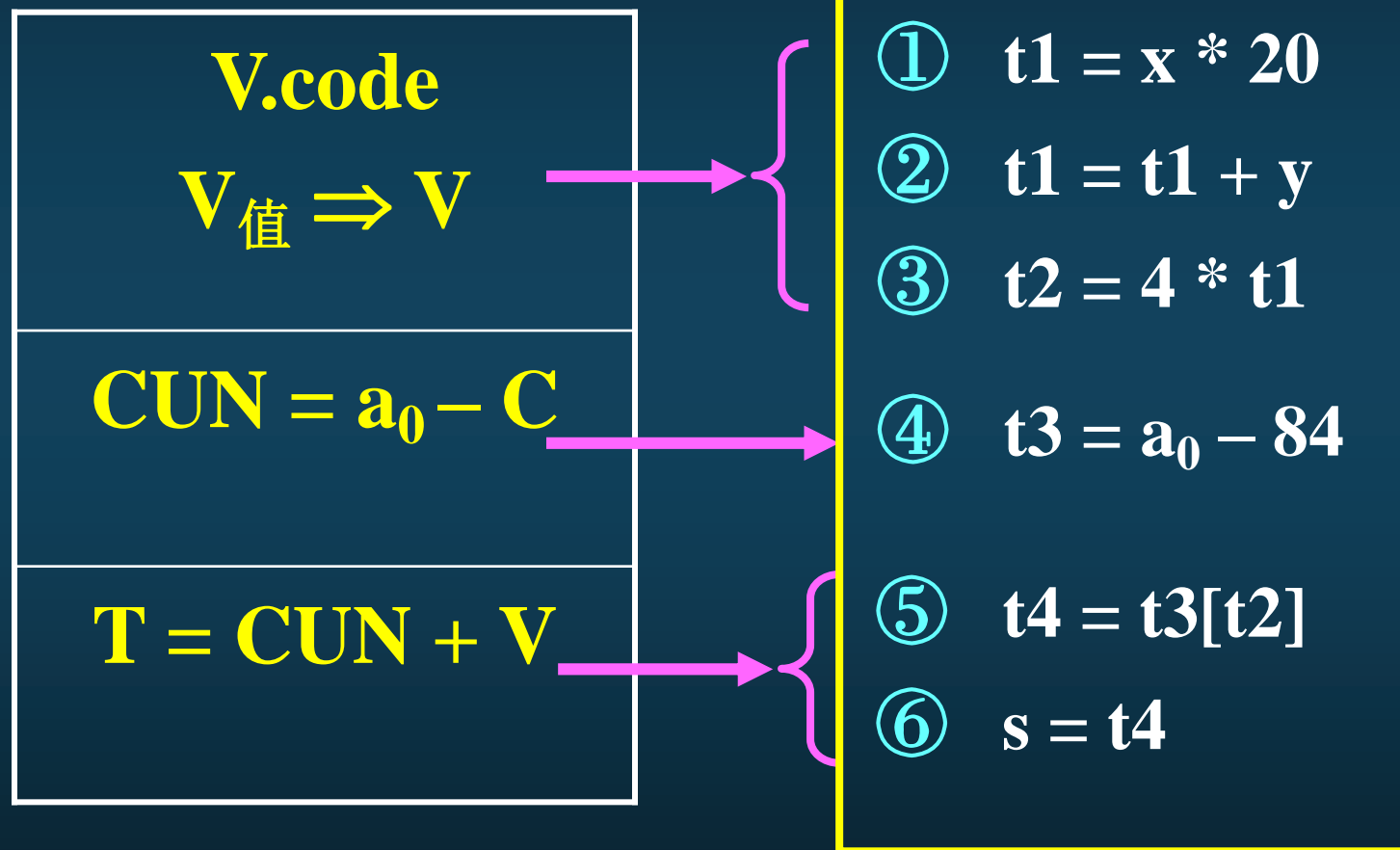
| name | kind  | type | addr |
|------|-------|------|------|
| a    | array | I    |      |
| ...  |       |      |      |

内情向量表

|                         |
|-------------------------|
| 10                      |
| 20                      |
| 2                       |
| $10 * 20 = 200$         |
| $C = (20 + 1) * 4 = 84$ |
| $a_0$                   |
| ...                     |

$s = a[x][y]$

$a[x][y]_{\text{addr}} \bullet \text{code}$



## ■ 动态数组？

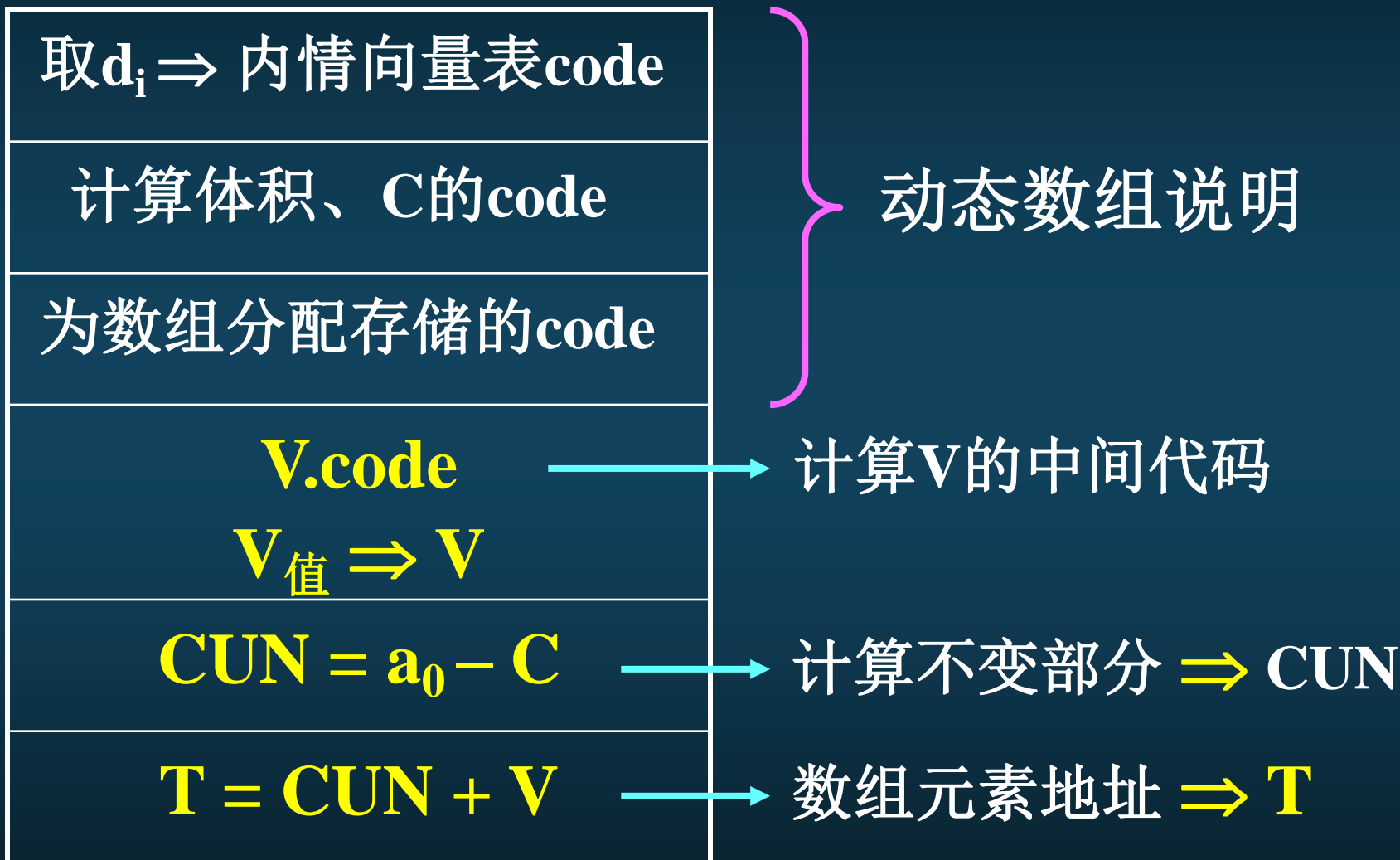
type A [ $d_1$ ][ $d_2$ ]... [ $d_n$ ];    ( $d_i$ 是变量)  
(每维大小、体积等信息程序运行时确定)

## ■ 动态数组处理

内情向量表在程序运行时建立，编译时仅分配占内情向量表存储空间(据维数)。即运行时， $d_i$ 确定后，计算体积、计算C、为数组分配存储空间且将这些信息登录内情向量表。

∴ 动态数组：产生此部分代码 + 静态数组元素引用

## ■ 动态数组处理目标结构





## 函数(过程)翻译



- 函数说明
- 函数调用

- 函数(过程)说明和函数调用是一种常见的语法结构,其形式随语言的不同而有所不同。
- 函数(过程)说明方式有的语言说明说明由关键字(**PROCEDURE**, **FUNCTION**)引导,有些语言则可直接定义。

## 一. 函数说明的翻译

1. 函数及局部量信息登录符号表,并填入有关的属性: 种属(过程或函数等)、是否为外部过程、数据类型(对函数而言)、形参个数、形参的信息(供语义检查用,如种属、类型等)、过程的入口地址等等。

函数形参的信息可以登录子表,并以某种方式和函数名的登记项连接起来。

## 一. 函数说明的翻译 (续)

2. 为每个形式参数分配相应的存储单元,称为形式单元,供形实结合时传递信息之用.并将形参的名字、相应形式单元的地址, 以及此形参的其它一些属性记入符号表。

具有嵌套结构的语言,为确保程序中全局量和局部量能得到正确的引用,符号表及各量的数据空间是按嵌套的层次建立、分配的。  
即记录函数中局部量的作用域(Level);

## 例6.6 设有C函数

```
fun1( int a,b)
{
    int c;
    c=a+b ;
    return ( c )
}
```

namelist

| name | kind | type | len | offset | addr |
|------|------|------|-----|--------|------|
| fun1 | 函数   | I    |     |        |      |
| a    | 形参   | I    | 4   | 0      |      |
| b    | 形参   | I    | 4   | 4      |      |
| c    | V    | I    | 4   | 8      |      |

code

## 一. 函数说明的翻译 (续)

3. 当扫描到函数说明中的函数体时，产生执行函数体的代码时需完成：

- (1) 产生将返回地址推入堆栈的代码；
- (2) 产生形实结合的代码；
- (3) 产生有关从函数返回的代码；

## 二. 函数调用的翻译

G:  $S \rightarrow \text{call id (Elist)}$

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

其中: **id:** 函数名;

**E:** 表达式。

**Elist:** 实参表;

- 函数（过程）说明和函数（过程）调用的形式因不同的语言而异，但在功能上和需做的语义处理工作上基本类似。
- 函数（过程）说明和函数（过程）调用的翻译,还依赖于形式参数与实在参数结合的方式以及数据存储空间的分配方式。

## ■ 函数调用的语义处理:

- ① 检查所调用的过程或函数是否定义；与所定义的过程或函数的类型、实参与形参的数量、顺序及类型是否一致；
- ② 给被调过程或函数申请、分配活动记录所需的存储空间；（ch7）
- ③ 计算并传送实参；
- ④ 加载调用结果和返回地址，恢复主调用过程或函数的继续执行；
- ⑤ 转向相应的过程或函数（转子指令）。

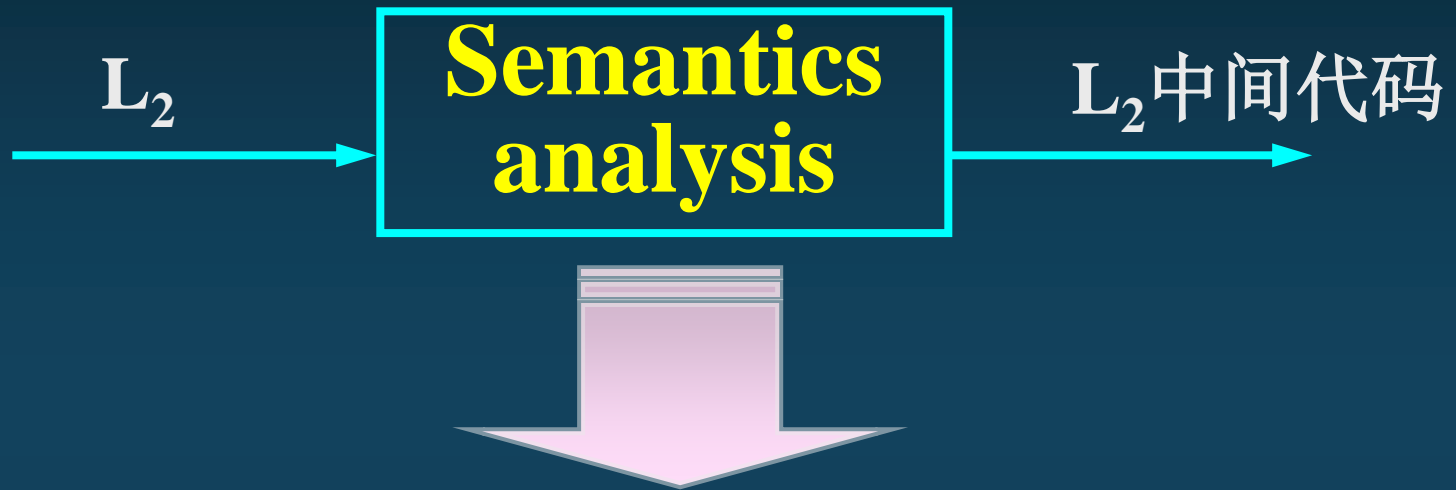


## ■ 函数调用返回的语义处理

- ① 返回一个或多个结果值存于指定位置;
- ② 恢复调用过程的活动记录;
- ③ 生成一条转移指令（返回）；

# about semantics analysis

---



根据语义进行语义检查和代码生成

对 $L_2$ 扫描的**顺序性**；产生的代码执行的**线性化**；  
语义映射的**准确性**；

# about semantics analysis

---

说明性语句



登录编译信息、为可执行语句提供语义检查和语句翻译的信息依据。

可执行语句



目标代码结构



中间代码

动态语义

语义动作